



INSA

INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
TOULOUSE

Rapport Projet POO Chat System

Mengxia SHI

Yifan WANG

4IR - Groupe B

Février 2021

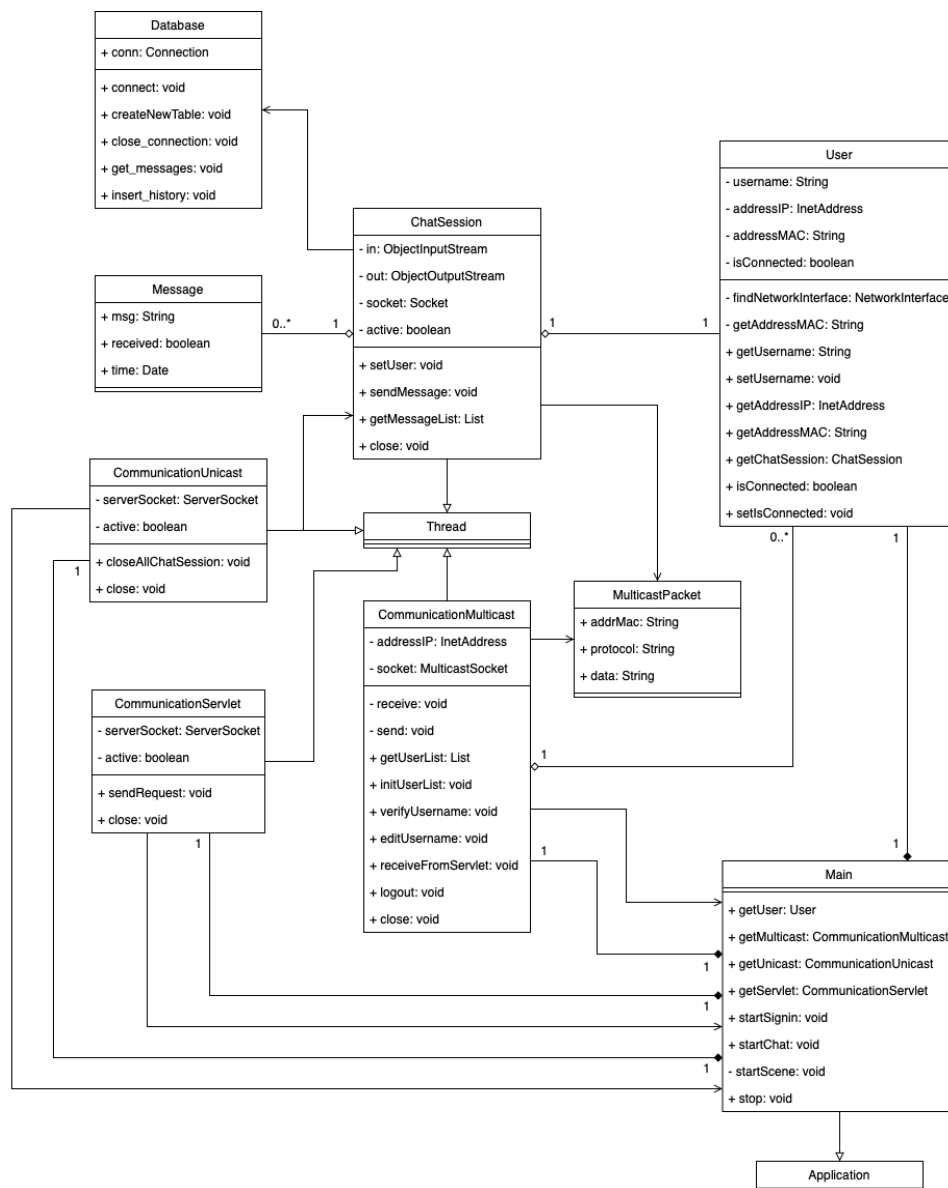
Sommaire

Partie 1: Architecture du système	3
I/ Architecture de la partie communication local	3
3) Communication Unicast	4
4) Communication Multicast	4
5) Multicast Packet	6
6) Chat Session	6
7) Database	6
8) Les contrôleurs	7
II/ Architecture de la partie JEE	7
Partie 2 : Choix techniques d'implémentation	8
I/ MVC	8
II/ JavaFX	9
Partie 3 : Test validation	10
Partie 4 : Manuel d'installation et d'utilisation	14
I/ Manuel d'installation	14
II/ Manuel d'utilisation	14

Partie 1: Architecture du système

Tout au long du projet, nous avons utilisé le modèle MVC (modèle, vue et contrôleur) pour développer notre système de chat sous l'IDE IntelliJ. Le modèle MVC nous permet d'avoir une structure flexible et rend le processus de développement plus efficace. En particulier, quand on est plusieurs à travailler sur le même projet. Nous avons choisi d'utiliser IntelliJ en tant qu'IDE car il propose plusieurs fonctionnalités intéressantes quand on travaille avec JavaFX. Ce système de chat est conçu totalement décentralisé. Les modèles représentent les différentes entités de notre système, comme les utilisateurs ou encore les différents canaux de communication. Les vues représentent nos interfaces de connexion et de chat. Les contrôleurs permettent le lien entre les modèles et les contrôleurs, entre autres la mise à jour des modèles en fonction des actions de l'utilisateur.

