

RAPPORT PROJET JAVA

4IR SI

Killian Gonet

Asmun Kanane

Joel Imbergamo Guasch

Introduction

Ce rapport décrit un projet de programmation qui a pour but de créer une application de chat en réseau local de connexion automatique décentralisée. L'application permettra aux utilisateurs de se connecter automatiquement et de discuter en toute sécurité. Les utilisateurs pourront également s'envoyer des messages instantanés. L'application sera développée avec les technologies les plus récentes afin de fournir une expérience de chat en réseau fiable et sécurisée. L'application permettra aux utilisateurs d'accéder à l'historique des messages, les utilisateurs pourront également consulter les messages envoyés et reçus par d'autres utilisateurs.

Dans ce rapport nous allons décrire le processus les choix de conception pris pour la réalisation de ce projet.

Vous trouverez tous les diagrammes UML en annexe ainsi que le manuel utilisateur de l'application

Architecture

Pour réussir à suivre le chairer des charges nous avons d'abord découpé le problème. Nous avons découpé le problème en trois parties :

- Front-end : interface graphique utilisateur.
- Back-end : Gestion du réseaux et base de données.
- Integration : communiquer le front et le back.

MVCS

Nous avons choisi d'utiliser une architecture MVCS pour notre architecture globale. MVCS c'est une extension de MVC. Un service est une classe implémentée comme un singleton utilisé pour gérer toutes les communications avec le monde extérieur et les données communes à l'ensemble de l'application.

Cette architecture est très utilisée dans différents frameworks front-end comme par exemple Angular ou Flutter.

Front-end

Afin de réaliser l'aspect front-end, nous avons décidé d'utiliser le Framework JavaFX, qui permet la réalisation d'interfaces graphiques en Java.

Il utilise pour l'aspect statique du système des fichiers FXML décrivant la structure de l'interface ainsi que les différents layouts des composants. Une application JavaFX est découpée en deux parties : une « Application » et un « Controller » : l'application permet de gérer les propriétés de la fenêtre, et le contrôleur permet de gérer les événements inhérents à la fenêtre.

Une application JavaFX s'exécute sur un thread unique, et il est donc impossible de modifier la fenêtre par un autre processus que celui déjà défini. De ce fait il est nécessaire d'utiliser la fonction « Platform.runLater() », qui permet de lancer la modification de la fenêtre lorsque le Thread exécutant JavaFX aura du temps CPU de libre.

Mise en place du backend

Le back-end regroupe toute la gestion de threads autres que l'interface, la communication et la base de données. De plus nous avons aussi placée la modélisation du problème dans la problématique back-end.

Dans un premier temps nous avons listé le nombre de threads indépendants dont on aurait besoin. Nous avons donc 4 threads (interface exclue) : serveur HTTP, serveur UDP, oubcle de suppression d'utilisateurs enjoignables et un thread implicite générée par Java automatiquement pour l'event loop des CompletableFutures.

Deuxièmement nous avons modélisée les données à gérer : utilisateurs, utilisateurs connectées, messages et pour regrouper toutes ces données d'une façon logique nous avons crée l'objet conversation qui regroupe tous les messages envoyés entre deux utilisateurs.

Les différents Services

Comme expliquée auparavant , nous avons décidé de rajouter les services au modèle MVC (donc MVCS) afin d'avoir le concept de services. Les services son des classes implémentes comme des singletons qui servent à gérer toutes les communications avec l'extérieur et les données communes à toute l'application.

StorageService

Pour la base de données nous avons crée la classe StorageService, elle est chargé de gérer la communication avec la base de donnés et de plus faire une abstraction de celle-ci afin de faciliter l'utilisation de la base de données sans avoir besoin d'écrire des requêtes SQL pour y interagir. Cette classe est un service donc elle suit le patron de conception singleton en plus d'offrir une façade pour le reste de l'application.

Grâce au fait que c'est conçu comme un singleton ce service permet l'utilisation d'une seule instance de connexion à la base de données.

Session Service

Pour la gestion de l'état de l'application nous avons crée le service SessionService. SessionService est un service dont l'objectif est le partages des données internes à l'application entre toutes les classes et threads. C'est la classe qui est chargée de maintenir l'état de l'application toujours à jour, par exemple si l'utilisateur a rentré son pseudo, la liste d'utilisateurs connectés au réseau, etc.

