TP 2
Application messagerie réseau en Java

Le but de ce TP est de réaliser une application réseau d'envoi et de réception de messages. Dans un premier temps, ces messages seront simplement des chaînes de caractères. L'application comporte deux parties : une partie Serveur et une partie Client. Chacune d'entre elle utilise plusieurs threads, et le protocole réseau choisi pour implémenter cette application est **TCP**. Dans un premier temps, les deux parties client et serveur s'exécuteront en ligne de commande.

Description simplifiée du serveur :

Les rôles du serveur sont les suivants :

- accepter les nouvelles connexions de clients
- recevoir les messages de chaque client, et les diffuser à tout le monde
- notifier les clients des nouvelles connexions et des déconnexions d'autres clients

Description simplifiée du client :

Un client doit d'abord se connecter au serveur. Lorsque la connexion est établie, il a deux tâches :

- inviter l'utilisateur à saisir un message pour qu'il soit ensuite envoyé au serveur
- réceptionner les messages du serveur : messages d'autres clients, connexions ou déconnexions d'autres clients

Architecture générale de l'application :

Créez deux **packages** : **client** et **server**. Dans le package **client**, créez une classe **MainClient** qui comportera la méthode main pour démarrer un client. Dans le package **server**, créez une classe **MainServer** qui comportera la méthode main pour démarrer un serveur. Le code de ces deux classes est donné en annexe de ce document.

0) Classes communes aux deux parties

Avant de commencer, créez un package **common** et ajoutez-y une classe **Message** qui servira à encapsuler nos messages.
La classe **Message** comporte deux attributs **sender** et **content** de type **String**. Elle possède un constructeur qui initialise les deux attributs, et une méthode **toString** qui retourne les deux attributs.
Puisque les messages seront échangés dans le réseau, il faut pouvoir les transformer en suite d'octets. On indique cette capacité grâce à l'interface **Serializable** que notre classe **Message** doit implémenter :

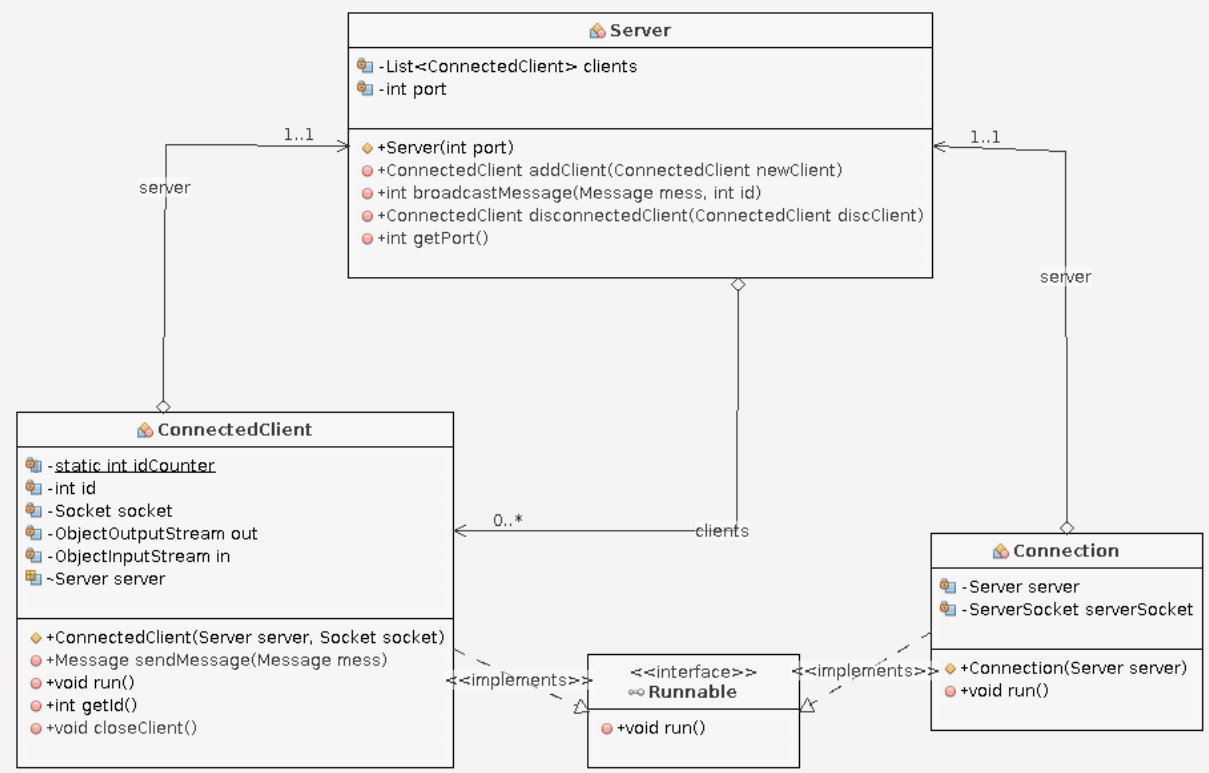
```
public class Message implements Serializable{
```

