INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Rapport de projet
Programmation Orientée Objets

Système de clavardage distribué interactif multi-utilisateur temps réel

Calmettes Pierre
Cotte Thomas
Fragonas Fabrice

Plan

Introduction

Dans le cadre du projet de programmation orientée objets, nous avons dû créer une application permettant à des utilisateurs de clavarder entre eux. Le cahier des charges spécifiait les contraintes que devait respecter l'application.
En outre, l'application devait distinguer deux types d'utilisateurs, les utilisateurs internes qui font partie du réseau local au sein de l'environnement de déploiement, et les utilisateurs externes qui n'en font pas partie.
Elle devait permettre à n'importe quel utilisateur de choisir un pseudonyme unique et de commencer une discussion avec un autre utilisateur tout en gardant un historique horodaté des conversations grâce à une base de données.
Finalement, elle devait pouvoir assurer l'envoi de fichiers et d'images entre utilisateurs. Le déploiement de l'application sur un poste devait être réalisable par un administrateur qui pourrait alors la configurer.

Ce rapport présentera les diagrammes UML ainsi que les choix techniques pris dans la conception de cette application.

Lien du dépôt git : https://github.com/PierreCalmettesInsa/ProjetPoo4A.git

Les diagrammes UML

Diagramme des cas d'utilisation

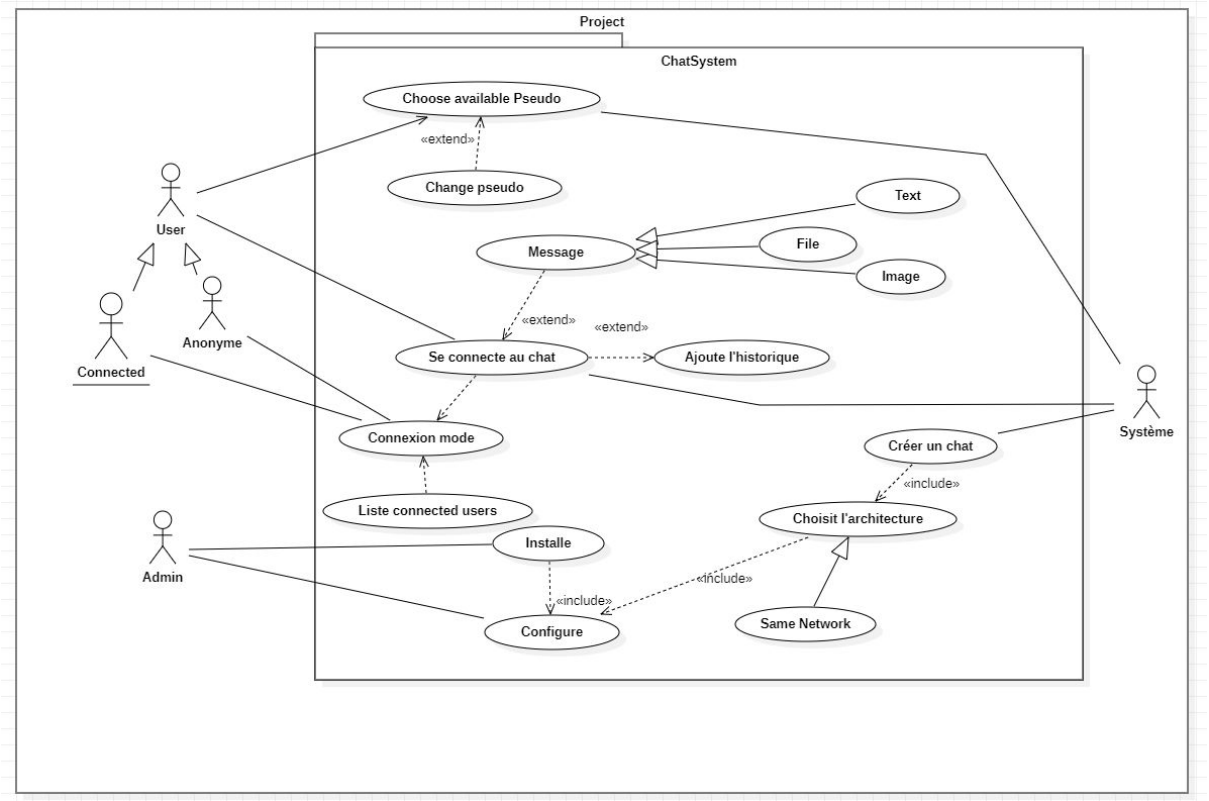

figure 1 : Diagramme des cas d'utilisation de notre application

Le diagramme de cas d'utilisation du ChatSystem permet de voir les fonctionnalités essentielles de l'application et leurs interactions lorsque l'utilisateur va interagir avec le système de clavardage.