Ce service permet le stockage et l'accès centralisée à toutes les données mutables de l'application. Cette centralisation facilite le développement et assurer l'intégrité des données qui sont partagées entre threads.

Broadcast UDP

Pour obtenir les adresses IP des personnes qui pourraient être connecté sur le réseau et à l'application nous avons procédé ainsi:

- Listing de toutes les adresses connectés au réseau
- Envoie d'un message en broadcast de la part de la personne qui vient de se connecter, en parallèle toute personnes connecté sur l'application écoute sur un port spécifique
- Envoie d'une réponse de la part du nouveau connecté avec son pseudo + adresse IP

Suite à cela l'utilisateurs qui a envoyé la notification de connexions se voit renvoyé ensuite la liste de tous les utilisateurs connectés.

Serveur HTTP

Pour le serveur HTTP nous avons utilisée la librairie standard http de Java. Nous avons crée un handler pour chaque endpoint. De plus ce service est observable avec des futures, c'est-à-dire, nous pouvons souscrire une CompletableFuture au serveur qui sera complété quand il y aura un event qui arrive à partir d'un endpoint.

Toutes les données transférées entre deux nodes sont dans le format JSON pour simplifier le débogage des communications.

Logger Service

Finalement, nous avons décidé d'ajouter un service de logging. Le logging consiste à ajouter des traitements dans les applications pour permettre l'émission et le stockage de messages suite à des événements. Le logging est utile pour tous les types d'applications en permettant par exemple de conserver une trace des exceptions qui sont levées dans l'application et des différents événements anormaux ou normaux liés à l'exécution de l'application.

Pour chaque méthode appelé que nous voulions suivre, nous avons ajouté un log associé qui sera transféré grâce à un handler sur un fichier au lieu de l'affichage sur console et grâce à un formatter le mettra sous un format html.

Justification des choix techniques

HTTP

Nous avons choisi d'utiliser le protocole HTTP qui s'appuie sur TCP au lieu de TCP directement. La librairie standard de HTTP de Java offre directement la gestion d'une queue, des threads de traitement, des endpoints et d'une notion de requête avec la possibilité d'envoi de données à l'allée et le retour.

De plus HTTP offre directement des interfaces async pour faire des requêtes ce qui facilite l'utilisation du réseaux pour ne pas faire des requêtes bloquantes ni au même temps devoir s'appuyer sur des observer qui déléguent la responsabilité à d'autres fonctions.

INTELLIJ

En tant qu'étudiants nous avons accès à la version Ultimate de IntelliJ, cet IDE offre des nombreuses fonctionnalités additionnelles par rapport à d'autres IDEs gratuits donc nous avons choisi de développer en utilisant cet environnement. Le programme développé par JetBrians offre des nombreuses fonctionnalités qui facilitent le processus de développement, comme une autocomplétion très puissante et des outils d'analyse et réorganisation du code ou la gestion de bases de données intégré. De plus nous étions habitués à travailler sous cet environnement donc nous sommes resté avec cet outil.

GRADLE

Gradle est un système de build automatisé , c'est le deuxième système de build plus utilisée pour le langage Java. Il offre des avantages au niveau de facilité d'utilisation mais nous avons décidé de l'utiliser principalement car un des membres du groupe l'avait utilisé pendant son stage et nous avons préféré de rester avec l'outil auquel on était habitués.

JavaFX

Nous avons décidé d'utiliser JavaFX au lieu de Java Swing pour plusieurs raisons. La première est la possibilité de définir l'aspect statique de la frame via un fichier FXML, fichier FXML pouvant être créé graphiquement à l'aide de l'outil Scene Builder. Cet aspect permet de gagner du temps sur l'aspect conception des fenêtres, en prenant en compte les différents layers dès la conception. Par ailleurs, JavaFX est un outil thread-safe par défaut, ce qui dans le cas d'une application utilisant des aspects réseaux comme celle ci, permet de s'assurer de ne pas avoir de perte d'information. Enfin la séparation entre contrôleur et application par défaut à permis de simplifier la structure de l'application, en permettant la mise en place de Listener plus simplement.

SQLITE

SQLite est une base de données stocké dans un fichier qui permet l'utilisation d'une base relationnelle sans avoir besoin d'un prosessus externe avec toute la complexité que cela implique. Le plus grand désavantage de SQLite est la performance, principalement en écriture car elle est limité à une écriture à la fois. Donc pour notre utilisation ou nous n'avons qu'un seul programme qui y écrit à la fois ce n'est pas un problème.

