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

2017 - 2018
Ateliers Logiciels

Ateliers logiciels

Architecture client/serveur

Client d'une messagerie instantanée

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

Sur la machine « prof » un serveur de messagerie instantanée développé en Java est en cours d'exécution. Le but de l'exercice est de développer sur votre machine un client qui se connecte à ce serveur et envoie des messages aux autres étudiants connectés au serveur. Ce TD nécessite de travailler sur **les machines de l'école** afin d'être sur le même réseau.

1.1 Projet Client

Ouvrez avec `Netbeans` le projet `ChatClientV1`. Ce projet est divisé en trois packages :

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

1.2 La classe `AbstractClient`

Commencez par analyser le contenu de la classe `AbstractClient` du package `be.he2b.atl.client`. Cette classe sert à gérer la communication avec un serveur. Elle présente les fonctionnalités suivantes :

1. se connecter au serveur ;
2. ouvrir les flux de communication avec le serveur ;
3. envoyer des messages vers le serveur ;
4. recevoir des messages du serveur ;
5. fermer les flux de communication ;
6. se déconnecter du serveur.

Détaillons l'implémentation de cette classe.

1.2.1 Attributs

Cette classe possède 7 attributs :

1. `String host` : adresse du serveur `localhost`, `192.168.16.222`,... ;
2. `int port` : numéro du port du serveur ;
3. `boolean readyToStop` : signale si une demande d'arrêt du client a été demandée ;
4. `Socket clientSocket` : socket de la connexion au serveur ;
5. `ObjectOutputStream output` : flux de sortie des données a envoyer vers le serveur ;
6. `ObjectInputStream input` : flux d'entrée des données venant du serveur ;
7. `Thread clientReader` : `Thread` écoutant les messages venant du serveur.

1.2.2 Constructeur

Le constructeur de `AbstractClient` initialise les attributs `String host` et `int port`.

1.2.3 Fermer les flux de communication

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

Question 1

L'ordre de fermeture des flux a-t-il une importance ?

1.2.4 Méthodes utilitaires

Vous trouverez ensuite deux méthodes liées à la gestion de la connexion.

1. `closeconnexion` : demande la fermeture de la connexion ;
2. `isConnected` : vérifie si le client est connecté au serveur.

Question 2

Pourquoi ces méthodes sont `final` ? Pensez-vous que l'accesseur de l'attribut `host` doit être `final` également ?

1.2.5 Hook méthode

Différents `hook`[2] sont ajoutés à la classe `AbstractClient` :

1. `connexionClosed` ;
2. `connexionException` ;
3. `connexionEstablished`.

Question 3

1. Pourquoi ces méthodes ont une visibilité `protected` ?
2. Ces méthodes étant vides, ne serait-il pas plus intéressant de les transformer en méthode abstraites ?

2. https://fr.wikipedia.org/wiki/Hook_(informatique)

1.2.6 Recevoir des messages du serveur

La méthode `handleMessageFromServer(Object msg)` sert à recevoir un objet du serveur. Elle **doit** être implémentée par toutes les classes héritant de `AbstractClient`.

L'interprétation de l'objet `msg` reçu en paramètre dépend de l'application développée. Notre application de messagerie instantanée s'attend à recevoir du texte, mais une application d'échange de fichier, par exemple, s'attendrait à un autre contenu. La méthode `handleMessageFromServer(Object msg)` devra donc être adaptée en conséquence.

1.2.7 Se connecter au serveur

Analysez le contenu de la méthode `openConnexion`. Cette méthode crée un objet `Socket` et ouvre les deux flux nécessaires pour la communication avec le serveur :

- `Socket clientSocket` : socket de la connexion au serveur ;
- `ObjectOutputStream output` : flux de sortie des données à envoyer vers le serveur ;
- `ObjectInputStream input` : flux d'entrée des données venant du serveur.

Question 4

Que deviennent les flux en cas d'erreur ?

Question 5

Quelle est l'utilité de la première instruction `return` au sein de cette méthode ?