Diagrammes de séquence

Démarrage de l'application

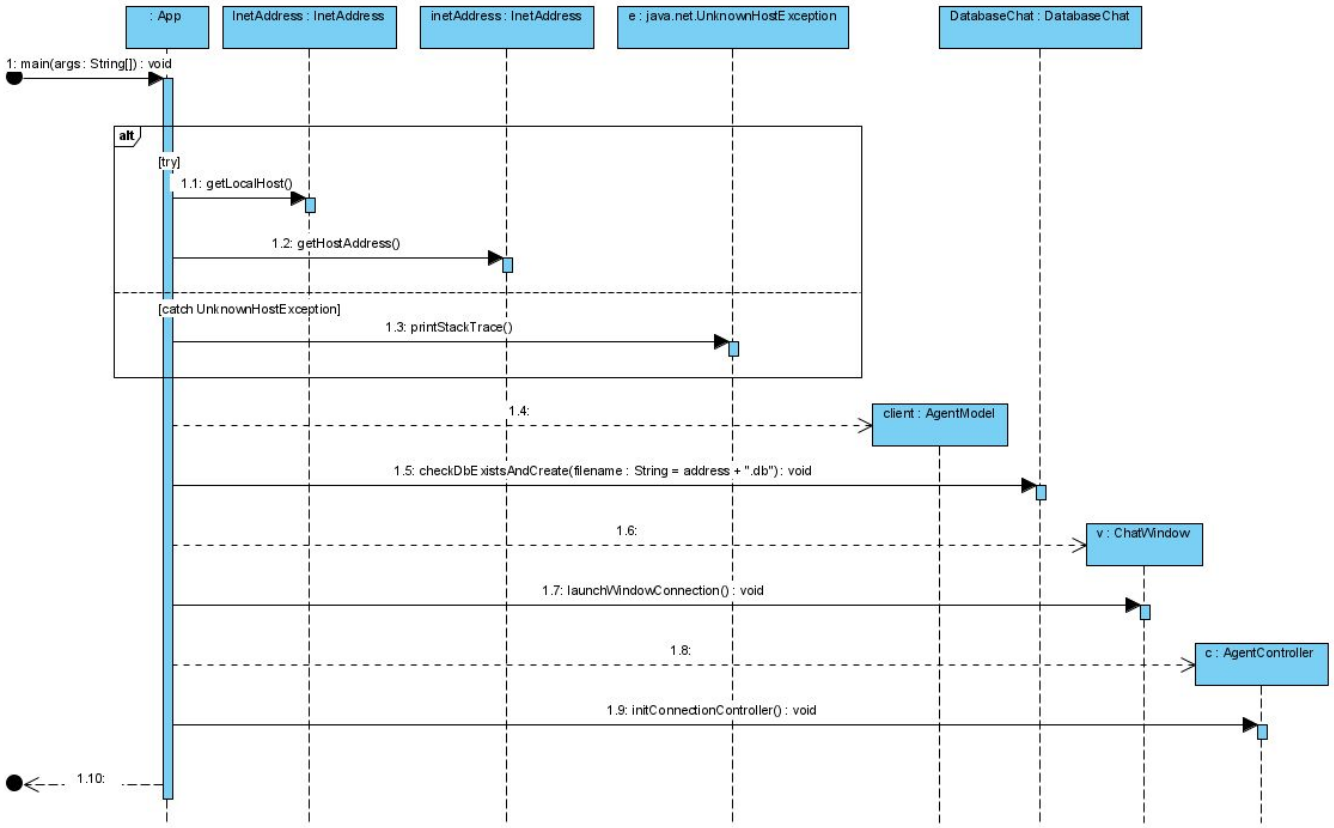

figure 2 : Lancement de l'application

Le diagramme de séquence du démarrage de l'application montre la première fonction lancée au démarrage de l'application, main() dans la classe App. Elle récupère l'adresse ip avec getLocalHost(), initialise la base de données, et les trois classes du modèle MVC.

Connexion

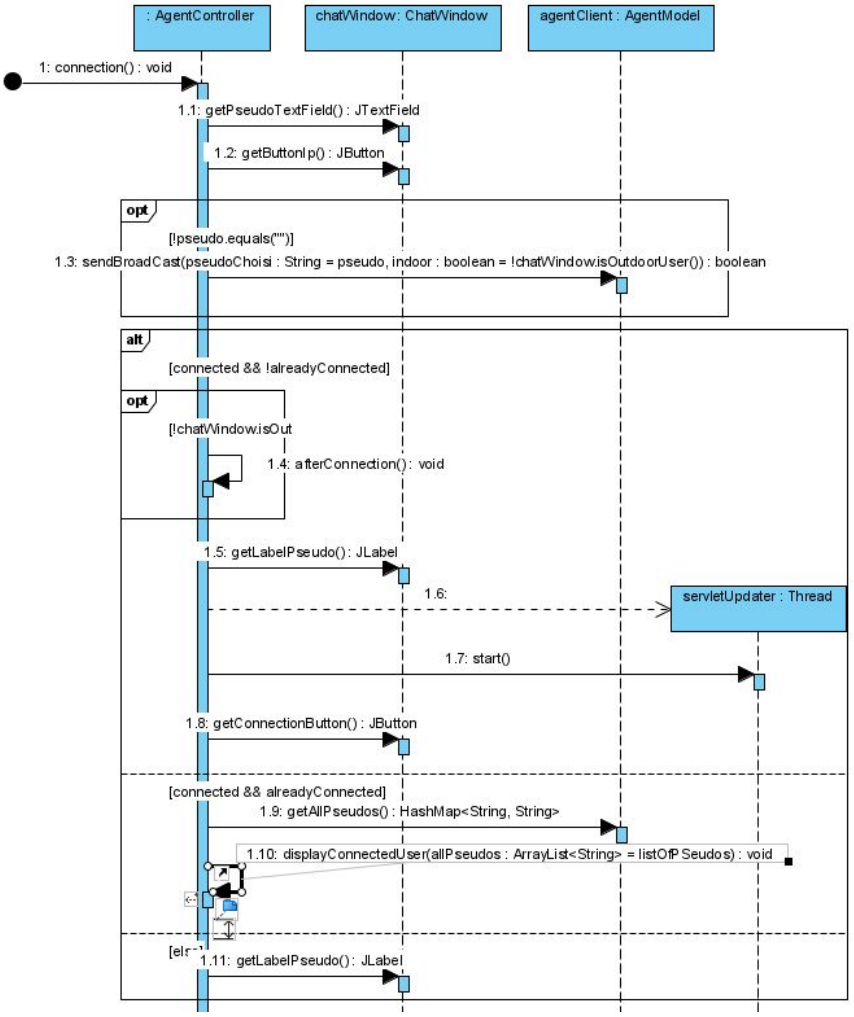

Figure 3 : Connexion de l'utilisateur

Le diagramme de séquence de connexion représente les différentes fonctions lancées lorsque l'on clique sur le bouton de connexion.

