**Messenger**

**Messenger v1.0 Report**

**Pr. Yann MATHET**

# **Résumé**

Développé par **GARNICA OLLARA Jesus** et **GRAFFE Hivinau**

Ce projet présente un système de chat fonctionnant sur le principe de socket.

Il est composé de deux applications Java ; d’une part, l’application Serveur qui consiste à gérer l’intégralité de la communication entre sockets ; et d’autre part, l’application Client qui permet aux utilisateurs de discuter entre eux à travers une interface graphique.

La particularité de ce chat est la fonctionnalité « Géographique », c’est le concept de « salon » sur lequel plusieurs utilisateurs se trouvant géographiquement proches, peuvent discuter entre eux sans que d’autres utilisateurs « Eloignés » puissent intervenir.

Ce travail a été effectué dans le cadre de la formation de Master DRN2i de 2ème année, basée à l’Université de Caen-Normandie pouvant être servi d’évaluation de nos capacités à développer une application Java selon le modèle Modèle-Vue-Contrôleur.

Synopsis

[Résumé 1](#_Toc476689789)

[Introduction 4](#_Toc476689790)

[Etude d’analyse 4](#_Toc476689791)

[**Présentation du projet** 4](#_Toc476689792)

[**Spécifications** 5](#_Toc476689793)

[Principaux aspects et fonctionnalités 12](#_Toc476689794)

[**Swing et les interfaces graphiques** 12](#_Toc476689795)

[Conclusion et améliorations 14](#_Toc476689796)

[Remerciements 15](#_Toc476689797)

[Annexes 15](#_Toc476689798)

# **Introduction**

Ce rapport présente le travail effectué dans le cadre de la formation Master DRN2i de 2ème année, basée à l’Université de Caen-Normandie.

Ce projet a été encadré par le professeur Yann MATHET, enseignant JAVA durant la période de début janvier 2017 au début du mois de mars.

Ce document se veut être la synthèse des différentes tâches accomplies. Il décrit les principaux axes de développement du projet, en précise l’état d’avancement et explique les méthodologies employées. Chaque étape de réalisation est exposée sous forme de prototype en passant par la conception, l’implémentation et la validation expérimentale.

## **Etude d’analyse**

### **Présentation du projet**

Un serveur propose à des clients de se connecter pour dialoguer. L’application client comporte une interface graphique permettant, de façon non limitative :

* D’afficher la liste des clients actuellement connectés,
* D’envoyer des messages privés à la personne sélectionnée dans la liste et de recevoir de tels messages,
* De dialoguer de façon publique, mais de façon circonscrite aux personnes les plus proches. L’interface montrera en effet une zone graphique dans laquelle chaque client apparaît via un avatar. Pour une meilleure visibilité, un client se voit graphiquement en rouge et voit les autres en bleu. Chaque client peut se déplacer à la souris par drag&drop mais n’a pas le droit de déplacer les autres clients. Tous les déplacements sont actualisés en direct sur chacun des clients graphiques. Chacun peut entendre ceux qui parlent de façon publique et qui sont situées à une distance inférieure à une valeur prédéfinie, visualisée via un cercle en pointillés.

