INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Rapport de projet

Logiciel de clavardage en Java : conception, développement et déploiement

Rédigé par :

Enjalbert Matéo - Gasc Mayeul

- Version du 27 janvier 2022 -

Rapport de projet

Logiciel de clavardage en Java : conception, développement et déploiement

Matéo Enjalbert (menjalbe@insa-toulouse.fr)
Mayeul Gasc (gasc@insa-toulouse.fr)

Projet de quatrième année IR

DGEI
INSA Toulouse
Version du 27 janvier 2022

Table des matières

Introduction

Objet de Discussion Digitale (ODD) est un logiciel de communication partiellement décentralisé, conçu et implémenté par Matéo Enjalbert et Mayeul Gasc en tant que projet de quatrième année Informatique et Réseaux à l'INSA Toulouse. Il permet un échange de messages entre utilisateurs présent sur le même réseau et il inclut des mécanismes de gestion de l'historique des messages ainsi que de découverte des utilisateurs locaux. Implémenté en Java, il est compatible avec la majeure partie des systèmes d'exploitation modernes supportant Java 11. Il est conçu pour être déployé sur un réseau d'entreprise fermé et il peut être utilisé sans connexion internet.

Le développement de ce logiciel s'est étalé sur deux mois (décembre 2021 et janvier 2022) et a permis d'aboutir à la validation d'une majeure partie des contraintes du cahier des charges. La méthodologie de développement s'est basée sur une méthode Agile et a fait appel à des technologies d'automatisation du développement (en particulier pour la gestion des dépendances et le déploiement du logiciel).

Ce document comprend deux manuels, un pour l'utilisateur final et le responsable du déploiement du système. Ces manuels sont suivis par une synthèse de la conception du logiciel, détaillant la méthodologie, les choix technologiques et les outils de développement utilisés.

1 Manuel utilisateur

Expliquant le fonctionnement de base du client, ce manuel sera utile à l'utilisateur final. L'utilisateur peut utiliser les systèmes d'exploitation GNU/Linux, MacOS ou bien Windows selon ses habitudes et son choix. En suivant les instructions suivantes, vous reconnaissez avoir pris connaissance la licence [1] et l'acceptez sans réserve.

1.1 Pré-requis

Afin de pouvoir installer le logiciel, Java en version 11 doit être installé. Pour cela, merci de se référer au site internet de l'éditeur. Alternativement, sur les systèmes GNU/Linux, l'installation du paquet `openjdk-11-jdk` est suffisante.

Obtenir Java : `https://www.java.com/fr/download/`

1.1.1 Téléchargement et installation

La version la plus récente du logiciel peut être directement récupérée en ligne sur la page *release* du répertoire GitHub :

`https://github.com/Enjmateo/distributed_chat_system/releases`

Sélectionnez la version souhaitée, cliquez sur **Assets** puis sur **release.jar**. Notez que les versions *nightly* ne sont pas recommandées, car moins stables, et que les différentes versions ne sont pas compatibles en elles. En cas de doute, référez-vous à votre administrateur système pour savoir quelle version installer.

Pour lancer le programme, double-cliquez sur le **release.jar**, une fenêtre doit apparaître. En cas de difficultés, vous pouvez également lancer le programme depuis la console avec la commande suivante :

```
java -jar release.jar
```

1.2 Initialisation

Lors du premier lancement, l'utilisateur devra importer le fichier de configuration fourni par l'administrateur. Pour cela, il doit choisir *Change config file* sur la première fenêtre (voir figure 1, page 4).

La configuration est alors automatiquement sauvegardée et cette manipulation n'a besoin d'être effectuée qu'une seule fois. Occasionnellement, votre administrateur réseaux peut vous demander de mettre à jour la configuration suite à la mise à jour du système. Pour la suite, merci de vous référer à la partie *Utilisation* ci-après.

1. `https://github.com/Enjmateo/distributed_chat_system/blob/main/license`

FIGURE 1 – Pour mettre à jour la configuration, sélectionnez *Change config file*

1.3 Utilisation

Dans le cas d'une utilisation courante, l'utilisateur doit commencer par choisir un pseudo. Pour cela, il renseigne dans le champ *pseudo* de la première fenêtre et lance le logiciel en validant avec le bouton *Connect* (voir figure 2, page 4). Attention, si le pseudo déjà utilisé, l'utilisateur sera averti et devra en choisir un nouveau pseudo avant de pouvoir continuer.

FIGURE 2 – Remplissez le champ *Pseudo* puis validez avec *Connect*

Le client est à présent dans la fenêtre de chat principale. Sur la gauche, il voit la liste des utilisateurs connectés. Ayant sélectionné aucune conversation, la zone à droite est vide (voir figure 3). Il peut à tous moments redimensionner la fenêtre selon ses envies en conservant une interface confortable.