1/ Architecture de la partie communication local



La figure ci-dessus montre le diagramme de classes du système de chat de base (sans les fonctions Servlet), nous avons ici principalement 8 classes pour réaliser les fonctionnalités de notre système:

1) User

La classe User est la classe essentielle du système. Elle représente un utilisateur via les attributs suivants :

```
private String username;  
private InetAddress addressIP;  
private String addressMAC;  
private ChatSession chatSession;  
private boolean isConnected;
```

L'objet chatSession permettra la communication avec cet utilisateur et les méthodes présentes dans cette classe sont de simples getters et setters pour obtenir les informations de l'utilisateur.

2) Message

```
final public String msg;  
final public boolean received;  
final public Date time;
```

La classe Message est l'objet essentiel pour la communication. Elle est composée du message, de sa date d'envoi et d'un booléen qui indique si le message a été reçu ou envoyé.

3) Communication Unicast

Cette classe est très importante pour la communication. Elle permet la partie communication TCP dans le système. Elle a 3 attributs:

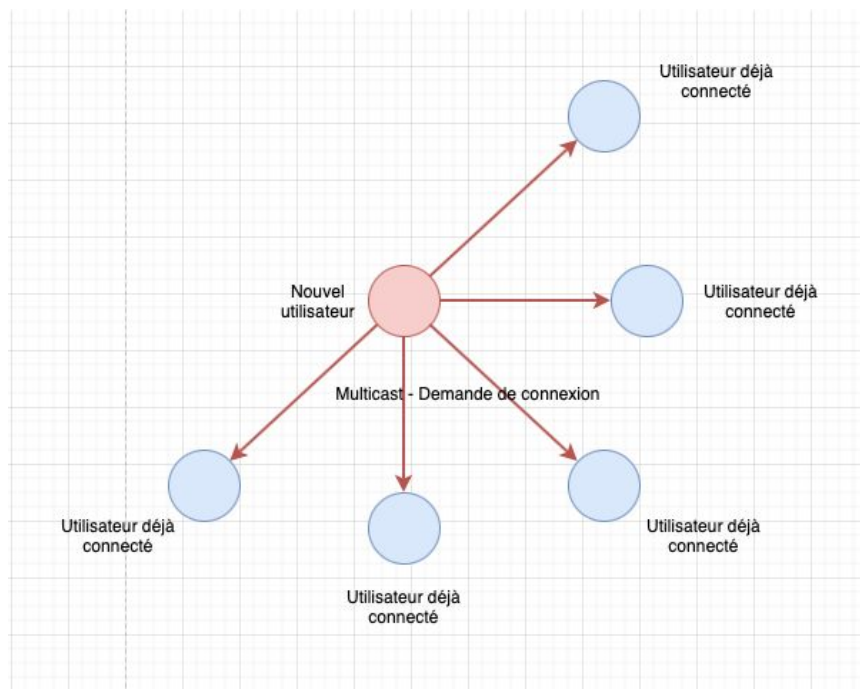
```
public static int port = 51000; Port par défaut  
private ServerSocket serverSocket; Serveur socket qui permet de recevoir les demandes de création de sessions de chat  
private volatile boolean active = true; Booléen qui représente l'état du serveur socket
```

Cette classe est une extension de Thread, chaque fois qu'un nouvel utilisateur essaye de se connecter avec le serveur, il y aura un thread de cette classe créé et qui lui permettra de réaliser la communication en mode TCP.