MVCS

Pour l'architecture globale de l'application nous avons choisi de partir sur un modèle MVSC (Model View Controller Service), ceci est un patron de conception qui rajoute le concept de service au modèle MVC classique. Un service est une classe qui suit le patron de conception singleton et qui a comme objectif donner un accès centralisé et standardisé à des données externe à l'application : base de données, serveurs web, ou même données propres au programme mais partagées dans tout le code.

Cette architecture permet d'avoir accès aux données à partir de toute les classes dans le programme au même temps qui permet la facile utilisation du patron de conception façade qui donne un accès facile à la base de données par exemple.

ASYNC

Pour la communication entre threads nous avons utilisé des « CompletableFuture » de la librairie standard de Java. Les CompletableFutures permettent l'exécution de code en asynchrone et facilitent la réaction à des événements asynchrones en utilisant de la programmation réactive. Ils utilisent un event loop pour gérer des tâches asynchrones sans transmettre la responsabilité d'exécution à une autre fonction.

Intégration continue

Nous avons deux travaux qui fonctionne en intégration continue : les tests et la documentation.

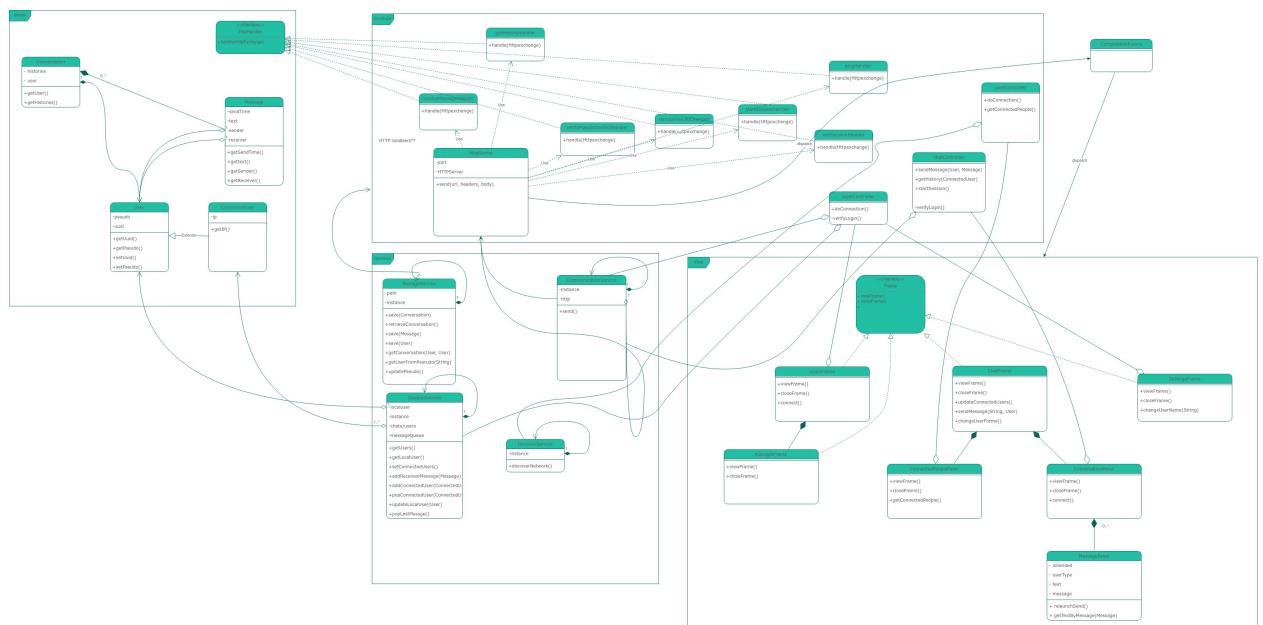
À chaque push sur la branche main il y a une action de github actions qui est lancée automatiquement et qui lance les tests et génère la documentation automatiquement publié sur github pages (https://joelimgu.github.io/POO_COO_4IR/javadoc/) avec des diagrammes de classes.