Vous noterez que cette interface ne nécessite l'ajout d'aucune méthode dans notre classe ; certains IDE vous demanderont l'ajout d'une variable statique

```
public static final long serialVersionUID
```

mais nous ne rentrerons pas dans la signification de cette variable. Affectez lui n'importe quelle valeur `long`, par exemple 1L

1) Package **server**



a) Commençons la classe **Server**

Cette classe comporte deux attributs :

- **port**, de type **int**, qui correspond au port sur lequel le serveur va écouter les nouvelles connexions
- **clients**, de type **List<ConnectedClient>**, qui va stocker la liste des clients connectés

Elle comporte un constructeur **Server(int)**, qui prend en entrée un port, et dont le rôle est le suivant :

- initialiser l'attribut **port**
- initialiser la liste **clients** (liste vide au départ) :

```
this.clients = new ArrayList<ConnectedClient>();
```

- lancer un thread à partir de la classe **Connection**. On rappelle le code pour lancer un tel thread :

```
Thread threadConnection = new Thread(new Connection(this));
threadConnection.start();
```

En effet, comme nous allons le voir, le constructeur de **Connection** prend en paramètre une référence vers le serveur (objet courant **this**).

b) la classe **Connection**

Cette classe représente un thread : elle doit donc implémenter l'interface **Runnable** (du package **java.lang**) qui force à déclarer une méthode **run()** : la méthode de lancement du thread. La classe **Connection** comporte deux attributs :

- un objet **server** de type **Server** (référence vers le serveur qui l'a créé).

- un objet **serverSocket** de type **ServerSocket**, permettant d'accepter de nouvelles connexions.

Le constructeur de la classe **Connection** comporte en paramètre le serveur qui l'a instancié. Il initialise l'attribut **server** avec ce paramètre, et initialise l'attribut **serverSocket** à l'aide du port du serveur :

```
this.serverSocket = new ServerSocket(server.getPort());
```

La méthode **run()** de la classe **Connection** est une boucle infinie attendant la connexion de nouveaux clients, et récupère un socket lorsqu'un client se connecte :

```
Socket sockNewClient = serverSocket.accept();
```

Lorsqu'une connexion est acceptée (on rappelle que la méthode **accept()** est bloquante l'exécution de l'application est stoppée tant qu'il n'y a pas de connexion), elle crée un nouvel objet **ConnectedClient** en lui spécifiant deux paramètres :

- une référence vers le **server**
- le socket créé **sockNewClient**

```
ConnectedClient newClient = new ConnectedClient(server, sockNewClient);
```

Il faut ensuite l'ajouter au serveur :

```
server.addClient(newClient);
```

(nous n'avons pas encore implémenté la méthode **addClient**, patience !).