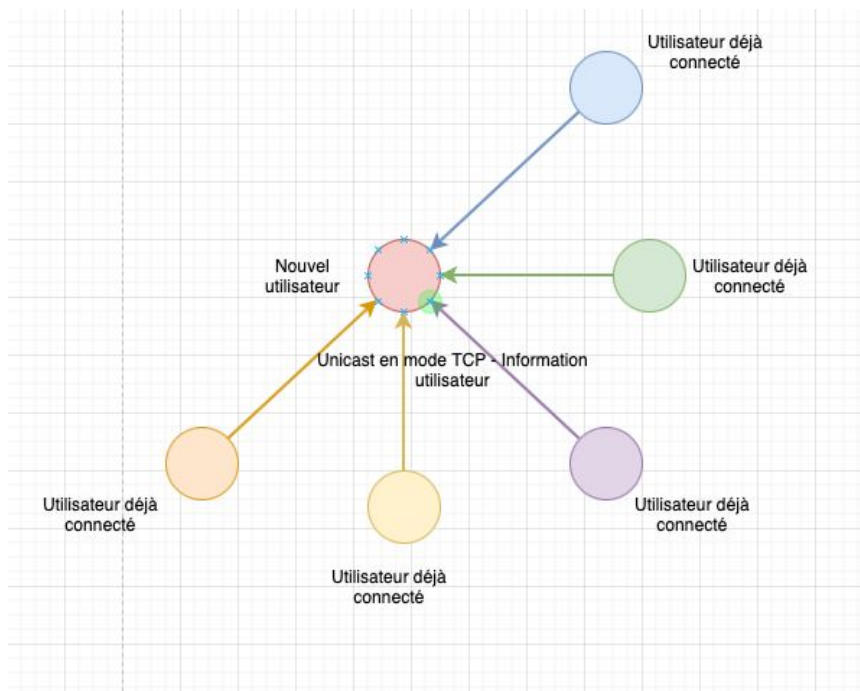
4) Communication Multicast

La classe Multicast est la classe qui permet de réaliser les connexions entre les différents utilisateurs de façon décentralisée. On a utilisé le multicast au lieu de broadcast car ces deux techniques fonctionnent presque de la même façon et que seulement les utilisateurs connectés reçoivent les paquets multicast mais pas toutes les machines du réseau local.

La première phase de la connexion consiste à envoyer un paquet multicast contenant les informations du nouvel utilisateur à tous les utilisateurs déjà connectés, comme le montre ce schéma :



Deuxième phase de la connexion, les utilisateurs qui ont reçu le paquet multicast envoient leurs informations à ce nouvel utilisateur. Si le pseudonyme choisi par le nouvel utilisateur n'est pas déjà utilisé, tout le monde va alors mettre sa liste d'utilisateurs à jour. Comme le montre le schéma ci-dessous :



L'utilisateur crée quant à lui sa liste d'utilisateurs et se connecte si son pseudonyme n'est pas déjà utilisé.

5) Multicast Packet

Multicast Packet définit la forme des paquets communiqués en mode multicast, cela nous permet de traiter facilement les communications multicast. On peut extraire les données plus facilement quand les données de paquets sont standardisées.

```
public final String addrMac;  
public final String protocol;  
public final String data;
```

6) Chat Session

ChatSession est la classe pour la communication entre les utilisateurs. Cette classe tient à jour une liste des messages échangés et communique tout changement aux listeners présents dans les vues. Bien sûr, cette classe est également une extension de Thread, toutes les sessions de chat sont des threads indépendants pour une exécution efficace.

7) Database

Cette classe permet de stocker les messages grâce à SQLite, elle génère l'historique de messages communiqués entre les utilisateurs. Elle contient les fonctionnalités suivantes:

```
public static void connect()
```

Connexion avec la base de données, l'historique est sauvegardé localement. Si la base de données n'existe pas, alors une nouvelle base de données sera créée automatiquement.

```
public static void createNewTable()
```

La fonction createNewTable est appelée automatiquement quand on essaie de se connecter avec la base de données, elle crée la table suivante:

m_id integer PRIMARY KEY AUTOINCREMENT
mac text NOT NULL
date text NOT NULL
message text
received boolean

```
public static void get_messages(String MAC , ListController<Message> messageList)
```

Permet d'obtenir l'historique d'une session de chat à partir de l'adresse MAC et de les mettre dans la liste des messages observée par la vue qui gère l'interface du chat.

```
public static void insert_history(String MAC, Message message)
```

Cette fonction permet d'insérer les messages dans l'historique, elle est donc appelée quand l'utilisateur envoie ou bien reçoit des messages.

8) Les contrôleurs

List controller est une classe pour faciliter l'implémentation et la réalisation des observers sur les listes avec JavaFX. Les listes de messages et la liste des utilisateurs connectés utilisent toutes les deux ce contrôleur vu que l'interface graphique a besoin d'être renouvelée chaque fois qu'un changement est réalisé.

ChatController et SigninController réalisent respectivement les modifications du chat et de la page de connexion lorsque les objets observés sont modifiés.