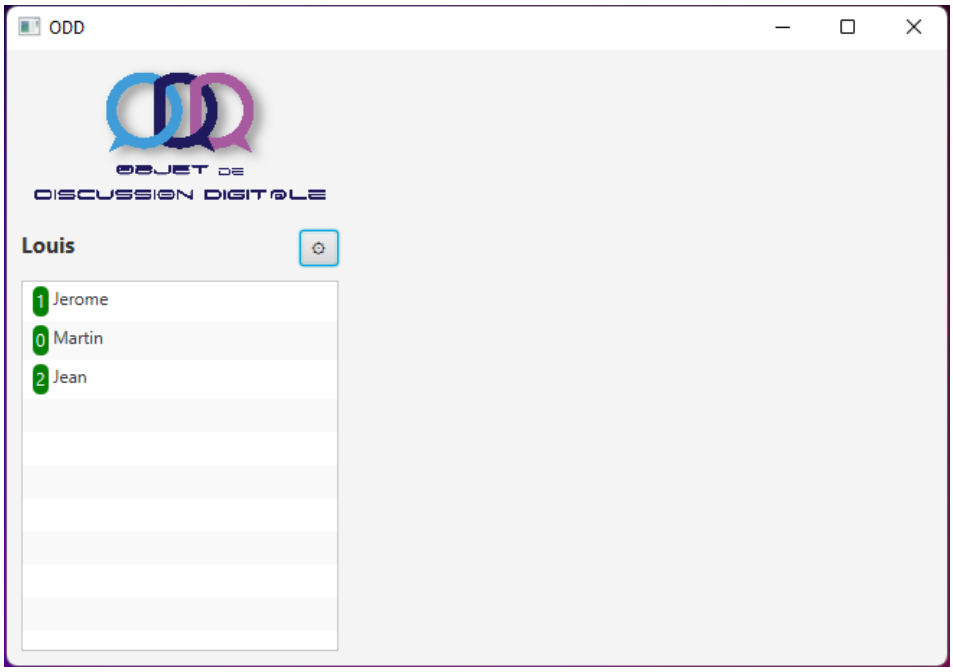

FIGURE 3 – Capture d'écran de la fenêtre principale après connexion.

Il peut converser avec n'importe quel utilisateur connecté en cliquant sur son nom. Si le client a déjà communiqué avec l'utilisateur, il voit alors l'historique de ses messages (voir figure 4, page 6). Ses messages sont à droite en bleu tandis que ceux de son correspondant sont à gauche en gris. Pour envoyer un message, il suffit de le saisir dans la zone en bas puis appuyer sur la touche *Enter* ou le bouton *Send*.

Le client a aussi accès au nombre de messages non lus dans les autres conversations, indiqué par un chiffre dans une bulle de couleur. Cette bulle est **verte** si l'utilisateur est connecté et **rouge** si l'utilisateur s'est déconnecté (voir figure 5, page 6). Dans ce second cas, le client peut essayer de se reconnecter à l'utilisateur en lui cliquant dessus.

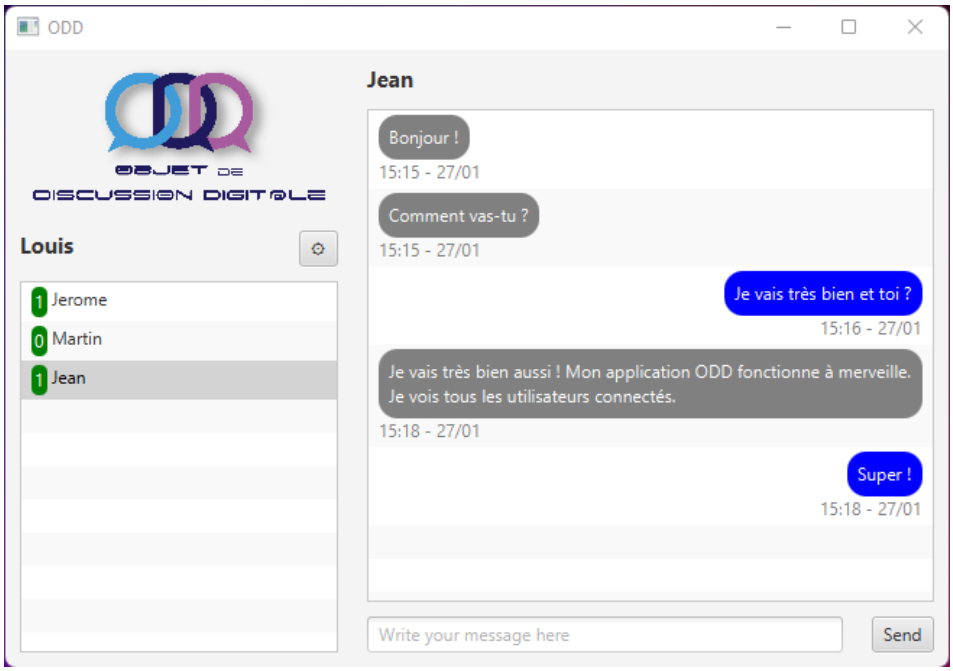

FIGURE 4 – Capture d'écran d'une conversation.