Question 6

Expliquez brièvement l'intérêt de créer un `Thread` contenant l'instance courante.

```
clientReader = new Thread(this);
```

Quelle interface doit implémenter la classe abstraite pour que cette instruction soit valide ?

1.2.8 Envoyer des messages vers le serveur

Regardez le contenu de la méthode `sendToServer`. Cette méthode envoie un `Object` vers le serveur. Pour ce faire, le flux de sortie des données **doit** être nettoyé avant chaque envoi.

Question 7

Quelle instruction effectue ce nettoyage ?

1.2.9 Écouter les messages du serveur

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

Question 8

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 `ChatClient`

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

1.3.1 Constructeur

Le constructeur du client `ChatClient` appelle le constructeur de son parent et ouvre une connexion avec le serveur via l'instruction `openConnexion()`. Remarquez qu'en cas d'erreur de connexion, l'exception est renvoyée.

1.3.2 Se déconnecter

Afin de permettre une déconnexion la méthode `quit` est implémentée. Cette méthode demande à son parent de fermer la connexion.

1.3.3 Écouter les messages du serveur

L'implémentation dans la classe `ChatClient` de la méthode abstraite `handleMessageFromServer` afin d'interpréter les messages reçus du serveur est obligatoire. Dans cette première version du client, les messages reçus sont directement transmis à l'observateur du client via l'appel de la méthode `notifyObservers(msg)`.

1.4 Vue console

La vue proposée au sein du package `be.he2b.atl.chat.view.console` via la classe `ChatClientConsole` est pour l'instant basique.

Cette vue se connecte au serveur en créant une instance de la classe `ChatClient`. Les réponses du serveur seront affichées automatiquement car la classe `ChatClient-Console` observe la classe `ChatClient`.

Exercice 1 Exécutez l'application en mode console. Quelle erreur obtenez-vous ?

Afin de se connecter au serveur, adaptez la valeur des variables suivantes :

— `String host = "localhost"` : encodez l'adresse IP de la machine « prof » ;

— `String name = "g12345"` : encodez votre numéro de matricule.

Comme montré sur la figure 1, à l'exécution du client, vous verrez la connexion de celui-ci dans la liste des clients connectés **du serveur**.

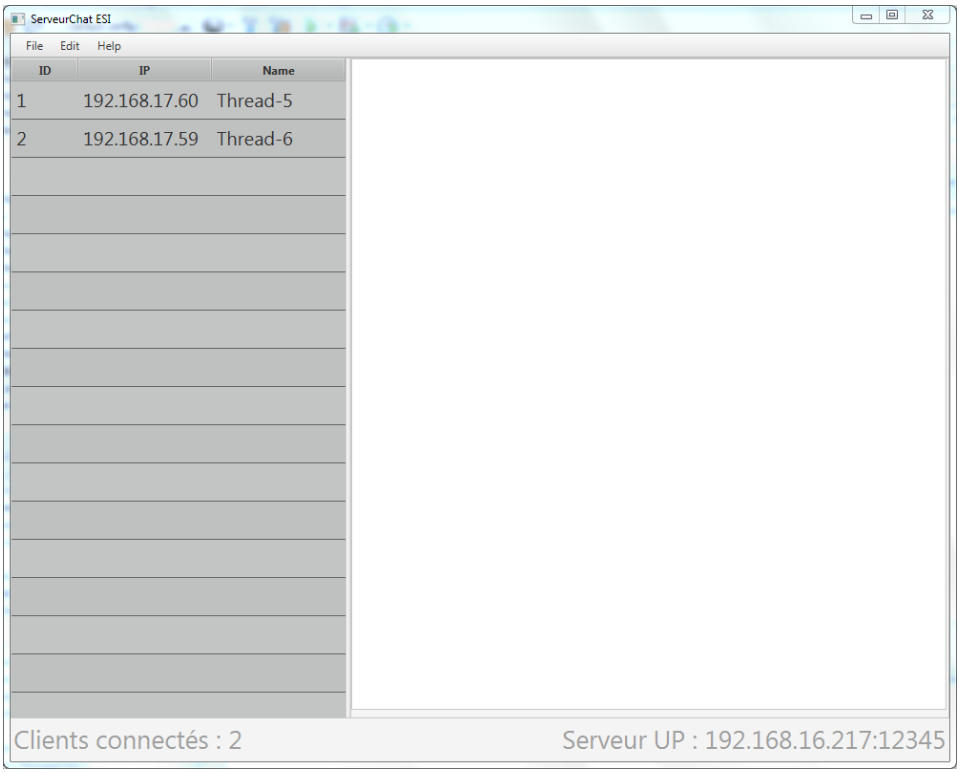

FIGURE 1 – Lorsqu'un client se connecte au serveur, la liste des utilisateurs connectés est mise à jour.