Envoi et réception de messages en indoor

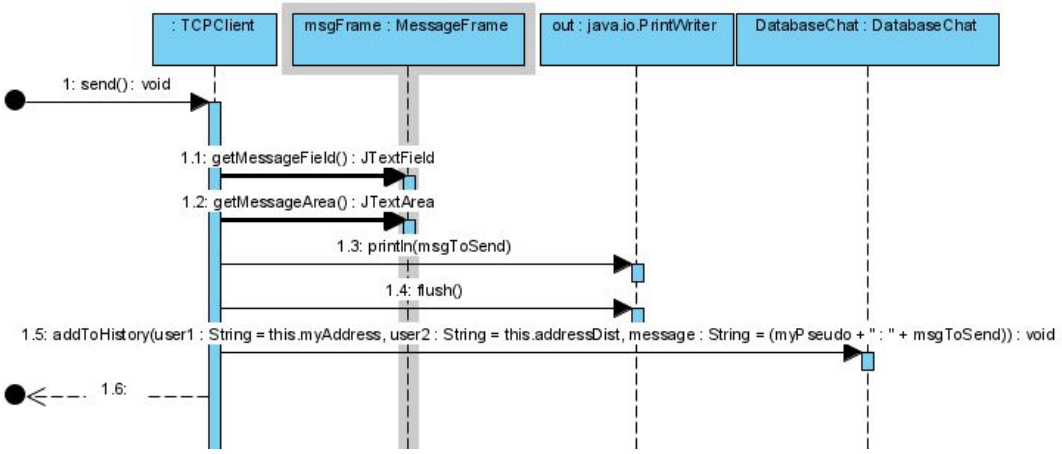

Figure 4 : Envoi de message en local

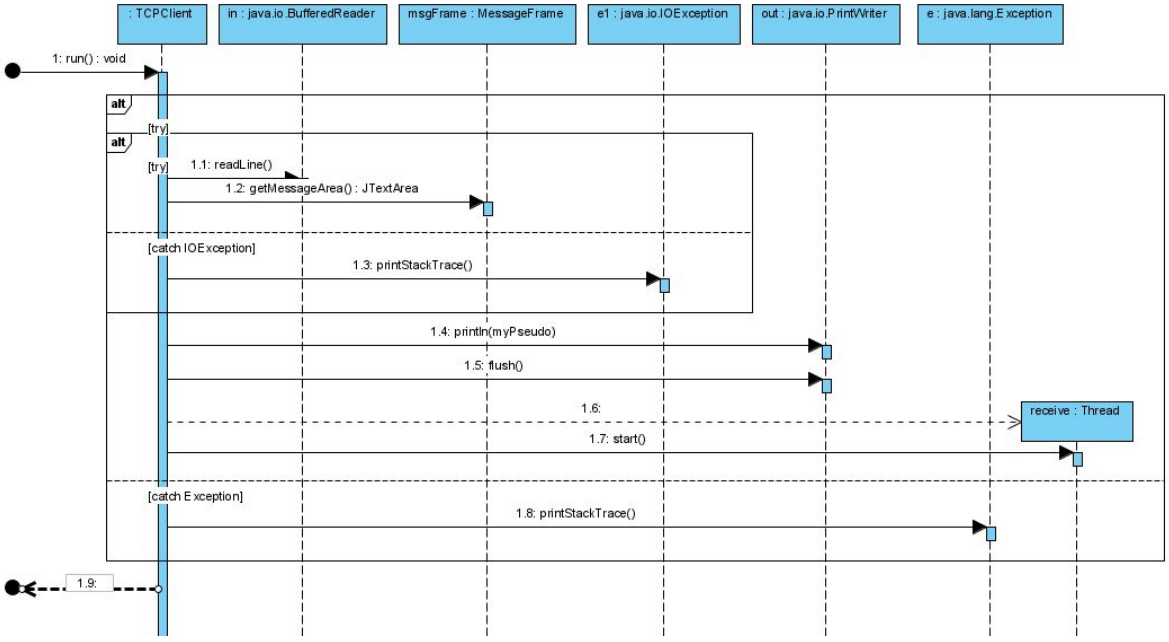

Figure 5 : Réception de message en local

Les diagrammes de séquence de messages Indoor montrent le fonctionnement de l'application lors de l'envoi et la réception de message par des utilisateurs présents sur le réseau local.

