

SR03
Rapport du Devoir n°1

Jian Chen Valentin Mention

12 avril 2020

Table des matières

1	Définition du projet	3
1.1	Contexte	3
1.2	Objectifs	3
1.3	Concepts	3
1.3.1	Les threads	3
1.3.2	Les sockets	4
2	Implémentation en Java	4
2.1	Classes du programme client	4
2.1.1	MessageInput	4
2.1.2	MessageReceptor	4
2.1.3	Client	4
2.2	Classes du programme serveur	4
2.2.1	Channel	4
2.2.2	Message	5
2.2.3	Serveur	5
3	Déploiement	5
3.1	Cloner le dépôt Git	5
3.2	Compiler l'application	5
3.2.1	Compilation du serveur	6
3.2.2	Compilation du client	6
3.3	Démarrer l'application	6
3.4	Exemple de fonctionnement	7
3.4.1	Instructions	7
3.4.2	Démonstration	8
4	Schémas de fonctionnement simplifié	9
4.1	Client	9
4.2	Serveur	10
A	Liens importants	11

1 Définition du projet

1.1 Contexte

Le premier devoir de SR03 consiste en la réalisation d'une application de messagerie entre plusieurs clients et un serveur, afin de créer un espace où plusieurs utilisateurs peuvent communiquer entre eux.

1.2 Objectifs

Nous avons fixés plusieurs exigences auxquelles doit répondre ce projet :

Côté client

- Le client doit disposer d'une interface console pour envoyer des messages au serveur
- L'actualisation des messages doit être automatique et le client doit pouvoir recevoir des messages pendant qu'il est en train lui-même de saisir un message
- Si le serveur n'est pas sur le port par défaut, le client doit pouvoir spécifier le port de connexion
- Le client doit pouvoir se déconnecter quand il le souhaite
- Chaque utilisateur doit être identifié par un nom d'utilisateur unique
- Si possible, les clients auront la possibilité d'échanger par messages privés

Côté serveur

- Le serveur doit récupérer les messages envoyés par un utilisateur et les redistribuer à tous les autres utilisateurs
- Le serveur doit accepter les connexions entrantes et gérer un certain nombre d'utilisateurs connectés
- Le serveur est responsable de la vérification des messages envoyés par l'utilisateur, et de du processus d'assignation de nom à chaque client connecté
- Le serveur doit pouvoir recevoir plusieurs messages d'utilisateurs différents simultanément
- Le serveur doit être résilient aux erreurs et ne pas planter en cas d'événement inattendu ou d'exceptions. Le nombre d'exceptions non-gérées doit être minime à nul.

1.3 Concepts

Afin de satisfaire ces objectifs, il est nécessaire de comprendre plusieurs concepts.

1.3.1 Les threads

Les threads sont une forme de "sous-processus" qui s'exécutent en parallèle du processus principal, et qui en partagent le même espace mémoire. Ils sont nécessaires dans notre projet du côté client, afin de pouvoir récupérer les entrées utilisateurs, sans bloquer la lecture des messages provenant du serveur. Ils sont également nécessaire au serveur, afin de pouvoir interagir simultanément avec

tous les clients connectés. Chaque connexion d'un utilisateur sera donc géré par un thread sur le serveur.

1.3.2 Les sockets

Les sockets sont une interface logicielle qui servent d'interface avec les fonctions réseau d'un système d'exploitation. Chaque socket est responsable d'une connexion avec un périphérique réseau, dans notre cas il s'agit de la connexion client-serveur. Ainsi, le client possède un unique socket qui gère les échanges avec le serveur, et le serveur dispose d'un socket par client.

2 Implémentation en Java

2.1 Classes du programme client

2.1.1 MessageInput

`MessageInput` gère les entrées utilisateurs. Il s'agit d'une tâche bloquante, donc cette classe implémente `Runnable` afin d'être exécutée dans son propre thread et de ne pas bloquer l'exécution de tout le programme. L'implémentation de `Runnable` est préférée de manière générale à `Threads`, car plus flexible [1]. Lorsque l'utilisateur saisit un message, la méthode bloquante *readstr* retourne le message saisi, qui est envoyé au serveur par la méthode *send*, sauf si le message est `"/exit"` qui provoque la déconnexion et l'arrêt du client.