Exercice 2 Modifiez le code afin d'afficher le message « `Client connecté` » dans la console lorsque la connexion est établie. Quelle méthode de la classe `Abstract-Client` peut-on réécrire pour implémenter cette fonctionnalité ?

Actuellement, sur l'interface du serveur à côté de votre adresse IP, le nom associé à votre connexion n'est pas utilisable (`Thread-5`, `Thread-6`). Pour y remédier, il faut

envoyer un message au serveur contenant votre numéro de matricule afin d'actualiser ses données vous concernant.

Exercice 3 Ajoutez à votre `main` l'instruction `client.sendToServer("g12345");` avant la boucle `while`. Quel affichage obtient-on en console et pourquoi ?

2 Développement des messages envoyés

Communiquer avec le serveur nécessite d'échanger des objets ayant un format précis, connu du client et du serveur. Dans le cas contraire, le message n'est pas interprétable.

Les messages utilisés par notre serveur de messagerie instantanée sont composés :

- d'une classe représentant un utilisateur ;
- d'une énumération représentant le type du message échangé ;
- de classes représentant les messages.

Fermez le projet `ChatClientV1` et ouvrez les projets `ChatClientV2` et `MessageChatV2`.

2.1 Définition d'un utilisateur

Au sein du package `be.he2b.atl.chat.users` du projet `MessageChatV2` vous trouverez la classe `User` représentant un utilisateur. Un utilisateur est défini sur le serveur via :

- un identifiant unique ;
- une nom ;
- une adresse IP.

Question 9

Quelle est l'utilité d'implémenter l'interface `Serializable` pour la classe `User` ?

Remarquez que deux utilisateurs particuliers sont définis au sein de la classe `User` :

- administrateur du serveur : `public static final User ADMIN = new User(0, "ADMIN")` ;
- utilisateur servant au broadcast : `public static final User EVERYBODY = new User(0, "EVERYBODY")`.

2.2 Les types de messages

Le package `be.he2b.atl.message` contient l'énumération `Type` représentant les différents types des messages échangés. Pour l'instant un seul type est défini : `PROFILE`.

Avec un message de ce type, le client et le serveur pourront s'échanger des informations concernant le profil de l'utilisateur : son nom, son identifiant et son adresse IP.

2.3 L'interface message

L'interface `Message` définie dans le package `be.he2b.atl.message` devra être implémentée par tous les messages échangés. Cette interface met à disposition les accesseurs des informations cruciales d'un message :

- `Type getType()` : son type ;
- `User getAuthor()` : son émetteur ;
- `User getRecipient()` : son destinataire ;
- `Object getContent()` : son contenu, qui est de type `Object` afin d'être le plus général possible.

Question 10

Quelle est l'utilité d'implémenter l'interface `Serializable` pour l'interface `Message` ?

2.4 Un message de type Profile

Pour échanger des informations concernant le profil des utilisateurs connectés au serveur, la classe `MessageProfile` est créée. Cette classe implémente l'interface `Message`.

Question 11

Quel est le destinataire des messages de type `Profile` et pourquoi ?

3 La classe `ChatClient` envoie le profil de l'utilisateur

Afin que le client puisse envoyer des messages, commencez par ajouter le projet `MessageChatV2` comme librairie du projet `ChatClientV2`.

La classe `AbstractClient` ne comporte aucun changement, les fonctionnalités supplémentaires ayant été ajoutées dans la classe `ChatClient`.