Envoi et réception de message en outdoor

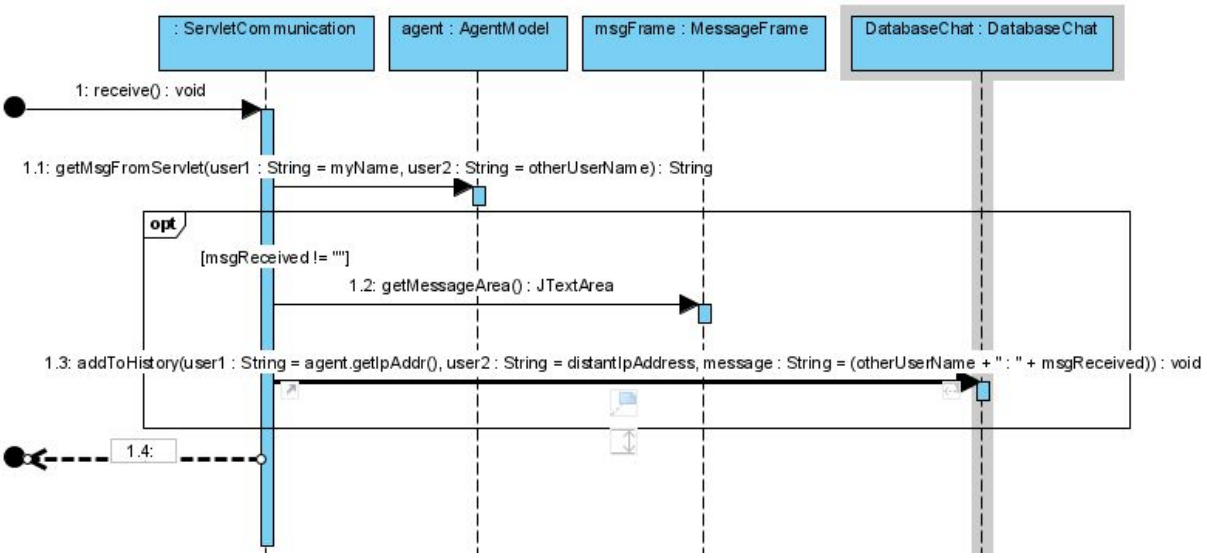

Figure 6 : Envoi de message avec le Web Server

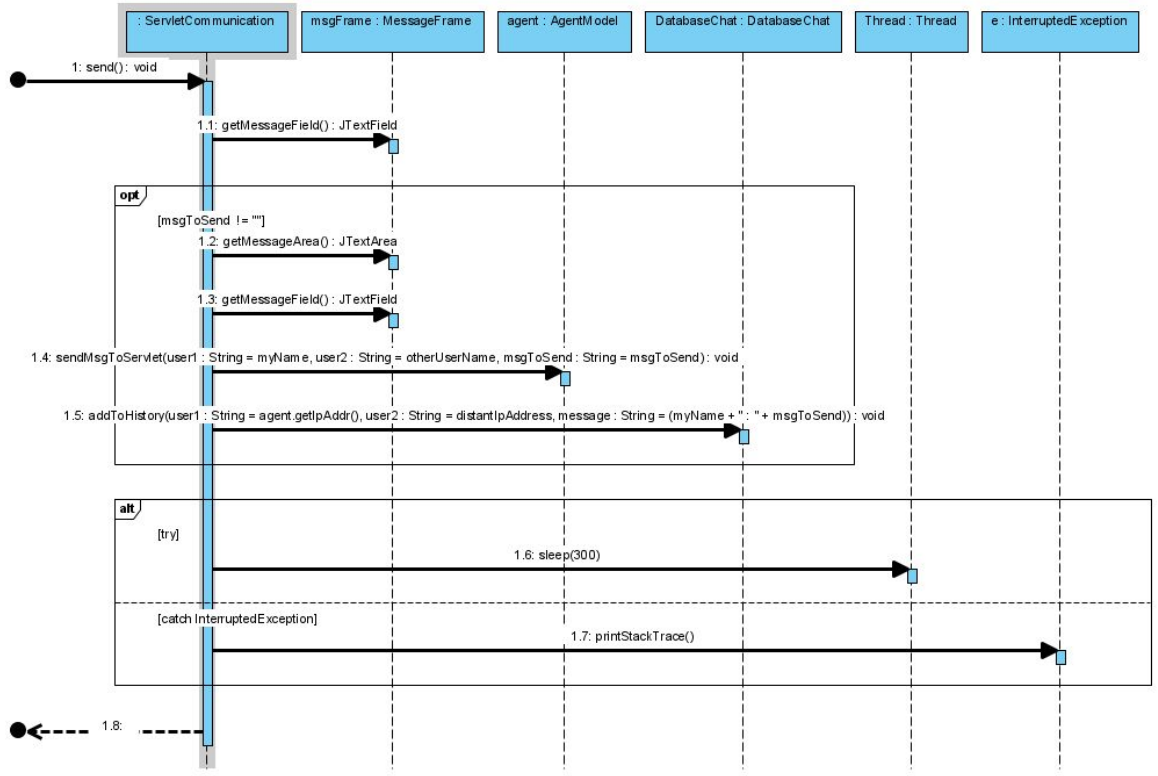

Figure 7 : Réception de message avec le Web Server

Les diagrammes de séquence de messages Indoor montrent le fonctionnement de l'application lors de l'envoi et la réception de message par des utilisateurs externes qui ne font donc pas partie du réseau local.

Diagramme de classe

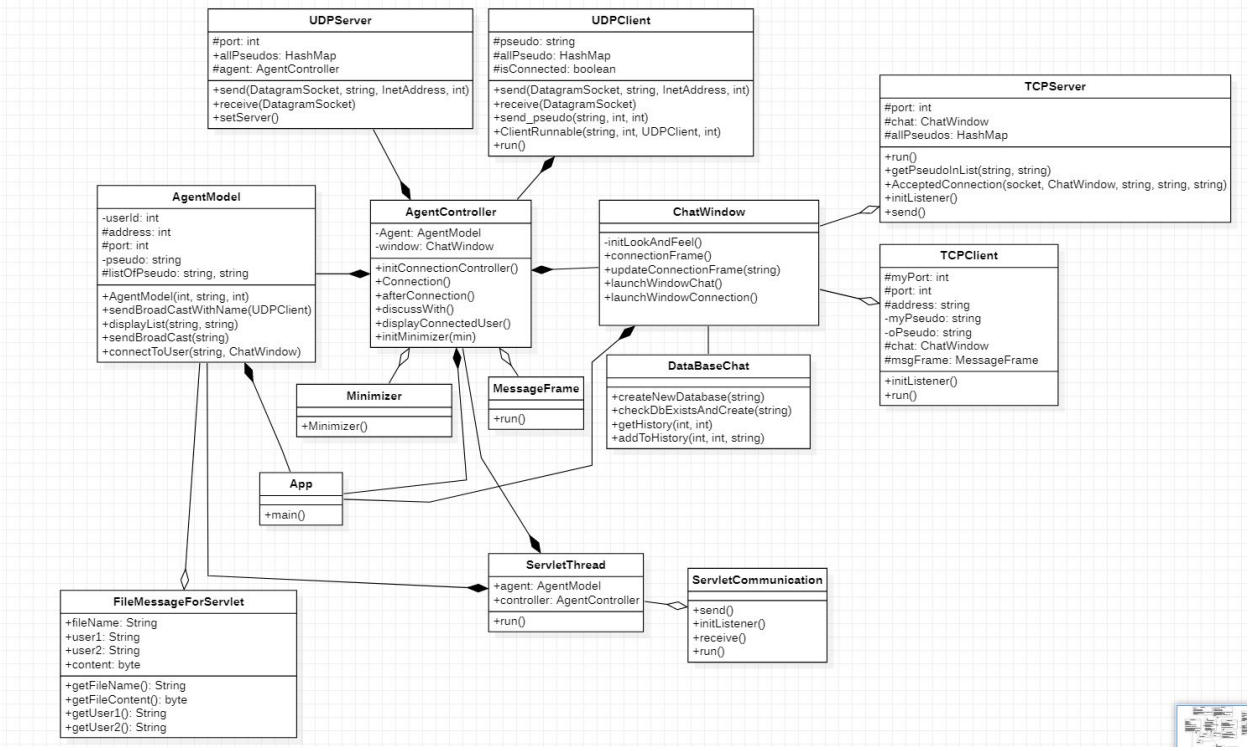

Figure 8 : Diagramme de classe de l'application de chat

Le diagramme de classe de l'application montre les différentes classes de l'application et les interactions entre elles, ainsi que leurs attributs et opérations importants.

