INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Célia CHAUZY
Maïa MANGIN
Mateo MILLE

Rapport de projet POO

Système de clavardage distribué interactif multi-utilisateurs temps réel

Janvier 2022 - promotion 56

Célia CHAUZY
Maïa MANGIN
Mateo MILLE

Rapport de projet POO

Système de clavardage distribué interactif multi-utilisateurs temps réel

Janvier 2022 - promotion 56

SOMMAIRE

INTRODUCTION

Dans le cadre de l'UF de COO et POO, nous avons été amenés à travailler sur un projet de système de clavardage distribué interactif multi-utilisateur temps réel. Pour ce faire, nous avons dans un premier temps abordé la conception de cette application. Tous les diagrammes réalisés sont disponibles sur notre GitHub. Nous avons dans un second temps travaillé sur l'implémentation de ce système.

Ainsi, nous présenterons tout d'abord la conception de notre application. Ensuite, nous exposerons l'architecture du système ainsi que les choix technologiques que nous avons pris (environnement de développement, implémentation des bases de données, implémentation de l'interface, architecture de communication, utilisation de Jenkins). Par la suite, nous expliquerons les différents tests que nous avons mis en place. Puis nous détaillerons la procédure d'installation et de déploiement de notre application, avant d'enfin proposer un manuel d'utilisation.

I. CONCEPTION ET DIAGRAMMES RÉALISÉS

Tous nos diagrammes sont disponibles sur notre dépôt git, en version initiale ainsi qu'en version modifiée à la fin du projet :
https://github.com/maiamn/Projet_Clavardage/tree/main/Conception
Dans cette partie, nous allons exposer uniquement le diagramme de cas d'utilisation et le digramme de classe.

1. Diagramme des cas d'utilisation

La partie COO, débutée avant l'implémentation de notre application, a été indispensable afin de nous questionner sur l'architecture de notre système et a ainsi été un gain de temps par la suite, même si nos classes ont beaucoup évolué.

Nous avons étudié les spécifications du cahier des charges afin de les retranscrire dans notre diagramme de cas d'utilisation, visible sur la figure 1 ci-dessous. Les trois acteurs représentés sont localUser qui est donc l'utilisateur qui utilise le système de clavardage, Interlocuteur qui représente un autre utilisateur en ligne avec lequel localUser pourra échanger, et enfin la base de données.

