Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

2017 - 2018
Ateliers Logiciels

Ateliers logiciels

Architecture client/serveur

Serveur d'une messagerie instantanée

Durant ce TD vous allez développer un serveur d'une application de messagerie instantanée (Chat) en respectant une architecture client-serveur donnée.

Ce TD est la suite directe du TD *Client d'une messagerie instantanée* Les ressources nécessaires au TD sont disponibles sur le repository ClientServerFramework [1] .

1. https://git.esi-bru.be/ATL/ClientServerFramework.git

Version du document : 18 mars 2018

1 Développement d'un serveur de messagerie instantanée

Ouvrez avec `Netbeans` le projet `ChatServer`. Ce projet est divisé en trois packages :

- `be.he2b.atl.server` : contient **les** classes permettant de développer un serveur multi-client ;
- `be.he2b.atl.chat.model` : contient la classe permettant de développer un serveur spécifique au chat ;
- `be.he2b.atl.chat.view.console` : contient la vue console de l'application.

1.1 La classe `AbstractServer`

Analysez la classe `AbstractServer` du package `be.he2b.atl.server`. Cette classe sert à écouter les demandes de connexion de la part des clients. Elle présente les fonctionnalités suivantes :

1. écouter les demandes de connexion ;
2. créer un `Thread` dédié à l'écoute d'un client particulier ;
3. envoyer des messages à tous les clients connectés ;
4. fermer tous les flux de communication avec les clients connectés ;
5. arrêter l'écoute des demandes de connexion.

Détaillons le contenu de cette classe.

1.1.1 Attributs

Cette classe possède 7 attributs :

1. `int port` : numéro du port du serveur ;
2. `boolean readyToStop` : signale si une demande d'arrêt du serveur a été demandée ;
3. `int timeout` : temps entre deux vérifications de demande d'arrêt du serveur ;
4. `int backlog` : nombre maximal de clients en attente de connexion ;
5. `ServerSocket serverSocket` : socket de la connexion au serveur ;
6. `Thread connexionListener` : `Thread` écoutant les demandes de connexion des clients ;
7. `List<Thread> threads` : liste des `Thread` dédiés à chaque client.

1.1.2 Constructeur

Le constructeur de `AbstractServer` initialise les attributs `List<Thread> threads` et `int port`.

1.1.3 Écouter les demandes de connexion

Afin de permettre l'écoute des demandes de connexion des clients, la méthode `listen` lance un `Thread` et le range dans l'attribut `connexionListener`. Cette procédure étant rendue possible grâce à l'implémentation de l'interface `Runnable` par la classe `AbstractServer`.

1.1.4 Arrêter le serveur

La méthode `close()` permet d'arrêter le serveur en n'écoutant plus les demandes de connexions et en fermant tous les flux ouverts.

1.1.5 Créer un `Thread` dédié à l'écoute d'un client particulier

La méthode `run` de la classe `AbstractServer` attend les demandes de connexion de la part de clients via l'instruction `Socket clientSocket = serverSocket.accept()`. Une fois la demande de connexion reçue, un `Thread` est créé pour communiquer avec le client via l'instruction `new ConnectionToClient(clientSocket, this);`.

Question 1

Pourquoi des blocs `synchronized` sont nécessaires au sein de la méthode `run` de la classe `AbstractServer` ?

Question 2

Les méthodes `hook` utilisées dans `AbstractServer` sont-elles les mêmes que celles utilisées dans `AbstractClient` ? Décrivez brièvement l'utilité de ces nouvelles méthodes.

Question 3

Pourquoi la méthode `handleMessageFromClient` est-elle abstraite ?

1.2 La classe `ConnexionToClient`

Ouvrez la classe `ConnexionToClient` du package `be.he2b.atl.server`. Cette classe sert à communiquer avec un client particulier. Elle présente les fonctionnalités suivantes :

1. ouvrir les flux de communication ;
2. envoyer des messages vers le client ;
3. recevoir des messages du client ;
4. enregistrer des informations concernant le client connecté ;
5. fermer les flux de communication ;

6. se déconnecter du client.

Décrivons les attributs et méthodes de cette classe.

1.2.1 Attributs

Cette classe possède 6 attributs :

1. `AbstractServer server` : référence au serveur ;
2. `Socket clientSocket` : socket de la connexion au serveur ;
3. `ObjectInputStream input` : flux d'entrée des données venant du client ;
4. `ObjectOutputStream output` : flux de sortie des données à envoyer vers le client ;
5. `boolean readyToStop` : signale si une demande d'arrêt du serveur a été demandée ;
6. `HashMap<String, Object> savedInfo` : `map` des différentes informations que l'on souhaite conserver sur le serveur concernant un client.

1.2.2 Constructeur

Le constructeur de `ConnexionToClient` initialise les attributs `Socket clientSocket` et `AbstractServer server`. Il ouvre également les flux de communications avec le client.