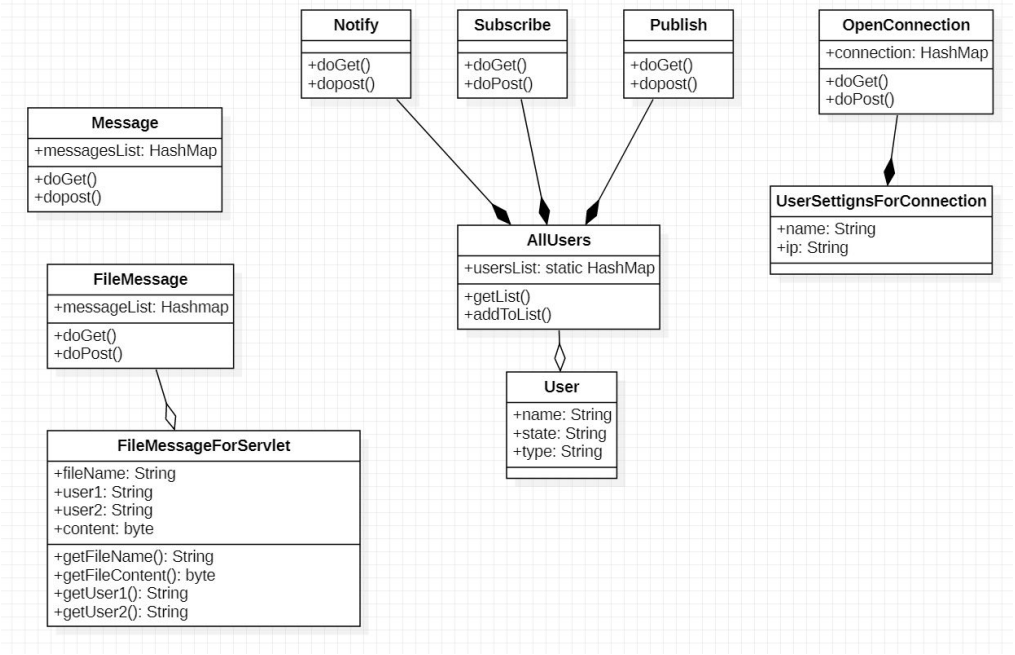

Figure 9 : Diagramme de classe du Web Server

Diagramme de machine à États

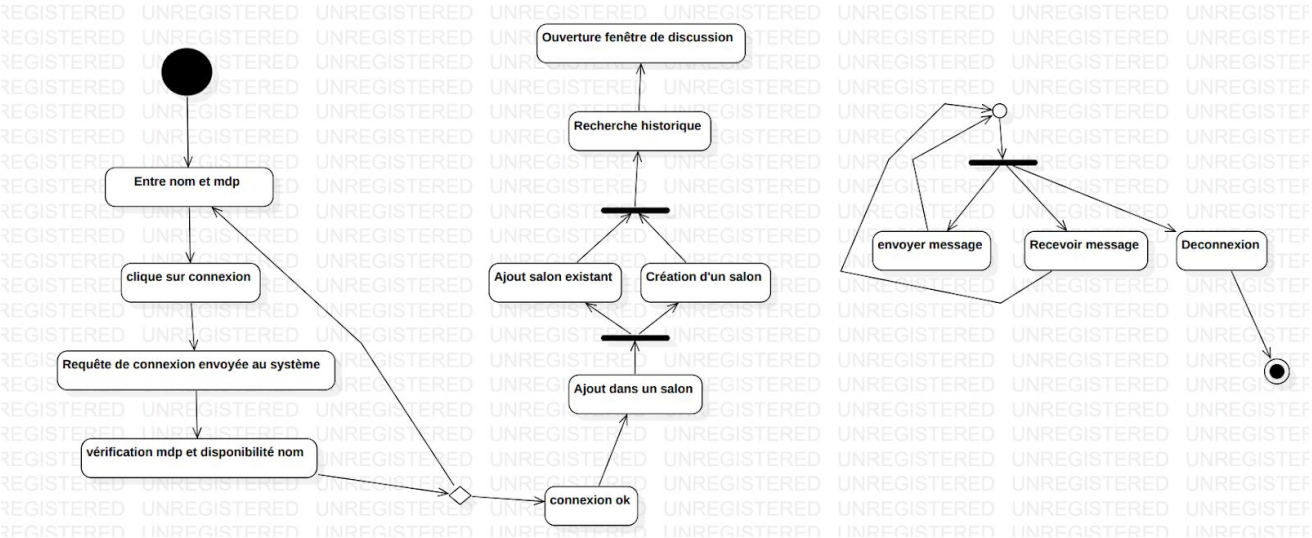

Figure 10 : Diagramme de machine à États de notre application

Installation de l'application

Organisation du git

Dans le git du projet sont présent :

- Ce rapport en pdf
- Un dossier contenant le projet maven java de l'application de chat
- Un dossier contenant le code source du Web Server

Utilisation du git pour installer l'application

Pour installer l'application il faut tout d'abord cloner le répertoire git :

git clone https://github.com/PierreCalmettesInsa/ProjetPoo4A.git

Il faut ensuite installer le Web Server sur un serveur Tomcat. Nous avons déjà installé notre serveur sur le site du gei, son nom est chatServletA2-2.
Pour lancer l'application il suffit de se placer dans le dossier chat, le chemin est : *ProjetPoo4A\maven_project_with_ip\chat* puis d'utiliser la commande suivante :

mvn clean compile assembly:single

Cette commande permet de créer un .jar exécutable. Au préalable il faut avoir installé java et maven sur la machine. Nous avons créé notre application avec Java SE Development Kit 11.0.7 car le Tomcat du gei est sur cette version. Nous conseillons donc d'utiliser celle-ci. Cette version de Java peut être trouvée ici :
https://www.oracle.com/fr/java/technologies/javase/jdk11-archive-downloads.html

Le .jar peut être lancé de deux manières :

- A partir d'un terminal avec la commande *java -jar chemin_du_jar* , cela nous a permis de déboguer le programme car les messages s'affichent dans le terminal
- Directement en double-cliquant sur le .jar

Le .jar créée avec la commande maven se trouve dans le dossier suivant :

ProjetPoo4A\maven_project_with_ip\chat\target

Pour les outdoors users, nous avons donc choisi d'utiliser le site du gei pour utiliser notre web server, donc pour pouvoir utiliser l'application sans avoir à installer le web server, il faut soit être connecté sur le réseau de l'INSA, soit utiliser un vpn. Cela n'est pas obligatoire pour utiliser le chat en indoor, l'application fonctionne sans le Web Server.

Guide d’utilisation de l’application

Il faut tout d’abord exécuter le .jar. Une fenêtre de connexion s’affiche.