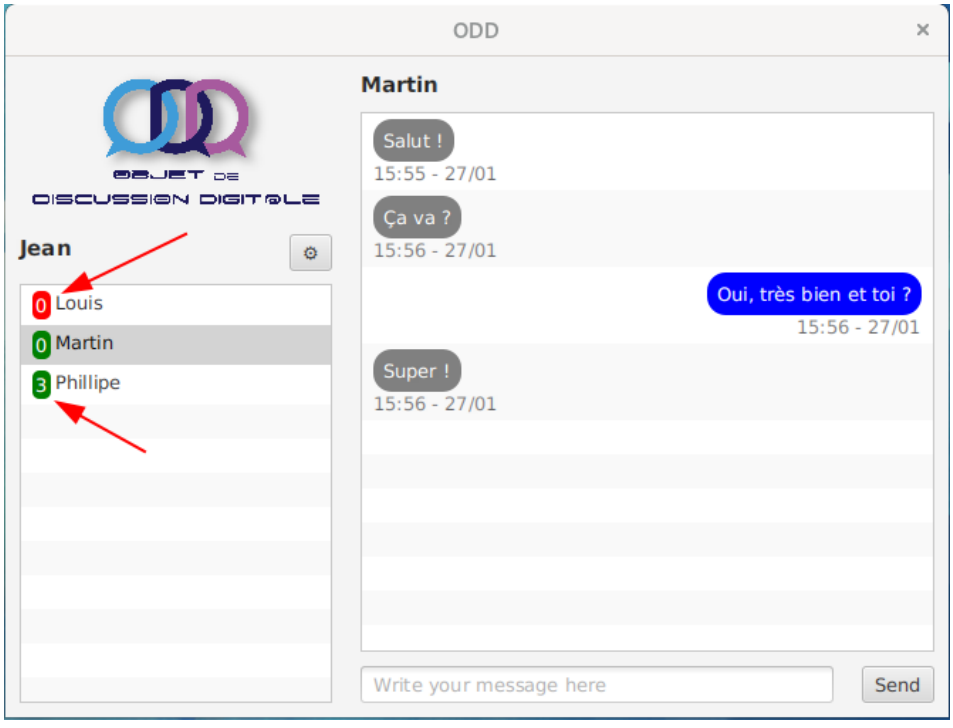

FIGURE 5 – Les flèches indiquent le statut des conversations.

Enfin, le client peut changer son pseudo à tout moment en cliquant sur son pseudo au-dessus de la liste des utilisateurs connectés. Il valide son nouveau pseudo avec le bouton à droite du champ *Pseudo* (voir figure 6, page 7).

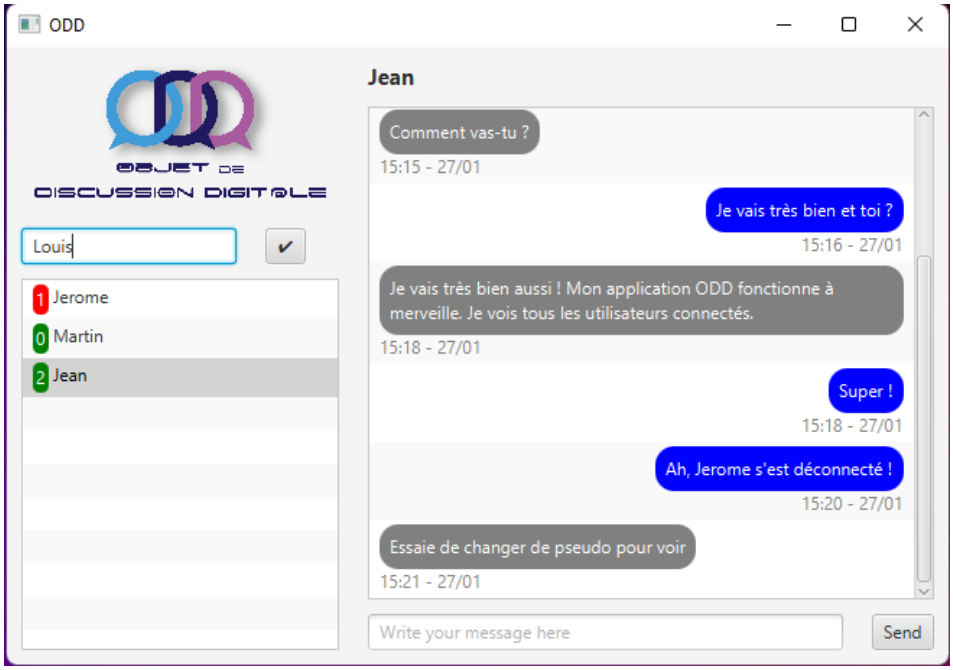

FIGURE 6 – L'utilisateur peut changer de pseudo en cliquant sur le pseudo courant.