2.1.2 MessageReceptor

`MessageReceptor` récupère les messages envoyés par le serveur et les affiche dans la console. Il n'y a aucun traitement des données côté client, les données sont affichées telles qu'elles sont reçues et c'est le serveur qui est chargé de la mise en forme.

2.1.3 Client

`Client` ne possède qu'une méthode `main`, qui constitue le point d'entrée du programme. Cette méthode vérifie les arguments qui lui sont passés, établit la connexion avec le serveur et lance les threads de `MessageInput` et `MessageReceptor`.

2.2 Classes du programme serveur

2.2.1 Channel

`Channel` est chargée de l'interaction avec un unique client, depuis le choix de son nom d'utilisateur jusqu'à sa déconnexion. Cette classe implémente `Runnable` et s'exécute dans son propre thread. Elle possède une méthode bloquante *init* appelée dans la méthode *run* au lancement du thread, qui récupère et vérifie le nom d'utilisateur du client. Quand l'utilisateur est correctement authentifié, le canal s'ajoute à la liste de tous les canaux actifs qui est partagée et synchronisée entre toutes les instances de `Channel`.

Une fois que l'utilisateur a été ajouté à la liste, la méthode *run* vérifie en continue si l'utilisateur envoie des messages. Dès qu'un message est reçu, il est analysé afin de déterminer dans un premier temps s'il est valide (c'est à dire non vide) et s'il s'agit d'une commande. Auquel cas la méthode *execCommand* est appelée pour prendre en compte cette commande. Sinon, on considère qu'il s'agit d'un message pour tous les autres utilisateurs et on appelle *broadcast* pour transmettre le message aux autres utilisateurs.

La diffusion du message se fait par itération sur la liste de canaux, en appelant la méthode *send* de chaque canal et en lui passant le message à transmettre.

Alternativement, il est possible pour les utilisateurs d'envoyer un message privé à un seul utilisateur. Ce cas est traité par *execCommand*, transmis ensuite à *privateSend* qui identifie dans la liste de canaux celui qui échange avec le destinataire du message et appelle sa méthode *send*.

2.2.2 Message

Message est une structure de donnée constituée de deux chaînes de caractères, une pour l'auteur du message et l'autre pour le contenu. Lorsque du texte doit être transmis aux clients, un objet *Message* est transmis à tous les *Channel*, qui l'enverront à leur client respectif, sauf s'il s'agit du client à l'origine du message.

2.2.3 Serveur

Serveur contient la méthode *main* et est le point d'entrée du programme serveur. Il accepte dans une boucle infinie toutes les connexions des utilisateurs et leur assigne un *Channel*. Il est possible de lui passer en paramètre le port sur lequel ouvrir la connexion.

3 Déploiement

3.1 Cloner le dépôt Git

Afin de déployer ce projet, il faut récupérer les fichiers sources disponibles sur Gitlab. Pour cela, le moyen préféré est de "cloner" le dépôt, bien qu'il soit également possible de télécharger le projet dans une archive.

La commande à utiliser pour cloner le projet est :

```
git clone git@gitlab.utc.fr:sr03_chen_mention/devoir1.git devoir1
```

Ce dépôt étant privé, il est nécessaire d'utiliser un compte autorisé à y accéder. De plus, git doit être correctement configuré pour utiliser le compte en question ou une clé SSH associée.

Il est recommandé de toujours utiliser la dernière version disponible de la branche master.

3.2 Compiler l'application

Afin de lancer les applications clients et serveur, il est nécessaire de les compiler. En ouvrant le projet avec un IDE, cela peut être réalisé automatiquement. Sinon, il est possible d'effectuer la compilation manuellement.

Pour compiler manuellement, se placer dans le répertoire racine du projet (c'est à dire le dossier "devoir1") et ouvrir un invite de commande. Les instructions ci-dessous sont peuvent être réalisés automatiquement en exécutant le fichier *build.sh* disponible à la racine du projet.

Attention! L'environnement de compilation doit disposer du JDK Java 8 au minimum et la version 13 est recommandée.