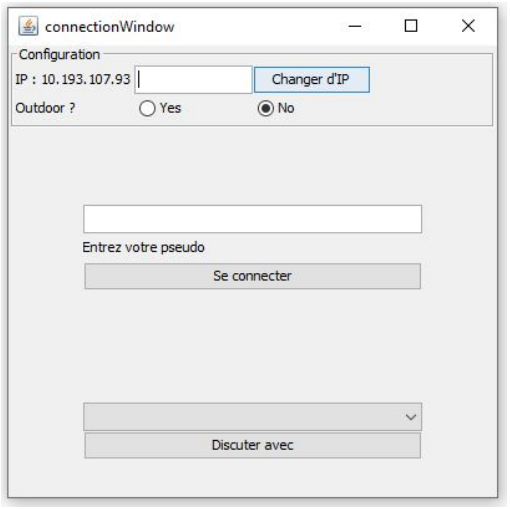

Figure 11 : Fenêtre de connexion

Il faut alors vérifier que l’application a choisi la bonne adresse ip, si ce n’est pas la bonne, il faut la modifier. Il faut ensuite choisir si l’on est un outdoor user ou non. On peut ensuite choisir son pseudonyme puis se connecter.

Si le pseudonyme est correct, la fenêtre affiche un message “Connected !” et le pseudonyme apparaît dans la liste. Il est possible de changer de pseudo à tout moment en écrivant un nouveau pseudo et en cliquant sur le bouton “Changer de pseudo”.

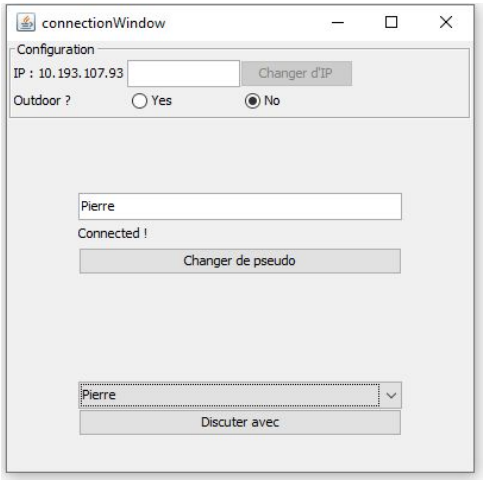

Figure 12 : Fenêtre de connexion une fois connecté

Une fois connecté on peut choisir un utilisateur dans la liste et cliquer sur “discuter avec”.

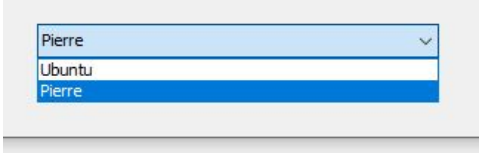

Figure 13 : Choix de l’utilisateur avec qui discuter

Une fenêtre de chat s’ouvre alors. L’historique des conversations antérieures avec l’autre utilisateur s’affiche.

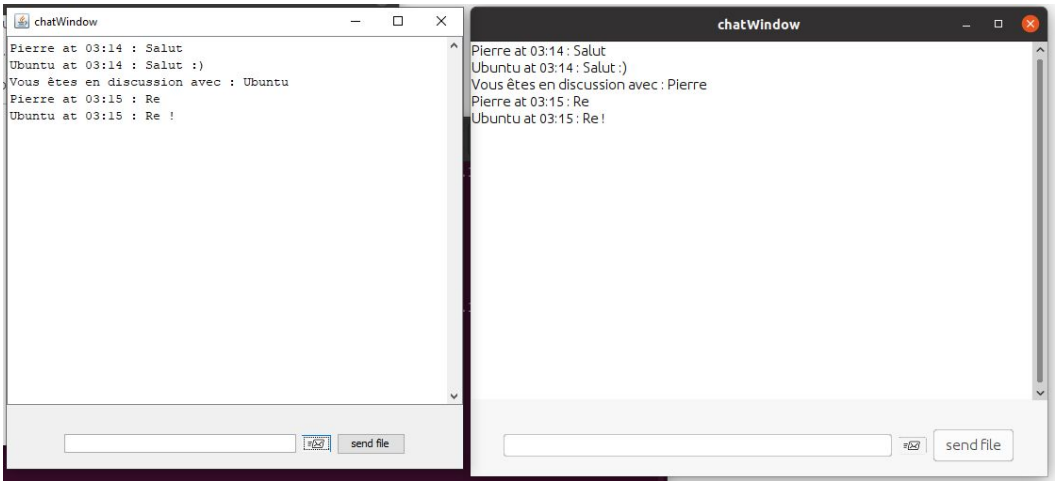

Figure 14 : Fenêtre de chat

On peut envoyer des messages en écrivant dans la barre de message puis cliquer sur le bouton avec l’icône de lettre.

On peut également envoyer des fichiers. Il faut cliquer sur le bouton “send file”, une fenêtre s’ouvre pour choisir le fichier que l’on souhaite envoyer.