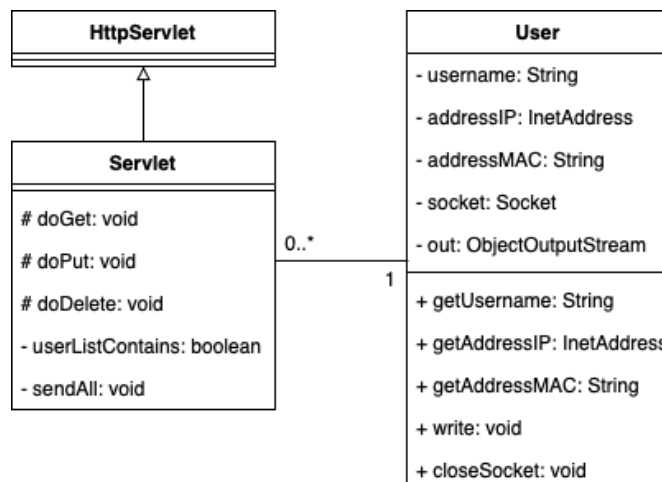
II/ Architecture de la partie JEE

Comme nous l'avons vu dans la partie précédente, nous utilisons le multicast pour découvrir les utilisateurs connectés à l'application. Afin que les utilisateurs hors du réseau privé de l'entreprise puissent également utiliser l'application, nous avons mis en place un servlet qui va jouer le rôle de "pont" avec le réseau public.

Pour les utilisateurs présents dans le réseau public, le fonctionnement ne change pas. Pour les utilisateurs hors du réseau et qui ne peuvent donc pas utiliser le multicast, ils communiqueront avec l'hôte sur lequel est lancé le servlet. Ce dernier est accessible depuis Internet et peut également communiquer avec les utilisateurs du réseau privé car il tient à jour une liste des utilisateurs connectés à l'application.

Une fois les utilisateurs connectés, ils pourront communiquer normalement entre eux via le protocole TCP.

Ci-dessous le diagramme UML qui représente l'architecture de la partie JEE :



La classe User est globalement la même que la classe User de l'application. A la différence, qu'elle a un socket qui permettra au servlet d'envoyer des paquets aux utilisateurs quand le multicast ne peut pas être utilisé.

Ensuite nous avons la classe Servlet qui étend la classe HttpServlet. Elle correspond à la route {host}/Servlet et implémente trois méthodes:

- GET: retourne la liste des utilisateurs connectés (on s'en servira uniquement pour le débogage)
- PUT: ajoute un nouvel utilisateur à la liste des utilisateurs connectés et envoie l'information à tous membres déjà présents dans la liste
- DELETE: retire un utilisateur de la liste des utilisateurs connectés et envoie l'information à tous les autres utilisateurs présents dans la liste

Grâce à ces trois routes, nous pouvons effectuer les mêmes actions que les utilisateurs du réseau local utilisent en multicast et ainsi permettre à tous d'utiliser l'application de chat.