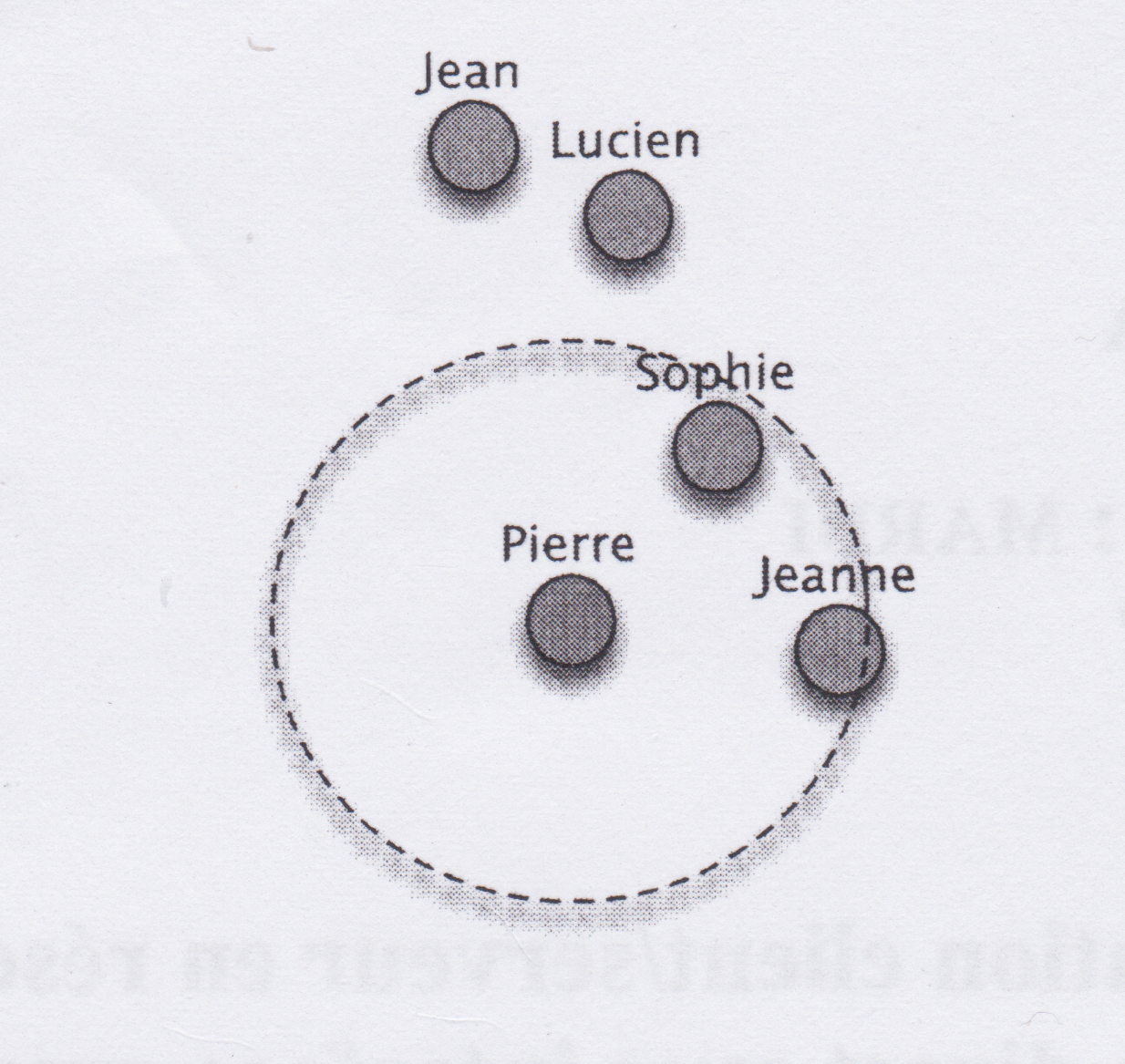


Figure . Exemple : Pierre entend, et est entendu par, Sophie et Jeanne seulement.

### **Spécifications**

#### **Les Sockets**

La communication serveur/client a été l’origine de notre analyse. Comment mettre en place un système communicant ? La réponse est : les Socket.

L’exemple comprend 2 programmes JAVA distincts.

##### **Application serveur :**

Le programme Serveur est composé de 2 classes (Server et ClientManager). La classe Server est le noyau du serveur grâce auquel le serveur se met en écoute, établit des connexions et lit ou écrit des messages aux clients. La classe ClientManager permet de gérer les différents protocoles que le serveur et les différents clients ont convenu à utiliser pour communiquer.

Le programme crée un ServerSocket dans la logique suivante :

On crée un objet ServerSocket

Tant que (infini) {

La serverSocket accepte une nouvelle connexion

On établit, dans un thread séparé, la communication avec le nouveau client

}

La classe ServerSocket est une classe du package java.net qui fournit les différents objets pouvant établir une connexion serveur/clients.

Le constructeur de ServerSocket lance une exception si elle ne peut pas écouter sur le port spécifié dans le cas où ce port est déjà utilisé par exemple :

try {

ServerSocket serverSocket = new ServerSocket(5000);

}  catch (IOException exception) {

//impossible de se lier au port 5000.

}

Lorsque le serveur se lie avec succès à son port, l’instance créée se met sur écoute de connexions entrantes et procède à l’acceptation d’un client à l’aide de la méthode accept().

La méthode accept() retourne une Socket avec laquelle le serveur peut communiquer :

while(true) {

Socket socket = serverSocket.accept();

}

On passe alors à la classe ClientManager qui doit être lancée dans un thread de la manière suivante :

new Thread(new ClientManager(socket)).start();

La nouvelle Socket représente le client nouvellement accepté par le serveur. La classe ClientManager permet maintenant de mettre en place les différents protocoles pour communiquer avec le serveur.