Nous avons essayé de faire des releases automatiquement aussi en utilisant une action de github aussi (<https://github.com/marketplace/actions/create-release>) et la génération de jar automatique par gradle avec « gradle fatJar ». Mais malheureusement à cause de manque de temps nous n'avons pas pu finir cette étape.

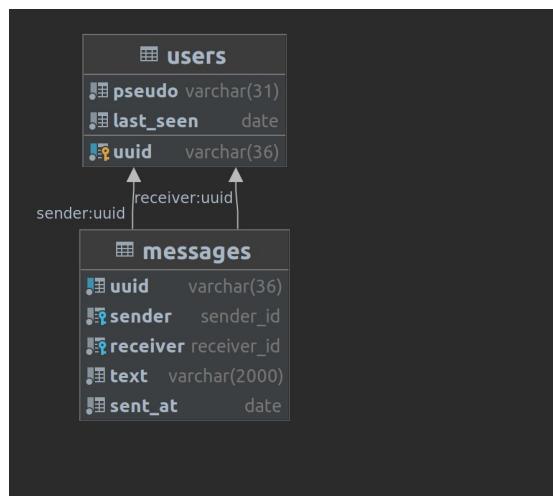
Annexes

Manuel d'utilisateur :

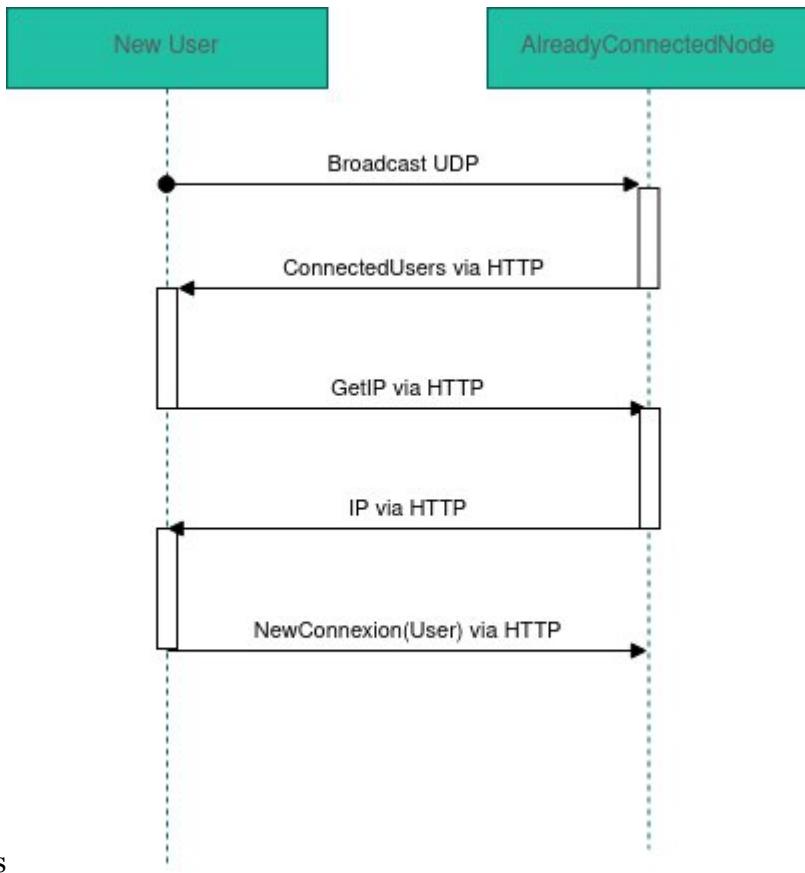
https://github.com/joelimgu/POO_COO_4IR/blob/main/User_Manual_Clavardix_2000.pdf