Partie 2 : Choix techniques d'implémentation

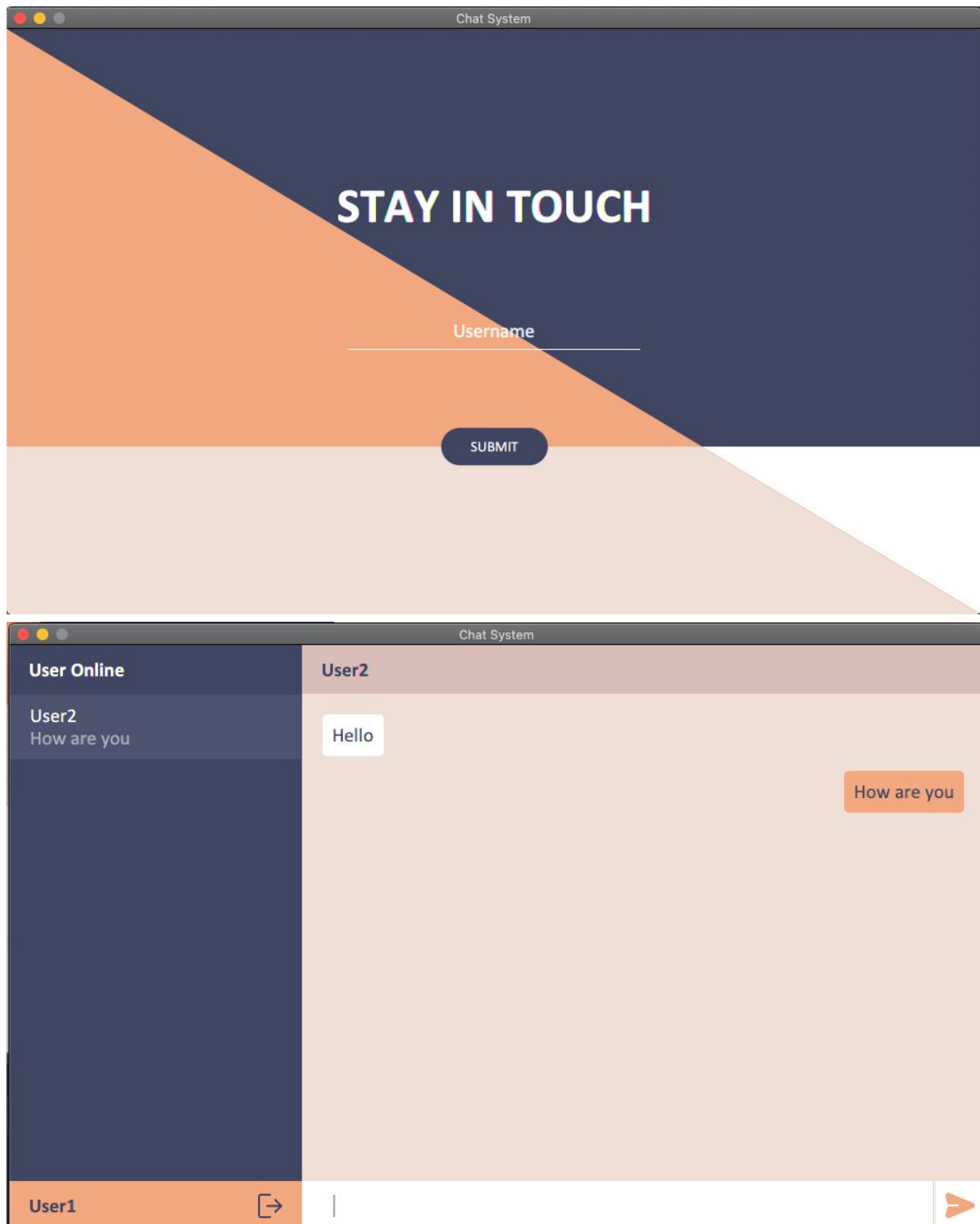
I/ MVC

On a choisi ce modèle d'architecture pour les avantages suivants :

1. Une conception claire et efficace grâce à la séparation des données de la vue et du contrôleur
2. Un gain de temps pour la maintenance et l'évolution de l'application
3. Une plus grande souplesse pour organiser le développement de l'application entre les différents développeurs (indépendance des données, de l'affichage et des actions)

II/ JavaFX

On a choisi JavaFX au lieu de Swing tout simplement parce qu'il nous permet de créer des interfaces graphiques plus modernes, plus facilement et plus efficacement. Ci-dessous les deux fenêtres conçues :



III/ SQLite

Il nous est demandé de tenir à jour une base de données des messages échangés. Cette base de données sera donc très simple et ne nécessite pas de requêtes complexes. Comme notre système est décentralisé, nous avons utilisé SQLite pour créer une base de données locale pour chaque utilisateur. SQLite est donc un choix convenable, il est léger et facile à implémenter.

Partie 3 : Test validation

Ci-dessous, nous avons passé en revue les fonctionnalités relatives à l'agent et les exigences opérationnelles exprimées dans le cahier des charges tout en précisant en quoi elles sont respectées ou non.

Le système doit pouvoir être déployé sur un poste de travail fonctionnant sur le système d'exploitation Windows : Respecté, JavaFX 11 est compatible avec Windows. Bien évidemment, il faudra que Java soit installé sur le poste, la version 10 au minimum.

Le système doit pouvoir être déployé sur un poste de travail fonctionnant sur le système d'exploitation Linux : Respecté, idem JavaFX 11 est compatible avec Linux.

Le système doit pouvoir être déployé sur un poste de travail fonctionnant sur le système d'exploitation OS X : Respecté, idem JavaFX 11 est compatible avec OS X.

Le système doit pouvoir être déployé sur un poste de travail fonctionnant sur le système d'exploitation Android : Partiellement respecté, JavaFX peut fonctionner sur Android grâce à JavaFXPorts (<https://gluonhq.com/products/mobile/javafxports/>) mais nous n'avons pas vérifié le bon fonctionnement de l'application avec cet outil.

Le déploiement du système devra se limiter à la copie sur le poste de travail d'une série de fichiers et la création d'un raccourci pour l'utilisateur : Respecté, comme nous le verrons dans la partie "Manuel d'installation", il suffit de copier l'archive JAR sur le poste pour commencer à utiliser l'application.