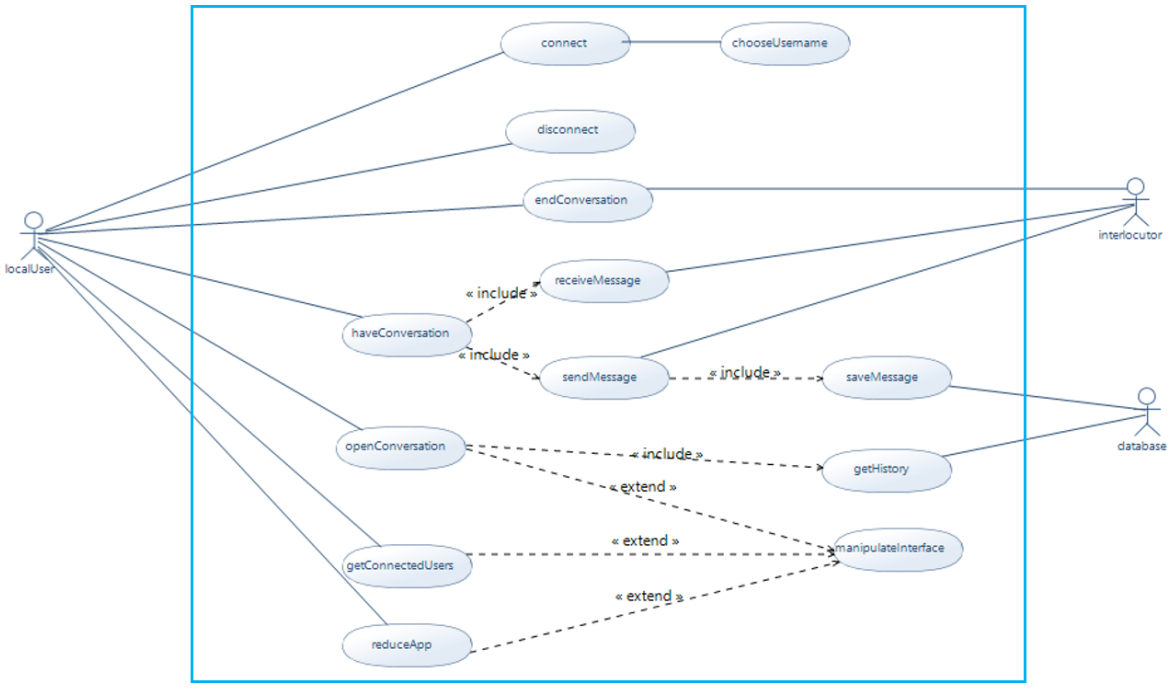

Figure 1 : diagramme des cas d'utilisation

2. Digramme de classes

Le système a été conçu de la manière suivante : il est découpé en trois packages principaux : databases, GUI, et network. Chacun de ces packages contient plusieurs classes avec des fonctionnalités spécifiques, ainsi qu'une classe qui opère en tant que manager.
Ainsi, notre quatrième package (Manager) permet de faire le lien entre nos différentes autres classes, plus spécifiques, en échangeant uniquement avec les managers des autres groupes.
Ceci est illustré sur la figure 2 ci-après, représentant notre diagramme de classes. Seules les principales classes sont illustrées (toutes les classes nécessaires pour l'interface n'y figurent

pas) et les fonctions et attributs ne sont pas représentés dans un souci de lisibilité. Le diagramme de classes complet est disponible sur le git du projet.
Cette architecture est intéressante car elle nous a permis de pouvoir modifier facilement nos différentes classes au cours de l'implémentation sans que cela n'impacte le reste du système.