Question 12

1. Quel attribut a été ajouté à la classe `ChatClient` ?
2. Comment le constructeur de la classe `ChatClient` transmet le nom de l'utilisateur au serveur ?
3. Quels changements ont été apportés à la méthode `handleMessageFromServer` de la classe `ChatClient` ?

Essayez à nouveau de vous connecter au serveur. Votre matricule apparaît dorénavant parmi la liste des utilisateurs connectés comme sur la figure 2.

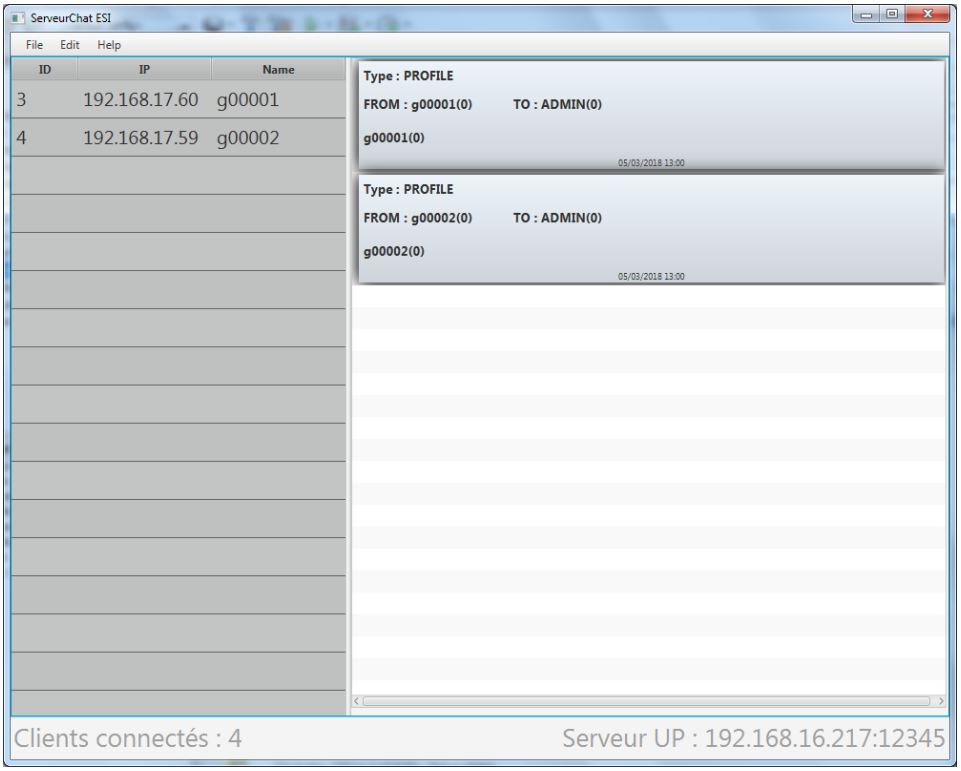

FIGURE 2 – La liste des utilisateurs connectés au serveur reprend le nom associé à l'utilisateur.

Exercice 4 Après une connexion, le client reçoit un message en réponse du serveur. Ce message contient le `userID` que le serveur a attribué. La classe `ChatClientConsole` essaye d'afficher le contenu du message via la méthode `update`, sans succès. Modifiez la méthode `update` afin d'afficher réellement le contenu du message de type `PROFILE` (`Type`, `Author`, `Recipient`, `Content`).

4 La classe `ChatClient` et la liste des utilisateurs connectés

Afin d'envoyer un message de chat à un utilisateur en particulier, il faut que la partie client de l'application dispose de la liste des utilisateurs connectés.

Pour ce faire nous allons à nouveau :

— ajouter un type de message ;

— ajouter une classe implémentant l'interface `Message` ;

— gérer le message au sein de la méthode `handleMessageFromServer` du client.

Ces trois opérations devront être effectuées **à chaque ajout d'un message** au sein du protocole de communication.

Fermez les projets `ChatClientV2` et `MessageChatV2` et ouvrez les projets `ChatClientV3` et `MessageChatV3`.

Afin que le client puisse envoyer des messages, commencez par ajouter le projet `MessageChatV3` comme librairie du projet `ChatClientV3`.