Pour quitter le programme, il suffit à l'utilisateur de fermer la fenêtre. Les messages qu'il a envoyés seront conservés et disponibles à la prochaine ouverture du programme.

2 Manuel administrateur

2.1 Utiliser les sources *(Facultatif)*

Pour pouvoir apporter des modifications visuelles et/ou vous assurer de l'authenticité du programme, vous pouvez construire le programme depuis les sources. Sinon, merci de vous référer au *manuel utilisateur*, section *Téléchargement et installation* (page 3 de ce document).

2.1.1 Préparation

L'environnement de développement se base sur **Maven** pour la gestion des dépendances et **Git** pour la gestion des versions. Le module JavaFX est également utilisé pour l'interface graphique [2]. Vous pouvez obtenir Maven sur le site de l'éditeur [3]. À l'instar de l'utilisateur, vous devez également disposer de Java 11.

Sur un système Ubuntu, l'ensemble des dépendances pour être installé avec la commande suivante :

```
sudo apt install openjdk-11-jdk maven git
```

2.1.2 Construire depuis les sources

Pour construire une version spécifique, récupérez les sources sur la page *release* du GitHub. Pour récupérer les dernières sources, clonez le projet sur votre machine :

```
git clone https://github.com/Enjmateo/distributed_chat_system
cd distributed_chat_system/
```

Vous pouvez maintenant lancer Maven :

```
mvn assembly:assembly
```

Une fois le *build* terminé, vous pouvez récupérer le JAR dans le dossier `target/` [4] :

```
$ ls target/
distributed_chat_system-0.0.7-jar.jar
distributed_chat_system-0.0.7-jar-with-dependencies.jar
```

Si vous souhaitez diffuser ce *build*, utiliser le JAR avec dépendances.

2. Maven doit résoudre la dépendance. En cas de difficultés, installez JavaFX manuellement.
3. Lien direct : `https://maven.apache.org/download.cgi`
4. Notez que le nom du fichier dépend de la version courante.

2.2 Préparer l'environnement

2.2.1 Base de données messages

Pour fonctionner avec l'ensemble des fonctionnalités (en particulier la gestion de l'historique des messages), le logiciel doit pouvoir accéder à une base de données commune. Pour configurer la base de données, l'administrateur doit utiliser le script `init_db.sql` (disponible sur Github). Pour utiliser ce script, modifiez la première ligne par le nom de la base de données que vous souhaitez utiliser puis lancer le script :

```
mysql -u my_db_user -h my-server.com -p < init_db.sql
```

Ce script permet également de réinitialiser la base de données. Ainsi, **tous les messages déjà présents seront définitivement perdus**.

2.2.2 Fichiers de configuration

Afin de simplifier la configuration pour l'utilisateur final, le logiciel accepte un fichier de configuration (voir *Configuration*, page 3 de ce document). Une fois importé dans le logiciel, le fichier est conservé par le programme. Ce fichier est au format JSON et comporte trois champs :

```
{
    "dbAddr": "mysql://my-server.com/my_DB",
    "dbUser": 'my_db_user",
    "dbPassword": "my_password"
}
```

Ajustez ces champs en fonction de votre environnement et assurez-vous de mettre à disposition ce fichier de configuration aux utilisateurs.

2.2.3 Configuration réseau

Pour permettre la communication entre les clients, les pare-feu doivent autoriser la communication sur 3 ports au minimum :

— UDP 4444
— TCP 4445
— TCP 4446

Note : *Au besoin, ces ports peuvent être modifiés dans* `utils/Consts.java` *(rebuild nécessaire). Ces ports ne sont pas filtrés par défaut sur les systèmes Debian et assimilés.*

3 Procédure de développement

3.1 Conception

3.1.1 Analyse du cahier des charges

Ce projet répond à un cahier des charges spécifique, la première étape de la conception a été la lecture et l'analyse de celui-ci. Il nous a donné une idée générale de ce qui était attendu, mais présentait des contraintes qui demandaient un temps de développement significativement plus élevé, nous avons donc dû faire des choix dans les spécifications implémentés. Nous avons décidé de nous en tenir à une implémentation simple (peu de sécurité, pas de portage Android, etc.) mais polyvalente et améliorable si le client souhaite investir plus de moyen sur ces points-là. De cette première étude et de ces choix, nous en avons déduit un diagramme des cas d'utilisation (voir figure 7, page 10).