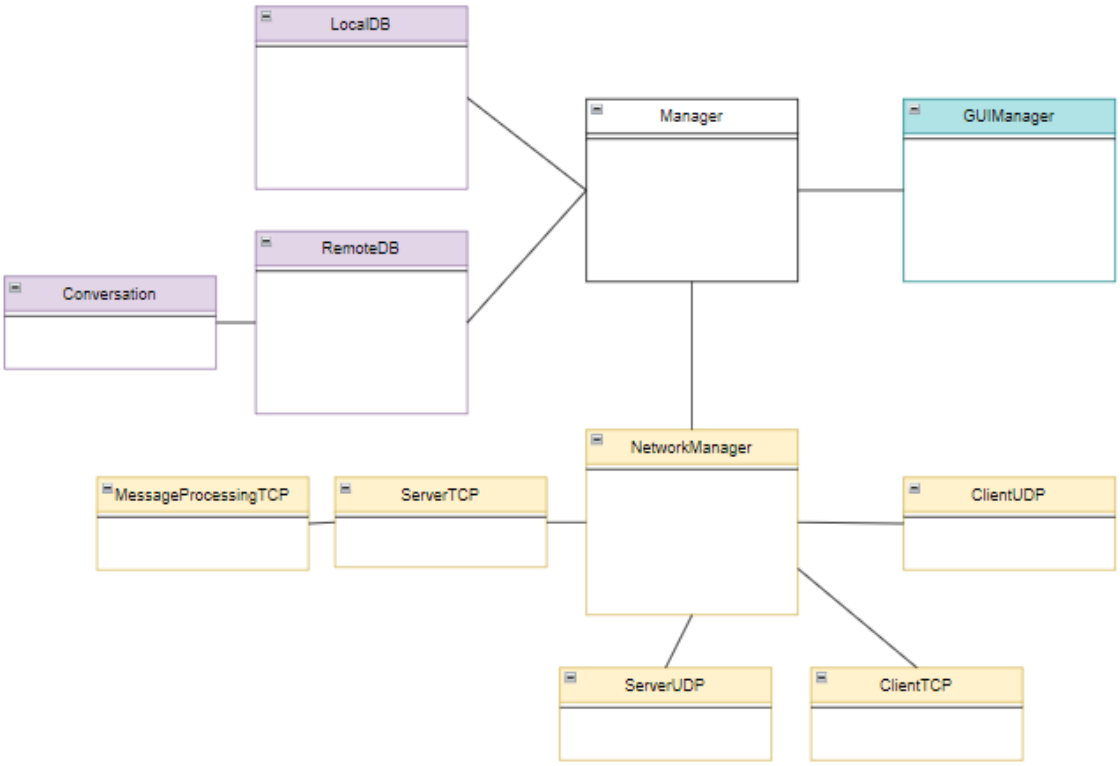

Figure 2 : diagramme de classes

II. ARCHITECTURE DU SYSTÈME ET CHOIX TECHNOLOGIQUES

1. Environnement de développement

Ce projet a été développé sous Eclipse. Nous avons utilisé la version 11 de Java, qui est compatible avec les machines de l'INSA ainsi que Jenkins.
Le projet réalisé est un projet Maven.

2. Implémentation des bases de données

Nous avons implémenté deux bases de données différentes : une locale et une distante.

a. Base de données pseudo/adresse IP

Cette base de données contient les correspondances entre les pseudos choisis par les utilisateurs en ligne et l'adresse IP de la machine avec laquelle ils sont connectés. Elle est implémentée localement, dans une optique de décentralisation.
La table créée se nomme *usernameToIP* et est représentée sur la figure 3. Les deux premières colonnes permettent de stocker la correspondance entre l'adresse IP des utilisateurs et le pseudo choisi. Ensuite, la colonne *isConnected* permet de savoir si l'utilisateur est en ligne ou non. Ainsi, nous pouvons afficher l'historique des conversations avec n'importe quel utilisateur

avec lequel nous avons échangé des messages, que celui-ci soit en ligne ou non, en gardant en mémoire son dernier pseudo afin de rendre l'affichage compréhensible par l'utilisateur.
Enfin, la dernière colonne *lastAccess*, permet de savoir la date à laquelle nous avons accédé pour la dernière fois à la conversation avec un interlocuteur donné. Ainsi, nous sommes capables de dire à l'utilisateur s'il a des messages non lus.

username	ip	isConnected	lastAccess
user1	192.168.15.1	1	« 08-01-2022 12:51:19 »
user2	192.168.15.2	0	« 08-01-2022 12:53:19 »
user3	192.168.15.3	1	« 01-01-2022 02:51:19 »

Figure 3 : représentation de la table locale usernameToIP

À chaque connexion, le pseudo choisi est envoyé à tous les utilisateurs connectés, via un broadcast. En réponse, chacun renvoie son propre pseudo, nous permettant ainsi de mettre à jour notre base de données. Deux situations peuvent avoir lieu :

- ✓ L'adresse IP est déjà présente dans la base de données, on met donc à jour le pseudo qui y est associé et on indique que l'utilisateur est connecté.
- ✓ L'adresse IP n'est pas encore présente dans la base de données, nous l'ajoutons donc.

Pour implémenter cette base de données, nous avons utilisé SQLite. En effet, cela nous permet de créer les tables locales de chaque utilisateur simplement. Pour exporter notre projet, la seule démarche nécessaire est de connaître le chemin du fichier .jar SQLite, que nous fournissons avec le .jar de notre application.

b. Base de données historique

Cette base de données distante contient l'ensemble des échanges entre les différents utilisateurs. La table créée se nomme *History* et est représentée sur la figure 4 ci-dessous. À chaque réception d'un message, celui-ci est ajouté à la base de données avec les informations suivantes : adresse IP de l'émetteur, adresse IP du récepteur, contenu du message, date de réception. Toutes ces informations sont stockées sous forme de chaînes de caractères.