Question 13

1. Quel type de message a été ajouté ?
2. À votre avis, pourquoi les messages `MessageMembers` sont envoyés en `broadcast` (destinataire `Everybody`) ?
3. Quelle classe hérite de l'interface `Serializable` et pourquoi ?
4. Quel attribut a été ajouté à la classe `ChatClient` ?
5. Quels changements ont été apportés à la méthode `handleMessageFromServer` de la classe `ChatClient` ?

Exercice 5 Modifiez votre interface console comme sur la figure 3 pour afficher la liste des utilisateurs dans la console lorsque l'utilisateur tape le mot `list`. Pour chaque utilisateur, son id, son adresse IP et son nom doivent être montrés.

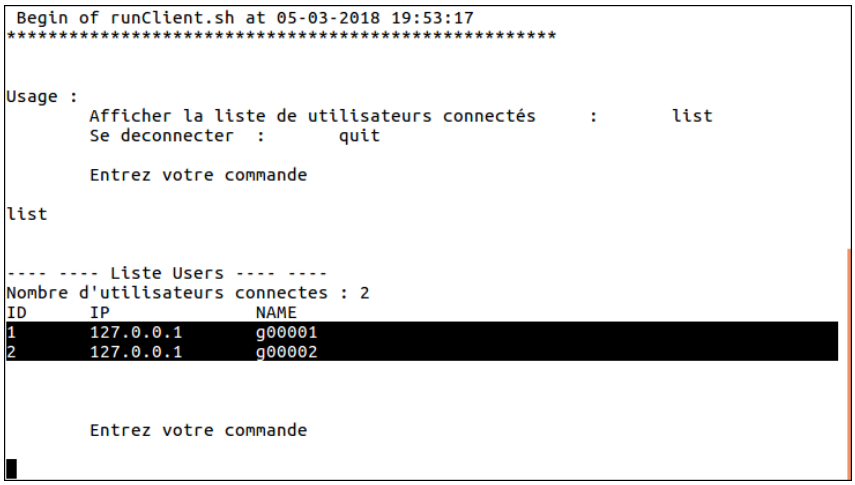

FIGURE 3 – L'utilisateur peut demander via la console la liste des utilisateurs connectés.

5 La classe `ChatClient` envoie un message à un autre utilisateur

Permettez maintenant au client d'envoyer des messages à un autre utilisateur connecté.

Pour ce faire nous allons devoir à nouveau :

— ajouter un type de message ;

— ajouter une classe implémentant l'interface `Message` ;

— gérer le message au sein de la méthode `handleMessageFromServer` du client.

Fermez les projets `ChatClientV3` et `MessageChatV3` et ouvrez les projets `ChatClientV4` et `MessageChatV4`.

Afin que le client puisse envoyer des messages, commencez par ajouter le projet `MessageChatV4` comme librairie du projet `ChatClientV4`.

Question 14

1. Quel type de message a été ajouté ?
2. Quelle classe hérite de l'interface `Serializable` et pourquoi ?
3. Quels changements ont été apportés à la méthode `handleMessageFromServer` de la classe `ChatClient` ?
4. Quel est le rôle de la nouvelle méthode `sendMessage` de la classe `ChatClient` ?

Exercice 6 Modifiez l'interface console comme sur la figure 4 pour qu'un message soit envoyé à l'utilisateur d'`id` donné en paramètre lorsque l'utilisateur entre la commande

```
send <id> <text>
```

Ce message transmis via le serveur provoque la mise-à-jour de l'affichage du serveur comme sur la figure 5.

Pensez également à mettre à jour la méthode `update` de l'observateur afin de gérer l'affichage des messages venant du serveur.