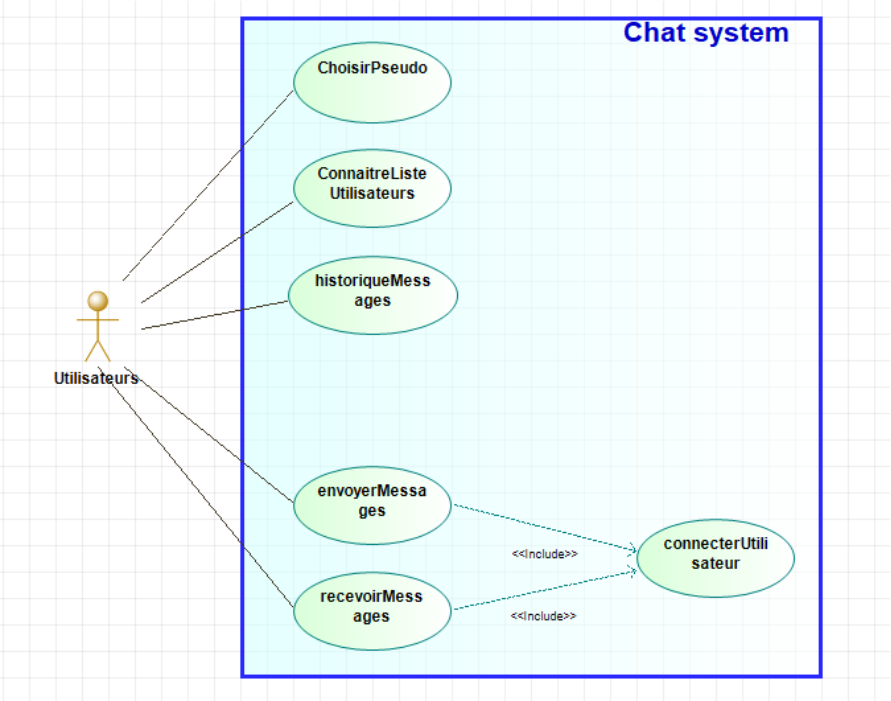

FIGURE 7 – Diagramme de cas d'utilisation réalisé sur Modelio

3.1.2 Structure générale

Durant la conception, nous avons établi des diagrammes de classe que nous avons fait évoluer tout au long de la production. Nous vous présentons ici des exemples diagrammes aboutis tels qu'ils sont en fin de production. [5]

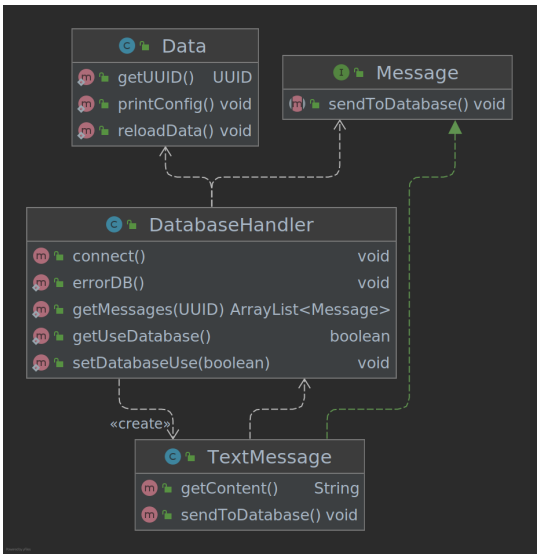

FIGURE 8 – Diagramme de classe du package communication

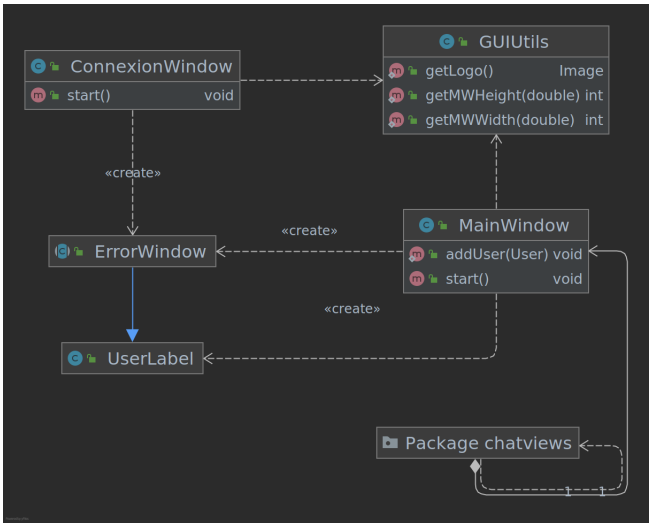

FIGURE 9 – Diagramme de classe du package GUI

5. Nous n'avons pas pu mettre l'ensemble des diagrammes bien trop volumineux

Package communication (voir figure 8, page 11) Il s'agit du package responsable de la communication haut niveau, de la structure des messages et de la gestion de la base de données. Elle se repose fortement sur le package *Utils* qui s'occupe pour sa part (notamment) des communications réseaux bas niveau (TCP & UDP).