1.2.3 Fermer les flux de communication

La méthode `closeAll()` permet de fermer les flux utilisés pour communiquer avec le client.

1.2.4 Arrêter la communication avec le client

La méthode `close()` permet d'arrêter la communication avec le client et de fermer les flux ouverts.

1.2.5 Envoyer des messages vers le client

La méthode `sendToClient` envoi un `Object` vers le client. Pour ce faire, le flux de sortie des données **doit** être nettoyé avant chaque envoi.

Question 4

Quelle instruction effectue ce nettoyage ?

1.2.6 Écouter les messages du client

Question 5

Quelle interface implémente la classe `ConnexionToClient` et pourquoi ?

La classe `ConnexionToClient` devant s'exécuter dans un `Thread` d'écoute d'un client, la méthode `run` est implémentée. Cette méthode attend que des données arrivent via le flux d'entrée des données par l'instruction `msg = input.readObject();`. Lorsqu'un message est reçu, il est envoyé à la classe `AbstractServer` afin d'être interprété par la méthode :

```
handleMessageFromClient(Object msg, ConnectionToClient client)
```

Question 6

1. Décrivez l'utilité des différentes gestions d'exception présentes dans cette méthode.
2. Que ce passerait-il si ces exceptions n'étaient pas gérées ?
3. Le bloc `finally` est-il pertinent ?
4. Quel est l'intérêt d'appeler des `hook` au sein de la méthode `run` ?

1.3 La classe `ChatServer`

Le serveur de messagerie instantanée peut maintenant être implémenté au sein du package `be.he2b.atl.chat.model`. Pour ce faire, la classe `ChatServer` hérite de `AbstractServer`.

Ouvrez la classe `ChatServer` et consultez le code des méthodes qu'elle contient. Une descritpion des méthodes de cette classe est donnée ci-dessous pour vous guider.

1.3.1 Constructeur

Le constructeur du client `ChatServer` appelle le constructeur de son parent et écoute les demandes de connexion via l'instruction `this.listen()`. En cas d'erreur, l'exception n'est pas gérée dans le constructeur.

1.3.2 Arrêter le serveur

Afin de permettre de stopper le serveur, la méthode `quit` est implémentée. Cette méthode demande à son parent d'arrêter d'écouter les demandes de connexions et de fermer toutes les connexions.

1.3.3 Écouter les messages des clients

La méthode abstraite `handleMessageFromClient` est réécrite dans `ChatServer` afin d'interpréter les messages reçus de chaque client.

1.3.4 Connexion d'un client

`ChatServer` peut réécrire la méthode `clientConnected` afin d'effectuer un traitement particulier lorsqu'un client vient de se connecter. Ce sera l'occasion pour notre serveur de mettre à jour la liste des utilisateurs connectés et d'envoyer cette nouvelle liste à **tous** ses clients. **Attention**, l'identifiant d'un utilisateur, une fois calculé, est conservé au sein de la `HashMap savedInfo` du client `clientConnected` donné en paramètre.

Question 7

Comment les identifiants des utilisateurs sont-ils attribués ?

Question 8

Pourquoi conserver l'identifiant de l'utilisateur au sein du `Thread clientConnected` ?

Exercice 1 Écrivez la méthode `clientDisconnected`. Cette méthode effectue un traitement particulier lorsqu'un client vient de se déconnecter. Ce sera l'occasion pour notre serveur de mettre à jour la liste des utilisateurs connectés et d'envoyer cette nouvelle liste à **tous** ses clients.

1.3.5 Gestion des erreurs clients

`ChatServer` peut réécrire la méthode `clientException` afin d'effectuer un traitement particulier lorsqu'un client commet une erreur. Ce sera l'occasion pour notre serveur d'envoyer une alerte au client si il est toujours connecté. Rappelez-vous que vous avez vu le résultat d'une erreur client lors de l'exercice 3 de la section 1.4 du TD *Client d'une messagerie instantanée*.

Exercice 2 Écrivez la méthode `sendToClient(Message message, int clientId)`. Cette méthode permet de parcourir la liste des `Thread` clients afin de trouver celui qui concerne le client d'identifiant donné en paramètre. Une fois le `Thread` identifié il suffit de lui envoyer le message via l'instruction `client.sendToClient(message)`.

1.4 Vue console

L'interface proposée au sein du package `be.he2b.atl.chat.view.console` via la classe `ChatServerConsole` est basique.

La méthode `main` de cette classe démarre le serveur en créant une instance de la classe `ChatServer`. Les messages reçus par le serveur seront affichés automatiquement car la classe `ChatServerConsole` observe la classe `ChatServer`.

Exercice 3

1. démarrez votre serveur ;
2. modifiez la variable `String host` de votre `ChatClientConsole` en `localhost` ;
3. démarrez une première instance de votre client ;
4. démarrez une seconde instance de votre client ;
5. échangez des messages entre vos clients.