La taille globale de l'ensemble des ressources composant le système ne devra pas excéder 50 Méga-Octets sous une forme non compressée quel que soit le système d'exploitation sur lequel il est déployé : Partiellement respecté, une fois l'archive JAR décompressée, on obtient une taille d'environ 55Mo. Comme nous avons créé une unique archive JAR qui fonctionne sous tous les systèmes d'exploitation, notre archive est plus lourde. Nous aurions pu faire une archive pour chaque OS et ainsi nous aurions respecté cette contrainte, mais nous aurions perdu en facilité de déploiement. L'archive JAR tant qu'à elle fait 22,2Mo et n'a pas besoin d'être décompressée pour utiliser l'application.

Le système doit permettre à l'utilisateur de choisir un pseudonyme avec lequel il sera reconnu dans ses interactions avec le système : Respecté, l'utilisateur choisit un pseudonyme au moment de se connecter.

Le système doit permettre à l'utilisateur d'identifier simplement l'ensemble des utilisateurs pour lesquels l'agent est actif sur le réseau local : Respecté, la liste des utilisateurs connectés est toujours visible à gauche de la fenêtre de chat.

Le système doit permettre à l'utilisateur de démarrer une session de clavardage avec un utilisateur du système qu'il choisira dans la liste des utilisateurs pour lesquels l'agent est actif sur le réseau local : Respecté, pour commencer une session de clavardage, il suffit de sélectionner un utilisateur dans la liste et la session sera automatiquement lancée.

Le système doit garantir l'unicité du pseudonyme des utilisateurs pour lesquels l'agent est actif sur un même réseau local : Respecté, avant d'accepter la connexion, le système vérifie si aucun autre utilisateur du réseau local est déjà connecté avec ce pseudonyme.

Tous les messages échangés au sein d'une session de clavardage seront horodatés : Respecté, les messages sont horodatés et cette donnée est également stockée dans notre base de données.

L'horodatage de chacun des messages reçus par un utilisateur sera accessible à celui-ci de façon simple : Respecté, il suffit de laisser la souris quelques secondes sur un message, pour que l'horodatage s'affiche.

Un utilisateur peut mettre unilatéralement fin à une session de clavardage : Respecté, l'utilisateur peut à tout moment fermer sa session ou la fenêtre, ce qui aurait pour effet de mettre fin à toutes les sessions de clavardage en cours.

Lorsqu'un utilisateur démarre une nouvelle session de clavardage avec un utilisateur avec lequel il a préalablement échangé des données par l'intermédiaire du système, l'historique des messages s'affiche : Respecté, toutes les sessions de clavardage sont stockées localement grâce à SQLite, si une nouvelle session démarre avec un utilisateur déjà connu du système, l'historique est alors chargé.

L'utilisateur peut réduire l'agent, dans ce cas, celui-ci se place discrètement dans la barre des tâches sous la forme d'une icône lorsque le système d'exploitation permet cette fonctionnalité : Respecté, comportement nativement possible avec JavaFX.

Le système doit permettre à l'utilisateur de changer le pseudonyme qu'il utilise au sein du système de clavardage à tout moment : L'utilisateur peut changer de pseudonyme en cliquant sur son pseudonyme dans la fenêtre de chat. L'unicité sera vérifiée pour le nouveau pseudonyme choisi.

Lorsqu'un utilisateur change de pseudonyme, l'ensemble des autres utilisateurs du système en sont informés : Respecté, les autres utilisateurs sont automatiquement informés d'un changement de pseudonyme via le multicast.

Le changement de pseudonyme par un utilisateur ne doit pas entraîner la fin des sessions de clavardage en cours au moment du changement de pseudonyme : Respecté, lors de la réception via le multicast d'un changement de pseudonyme, les données sont mises à jour mais cela n'affecte en aucun cas les sessions de clavardage en cours.

Le déploiement du système doit être réalisable en 2 heures à partir de la prise de décision de déploiement : Respecté, comme nous l'avons dit, il suffit de copier un fichier JAR sur chaque poste pour déployer le système. Si on veut que le système soit accessible depuis l'extérieur du réseau local,

il faudra également démarrer le servlet sur un poste accessible du réseau local et de l'internet. Ces étapes pourront facilement être réalisées en 2 heures.

Le changement de pseudonyme d'un utilisateur doit être visible de l'ensemble des autres utilisateurs dans un temps inférieur à 20 secondes : Respecté, pour réaliser ce test, nous avons modifié le pseudonyme 100 fois de suite et nous avons calculé la durée entre le premier envoi et la réception du 100ème changement par l'autre utilisateur.

```
369512043123952
USER: 0 - G22A8A9394E2 (/fe80:0:0:0:502a:8aff:fe93:94e2%1lw0)
M-S: G22A8A9394E2:editUser:0
```

```
369512080870584
M-R: G22A8A9394E2:editUser:99
USER: 99 - G22A8A9394E2 (/10.189.3.237)
[99]
```

Si on calcule la différence entre les deux, on obtient une durée de 37,7 ms pour 100 changements de pseudonyme. On est donc bien en-dessous de la limite des 20 secondes.

Le temps écoulé entre l'envoi d'un message par un utilisateur et la réception par un autre utilisateur ne doit pas excéder 1 seconde : Respecté, idem, nous avons envoyé 100 messages et nous avons calculé la durée entre l'envoi du premier message et la réception du 100ème message par l'autre utilisateur.

```
372141716595825 372142337721988
U-S: 0           U-R: 99
```

La durée est de 621,1 ms pour l'envoi et la réception de 100 messages, pour un seul message on est donc bien inférieur à 1 seconde.

Le système doit permettre la mise en place de 1000 sessions de clavardage simultanées au sein de celui-ci : Nous n'avons pas su comment vérifier cette contrainte.

L'agent doit permettre la mise en place de 50 sessions de clavardage simultanées : Nous n'avons pas su comment vérifier cette contrainte.

Lorsque la vérification de l'unicité du pseudonyme de l'utilisateur échoue, l'utilisateur doit en être informé dans une période ne dépassant pas 10 secondes : Respecté, pour vérifier cette contrainte, nous nous sommes connectés 10 fois de suite avec un pseudonyme déjà connecté et nous avons mesuré le délai entre la demande de connexion et le retour de l'erreur.

$(2896.2\text{ms} + 1928.1\text{ms} + 1487.3\text{ms} + 1390.0\text{ms} + 1031.6\text{ms} + 1256.1\text{ms} + 1317.3\text{ms} + 1048.4\text{ms} + 1409.3\text{ms} + 1287.3\text{ms}) / 10 = 1505.2\text{ms}$

Avec une moyenne de 1.5s, on est bien en-dessous de la limite de 10 secondes.


Le temps d'apparition des utilisateurs au sein de la liste des utilisateurs pour lesquels l'agent est actif ne doit pas excéder 10 secondes : Respecté, pour vérifier cette contrainte, nous nous sommes connectés 10 fois de suite et nous avons mesuré le délai entre la connexion et l'apparition dans la liste des utilisateurs connectés de l'autre utilisateur.

$(25.5\text{ms} + 9.5\text{ms} + 3.2\text{ms} + 2.5\text{ms} + 3.6\text{ms} + 2.7\text{ms} + 3.0\text{ms} + 3.1\text{ms} + 4.0\text{ms} + 3.0\text{ms}) / 10 = 6.0\text{ms}$


Avec une moyenne de 6 ms, on est très loin de la limite de 10 secondes.

Le système doit permettre un usage simultané par au minimum 100 000 utilisateurs : Nous n'avons pas su comment vérifier cette contrainte.

Le système doit avoir une empreinte mémoire inférieure à 100Mo : Non respecté, durant nos tests nous étions davantage autour des 200Mo de mémoire utilisée.

Nom du processus	Mémo... ▾
 java	235,5 Mo

Lors de son exécution, le système ne doit pas solliciter le processeur plus de 1% du temps lorsque la mesure est réalisée sur un intervalle de 5 secondes : Non respecté, sur 12 secondes de mesure, l'application sollicitait le processeur 2% du temps.

Nom du processus	% processeur ▾	Temps de traitement
 java	2,0	10,00

Le système doit présenter une réactivité normale pour une application de clavardage : Respecté, lors de nos différents tests nous n'avons pas observé une réactivité anormale.

Le système doit garantir une intégrité des messages supérieure à 99,999% : Respecté, pour vérifier cette contrainte, nous avons envoyé 100 000 messages et nous avons mesuré le nombre de messages reçus par l'autre utilisateur.

```
U-R: 99999
100000
```

On voit que la totalité des messages envoyés ont été reçus, on est donc bien au-dessus de 99,999%.

Une utilisation normale du système ne doit pas avoir d'impact sur le reste du système : Respecté, lors de nos différents tests nous n'avons pas observé d'impact négatif suite à une utilisation normale du système.

Le système doit permettre une extension simple des fonctionnalités par l'utilisation d'un système de modules qui fera l'objet d'une standardisation : Non respecté, nous n'avons pas eu d'indication sur la standardisation des modules et nous n'avons donc pas traité cette contrainte.