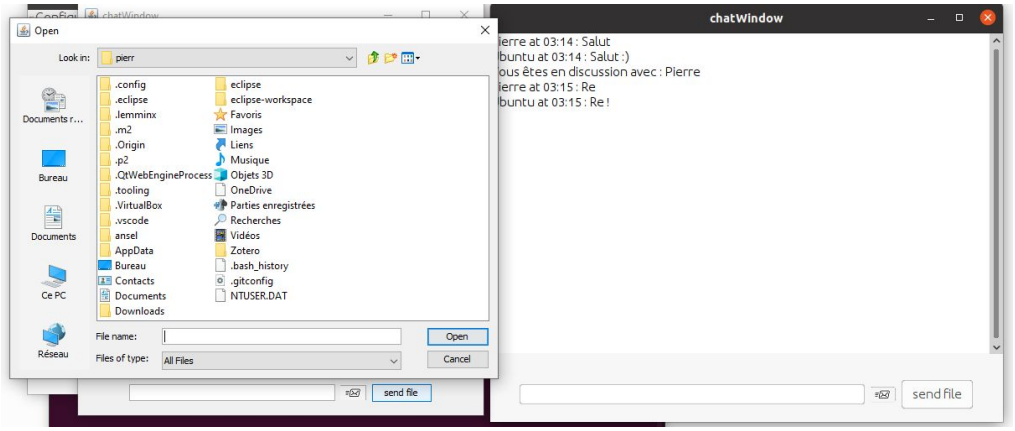

Figure 15 : Exemple d’envoi de fichier

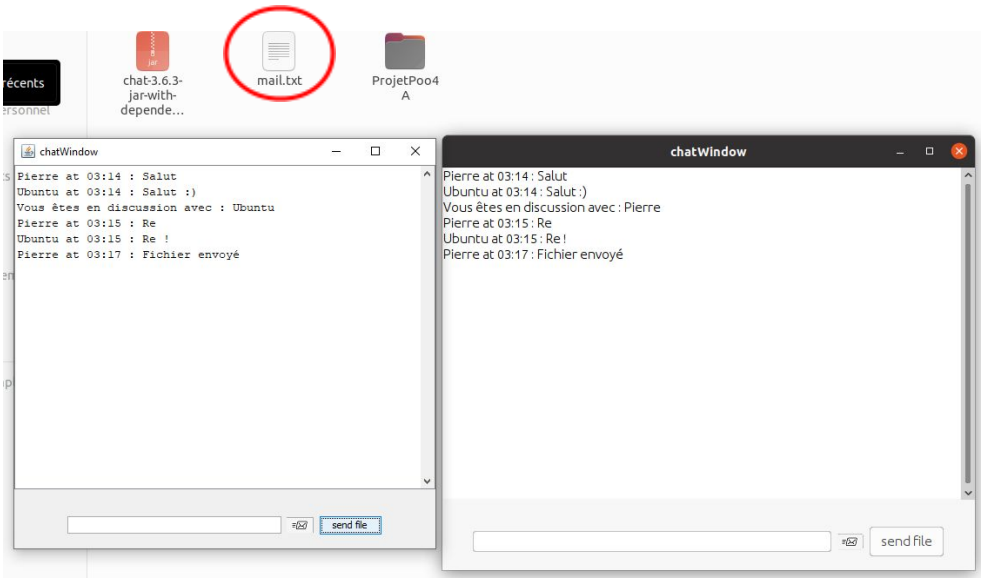

Figure 16 : Réception d'un fichier

Un message est envoyé dans le chat et le fichier est téléchargé et se trouve à l'emplacement du .jar (ici mail.txt entouré d'un rond rouge).

Il est possible de discuter avec plusieurs utilisateurs en même temps.
A la fermeture d'une conversation, l'utilisateur distant reçoit un message "User disconnected".

```
Pierre at 03:17 : Fichier envoyé
User disconnected
```

Figure 17 : Déconnexion d'un utilisateur

Lorsque l'on ferme la fenêtre principale, sur les OS qui le permettent (en particulier Windows), l'application se minimise dans les icônes cachées en bas à droite de l'écran, l'icône est une bulle de chat. On peut alors faire un clic droit dessus et choisir de réafficher l'application ou de la fermer définitivement.

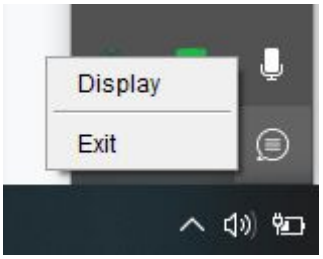

Figure 18 : Application minimisée

Architecture du logiciel

Outils utilisés pour réaliser le projet

Utilisation de **JDK 11.0.7** pour avoir la même version que le Tomcat de l'INSA.

Le projet a été réalisé avec **Maven** pour pouvoir facilement le compiler et l'exporter en .jar, ce qui facilite le déploiement. Maven permet également une gestion facile des dépendances (driver SQLite par exemple).

Utilisation de l'IDE **Eclipse for Enterprise Java Developer** pour le développement du Web Server.

Utilisation de **SQLite** pour la base de données.

Choix pour l'interface graphique

Pour l'interface graphique nous avons choisi d'utiliser **Swing**.
Nous avons suivi le motif d'architecture **Modèle-Vue-Contrôleur**.
Dans nos classes, le **modèle** est la classe **AgentModel**, cette classe contient toutes les fonctions permettant de modifier la vue selon ce que demande le contrôleur.
La **vue** est la classe **ChatWindow**, elle permet de créer la fenêtre JFrame avec Swing.
Le **contrôleur** est la classe **AgentController**, elle met en place les listener sur les différents modules de la JFrame et y associe des fonctions du modèle.

Démarrage de l'application

Au lancement du .jar, c'est la fonction main de la **classe App** qui est exécutée. Cette fonction effectue plusieurs choses. Tout d'abord elle récupère l'adresse ip de la machine. Si celle-ci a plusieurs adresses ip et la fonction choisit la mauvaise, l'adresse pourra être modifiée plus tard dans la fenêtre de connexion.

Cette fonction créée ensuite trois instanciation de AgentModel, ChatWindow et AgentController pour initialiser le modèle MVC. Elle initialise aussi la base de données.

Fonctionnement de la connexion et de la découverte des autres utilisateurs

Pour que la connexion fonctionne il faut au préalable vérifier l'adresse ip et choisir si l'on est un outdoor user ou non. Au clic sur le bouton connexion, la fonction **connection**() dans l'AgentController est executée. La fonction **sendBroadcast**() de AgentModel est alors effectuée. Si l'utilisateur est un outdoor la fonction ne fait rien et passe à la fonction **sendToServlet()**.
L'adresse ip de broadcast est tout d'abord récupérée. Un nouveau **client UDP** est créé puis exécuté dans un **ExecutorService**. Le thread est kill au bout de deux secondes, en effet, si l'utilisateur est le premier à se connecter personne ne va lui répondre, pour que le thread n'attende pas indéfiniment nous avons choisi de lui laisser deux secondes ce qui devrait être largement suffisant, les autres utilisateurs sont en local, les échanges de messages en UDP sont donc très rapide (ms).
Voici l'ordre des messages échangés :

- Le client UDP envoie sur l'adresse de broadcast un message
- Il reçoit toutes les adresses ips des utilisateurs locaux et les stocke dans un array
- Pour chaque ip, l'utilisateur envoie son pseudo
- Il envoie ensuite son adresse
- S'il reçoit d'un des utilisateurs un message "Refused" cela veut dire que le pseudo est déjà utilisé, un boolean passe à false, la connexion échoue et un message demande à l'utilisateur de choisir un autre pseudo
- Sinon il reçoit les adresses et les pseudos de chaque utilisateur

Une fois le sendBroadcast() terminé, la fonction sendToServlet() est executée, l'utilisateur envoie une requête **HTTP doGet** au servlet avec son pseudo, son type (outdoor ou indoor) et son adresse ip. Il envoie ensuite une deuxième requête pour passer son statut en online. Si le sendBroadcast ou le sendServlet renvoie false, ce qui signifie que le pseudo n'est pas valide, l'utilisateur doit changer de pseudo.
Pour le sendBroadCast les utilisateurs vérifient que le nouveau pseudo n'est pas déjà présent dans la liste. S'il l'est, ils vérifient aussi si l'adresse ip est différente, et alors la connexion est refusée. Pour le Servlet, celui-ci vérifie également que l'utilisateur avec le même pseudo est online, s'il ne l'est pas alors le pseudo est correct.