Sender	Receiver	Message	Date
192.168.15.1	192.168.15.2	Hello user2	08-01-2022 12:51:19
192.168.15.2	192.168.15.3	Hello user3	08-01-2022 12:53:47
192.168.15.1	192.168.15.3	Hello user3!	08-01-2022 12:54:23

Figure 4 : représentation de la table History

Pour implémenter cette base de données, nous avons utilisé MySQL. En effet, le serveur contenant la table est distant, la démarche pour s'y connecter est donc toujours la même. Pour initialiser la connexion, nous avons cependant besoin de connaître le chemin du connecteur MySQL, fourni dans le .jar de notre application.

3. Implémentation de l'interface

Pour notre interface, nous avons choisi d'utiliser Java SWING. La gestion de notre interface est réalisée par une classe appelée *GUIManager* qui permet de faire le lien entre toutes les classes appartenant au package *GUI*. En effet, le fonctionnement de notre application repose

sur l'existence de diverses classes qui correspondent chacune à une interface permettant à l'utilisateur de réaliser plusieurs actions.

- *AuthentificationGUI* : Interface d'authentification
- *ConnectionGUI* : Interface de connexion
- *HomePageGUI* : Interface de page d'accueil
- *ConnectedUsersGUI* : Interface permettant de visualiser la liste des utilisateurs connectés
- *SendMessageGUI* : Interface permettant de choisir l'utilisateur (connecté) avec lequel on veut communiquer
- *ChatGUI* : Interface permettant de visualiser l'historique d'une conversation et d'envoyer un message
- *NewMessagesGUI* : Interface permettant de voir s'il existe des messages non lus
- *HistoryGUI* : Interface permettant de voir l'historique des conversations (pour des utilisateurs connectés ou déconnectés)
- *ChangeUsernameGUI* : Interface permettant de changer de nom d'utilisateur
- *DisconnectionGUI* : Interface de déconnexion

4. Architecture de la communication

En ce qui concerne la communication, elle est gérée exclusivement par le package *Network*. Nous utilisons le protocole UDP pour les broadcasts : demande de la disponibilité d'un pseudo, connexion, déconnexion… L'échange entre les utilisateurs se fait avec le protocole TCP. En effet, cela nous permet de garantir le transfert de ces messages de façon fiable.
Nous avons décidé de définir un format de messages précis afin de faciliter leur traitement dans la classe NetworkManager, suivant la situation : notification d'une connexion, réception d'un message, etc. Ce format est illustré par l'annexe 1, page I.

III. PROCÉDURES D'ÉVALUATION ET DE TESTS

1. Mise en place de test JUnit

Afin de tester le bon fonctionnement des bases de données que nous avons créées, nous avons implémenté des tests JUnit, correspondant aux classes LocalDBTest et RemoteDBTest.
À l'intérieur de ces fichiers, nous avons créé des tests pour chacune des fonctions existant dans les classes LocalDB et RemoteDB. Tous les tests passent, nous pouvons donc nous assurer du bon fonctionnement de nos bases de données.

2. Test de la communication

Au sujet des tests concernant la communication, nous n'avons pas implémenté de tests JUnit. En effet, créer ce type de tests pour tester des protocoles UDP et TCP nous paraissait trop complexe à mettre en place, à cause notamment de la création de serveurs et de clients. Cependant, nous avons toutefois testé au fur et à mesure chacune des fonctions que nous avons codées dans le package Network de notre projet. Ainsi, nous avons utilisé plusieurs ordinateurs dans une salle de l'INSA pour tester nos fonctions de broadcast, et nos échanges de messages, entre autres.

IV. PROCESSUS DE DÉVELOPPEMENT AUTOMATISÉ

1. Utilisation d'un dépôt git

Pour notre projet, nous avons travaillé avec GitHub. Notre dépôt est disponible à l'adresse suivante : https://github.com/maiamn/Projet_Clavardage

L'avantage est de pouvoir travailler de manière collaborative sur ce projet que nous avons réalisé à trois étudiants. De plus, il est possible de relier notre git avec Eclipse afin de pouvoir réaliser des push ou pull directement depuis Eclipse.