Partie 4 : Manuel d'installation et d'utilisation

I/ Manuel d'installation

Etape 1 : Cloner le répertoire de l'application sur l'hôte

```
git clone git@github.com:sandrrr/ChatSystem.git
```

Etape 2 : Exécuter l'archive JAR qui se trouve dans le dossier jar du répertoire cloné à l'instant

```
java -jar jar/POO_Chat_System_SHI_WANG-1.0-SNAPSHOT.jar
```

Etape 2 Bis : Si vous préférez, vous pouvez également utiliser maven pour créer votre propre archive JAR avec la commande suivante avant de l'exécuter :

```
mvn package
```

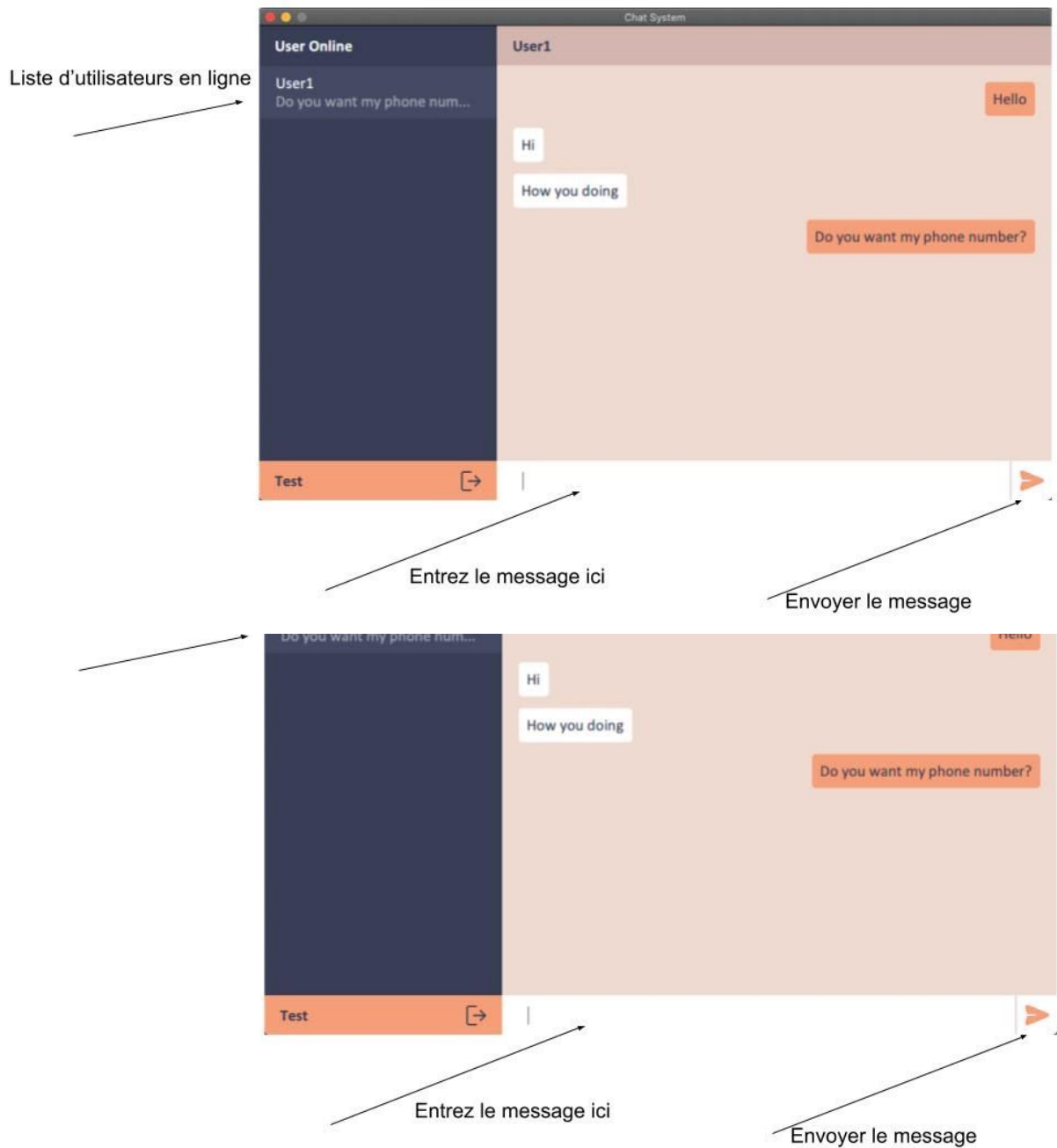
II/ Manuel d'utilisation

Ce système de chat a pour but d'être simple et intuitif. Il est donc très simple à utiliser. Les utilisateurs ont seulement besoin de choisir un pseudonyme pour commencer à chatter.

Phase 1 : Se connecter



Phase 2 : Communiquer



Après cette phase, vous pouvez déjà tester les fonctionnalités du système.

Vous pouvez changer votre pseudo à tout moment.