Enfin, elle lance un thread à partir de l'objet **ConnectedClient** tout juste créé :

```
Thread threadNewClient = new Thread(newClient);
threadNewClient.start();
```

Attention : ne pas oublier de placer des instructions **try { ....} catch (...) { ... }** autour du code afin de capturer les éventuelles exceptions. Tout bon IDE (Eclipse, NetBeans...) vous l'aura alors suggéré.

c) La classe **ConnectedClient**

Cette classe comporte 6 attributs :

- **idCounter**, de type **int**. Attribut **static**, il va permettre d'attribuer des identifiants uniques à chaque client. C'est simplement un compteur d'instances créées
- **id**, de type **int**, qui est l'identifiant du client
- **server**, de type **Server**, une référence vers notre serveur
- **socket**, de type **Socket** : le socket utilisé par le serveur pour communiquer avec le client
- **out**, de type **ObjectOutputStream**, qui va permettre d'envoyer des messages au client
- **in**, de type **ObjectInputStream**, qui va permettre de réceptionner les messages du client

Le constructeur de la classe **ConnectedClient** prend en paramètre une référence vers le **server**, une référence vers le **socket**, et les utilise pour initialiser ses attributs correspondants. Il initialise également son attribut **id** avec la valeur courante de **idCounter**, en incrémentant ce dernier. Enfin, il crée le flux **out** à partir du **socket** :

```
out = new ObjectOutputStream(socket.getOutputStream());
```

Enfin, puisque ce constructeur sera appelé lorsqu'un nouveau client sera connecté, il peut également afficher dans la sortie standard un message correspondant.

```
System.out.println("Nouvelle connexion, id = " + id);
```

Nous l'avons vu à l'étape précédente, la classe **ConnectedClient** est un thread. Elle doit donc implémenter l'interface **Runnable**, qui la contraint à déclarer la méthode **run()**. Cette dernière a pour but de réceptionner les messages du client. Elle doit d'abord créer l'objet **in** à partir du **socket** :

in = new ObjectInputStream(socket.getInputStream());

Puis, tant que la connexion est active :

```
boolean isActive = true;
while (isActive) {
```

on attend un nouveau message grâce à l'attribut **in** :

```
Message mess = (Message) in.readObject();
```

Cette instruction est **bloquante.** Dès qu'on en reçoit un, on teste d'abord s'il n'est pas **null.** Puis, on initialise l'attibut **sender** avec l'id du client.

mess.setSender(String.valueOf(id));

Puis, on indique au serveur de diffuser ce message à tous les clients, en indiquant que ce message vient du client courant (en lui passant l'indentifiant):

```
server.broadcastMessage(mess, id);
```

Si au contraire le message est **null**, alors il faut terminer proprement la connexion. Une façon de le faire est de lancer une exception :

```
throw new IOException("Client déconnecté");
```

Dans le bloc catch (qui capture votre exception, ou bien toute exception levée par `in.readObject()`, vous arrêtez proprement la boucle et fermez la connexion :

```
server.disconnectedClient(this);
isActive = false;
```

Ici aussi, ne pas oublier de capturer les éventuelles exceptions, notamment les EOFException, qui sont levées lorsque le client se déconnecte brutalement.

Les méthodes **broadCastMessage** et **disconnectedClient** de la classe **Server** seront détaillées plus tard.

La classe **ConnectedClient** comporte également deux méthodes assez simples :

- la méthode **sendMessage(Message mess)** envoie le message au client en utilisant **out** :

```
this.out.writeObject(mess);
this.out.flush();
```

En Java, les flux sont gérés par des tampons (buffer). Si on veut s'assurer que le tampon est vidé dès ce message, on fait appel à la méthode **flush**. Le risque, en ne l'appelant pas, serait que la machine virtuelle attende que le tampon se remplisse avant d'envoyer le message.

- la méthode **closeClient()**, qui ferme les deux flux **in** et **out**, ainsi que le socket :

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

d) Fin de la classe **Server**

Il nous reste à implémenter les méthodes **addClient**, **broadcastMessage** et **disconnectedClient.**

- La méthode **addClient(ConnectedClient newClient)** ajoute le client connecté, passé en paramètre, à notre liste **clients :**

```
this.clients.add(newClient);
```

Puis elle envoie un message à tous les clients pour annoncer la nouvelle connexion, en utilisant la méthode **broadcastMessage** (qui n'est pas encore écrite) **:**

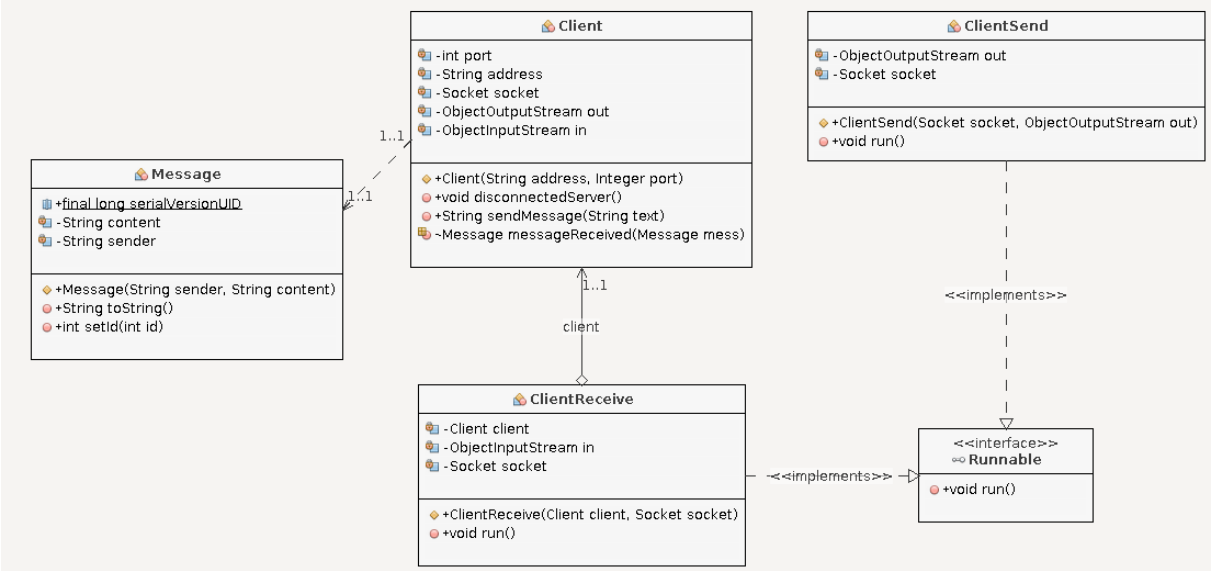
```
Message m = new Message(newClient.getId()+" vient de se connecter", newClient.getId())
broadcastMessage(m, newClient.getId());
```

- la méthode **broadcastMessage(Message mess, int id)** envoie le message **mess** à tous les clients sauf celui dont l'identifiant est **id**. Pour cela, on appellera la méthode **sendMessage** de la classe **ConnectedClient** :

```
for (ConnectedClient client : clients) {
	if (client.getId() != id) {
		client.sendMessage(mess);
	}
}
```

- la méthode **disconnectedClient(ConnectedClient discClient)** doit appeler la méthode **closeClient()** du client qui se déconnecte, le supprimer de la liste **clients**, et enfin prévenir les clients restants que **discClient** vient de se déconnecter, en leur envoyant un message via **broadCastMessage**

2) Le package **client**



a) La classe **Client**

La classe **Client** comporte 5 attributs :

- **address**, de type **String** : l'adresse IP du serveur
- **port**, de type **int** : le port du serveur
- **socket**, de type **Socket** : le socket pour communiquer avec le serveur
- **in**, de type **ObjectInputStream**, pour recevoir des messages du serveur
- **out**, de type **ObjectOutputStream**, pour envoyer des messages au serveur

Le constructeur prend en paramètre l'adresse IP du serveur ainsi que son port, il initialise les attributs correspondants, crée le **socket** à partir de ces informations, et crée l'attribut **out** à partir du **socket**. Puis, il crée deux threads : un pour émettre des messages, de type **ClientSend** l'autre pour en recevoir, de type **ClientReceive**. Maintenant, vous devez être capable de faire ça tout seul, alors on ne vous donne plus le code :-)

La méthode **disconnectedServer()** appelle la méthode **close()** sur les attributs **out**, **socket**, **in** (en vérifiant qu'il n'est pas **null**) et quitte l'application (`System.exit(0);`).

Enfin, la méthode **messageReceived** affiche simplement dans la console le message reçu.

b) La classe **ClientSend**

La classe **ClientSend** comporte un attribut **socket** de type **Socket**, ainsi qu'un attribut **out**, de type **ObjectOutputStream**. Son constructeur initialise ces attributs avec des objets passés en paramètre.