Cela consiste à envoyer ou recevoir des chaînes de caractères.

Ainsi, on récupère les flux d’entrée et de sortie permettant distinctivement de lire les messages venant du client et d’écrire des messages à ce client :

try {

PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);

BufferedReader reader = new BufferedReader(new InputStreamReader( socket.getInputStream()));

Pour envoyer un message à un client, le serveur procède comme suite :

String message = "message à envoyer";

//on envoie le message au client avec la méthode println()

writer.println(serverMessage);

Enfin, pour recevoir un message du client :

String clientMessage;

//le serveur reçoit le message du client avec la méthode readLine()

//dans une boucle

while ((clientMessage = reader.readLine()) != null) { }

##### **Application client :**

Le programme Client doit procéder de la même manière.

Avant tout, le serveur doit être lancé et en écoute sur un port spécifique.

La 1ère chose que la classe Client fait est d’ouvrir une Socket connecté au serveur :

try {

Socket socket = new Socket(host, port);

}

Une fois la socket connectée, on établit la communication dans une boucle.

C’est-à-dire que le client peut envoyer ou recevoir des chaînes de caractères.

Pour cela, on récupère les flux d’entrée et de sortie permettant respectivement de lire les messages du serveur et d’écrire des messages au serveur :

PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);

BufferedReader reader = new BufferedReader(new InputStreamReader( socket.getInputStream()));

Pour envoyer ou recevoir un message au serveur, le client procède comme suite :

String clientMessage = "message à envoyer";

String serverMessage;

//le client reçoit le message du serveur avec la méthode readLine()

while ((serverMessage = reader.readLine()) != null) {

// on envoie le message au serveur avec la méthode println()

writer.println(clientMessage);

}

#### **Les protocoles**

Les protocoles permettent que le serveur et les clients potentiels puissent communiquer entre eux avec des règles de conduite imposées. En autre, un client ne peut pas faire ce qu’il veut pour communiquer avec le serveur ou d’autres clients, il est obligé de suivre les « questions-réponses » du serveur qui font office de barrière, afin d’obtenir un « laissez-passer ». Notre système comprend trois protocoles que sont « authentification », « envoi-réception d’un message public » et « envoi-réception d’un message privé ».

##### **Le protocole « authentification » :**

Ce protocole permet de se connecter au réseau. La contrainte imposée est qu’un client doit envoyer un profil utilisateur dans le but d’être identifié à partir de ce profil. Ainsi, si un client A envoie un message au client B, le serveur notifie au client B que le client A, identifié par un certain profil utilisateur, a envoyé un message.



Figure . Diagramme de séquences : authentification d'un client.

Explications de chaque étape :

* Le serveur se met sur écoute sur un port défini, c’est-à-dire qu’il attend les connexions entrantes de clients.
* Un client se connecte au serveur sur le port d’écoute spécifique.
* Le serveur a accepté le client, le protocole d’identification démarre.
* Le serveur demande au client « qui êtes-vous ? ».
* Le client envoie au serveur son profil utilisateur. Tant que le serveur n’a pas reçu le profil utilisateur du client, le client reçoit la même demande du serveur à chaque fois qu’il essaye d’établir le dialogue avec le serveur.

##### **Le protocole « envoi-réception de message public » :**

Un client A vient se connecter au réseau. Le serveur notifie les clients que le client A vient d’être connecté. Le client A envoie « bonjour » à tous les clients connectés. Le serveur envoie « bonjour » à tous les clients de la part du client A. Tout ceci est possible si le protocole « envoi-réception de message public » est respecté. Ce protocole permet en effet de notifier tous les clients connectés d’un évènement prédéfini (tels que la connexion/déconnexion d’un client, l’envoi d’un message, …).



Figure . Diagramme de séquences : Envoi-réception d’un message public.

Explications de chaque étape :

* Le client envoie un message au serveur.
* Le serveur identifie le client à partir de son profil utilisateur.
* Le serveur notifie tous les clients déjà enregistrés qu’un client a envoyé un message.

##### **Le protocole « envoi-réception de message privé » :**

Même procédure que le protocole « envoi-réception de message public », mis à part que le client qui envoie le message précise au serveur le profil utilisateur auquel le message est destiné, et le serveur va rediriger le message au client identifié par le profil utilisateur spécifié par l’envoyeur.

Implémentation UML :



Figure . Diagramme de classes : classes pour établir le serveur.

On aperçoit sur ce diagramme une entité « Server ». Elle permet de créer et gérer les connexions entre clients.