Une fois la connexion établie, trois threads sont créés. Un thread UDP serveur qui répond aux clients UDP pour leur connexion en créant des threads fils (il peut donc recevoir plusieurs demandes de connexion en simultanée). Un thread pour le servlet qui de temps en temps effectue un notify pour voir s'il y a des nouveaux outdoors users, et qui regarde également si un utilisateur veut discuter via le servlet. Pour finir, un thread pour le socket TCP qui attend qu'un client TCP lui demande de se connecter, dans ce cas-là un nouveau thread fils est créé, cela permet d'avoir plusieurs discussions en simultanée.

Echange de messages en indoor

L'échange de message en local s'effectue à l'aide de **socket TCP**. Une JFrame pour l'affichage de la conversation est créée pour les deux utilisateurs. Le TCP client demande la connexion, le TCP serveur l'accepte. L'envoi de message s'effectue lorsque le bouton d'envoi est appuyé. Pour la réception, un thread attend dans une boucle infinie de recevoir des messages. Si ce thread reçoit un message spécial, "---Sending file--- code : 12976#", alors il sait que l'autre utilisateur s'apprête à envoyer un fichier.
Pour envoyer un fichier nous avons fait le choix d'utiliser des **mutex**, l'envoi se fait dans une fonction **sendFile**(), pour le bon déroulement de l'envoi cette fonction doit recevoir des messages de confirmation, sauf que le thread de réception est toujours actif et empêcherait la réception des messages par la fonction sendFile(). Nous avons donc mis un mutex **ReentrantLock** qui permet de bloquer le thread de réception le temps de l'envoi de fichier.
Nous avons également dû désactiver le bouton d'envoi, en effet comme la réception est bloquée le temps de l'envoi de fichier pour l'autre utilisateur, il ne faut pas que des messages soient envoyés pendant ce temps sinon ils seraient perdus ou même empêcheraient le bon déroulement de l'envoi. Le bouton est réactivé une fois l'envoi terminé.

Echange de message en outdoor

Pour entrer en contact avec un autre utilisateur en mode outdoor, il faut faire une requête **HTTP Post** au Web Server pour ajouter dans un HashMap que l'on veut parler avec la personne. Les utilisateurs vérifient en permanence grâce à un thread et des requêtes au Web Server si quelqu'un veut discuter avec eux. Si c'est le cas, les deux utilisateurs créent un thread **ServletCommunication** qui ouvre une fenêtre de chat. Pour envoyer un message il faut faire une requête POST, le message est alors stocké dans un **HashMap**. Pour recevoir le message, il faut faire une requête doGet qui va vérifier dans le HashMap, renvoyer le message et l'effacer du HashMap. Pour éviter de perdre des messages, un petit délai de quelques millisecondes a été ajouté entre l'envoi des messages, cela permet à l'autre utilisateur de recevoir le message avant qu'un autre ne soit envoyé. En effet en testant nous avons vu qu'en envoyant des dizaines de messages très rapidement certains pouvaient être perdus. Ce délai ne se remarque pas à l'utilisation.
A la déconnexion un message spécial, "*// Disconnected \\ code 12548trfsg58K*" , est envoyé, l'utilisateur est ainsi averti que l'autre utilisateur s'est déconnecté avec le message "User disconnected".
Pour l'échange de fichier, nous avons créé une classe **FileMessageForServlet** contenant toutes les informations du fichier que l'on souhaite envoyer (nom du fichier, contenu, destinataire, propriétaire). Un nouvel **objet** représentant le fichier est créé puis **sérialisé** en un **array de bytes** (byte[]) et ces bytes sont envoyés au serveur http qui va les **désérialiser** pour reconstruire l'objet et ainsi avoir toutes les informations nécessaires pour stocker le fichier au bon endroit. Le destinataire envoie une requête http au serveur pour demander si des fichiers le concernant sont en attente, le Web Server sérialise de nouveau l'objet pour l'envoyer en byte[] au destinataire qui à son tour le désérialise puis créé un nouveau fichier avec le bon nom et y ajoute le contenu.

Fonctionnement de la base de donnée

Pour la base de données, nous avons choisi une **architecture décentralisée**. Nous avons choisi **SQLite** car il est facile à utiliser. De plus, comme nous avons utilisé maven, le driver est automatiquement téléchargé avec une bonne configuration du POM.xml. Dès qu'un utilisateur envoie ou reçoit un message, celui-ci est ajouté dans sa base locale. Les primary keys de la table history sont les adresses ip, donc si un utilisateur change de pseudonyme, il a toujours accès à son historique. A l'ouverture d'une conversation, une requête est envoyée à la base de données pour obtenir tous les anciens messages échangés entre les deux utilisateurs. La base de donnée locale est créée à l'emplacement du .jar avec comme nom de fichier l'adresse ip de l'utilisateur.

Tests de l’application

Nous avons tout d’abord testé notre application en local avec une machine et une machine virtuelle.
Nous avons testé l’établissement de la connexion, chaque utilisateur découvre bien les autres. Puis nous avons testé le changement de pseudo, les autres utilisateurs reçoivent rapidement le changement.
Nous avons ensuite testé l’échange de messages, de fichiers et l’historique. Pour finir nous avons testé la déconnexion de l’utilisateur et la minimisation. Ces tests nous ont également permis de vérifier que l’application fonctionne sur plusieurs OS, ici Windows et Linux (Ubuntu). Nous avons rencontré un problème au niveau du pare-feu, si l’établissement de la connexion ou l’échange de message ne fonctionne pas, cela peut être dû à une règle manquante autorisant Java à envoyer des messages TCP/UDP sur les ports utilisés.

Nous avons ensuite testé notre application avec trois machines, 2 machines + 1 machine virtuelle. Nous avons vérifié que l’établissement de la connexion fonctionne toujours bien lorsqu’il y a plus de deux personnes. Nous avons également vérifié la possibilité de discuter avec plusieurs utilisateurs en même temps.

Nous avons ensuite vérifié le bon fonctionnement de l’application avec des utilisateurs outdoors avec tout d’abord une machine plus une machine virtuelle connectée en NAT utilisant le même VPN que la machine pour accéder au Web Server. Puis nous avons testé en nous envoyant des messages entre nous alors que chacun était chez soi sur un réseau différent.