Cette classe est un thread, elle implémente donc l'interface **Runnable** qui la contraint à déclarer la méthode **run()**. Cette méthode est une boucle infinie qui invite l'utilisateur à saisir un message, et l'envoie :

```
Scanner sc = new Scanner(System.in);
while (true) {
	System.out.print("Votre message >> ");
	String m = sc.nextLine();
	Message mess = new Message("client", m);
	out.writeObject(mess);
	out.flush();
}
```

Si vous le souhaitez, vous pouvez rajouter dans cette méthode la possibilité de quitter le programme lorsque l'utilisateur saisit une instruction particulière (telle que « bye » ou « exit »).

c) La classe **ClientReceive**

La classe **ClientReceive** possède trois attributs :

- **client**, de type **Client** : une référence vers le client qui l'a créé
- **socket** de type **Socket**, le socket du client
- **in**, de type **ObjectInputStream**, qui permet de pouvoir recevoir des messages

Son constructeur initialise les deux premiers attributs grâces aux paramètres reçus. Nous créerons le flux **in** une fois le thread lancé.

En effet cette classe est un thread, elle implémente donc l'interface **Runnable** qui la contraint à déclarer la méthode **run()**. Cette méthode crée l'object **in** à partir du socket, puis dans une boucle, elle attend un nouveau message à l'aide de **in** tant que la connexion est active. Si le message reçu n'est pas **null**, on l'affiche sur la sortie standard (c'est le travail de la méthode messageReceived(Message) de la classe Client). Si le message reçu est **null**, la connexion devient inactive, et on appelle la méthode **disconnectedServer()** de l'attribut **client** :

```
boolean isActive = true ;
while(isActive) {
	mess = (Message) in.readObject();
	if (mess != null) {
		this.client.messageReceived(mess);
	} else {
		isActive = false;
	}
}
client.disconnectedServer();
```

Ici encore, il faudra rajouter les blocs try{ … } catch adéquats, notamment lorsque l'exception EOFException est levée, qui signifie que le serveur s'est déconnecté.

Annexe 1 : la classe MainServer

```java
package server;

import java.io.IOException;

/**
 * start a server. Reads the server's port from the command line argument
 * @author Remi Watrigant
 *
 */
public class MainServer {

	/**
	 * creates a new server
	 * @param args
	 */
	public static void main(String[] args) {


		try {
			if (args.length != 1) {
				printUsage();
			} else {
				Integer port = new Integer(args[0]);
				Server server = new Server(port);
			}
		} catch (IOException e1) {
			e1.printStackTrace();
		}
	}

	private static void printUsage() {
		System.out.println("java server.Server <port>");
		System.out.println("\t<port>: server's port");
	}
}
```

Utilisation de l'application en ligne de commande : se placer dans le répertoire contenant les deux packages **client** et **server** du répertoire **build**, puis lancer :

```
java server.MainServer <port>
```

Où <port> est le port du serveur : un entier entre 1025 et 49151.

Annexe B : la classe MainClient

```
package client;

import java.io.IOException;
import java.net.UnknownHostException;

/**
 * starts a client. Reads the address and port from the command line
argument
 * @author Remi Watrigant
 *
 */

public class MainClient {

	/**
	 * construct a new client
	 * @param args
	 */
	public static void main(String[] args) {

		try {
			if (args.length != 2) {
				printUsage();
			} else {
				String address = args[0];
				Integer port = new Integer(args[1]);
				Client c = new Client(address, port);
			}
		} catch (UnknownHostException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	private static void printUsage() {


		System.out.println("java client.Client <address> <port>");
		System.out.println("\t<address>: server's ip address");
		System.out.println("\t<port>: server's port");
	}
}
```

Utilisation de l'application en ligne de commande : se placer dans le répertoire contenant les deux packages **client** et **server**, puis lancer :

```
java client.MainClient <adresse> <port>
```

Où <adresse> et <port> sont respectivement l'adresse et le port du serveur.

Pour tester en local, mettre comme adresse 127.0.0.1.