Elle s’utilise très simplement, de la manière suivante :

try {

Server server = new Server();

//permet de :

// - lancer le serveur,

// - récupérer l'adresse IP et le port d'écoute du serveur.

InetAddress address = server.start();

}

Lorsqu’un client tente de se connecter au serveur, ce dernier lui impose le protocole « authentification ». Une fois le client identifié, le serveur le sauvegarde dans la liste de clients et envoie un message à tous les clients connectés pour notifier la connexion d’un nouveau client, en utilisant le protocole « envoi-réception de message public ».

A partir de là, le nouveau client peut lancer n’importe quelle discussion (publique ou privée).

On peut noter que l’utilisation de la méthode « yield » force l’interruption du serveur.



Figure . Diagramme de classes : classes pour établir le client.

Nous remarquons particulèrement que l’entité « Client » implémente l’interface « Runnable ». Il faut noter que chaque instance de cette classe doit être créée dans un processus parallèle :

try {

Client client = new Client(user, host, port);

client.addClientListener(new ClientListener() {

@Override

public void messageReceived(User user, Message message) {

//TODO: lire message d'un client identifié par le profil utilisateur //spécifié

}

@Override

public void handleState(boolean connected) {

//TODO: si false: le client est hors-ligne

}

});

}

## **Principaux aspects et fonctionnalités**

### **Swing et les interfaces graphiques**

#### **Application serveur**

Par défaut, le constructeur de la classe Server instancie l’object serverSocket avec le port 5000. Cependant, il est possible d’utiliser le constructeur comme suit :

try {

int port = 5000;

//voir annexes pour connaître les paramètres modifiables

ServerConfiguration configuration = ServerConfiguration.getInstance()

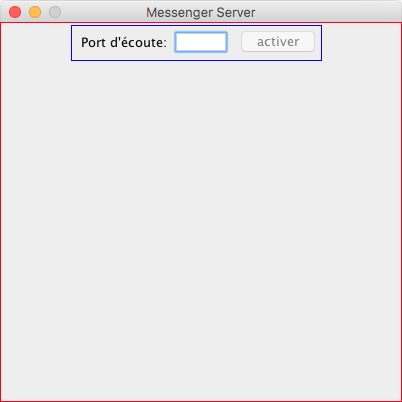
.setPort(port);

//par défaut, le port est initialisé à 5000

/\*Server server = new Server();\*/

Server server = new Server(configuration);

Afin de laisser la possibilité à l’utilisateur de modifier le port d’écoute par défaut, on a opté pour une interface graphique simple comme suit :



RootController est une JFrame qui contient :

* SettingsView pour saisir le port d’écoute,
* BorderLayout pour enlever les espaces inutiles

SettingsView est un JPanel qui contient :

* JLabel pour « Port d’écoute »,
* JTextField pour saisir le port,
* JToggleButton pour activer/désactiver le serveur

Figure . Interface graphique de l'application Serveur.

Ainsi, lorsque l’utilisateur clique sur le bouton « activer », il suffit de récupérer le port à partir de la méthode getPort() de la SettingsView comme suit :

int port = settingsView.getPort();

La déconnexion du serveur est faisable en recliquant sur le bouton « activer ». De ce fait, on fait appel à la méthode yield() de la classe Server :

server.yield();

//on s’assure que cette instance passe au Garbagge Collector

//afin qu’aucune connexion ne puisse être établie à nouveau sur cette instance.

server = null;



Figure . Diagramme de classes : classes pour l'application Serveur.

A travers ce diagramme, on note que l’entité RootController permet de lancer ou arrêter le serveur (instance de Server) lorsque la méthode stateChanged est appelée, en fonction du paramètre « state » :

@Override

public void stateChanged(final SettingsView settingsView, final boolean state) {

try {

if(state) {

ServerConfiguration configuration = ServerConfiguration.getInstance()

.setPort(port);

Server server = new Server(configuration);

InetAddress address = server.start();

} else {

if(server != null) {

server.yield();

server = null;

}

}

#### **Application client**

# **Conclusion et améliorations**

Results are mixed:

On the one hand, I produced all reasonable requests. As continuous process to update location of mobile phones or simultaneous display track on map we can enhance authentication.

On the other hand, because of lack of time, there are no alerts to prevent thresholds are missed. Moreover, we can enrich GUI.

# **Remerciements**

I would like to thank M. Marc SPANIOL offering me opportunity to highlight my mobile-developer skills and by what I learnt courses.

# **Annexes**

Please, look at [online project](https://github.com/hivinau/MobileNanny) to study source code or to share your comments.