2. Pipelines et jobs Jenkins

Sur Jenkins, nous avons créé deux jobs. Le premier permet de faire un build automatique du projet lorsqu'un changement a lieu sur le git. Le second permet de réaliser le déploiement de ce projet. Ensuite, nous avons créé un pipeline de ces deux jobs, comme illustré sur la figure 5 ci-dessous. Le déploiement du projet a lieu uniquement si le build automatique a réussi.

Remarque : les tests effectués pour notre base de données distante ne passent pas lorsqu'ils sont effectués depuis Jenkins car il ne parvient pas à se connecter au serveur distant. Nous avons ignoré ce problème car les tests sont validés lorsqu'ils sont lancés depuis Eclipse.

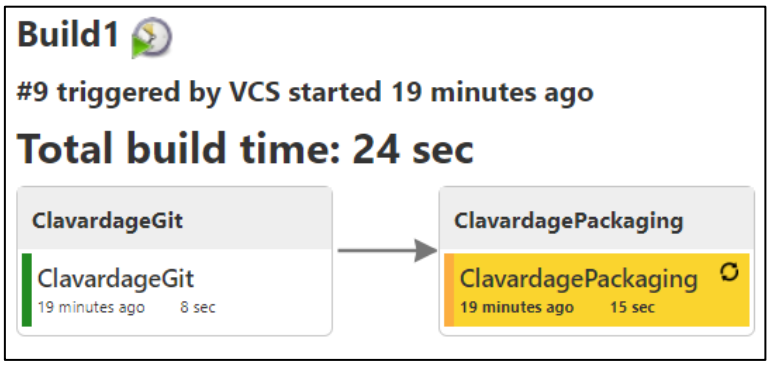

Figure 5 : pipeline du projet sous Jenkins (Delivery Pipeline View)

3. Application de la méthode Agile grâce à Jira

Dès la conception du projet, nous avons utilisé Jira afin de nous fixer des sprints pour chaque semaine.

Cela nous a permis de ne pas nous éparpiller, et de nous focaliser uniquement sur les tickets présents dans le sprint correspondant à la semaine en cours. Cependant, nous avons rencontré quelques difficultés. En effet, comme il s'agissait de la première fois que nous travaillions sur un projet aussi poussé, nous avons eu du mal à estimer la durée que les tâches allaient prendre, lorsque nous avons appliqué la méthode Poker pour les définir. Certaines fois, nous avons donc surestimé la durée de certains tickets et avons eu besoin d'en rajouter dans le sprint au cours de la semaine, alors que d'autres fois nous avons sous-évalué la quantité de travail nécessaire pour un ticket et nous avons dû le continuer au sprint suivant.

V. PROCÉDURE D'INSTALLATION ET DE DÉPLOIEMENT

Afin d'installer notre application, il suffit de télécharger le fichier exécutable jar intitulé *messengIR.jar* présent sur notre git à cette adresse :
https://github.com/maiamn/Projet_Clavardage/blob/main/Executable

Ce fichier contient toutes les librairies nécessaires au bon fonctionnement de l'application, et n'excède pas les 50Mo, comme précisé dans le cahier des charges. Une version de java est nécessaire pour pouvoir lancer l'exécutable.
Remarque : si l'application est exécutée sur une machine autre que celles de l'INSA, un VPN sera indispensable afin de pouvoir accéder à la base de données distante contenant l'historique des messages.

VI. MANUEL D'UTILISATION SIMPLIFIÉ

Pour lancer notre application, il suffit de double cliquer sur le fichier messenger.jar sous Windows, ou de l'exécuter depuis un terminal avec la commande suivante :

```
java -jar messengIR.jar
```

La fenêtre d'authentification s'ouvre alors automatiquement, comme suit.