3.2.1 Compilation du serveur

```
> mkdir .binaries
> javac -d ../binaries ./src/server/*.java
> cd ../binaries
> jar cvmf ../src/server/META-INF/MANIFEST.MF Server.jar server
> cd ../
```

3.2.2 Compilation du client

```
> mkdir .binaries
> javac -d ../binaries ./src/client/*.java
> cd ../binaries
> jar cvmf ../src/client/META-INF/MANIFEST.MF Client.jar client
```

Si la compilation aboutit, il est à présent possible dans le dossier *.binaries* d'exécuter les fichiers *Client.jar* et *Server.jar*.

3.3 Démarrer l'application

Pour lancer l'exécution du client ou du serveur, définir le répertoire actif d'un invite de travail sur *.binaries* ou tout autre dossier contenant les fichiers exécutables. Bien qu'il soit possible d'exécuter l'application avec les fichiers *.class* également générés, nous ne traiterons pas ce cas ici.

Il est nécessaire de démarrer le serveur avant le client. Il faut un invite de commande par programme lancé.

Pour lancer le serveur

```
> java -jar Server.jar
```

Pour lancer le serveur en spécifiant le port

```
> java -jar Server.jar 1234
```

Pour lancer le client

```
> java -jar Client.jar
```

Pour lancer le client en spécifiant l'IP du serveur

```
> java -jar Client.jar 192.168.0.1
```

Pour lancer le client en spécifiant l'IP et le port du serveur

```
> java -jar Client.jar 192.168.0.1 1234
```

Attention, pour le client on ne peut pas spécifier le port sans spécifier l'IP

3.4 Exemple de fonctionnement

3.4.1 Instructions

Pour démontrer les fonctionnalités de notre application il est nécessaire d'avoir au moins 3 invites de commandes lancées, 4 étant préférables afin de pouvoir constater le fonctionnement des messages privés. Il est également préférable d'utiliser des consoles de type Bash, qui supportent les caractères accentués (non supporté par Windows, ce qui cause des problèmes d'affichage).

En suivant les instructions données en 3.3, lancer une instance serveur et trois instances clients sans passer de paramètres (les paramètres par défaut sont utilisés). Si demandé par le système d'exploitation, autoriser l'application à accéder au réseau.

Constater la bonne connexion des clients au serveur, puis entrer un nom d'utilisateur pour le premier client. Pour le second client, essayer de saisir le même nom que précédemment (sensible à la casse, mais pas aux espaces, qui sont supprimés), et constater le refus du serveur. Dans la console serveur, on pourra remarquer que seules les connexions et déconnexions des sockets sont indiquées, sans préciser si l'utilisateur est identifié ou pas. Il s'agit d'un choix de conception.

Saisir un nom valide pour les deux clients restants, et vérifier dans l'invite du premier client l'annonce de la connexion des deux autres. Envoyer un message depuis un client : le message est transmis aux deux autres avec le pseudonyme de celui l'ayant envoyé.

Tester à présent les commandes, en tapant /help pour en afficher la liste. On note que la commande n'est pas transmise aux autres utilisateurs. Tenter d'envoyer un message privé à un autre utilisateur, en utilisant :

```
> /Bob Bonjour Bob !
```