Package GUI (voir figure 9, page 11) Responsable de l'affichage de l'interface graphique à l'utilisateur, c'est la partie *view* de notre modèle MVC. Deux fenêtres principales y sont définis (*ConnectionWindow* et *MainWindow*) et font appel à différents utilitaires, notamment *GUIUtils* qui est une boîte à outil pour l'interface graphique.

3.1.3 Diagramme de séquence : découverte des utilisateurs

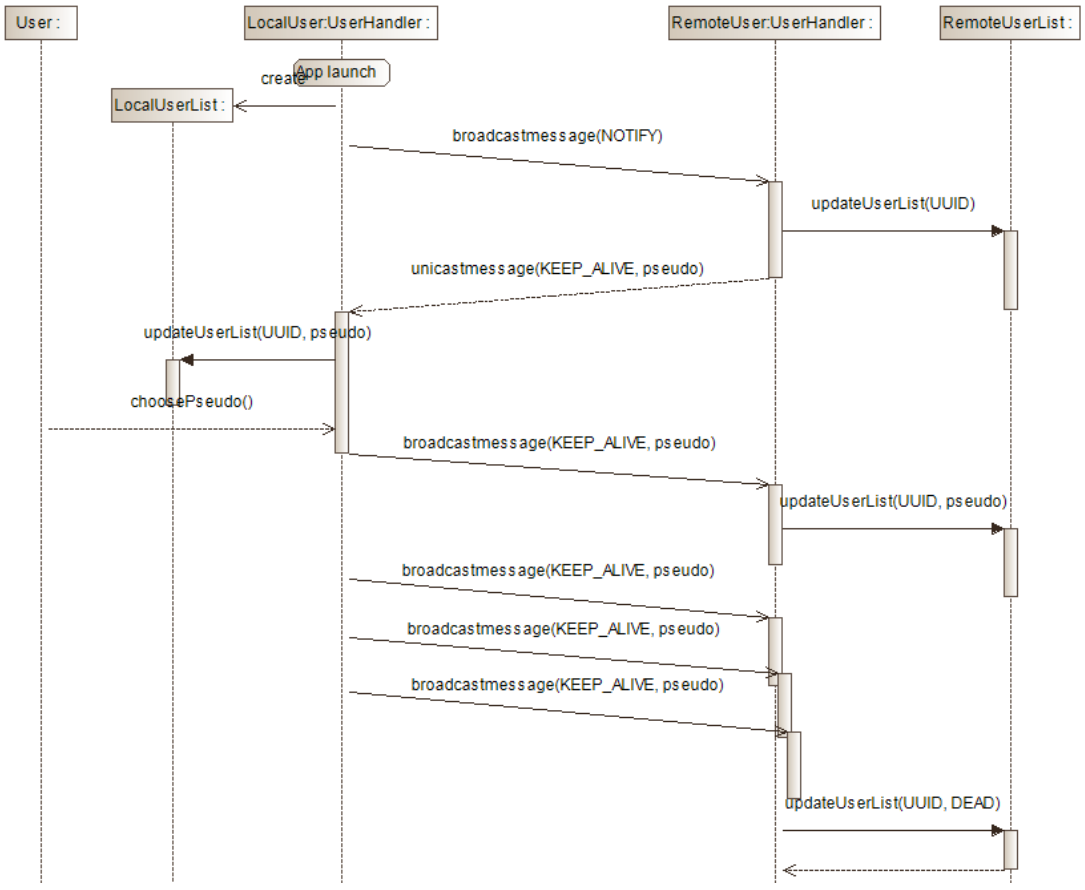

FIGURE 10 – Diagramme de séquence simplifié réalisé sous Modelio modélisant la découverte des autres utilisateurs en UDP.

Une des particularités de notre application est son système de découverte des utilisateurs en UDP, imposé par le cahier des charges. Nous avons illustré notre implémentation par un diagramme de séquence (voir figure 10, page 12).

Un client qui vient de démarrer commence par broadcast un message *NOTIFY* pour prévenir les utilisateurs distants de son existence. À la réception de ce message, les autres clients répondent tous un unicast *KEEP_ALIVE* au nouveau client en leur indiquant leur pseudo. À partir de ces informations, l'utilisateur peut choisir un pseudo. Une fois ce choix, le nouveau client, à l'instar des autres clients, se met à broadcast régulièrement un message *KEEP_ALIVE* avec son pseudo. Lorsque que le client s'arrête, les autres clients cessent de recevoir les *KEEP_ALIVE* et finissent par mettre à jour le client au statut *DEAD*.

3.2 Implémentation

3.2.1 Technologies utilisées

Lors du développement du logiciel, nous avons été amenés à choisir entre différentes technologies. Pour effectuer ces choix, nous nous sommes basés sur nos connaissances dans le domaine et sur la qualité de la documentation à notre disposition. Nous détaillions ci-après ces différents choix.