1. Phase d'authentification

Figure 6 : fenêtre d'authentification

La première étape consiste à choisir le pseudo sous lequel l'utilisateur va être reconnu par ses interlocuteurs. Dans le cas où le pseudo n'est pas valide (i.e. s'il contient un caractère spécial, moins de 1 caractère ou plus de 30 ou s'il est déjà utilisé), une fenêtre pop-up s'ouvre avec un message d'alerte, comme illustré par la figure 7 ci-dessous. L'utilisateur a alors la possibilité de choisir un autre pseudo.

Figure 7 : fenêtre pop-up en cas de pseudo non valide

2. Phase de connexion

Une fois que l'utilisateur a choisi son pseudo, il a la possibilité de se connecter avec le pseudo choisi ou de revenir à l'interface d'authentification pour choisir un autre pseudo (dans le cas où il aurait commis une erreur par exemple).

Figure 8 : validation du pseudo

3. Page d'accueil

Sur la page d'accueil, l'utilisateur peut alors choisir le service auquel il veut accéder. Ces possibilités sont listées grâce à différents boutons sur lesquels il peut cliquer.

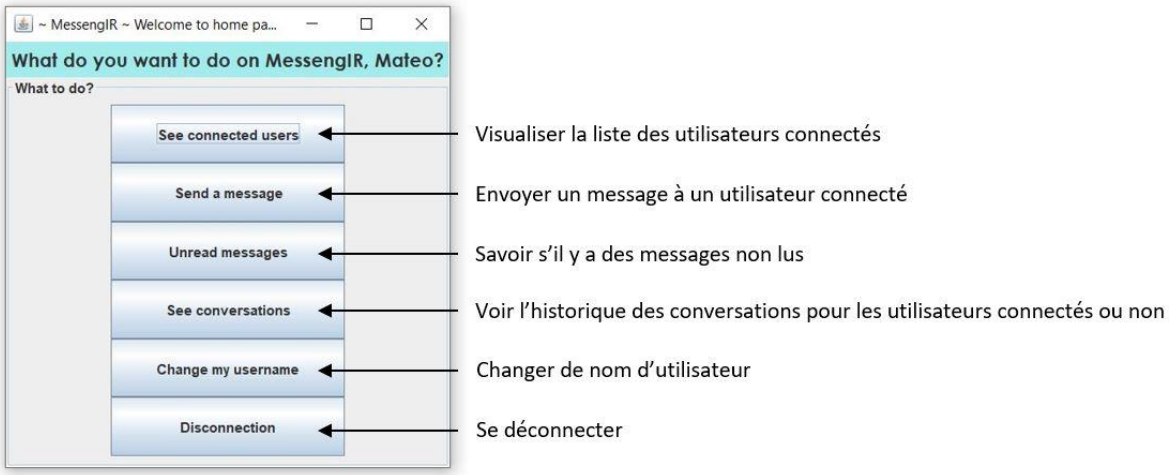

Figure 9 : page d'accueil

CONCLUSION

En somme, nous avons conçu et implémenté une application de clavardage distribué interactif multi-utilisateur temps réel. Avec notre système, un utilisateur peut ainsi :

- ✓ Se connecter en choisissant un pseudo non utilisé,
- ✓ Changer ce pseudo à tout instant et en informer automatiquement les autres utilisateurs tout en conservant l'intégralité de ses conversations passées,
- ✓ Accéder à l'historique des conversations, même si l'interlocuteur n'est pas en ligne actuellement,
- ✓ Voir le nom des utilisateurs connectés actuellement,
- ✓ Voir s'il existe des messages non lus et accéder au nom de l'interlocuteur concerné si tel est le cas,
- ✓ Envoyer et recevoir des messages textuels en temps réel,
- ✓ Se déconnecter.

Ainsi, l'application que nous avons conçue est aujourd'hui utilisable pour tchatter sur un même réseau local. Dans le futur, nous pourrons ajouter les fonctionnalités suivantes : pouvoir échanger des données non textuelles, et pouvoir communiquer sur un réseau distant. Grâce à l'architecture de notre code, seules les classes du package Network seraient concernées par ce dernier changement, ce qui le faciliterait.