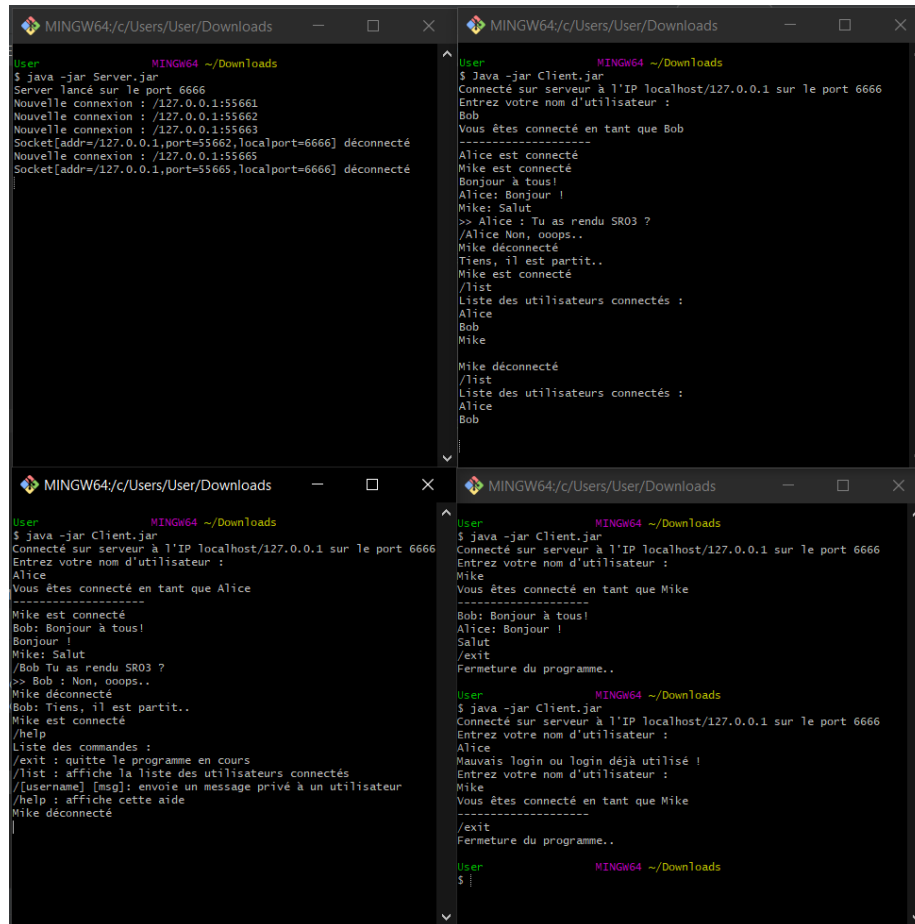
S'il existe un utilisateur nommé Bob (encore une fois, la casse est importante), on observe dans son invite de commande (et on remarque que le troisième client ne voit rien de cet échange) :

```
> >> Alice: Bonjour Bob !
```

Les messages privés sont reconnaissables aux doubles carets >> les précédents. S'il n'existe pas d'utilisateur nommé Bob, Alice verra marqué dans son invite :

```
> Commande ou utilisateur inconnu ! /help pour la liste des commandes
```