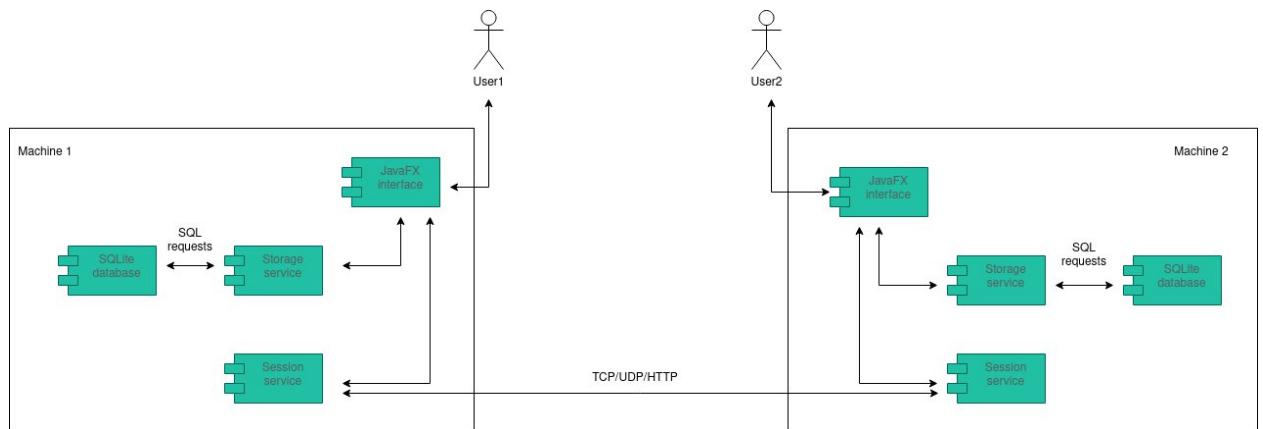
1 - Diagramme de classes initial



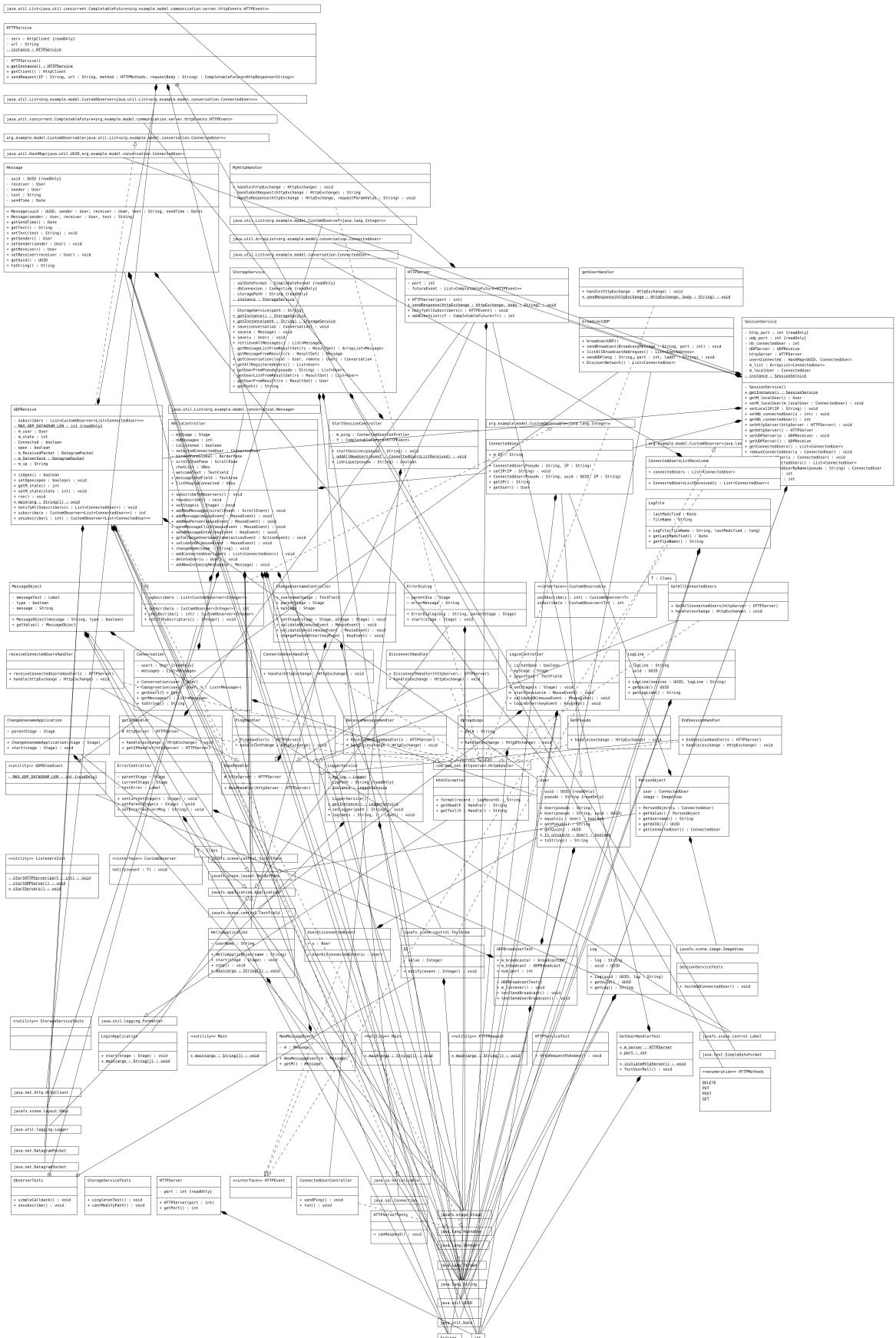
1 - Diagramme de classes base de données