ANNEXE : FORMATS MESSAGES PROJET CLAVARDAGE

Format général : `TYPE | username | content`

Émission (Client)

Gestion des usernames disponibles

1. Demander si le username est disponible :
   [ClientUDP] `USERNAME_BRDCST | username | @IP`
   - Username = celui qu'on veut utiliser
   - @IP = la nôtre pour qu'on puisse nous répondre si jamais le username est déjà utilisé

2. Réponse en TCP (si le username est déjà le nôtre) :
   [ClientTCP] `USERNAME_BRDCST | username | null`

Gestion des connexions

1. Notifier les autres qu'on est connecté :
   [ClientUDP] `USERNAME_CONNECTED | username | @IP`
   - Username = le nôtre
   - @IP = la nôtre pour que l'interlocuteur puisse l'ajouter à sa BDD locale

Gestion des déconnexions

1. Notifier les autres qu'on est déconnecté :
   [ClientUDP] `USERNAME_DISCONNECT | username | null`
   - Username = le nôtre

Demande des noms des utilisateurs connectés

1. Demander aux autres leur username et leur @IP :
   [ClientUDP] `GET_USERNAMES | username | null`
   - Username = le nôtre
   - Null : car on a déjà envoyé notre @IP quand on s'est connecté

2. Notifier notre username et notre @IP à celui qui les a demandés :
   [ClientTCP] `GET_USERNAMES | username | @IP`
   - Username = le nôtre
   - @IP = la nôtre pour que l'utilisateur puisse l'ajouter à sa BDD locale

Changement de nom d'utilisateur

1. Notifier les autres qu'on a changé de nom d'utilisateur :
   [ClientUDP] `USERNAME_CHANGED | new_username | @IP`
   - New_username = notre nouveau pseudo
   - @IP = la nôtre pour que l'interlocuteur puisse modifier sa BDD locale

Échange d'un message

1. Envoi d'un message :
   [ClientTCP] `MESSAGE | username | content`
   - Username = le nôtre
   - Content = le message qu'on veut envoyer

Réception (Serveur)

Gestion des usernames disponibles

1. Demande d'utilisation d'un username :
   [ServeurUDP] `USERNAME_BRDCST | username | @IP`
   - Username = celui que l'interlocuteur veut utiliser
   - @IP = la sienne

   On envoie le 2 d'émission

2. Username déjà utilisé :
   [ServeurTCP] `USERNAME_BRDCST | username | null`
   ➔ `Set usernameunavailable`

Gestion des connexions

1. Notification d'un utilisateur connecté :
   [ServeurUDP] `USERNAME_CONNECTED | username | @IP`
   - Username = le nouveau connecté
   - @IP = la sienne

   ➔ On l'ajoute à la LocalDB

Gestion des déconnexions

1. Notification d'un utilisateur déconnecté :
   [ServeurUDP] `USERNAME_DISCONNECT | username | null`
   - Username = le sien

   ➔ On l'enlève de la LocalDB

Demande des noms des utilisateurs connectés

1. Réception d'une demande de notre username et notre @IP depuis un broadcast :
   [ServeurUDP] `GET_USERNAMES | username | null`
   - Username = la personne qui demande nos infos

   ➔ On répond avec l'émission 2
2. Réception d'un username et d'une @IP demandés :
   [ServeurTCP] `GET_USERNAMES | username | @IP`
   - Username = le sien
   - @IP = la sienne

   ➔ On peut l'ajouter à la BDD locale

Changement de nom d'utilisateur

1. Notifier les autres qu'on a changé de nom d'utilisateur :
   [ServeurUDP] `USERNAME_CHANGED | new_username | @IP`
   ➔ On peut modifier la ligne correspondante dans notre BDD locale

Échange d'un message

1. Réception d'un message :
   [ServeurTCP] `MESSAGE | username | content`
   - Username = celui de l'interlocuteur
   - Content = le message qu'il nous a envoyé

   ➔ On peut répondre en revenant en émission 1

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