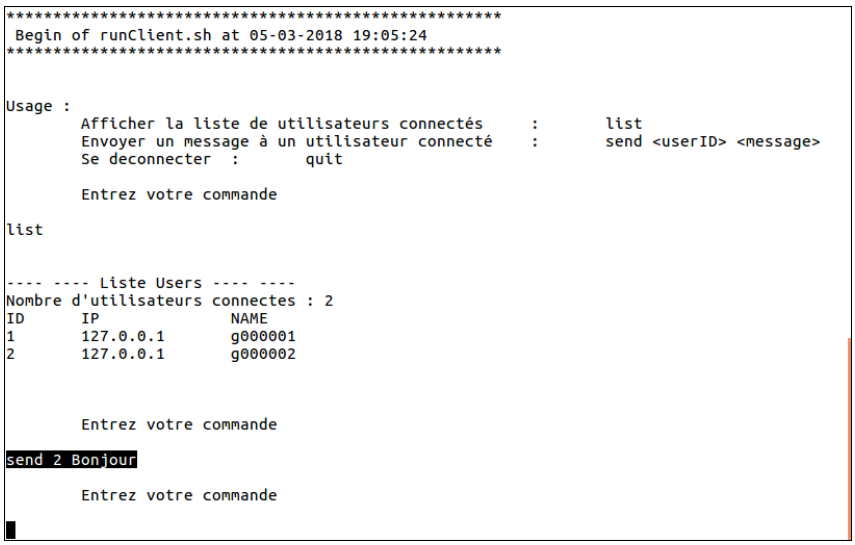

FIGURE 4 – Pour envoyer un message à un autre utilisateur, l'utilisateur doit demander via la console la liste des utilisateurs connectés. Une fois l'id de la personne à contacter noté, il peut envoyer un message via la commande `send <id> <text>`.

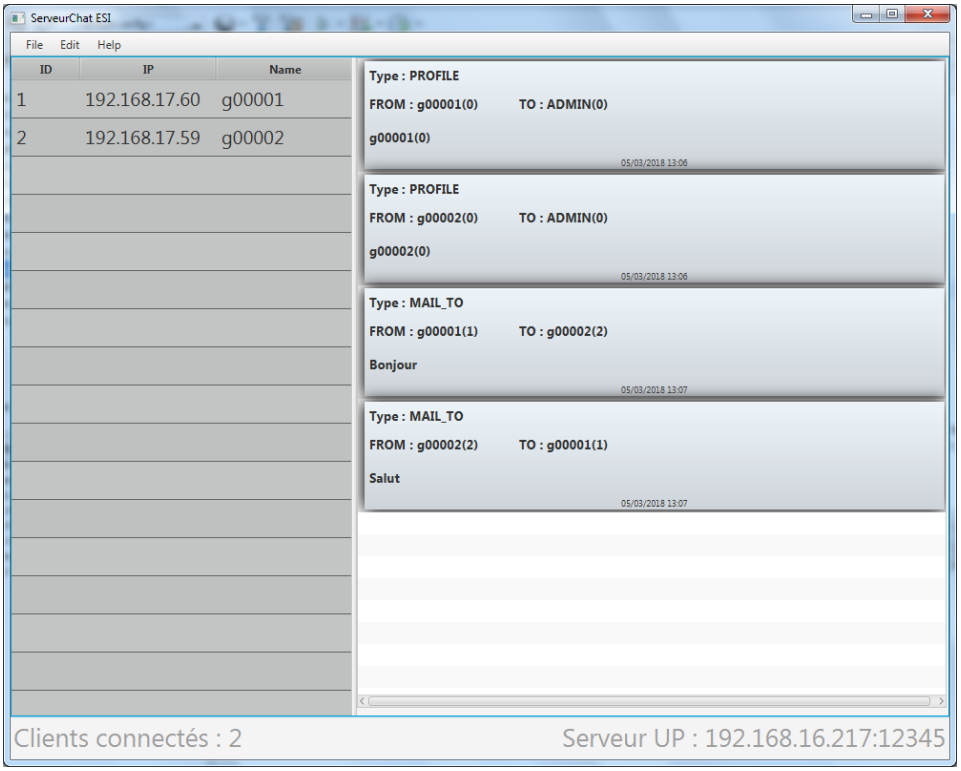

FIGURE 5 – Lorsqu'un client envoie un message à un autre utilisateur, le serveur reçoit ce message et le transfère vers l'utilisateur concerné. Le contenu du message est dans notre cas également affiché sur l'interface graphique du serveur.