Maven Nous avons retenu Maven pour la gestion du projet et la gestion des dépendances. Ce qui nous a motivé dans ce choix est la très forte capacité d'automatisation de Maven en termes de gestion des dépendances, y compris en cross-système (GNU/Linux et Windows étaient utilisés pour le développement sans problème de compatibilité).

JDBC Lorsque nous avions dû choisir une méthode pour gérer notre base de données, nous nous sommes tournés vers JDBC, principalement par sa simplicité d'utilisation et par sa documentation complète.

JavaFX Bien que nous eussions commencé le développement de l'interface graphique avec *Swing*, nous avons rapidement basculé vers JavaFX, notamment pour disposer de meilleures options en termes de réactivité de l'interface, et car JavaFX offre une simplicité de développement plus intéressante.

3.2.2 Environnement de développement

IDE Le projet a été développé sur **Visual Studio Code** avec l'add-on *liveshare*, lequel nous a permis un processus de développement accéléré (l'add-on permettant

l'édition du code en simultané avec mise à jour en temps réel). À cela venait s'ajouter **Git** pour gérer le versionnage, lequel était synchronisé avec Github [6]. L'avancement du projet peut donc être suivi sur la page Github du projet.

Suivi de l'avancement Le suivi de l'avancement a été effectué avec **Jira** [7] avec un total de 4 *sprints* ayant chacun une durée comprise entre une et trois semaines. Les *sprints*, établis avant le commencement du projet, étaient les suivants :

— Network 1 (Discovery & UDP)
— Network 2 (Communication & TCP)
— Database (SQL)
— Interface utilisateur

Le premier *sprint* est celui qui nous a pris le plus de temps, car il fallait également mettre en place l'environnement et la structure initiale. À cause de la contrainte de temps, nous avons introduit une nouvelle catégorie de ticket dans *Jira* : *`Reporté à plus tard`*, qui correspond aux tickets étant trop long à implémenter et que nous avons dû reporter afin de rester dans les temps.

Important : *Afin de permettre l'évaluation du Jira et selon instructions de l'encadrant, aucun sprint n'a été fermé. Nous avons considéré un sprint comme terminé lorsque l'ensemble des tickets étaient indiqués* `Terminé` *ou* *`Reporté à plus tard`*. *Notez alors que le tableau de bord n'a pas été utilisé.*

3.3 Déploiement

Pour des raisons de polyvalence et de facilité de déploiement, nous avons décidé d'utiliser **Github Action** en lieu et place de *Jenkins* qui était recommandé pour ce projet [8]. Les deux systèmes ont des objectifs similaires, mais des méthodes différentes. Nos principaux usages de *GitHub Action* ont été l'automatisation des *builds* et le déploiement automatique des *nightly* et des *release* qui s'intègre directement à Github. Ainsi, notre page Github héberge les derniers *builds* téléchargeables par n'importe quel utilisateur.

La configuration de *Github Action* s'effectue dans un fichier YAML (cf. `.github/workflows/release.yml` sur Github). Nous avons configuré trois routines, toutes déclenchées par la présence d'un tag spécifique dans le message du commit.

6. Lien Github : `https://github.com/Enjmateo/distributed_chat_system`
7. Lien Jira : `https://team-1635954518853.atlassian.net`
8. En faisant ce choix, nous avons consulté nos professeurs qui ont exprimé leur accord.

3.3.1 Routine *Build*

Déclenchée par le tag `[build]` dans le message du commit, la routine prépare un environnement puis essaye de compiler le programme. La sortie de la routine informe du succès ou de l'échec de la compilation. Un exemple de sortie est fournis à la figure 11 (page 15). Au besoin, cette routine peut également être déclenchée manuellement depuis l'onglet *Actions*.