3.4.2 Démonstration



The figure displays four terminal windows arranged in a 2x2 grid, all titled 'MINGW64/c/Users/User/Downloads'. The top-left window shows the server running 'Server.jar' on port 6666, logging connections from 127.0.0.1 on ports 55661, 55662, and 55663. The top-right window shows the first client running 'Client.jar', logging in as Bob and interacting with the server. The bottom-left window shows the second client logging in as Alice and interacting with the server. The bottom-right window shows the third client logging in as Mike, interacting with the server, and then running the client again to show a 'Mauvais login' error.

```

User MINGW64 ~/Downloads
$ java -jar Server.jar
Server lancé sur le port 6666
Nouvelle connexion : /127.0.0.1:55661
Nouvelle connexion : /127.0.0.1:55662
Nouvelle connexion : /127.0.0.1:55663
Socket[addr=/127.0.0.1,port=55662,localport=6666] déconnecté
Nouvelle connexion : /127.0.0.1:55665
Socket[addr=/127.0.0.1,port=55665,localport=6666] déconnecté

User MINGW64 ~/Downloads
$ java -jar Client.jar
Connecté sur serveur à l'IP localhost/127.0.0.1 sur le port 6666
Entrez votre nom d'utilisateur :
Bob
Vous êtes connecté en tant que Bob
-----
Alice est connecté
Mike est connecté
Bonjour à tous!
Alice: Bonjour !
Mike: Salut
>> Alice : Tu as rendu SR03 ?
/Alice Non, ooops..
Mike déconnecté
Tiens, il est parti..
Mike est connecté
/list
Liste des utilisateurs connectés :
Alice
Bob
Mike
Mike déconnecté
/list
Liste des utilisateurs connectés :
Alice
Bob

User MINGW64 ~/Downloads
$ java -jar Client.jar
Connecté sur serveur à l'IP localhost/127.0.0.1 sur le port 6666
Entrez votre nom d'utilisateur :
Alice
Vous êtes connecté en tant que Alice
-----
Mike est connecté
Bob: Bonjour à tous!
Bonjour !
Mike: Salut
/Bob Tu as rendu SR03 ?
>> Bob : Non, ooops..
Mike déconnecté
Bob: Tiens, il est parti..
Mike est connecté
/help
Liste des commandes :
/exit : quitte le programme en cours
/list : affiche la liste des utilisateurs connectés
/[username] [msg]: envoie un message privé à un utilisateur
/help : affiche cette aide
Mike déconnecté

User MINGW64 ~/Downloads
$ java -jar Client.jar
Connecté sur serveur à l'IP localhost/127.0.0.1 sur le port 6666
Entrez votre nom d'utilisateur :
Mike
Vous êtes connecté en tant que Mike
-----
Bob: Bonjour à tous!
Alice: Bonjour !
Salut
/exit
Fermeture du programme..

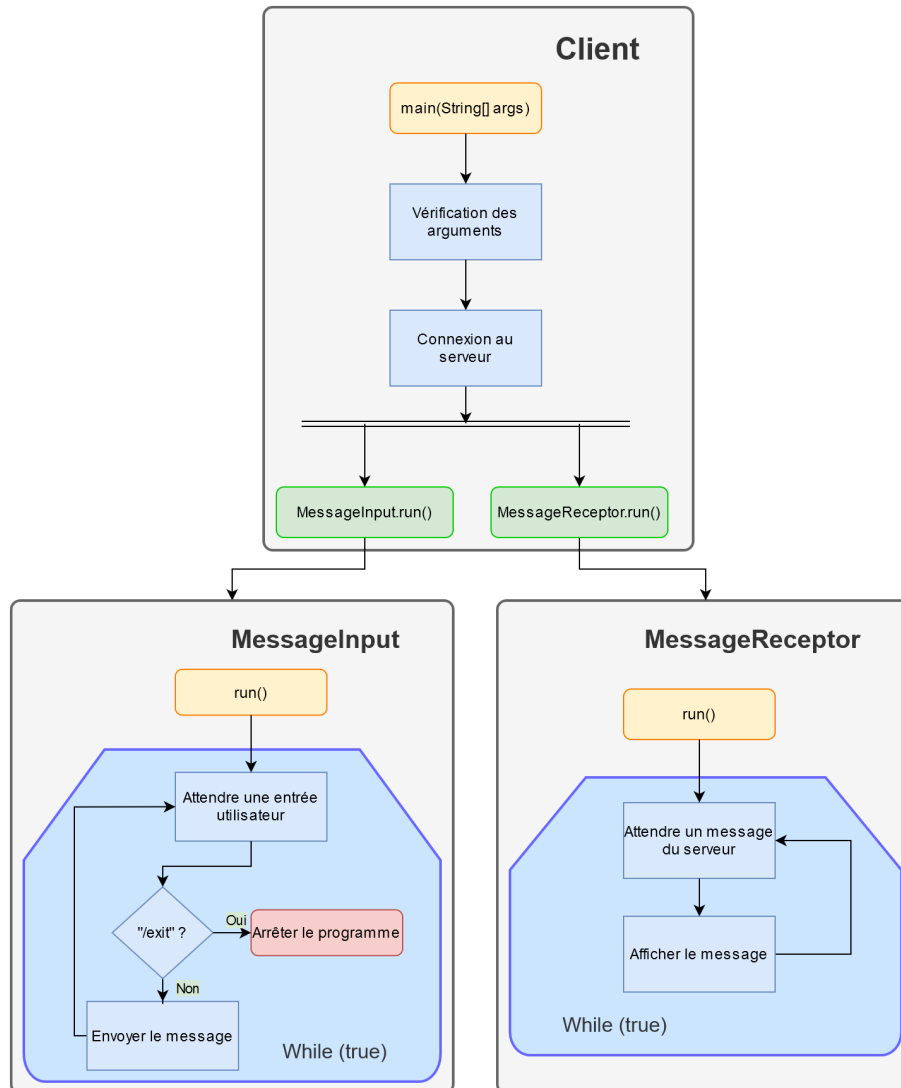
User MINGW64 ~/Downloads
$ java -jar Client.jar
Connecté sur serveur à l'IP localhost/127.0.0.1 sur le port 6666
Entrez votre nom d'utilisateur :
Alice
Mauvais login ou login déjà utilisé !
Entrez votre nom d'utilisateur :
Mike
Vous êtes connecté en tant que Mike
-----
/exit
Fermeture du programme..

User MINGW64 ~/Downloads
$
```

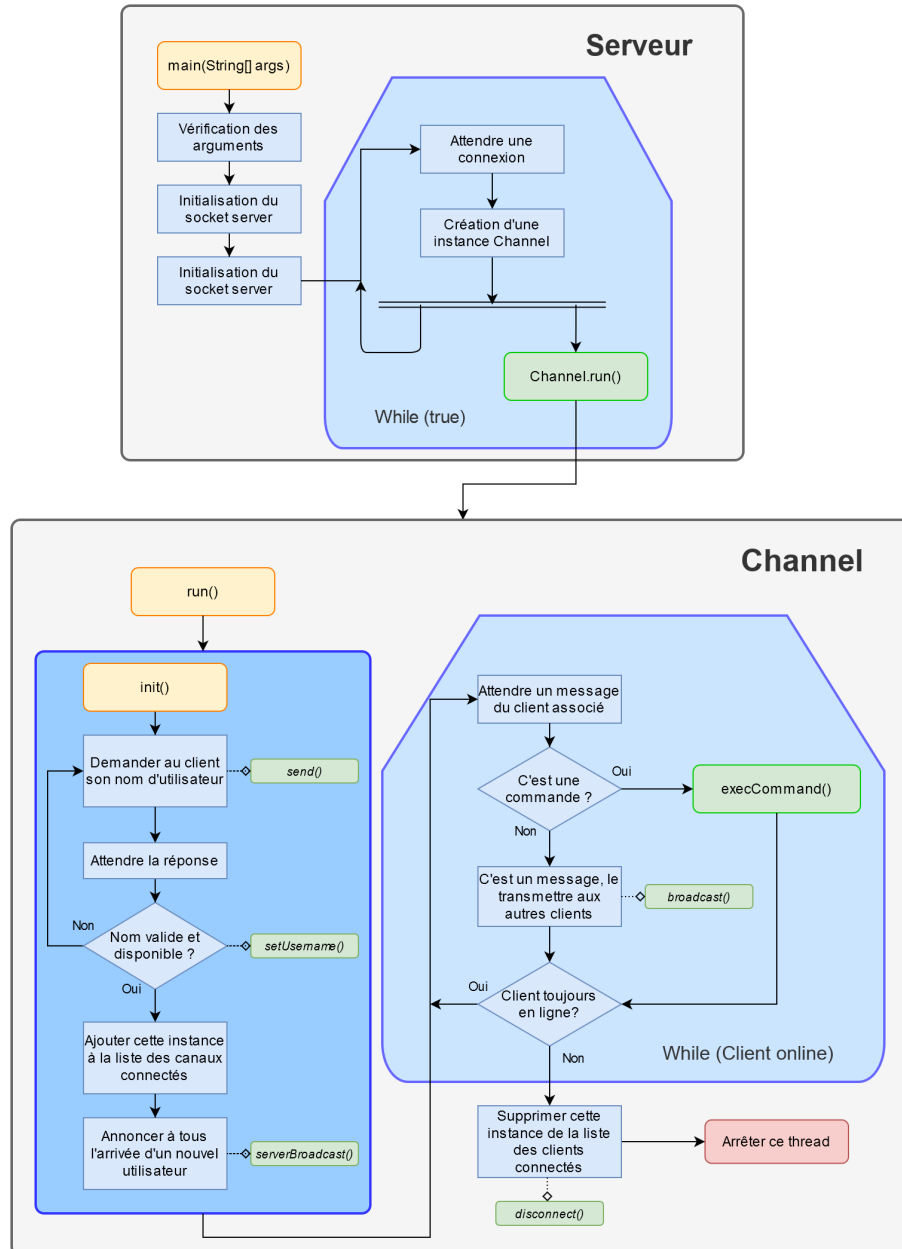
FIGURE 1 – Exemple de fonctionnement avec un serveur et trois clients

4 Schémas de fonctionnement simplifié

4.1 Client



4.2 Serveur



A Liens importants

Javadoc du projet : https://sr03_chen_mention.gitlab.utc.fr/devoir1/index.html

Page des releases : https://gitlab.utc.fr/sr03_chen_mention/devoir1/-/releases

Racine du dépôt git : https://gitlab.utc.fr/sr03_chen_mention/devoir1/-/tree/master

Références

- [1] <https://stackoverflow.com/questions/541487/implements-runnable-vs-extends-thread-in-java>