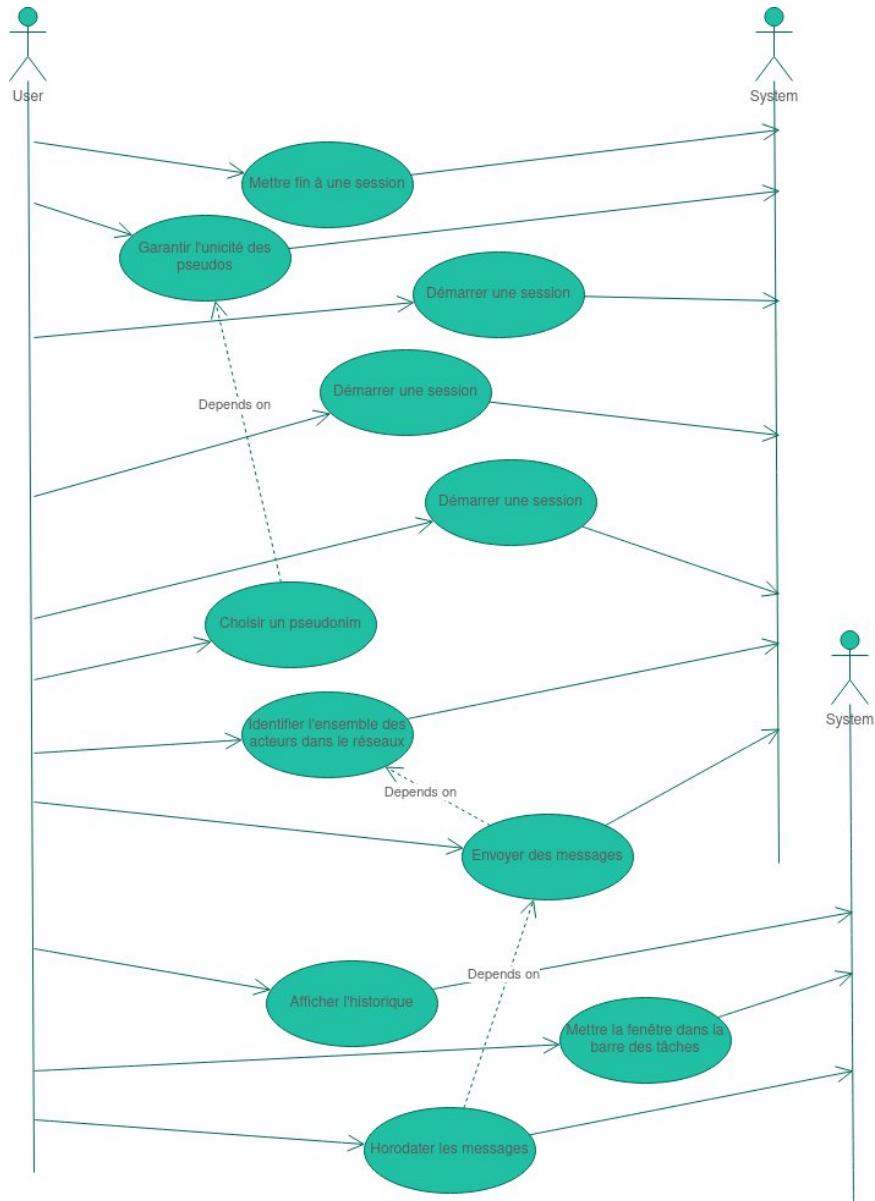
2 - Diagramme de séquence connexion au réseau



3 - Diagramme de déploiement



1 - Diagramme de classes complet du projet final



3- Usecase diagram