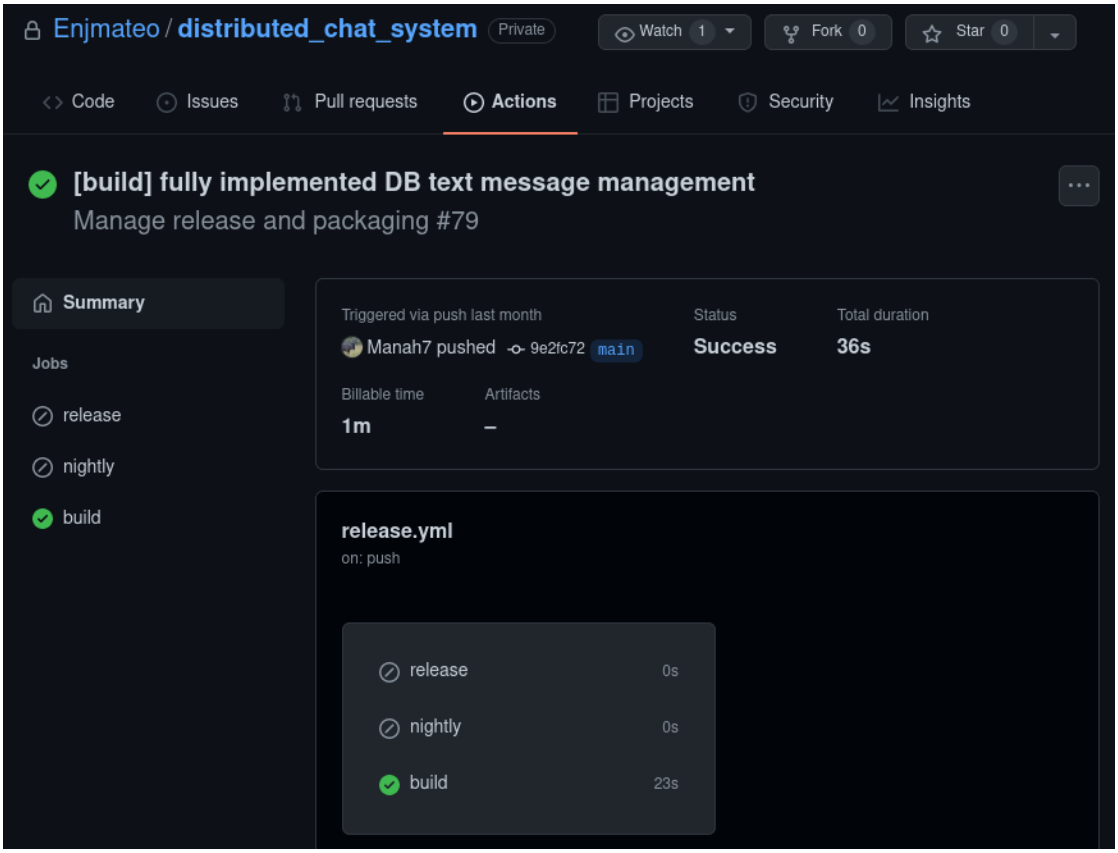

FIGURE 11 – Exemple de sortie d'une routine *build*

3.3.2 Routine *Nightly*

Déclenchée par le tag `[nightly]` dans le message du commit, la routine effectue une compilation à l'instar de la routine *Build*. Si la compilation réussie, la routine va ensuite déployer le fichier JAR produit dans une *pre-release* sur Github avec la mention "nightly". Cette procédure est utile dans le cadre du développement pour déployer un fichier JAR partageable facilement. Cependant, **les fichiers produits**

par ces routines n'ont pas pour objectif d'être déployés en production et sont réservés pour le développement.

3.3.3 Routine *Release*

Déclenchée par le tag `[release]` dans le message du commit, la routine effectue un travail très similaire à la routine *nightly*. La routine aboutie cependant à la publication d'une version stable sur la page *Release* du Github. Ces versions sont destinées à être utilisées par le client, elle n'est cependant pas immune aux bugs et seule la *release* la plus récente est recommandée.

3.4 Tests

Pour nous assurer du bon fonctionnement de notre programme avant la publication de *releases*, nous avons effectué une série de tests. Nous décrivons notre méthode dans cette section.

3.4.1 Environnement de test

Tests sur les ordinateurs de développement La majorité des tests, en particulier ceux effectués pendant le développement, ont été réalisés dans un environnement simulé de réseau d'entreprise. Pour cela, nous avons utilisé le logiciel ***ZeroTier*** qui permet d'émuler un réseau local à travers internet.

Tests en salle de TP Nous avons également testé notre logiciel sur les machines de TP INSA, car le réseau sur lequel elles se trouvent est typé d'un réseau d'entreprise et nous a permis de valider nos tests en conditions réelles. Cependant, l'environnement de développement n'étant pas disponible sur ces machines et ne souhaitant pas mélanger les configurations IDE, nous avons privilégié l'utilisation de *pre-release nightly* rapidement et simplement déployable sur ces machines. Cette configuration nous permettait également de testé la viabilité de notre package JAR.

3.4.2 Conclusion des tests

Dans les deux environnements, les tests se sont montrés concluants et les spécifications visées ont été atteintes. Nous avons notamment utilisé ODD afin d'échanger pendant la rédaction de ce rapport. Cependant, nous avons constaté que sur un réseau particulièrement instable, la connexion à la base de données présente des erreurs. Pour palier ce problème, nous avons ajouté une option au démarrage permettant d'utiliser l'application sans l'historique des messages. Dans ce cas, le fichier de configuration n'est pas requis. Cette option nous a également facilité les tests sans imposer une connexion systématique à la base de donnée.

INSA Toulouse
135, Avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

Université Fédérale Toulouse Midi-Pyrénées

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE
MINISTÈRE
DE L'ÉDUCATION NATIONALE,
DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE