PROJET CHAT SYSTEM

RAPPORT TECHNIQUE DU PROJET CHAT SYSTEM
UF POO 4A IR

CADOT Benjamin
DECORE Jérôme
Génie Électrique et Informatique
INSA Toulouse

Table des matières

Introduction

Le but de ce projet est de développer un système de communication qui sert de support aux équipes et groupes de l'entreprise afin de leur permettre d'accroître leur efficacité naturelle. Le système utilise un(des) agent(s) afin d'accomplir sa mission.

En tant que relais de communication, le système permet aux intervenants de se coordonner par l'échange de messages textuels entre eux mais également des illustrations sous la forme d'images, des schémas en plus de permettre l'échange de fichiers de travail.

Le système supporte la dynamicité du contexte. Il détecte et prend en compte les connexions des utilisateurs subites au fil de l'eau (en cours d'exécution).

Selon la situation, l'architecture du système change. Il prévoit des comportements et des procédures de communication différents et adaptés. En entreprise, lorsque tous les utilisateurs sont sur le réseau propriétaire, le système est dépourvu d'une architecture centralisée.

Dans ce rapport, nous allons vous présenter les choix technologiques et architecturaux qui ont été faits. Dans un premier temps nous détaillerons comment nous avons conçu le projet à l'aide de diagrammes UML. Dans un second temps, nous décrirons en détail les classes implémentées. Nous terminerons avec une présentation succincte des procédures de tests et de validation mis en place.

1

Conception à l'aide de diagrammes UML

Cette partie a pour but de présenter les diagrammes initiaux lors de la conception de notre système. Nous en présenterons d'autres dans la partie suivante pour montrer l'état actuel des choses. Dans l'ensemble, nous avons suivi les idées originales.

Premièrement, pour le choix et le changement de pseudonyme puis pour l'envoi de messages.

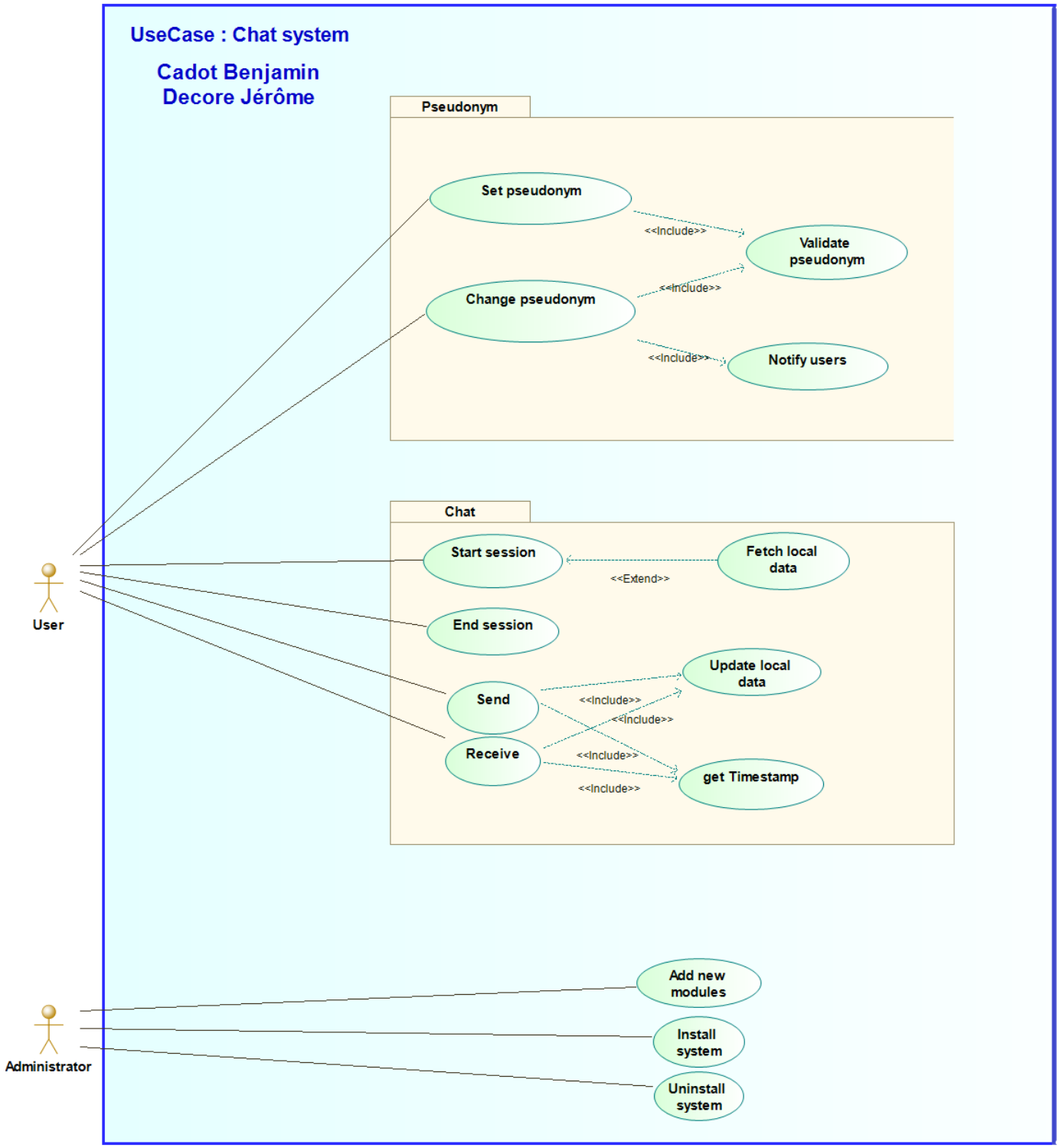

FIGURE 1.1 – Diagramme des cas d'utilisation lors de la conception

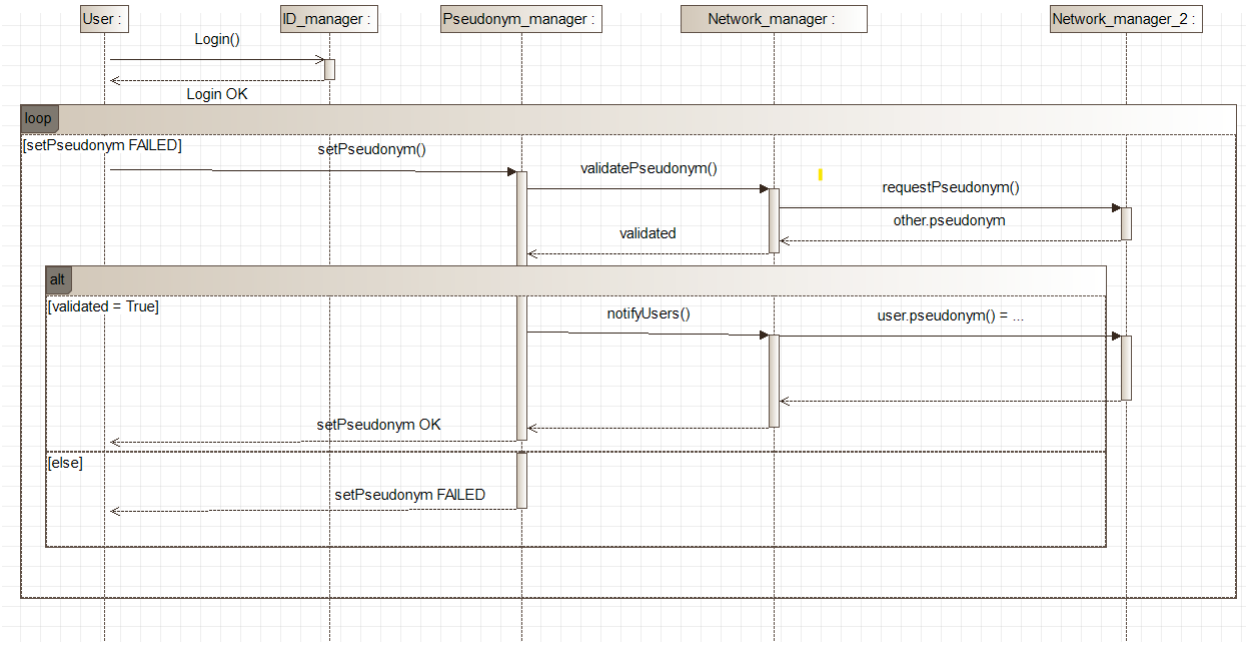

FIGURE 1.2 – Diagramme de séquence du pseudonyme pendant la conception

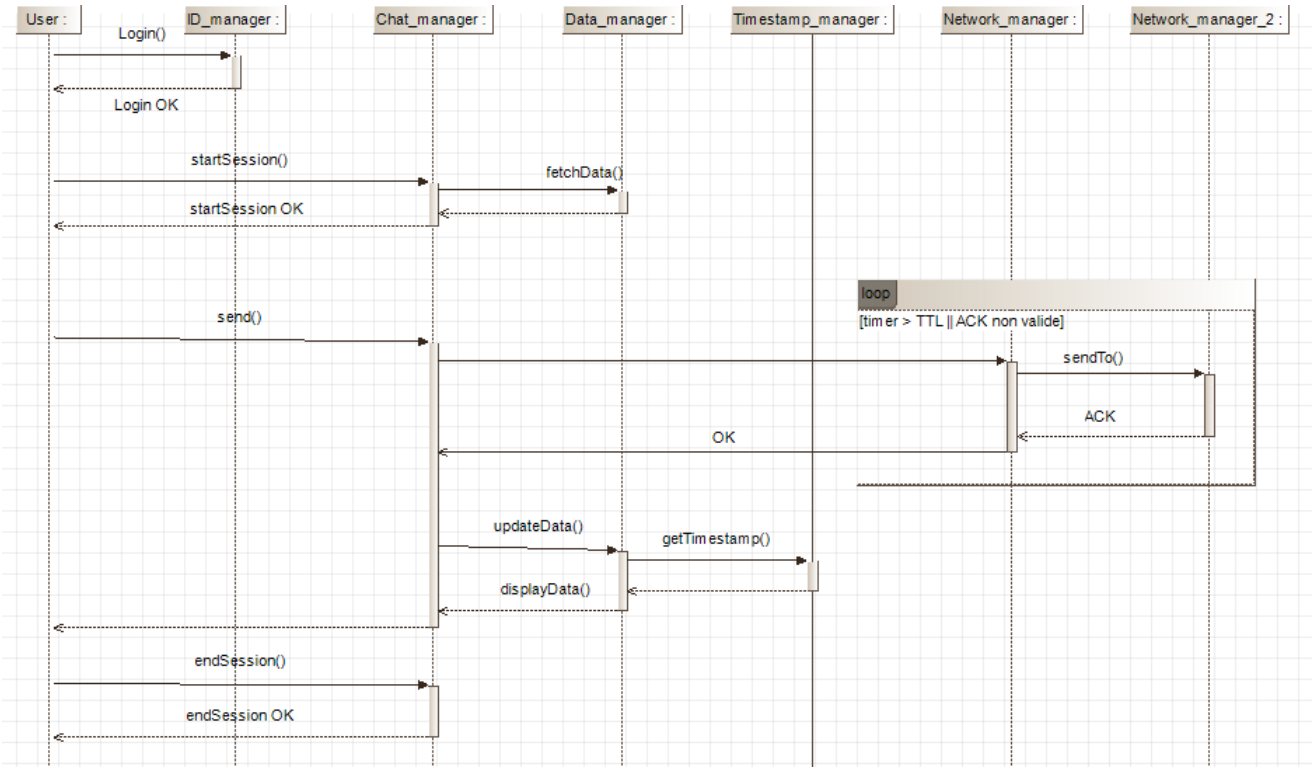

FIGURE 1.3 – Diagramme de séquence pour l'envoi de messages au moment de la conception

2

Description des classes implémentées

2.1 Partie utilisateur

Dans ce segment, nous allons décrire en détail l'architecture utilisée pour la partie utilisateur de notre programme.

2.1.1 Classe Agent

La classe Agent est la classe "racine" de notre application. En effet, elle contient une occurrence de chacune des classes qui seront utilisées par la suite. Son constructeur permet d'initialiser toutes les occurrences de classe ainsi que de lancer les serveurs UDP et TCP sous forme de Threads. Nous reviendrons en détail sur l'implémentation de ces derniers.

2.1.2 Classe Id_Manager

L'agent dont on a parlé précédemment est identifié par un id. Celui-ci est initialisé dans le constructeur par défaut en récupérant l'adresse IP de la machine qui exécute l'application. En effet, étant sur un réseau local, cette adresse IP sera unique et pourra donc servir d'identifiant. Nous supposons que cette adresse sera de classe C, c'est à dire de la forme 192.168.x.x. Pour ce faire, nous récupérons la liste des interfaces réseau de la machine. Pour chaque interface, nous récupérons la liste des adresses associées et la comparons au String "192.168.x.x" grâce à une expression régulière. Dès que nous avons trouvé une adresse de cette forme, nous cessons les itérations et mettons à jour le champ *id*.

2.1.3 Classe Pseudonym

Un agent doit pouvoir choisir un pseudonyme. De ce fait, cette classe possède un attribut *pseudonym* de type String. Celui-ci peut être changé à tout moment à condition qu'il soit différent d'un pseudonyme déjà utilisé par un autre agent sur le même réseau. Pour pouvoir effectuer ces vérifications, plusieurs méthodes ont été définies. Tout d'abord nous avons implémenté une méthode

setPseudonym() prenant comme paramètre un String. Cette méthode est donc un setteur pour l'attribut *pseudonym*. Elle renvoie un booléen à vrai si le pseudo a bien été changé, et faux sinon. Elle vérifie d'abord que le pseudonyme passé en argument n'est pas égal au pseudonyme que possède déjà la classe appelante. Ensuite, elle invoque une autre méthode *validatePseudonym()* qui prend en paramètre un String. Cette fonction renvoie vrai si le pseudonyme est valide, faux sinon. Elle envoie le String reçu en argument en broadcast UDP et va chercher à savoir si une réponse à été reçue à la suite de ce broadcast. Si une réponse est reçue, cela signifie qu'un autre utilisateur s'est manifesté lors de la réception du broadcast et a renvoyé un paquet informant que le pseudonyme n'est pas disponible. Si aucune réponse n'est reçue au bout d'un certain temps, on considère qu'il est disponible et donc la méthode *validatePseudonym()* renvoie vrai. Ainsi, dans la méthode *setPseudonym()*, nous pouvons à présent modifier le pseudonyme. Après modification, il est nécessaire d'informer les autres utilisateurs du réseau. La méthode *notifyUsers()* permet cela. Cette méthode effectue un broadcast UDP contenant le flag *notificationPseudonym* d'un objet de type *User* représentant l'utilisateur en train de changer son pseudonyme.

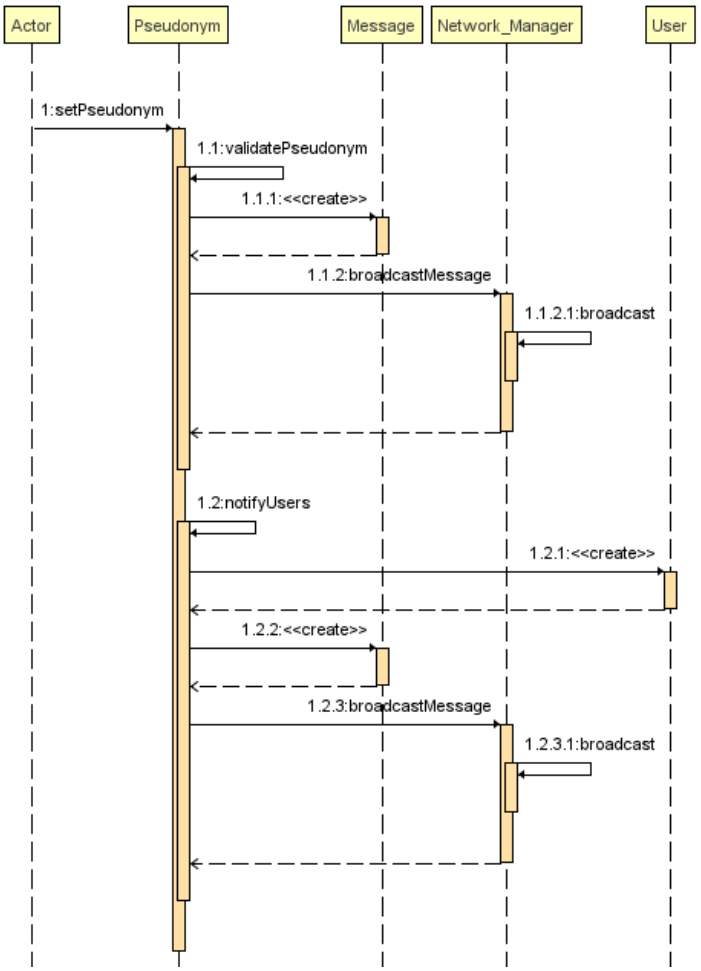

FIGURE 2.1 – Diagramme de séquence de *setPseudonym()*

2.1.4 Classe User_Manager

L'agent doit connaître les autres utilisateurs actifs. La classe User_Manager possède une liste d'utilisateur de type *ArrayList<User>* nommée *activeUsers*. Elle implémente diverses méthodes pour récupérer des informations de la liste telles que l'IP, le port ou un utilisateur lui-même en fonction de l'argument donné. Ces méthodes sont utilisées notamment dans la partie communication.

2.1.5 Classe User

Cette classe représente un utilisateur externe du point de vue de l'agent. Elle possède 3 attributs : *ip* de type String, *name* de type String également et *rcvPort* de type int correspondant au numéro de port à utiliser pour contacter cet utilisateur. Ces trois éléments définissent totalement un utilisateur. Cette classe implémente l'interface *Serializable* pour être envoyée sur le réseau par le biais de la classe Message.

2.2 Partie communication

Nous allons maintenant préciser les classes utilisées pour communiquer avec les autres agents.

2.2.1 Classe Message

Toutes les communications se font à l'aide de la classe Message. Cette classe possède six attributs qui vont être initialisés ou non selon le contexte : *message* de type String, *user* de type User, *type* de type String, *timestamp* de type LocalDateTime, *file* de type File et *fileBytes* de type byte[]. Elle possède également cinq constructeurs représentant cinq contextes d'utilisation. Premièrement, pour les messages de contrôle, dans lequel est spécifié un message et le type du message. Il est utilisé dans la classe Pseudonym en précisant les types suivants : *notificationPseudonym, requestValidatePseudonym* ou *answerValidatePseudonym*. Le second est utilisé pour broadcast un nouvel utilisateur sur le réseau lors de son enregistrement. Le type *notificationPseudonym* est alors renseigné. Le troisième représente un message textuel envoyé lors d'une communication. On lui fournit le message envoyé et il met à jour le champ *timestamp* avec la date et l'heure locales. Il est utilisé dans les classes de test. La quatrième prend en plus un User et a le même fonctionnement. On l'utilise lorsqu'on appelle la méthode *Network_Manager.send()*. Le dernier est utilisé lorsqu'on veut envoyer un fichier. Il met à jour le champ *file* mais également le champ *fileBytes* à l'aide de la méthode privée *prepareFile()* qui retourne un tableau d'octets à partir d'un fichier.

Cette classe implémente l'interface *Serializable* puisqu'elle sera envoyée dans le réseau sous forme de flux binaire. Nous n'avons pas précisé de taille maximale pour l'attribut *message*, ni pour *fileBytes*. Cependant, la limitation viendra du choix fait dans la mise en place de la base de données à propos du type *Blob*.

2.2.2 Classe Chat

La classe Chat fait le lien entre l'agent et la communication. Elle regroupe la classe Data_Manager et la classe Network_Manager.

2.2.3 Classe Network_Manager

Les interactions faisant appel à des primitives réseau pour l'envoi de données sont gérées par cette classe. Elle possède plusieurs attributs tels qu'un attribut de type *UDP_Serv* et un attribut de type *TCP_Serv*. Elle contient également deux autres attributs pour l'adresse IP de l'utilisateur et l'adresse IP de broadcast. Le constructeur de cette classe initialise nos deux serveurs TCP et UDP puis affecte l'adresse IP utilisateur et l'adresse de broadcast. Pour l'attribution de l'adresse de broadcast, la méthode *getBroadcastAddress()* est implémentée.

Venons en maintenant aux fonctionnalités réseau qu'implémente cette classe. Tout d'abord, la méthode *broadcast()* permet, comme son nom l'indique, d'effectuer un broadcast UDP sur le réseau. Elle prend en paramètres l'objet *Message* à envoyer ainsi que l'adresse de broadcast. Elle crée d'abord un socket et active l'option de broadcast sur celui-ci. Ensuite, elle transforme l'objet *Message* à envoyer en un tableau d'octets. Cette transformation peut être effectuée car *Message* implémente l'interface *Serializable*. Elle créé finalement un datagramme UDP et l'envoie via la primitive *send()* du socket créé précédemment. A la fin de l'opération, le socket est fermé.

La méthode *send(Message, User)* permet d'envoyer un message à un autre utilisateur. Pour ce faire, nous récupérons l'IP et le port de ce User puis créons un socket à partir de ces informations. Nous récupérons le OutputStream du socket créé, l'associons à un ObjectOutputStream et nous pouvons écrire notre message. Ensuite, nous fermons le ObjectOuputStream puis insérons le message écrit dans la base de données. A la fin, nous fermons le socket. Si une exception est levée pendant l'exécution de la méthode (soit à l'association du socket, soit en récupérant le OutputStream) nous en déduisons que l'utilisateur distant s'est déconnecté et nous le supprimons donc de notre liste d'utilisateurs connectés. Nous pourrions également le signaler aux autres agents à l'aide d'un broadcast mais nous ne l'avons pas implémenté. Le diagramme de séquence de cette méthode est illustré sur la figure 2.2.

Si nous voulons envoyer un fichier, nous utilisons la méthode *sendfile(File, User)*. Nous convertissons le fichier passé en argument en Message puis nous appelons la méthode décrite ci-dessus.

2.2.4 Classe Data_Manager

Cette classe met en place la connexion avec la base de données. Elle possède les attributs *url*, *user* et *password* de type String dans son constructeur et *con* de type Connection. Lors de l'appel à son constructeur, l'attribut *con* qui représente la connexion est mis à jour grâce à la méthode *getDBinfo()* qui récupère les informations de connexion dans un fichier texte database_setup.txt. A partir de cette connexion, nous pouvons désormais effectuer des requêtes SQL.
La méthode *fetch()* est utilisée pour récupérer l'historique des messages entre l'agent et le destinataire.

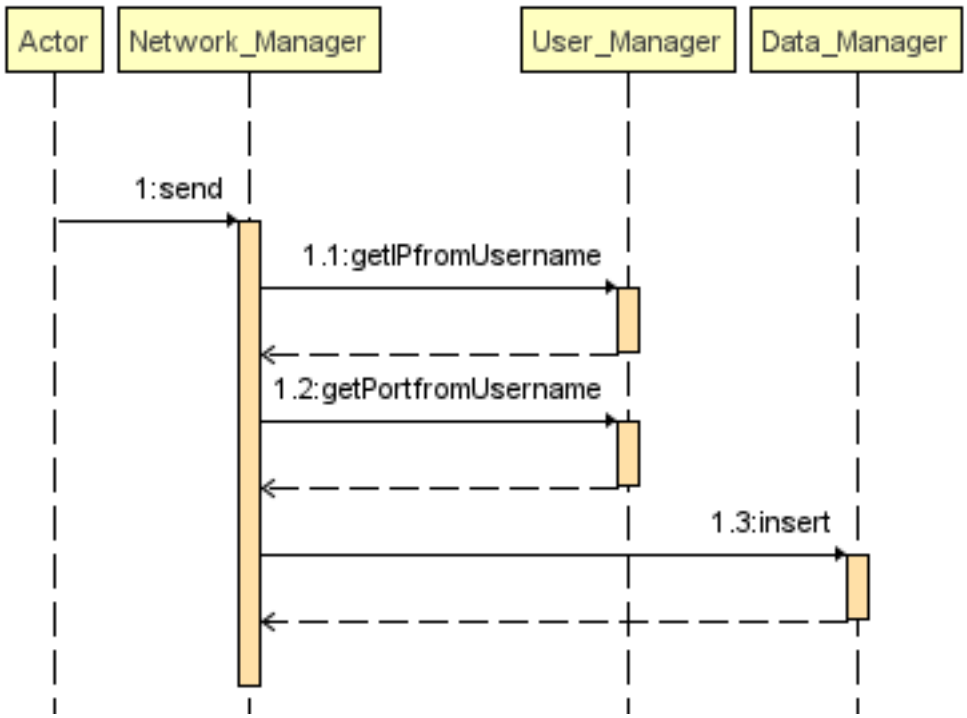

FIGURE 2.2 – Diagramme de séquence de l'envoi des messages

Nous créons une instance de Blob, récupérons son InputStream pour définir un ObjectInputStream sur lequel nous effectuons un cast pour avoir un objet de type Message. La méthode *insert()* quant à elle insère dans la base de données un message entre l'agent et le destinataire. On utilise pour cela un PreparedStatement dans lequel on définit les valeurs *src*, *dest* et *msg*. [1]
En cas d'exception, la méthode retourne l'erreur associée.

2.2.5 Classe UDP_Serv

Un serveur UDP est nécessaire pour notre application. En effet, pour la découverte du réseau ainsi que pour les demandes de vérification entre utilisateurs, des segments UDP sont créés et envoyés. Il est donc nécessaire de pouvoir les recevoir et les traiter. La classe *UDP_Serv* va donc implémenter les fonctionnalités nécessaires pour remplir cette mission.

Elle possède trois attributs : un booléen *answerReceived* initialisé à false, un entier *port* initialisé à 1234 et un Network_Manager *network*. Le constructeur prend comme argument un objet *Network_Manager* et l'affecte à son attribut *network*.

Cette classe implémente l'interface *Runnable*, ce qui va lui permettre de s'exécuter en tant que Thread. La fonction qui sera exécutée par le Thread est définie dans la méthode abstraite *run()*. Dans cette méthode, les actions du serveur UDP sont donc implémentées. Dans un premier temps, le serveur va créer un socket d'écoute sur le *port* 1234. Lors de la réception d'information binaire sur

1. Voir le manuel d'utilisation pour la structure de la base de données

ce port, le serveur va la convertir en un objet de type *Message*. A partir de cet objet, il est possible de récupérer le *type* du message. Avant de traiter le message reçu en fonction de son type, le serveur vérifie d'abord qu'il ne provient pas de son propre *Network_Manager*. Une fois la vérification faite, nous rentrons dans un bloc case qui repose sur le type de message reçu.

Si le message reçu est de type *requestValidatePseudonym*, cela veut dire que le serveur UDP reçoit une requête de changement de pseudonyme de la part du *Network_Manager* d'un autre utilisateur. Le serveur UDP compare alors le pseudo reçu avec le sien. S'ils sont égaux, le serveur renvoie un message de type *answerValidatePseudonym* à destination de l'émetteur du message pour lui dire que le nom est déjà utilisé.

Si le message reçu est de type *answerValidatePseudonym*, cela veut dire que le serveur UDP reçoit une réponse suite à une requête de changement de pseudonyme. L'attribut booléen *answer_received* de la classe *UDP_Serv* est alors affecté avec la valeur true.

Enfin, si le message reçu est de type *notificationPseudonym*, cela veut dire qu'un utilisateur sur le réseau vient d'invoquer la méthode *notifyUsers()* définie dans la classe *Pseudonym*. Le serveur met donc à jour la liste des utilisateurs contenue dans la classe *User_Manager* en y ajoutant le *User* reçu.

Ces actions sont représentées dans le diagramme de séquence de la figure 2.3.

2.2.6 Classe TCP_Serv

Pour la réception des messages, nous avons ajouté un serveur TCP. De même que la classe UDP_Serv, TCP_Serv implémente l'interface *Runnable* pour s'exécuter en tant que Thread. Nous créons un serverSocket et l'allouons au port 1234 choisi arbitrairement. On attend une connexion grâce à la primitive *accept()*, récupérons l'InputStream du socket créé et lisons le message. A ce stade-là, le message peut soit contenir un fichier, soit du texte. Ceci sera déterminé par la valeur du flag *type*. Si c'est un fichier, nous en créons un vide (portant le même nom que celui envoyé) dans le home directory de l'utilisateur. Nous écrivons ensuite son contenu dans le fichier vide. Cette façon de faire n'est pas sécurisée bien entendu, il faudrait trouver un moyen de rendre le fichier téléchargeable depuis la fenêtre de discussion mais nous n'avons pas eu le temps. Ensuite, nous affichons le message reçu dans la fenêtre prévue à cet effet en appelant diplayMessageReceived en tant que Thread.

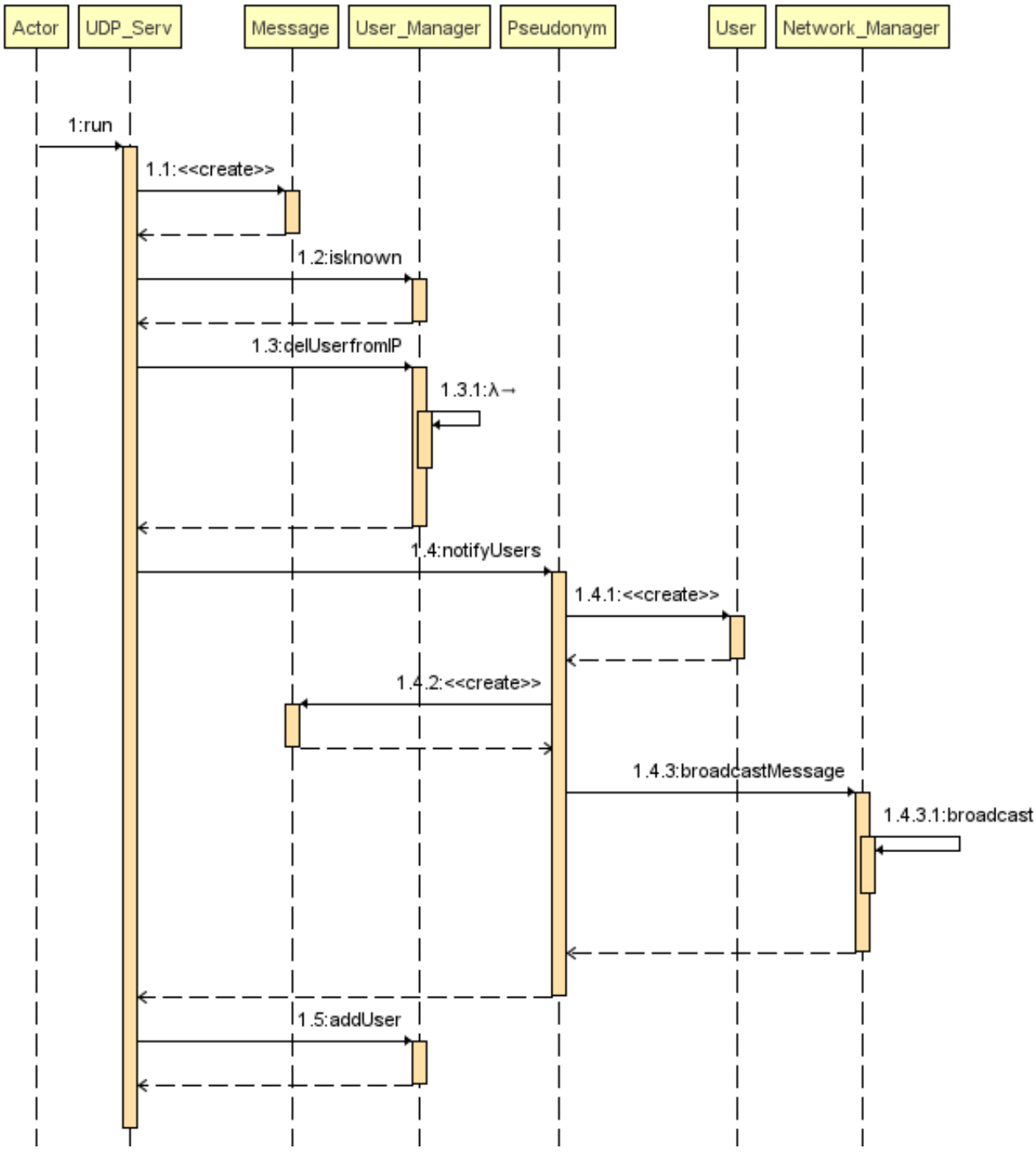

FIGURE 2.3 – Diagramme de séquence du serveur UDP

2.3 Partie graphique

2.3.1 Overview

Pour l'interface graphique, nous avons utilisé un fichier *App.form* intégré dans l'IDE Intellij IDEA. A ce fichier est associée la classe *App* où toute l'interface graphique est gérée. Il nous a donc été possible de placer des éléments JSwing à la volée à partir d'une interface graphique. Chaque élément est contenu dans un JPanel nommé *MyPannel*. En bas à gauche, nous avons un élément JTextField *pseudoInput* pour rentrer le pseudonyme voulu. A cet élément est associé un JButton *button1* sur lequel un listener est en attente d'un clic. En haut à droite, un JTextArea *pseudo* est placé. Celui-ci affiche le nom que l'utilisateur possède. En haut à gauche, nous avons un autre JTextArea *PSEUDO-TextArea* contenant le champ PSEUDO. Enfin, au milieu de la fenêtre, nous avons une JList *usersList* contenant la liste des utilisateurs présents sur le réseau.

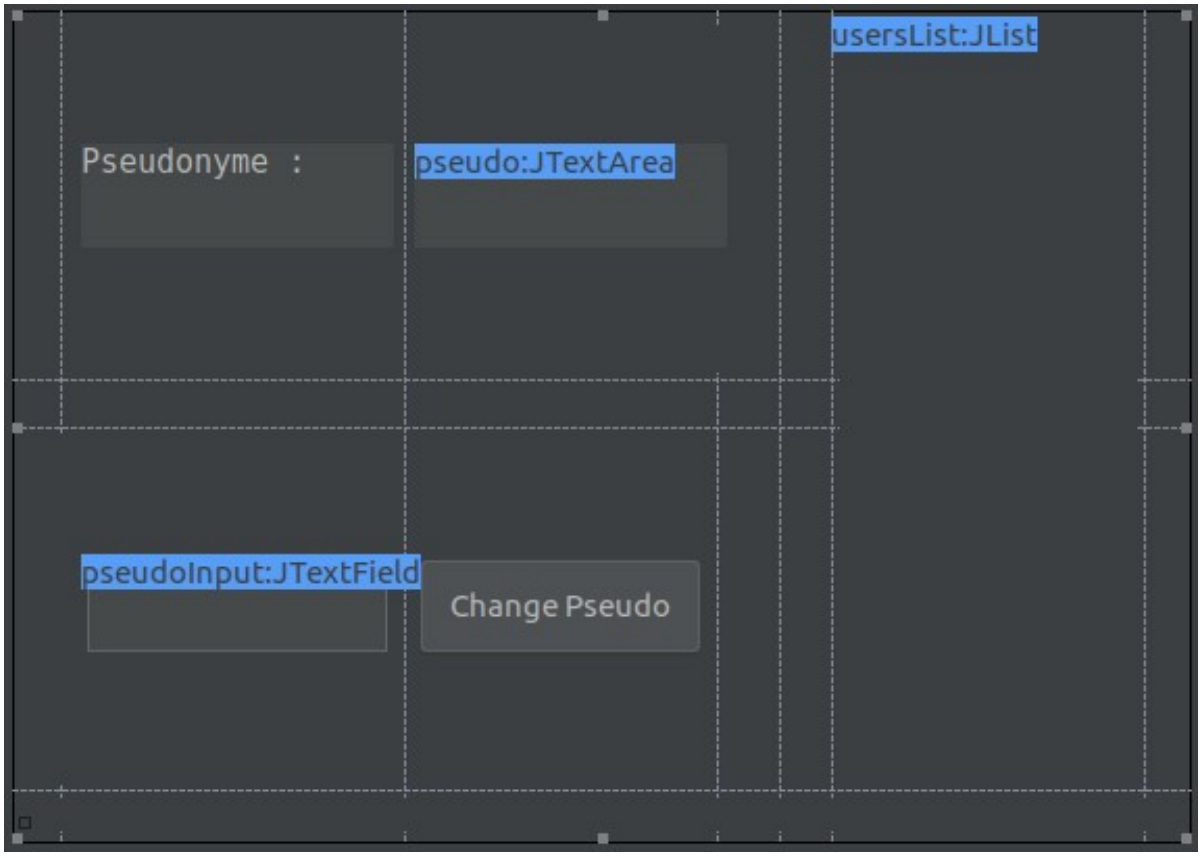

FIGURE 2.4 – Présentation de la fonctionnalité .form

Au final, voici la fenêtre obtenue à l'exécution (illustré à la figure 2.5).

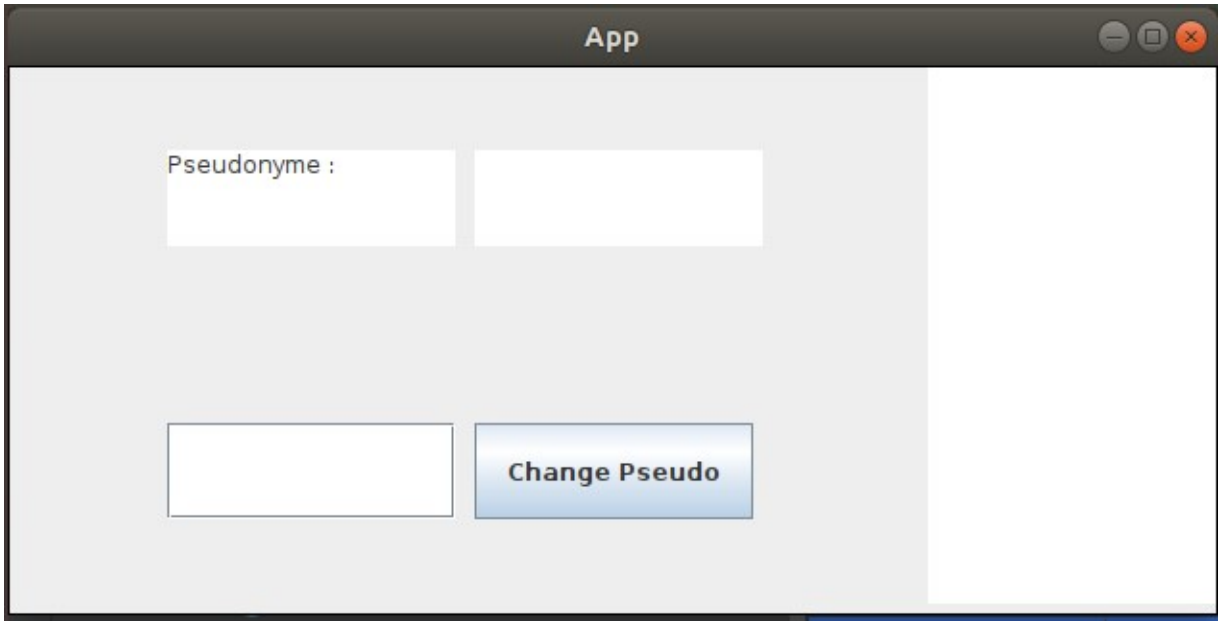

FIGURE 2.5 – Fenêtre de l'application

2.3.2 Classe App

C'est à l'intérieur de cette classe que sont initialisés tous les composants cités précédemment. De plus, un objet de type *Agent* y est créé ce qui permet à la classe *App* d'avoir accès à toutes les classes du programme. Un attribut *destinationUser* de type User permettant de connaître notre destinataire est aussi défini. De plus, on peut trouver cinq autres attributs : *newFrame* de type JFrame, *sendMessage* de type JButton, *sendFile* de type JButton, *messageBox* de type JTextField et *chatBox* de type JTextArea. Ces attributs permettront par la suite de gérer une nouvelle fenêtre pour le chat entre utilisateurs. Dans le constructeur de notre classe *App* sont gérés la création des Listener, le lancement du Thread permettant de mettre à jour la liste des utilisateurs connectés ainsi la création de la fenêtre de chat.

2.3.3 Mise à jour du pseudo

Quand un utilisateur lance l'application, son nom initial est son adresse IP. Pour changer de pseudonyme, il doit le saisir dans le JTextArea *pseudoInput*. Il doit ensuite cliquer sur le JButton *button1*. Celui-ci possède un listener dans lequel est invoquée la méthode *setPseudonym()* de la classe *Pseudonym* avec en argument le champ contenu dans *pseudoInput*. Si la méthode renvoie vrai, le pseudo a été changé avec succès et le champ du JtextArea *pseudo* est mis à jour avec le pseudo saisi. Si la méthode renvoie faux, un pop up demandant à l'utilisateur de choisir un autre pseudo apparaît.

2.3.4 Mise à jour et affichage de la liste des utilisateurs connectés

La liste des utilisateurs connectés doit s'afficher et s'actualiser dynamiquement en parallèle du reste du programme dans la JList *usersList*. Pour se faire, l'usage d'un Thread est nécessaire.

Nous avons donc créé une nouvelle classe *JListUpdate* qui implémente l'interface Runnable. Elle possède notamment un objet *DefaultListModel* nécessaire à l'actualisation de la JList. Une méthode *getUsersOnline()* est définie. Elle renvoie une liste de type *DefaultListModel* contenant le pseudo de tous les utilisateurs présents sur le réseau à partir de l'ArrayList<User> contenue dans la classe *User_Manager*.

La méthode abstraite *run()* est ensuite implémentée. Dans celle-ci, une boucle infinie tourne dans laquelle nous faisons appel à la fonction *getUsersOnline()* définie plus haut. Il est à présent possible de mettre à jour notre JList grâce à la méthode *JList.setModel()* qui attend comme argument une *DefaultListModel*. Le Thread est ensuite mis en sommeil pendant dix secondes de manière à ce que la JList soit actualisée périodiquement.

2.3.5 Commencer un chat avec un autre utilisateur

Pour démarrer un chat avec un autre utilisateur, il faut sélectionner ce-dernier dans la JList affichant les utilisateurs présents sur le réseau. Pour cela, un listener a été positionné sur la JList. Au sein de la classe *App* a été définie une autre classe *usersListSelectionListener* qui implémente l'interface *ListSelectionListener*. Sur occurrence d'un évènement de type *ListSelectionEvent*, nous mettons d'abord à jour l'attribut *destinationUser* de la classe *App* à partir de l'utilisateur sélectionné dans la liste. Nous savons ainsi avec qui l'on veut communiquer. Ensuite, nous rendons visible la frame contenant tous les éléments de la fenêtre de chat. Cette action fait donc apparaître à l'écran la nouvelle fenêtre utilisée pour échanger avec un autre User.

En plus de cela, si les deux utilisateurs impliqués dans la communication ont déjà échangé par le passé, l'historique de leur communication est affiché.

Vous pouvez avoir un aperçu de cette dernière à la figure 2.6.

2.3.6 Envoi et affichage de message à destination d'un autre utilisateur

Pour envoyer un message, il faut saisir du texte dans le JTextField *messageBox*. Un listener a été positionné sur le JButton *sendMessage*. Sur occurrence d'un évènement, le texte saisi dans la *messageBox* est récupéré. Si le texte saisi est *.clear*, la *chatBox* supprime tout le contenu. Sinon, elle affiche le texte suivi de la date. En plus de cela, la méthode *send()* de la classe Network_Manager est invoquée ce qui permet d'envoyer le texte saisi sous forme de *Message* à destination du *destinationUser*.

2.3.7 Affichage de message venant d'un autre utilisateur

L'affichage des messages reçus se fait à partir de la classe *TCP_Serv*. Lorsqu'un message est reçu sur le serveur TCP, celui-ci crée une instance de la classe *displayMessageReceived*. C'est elle qui gère

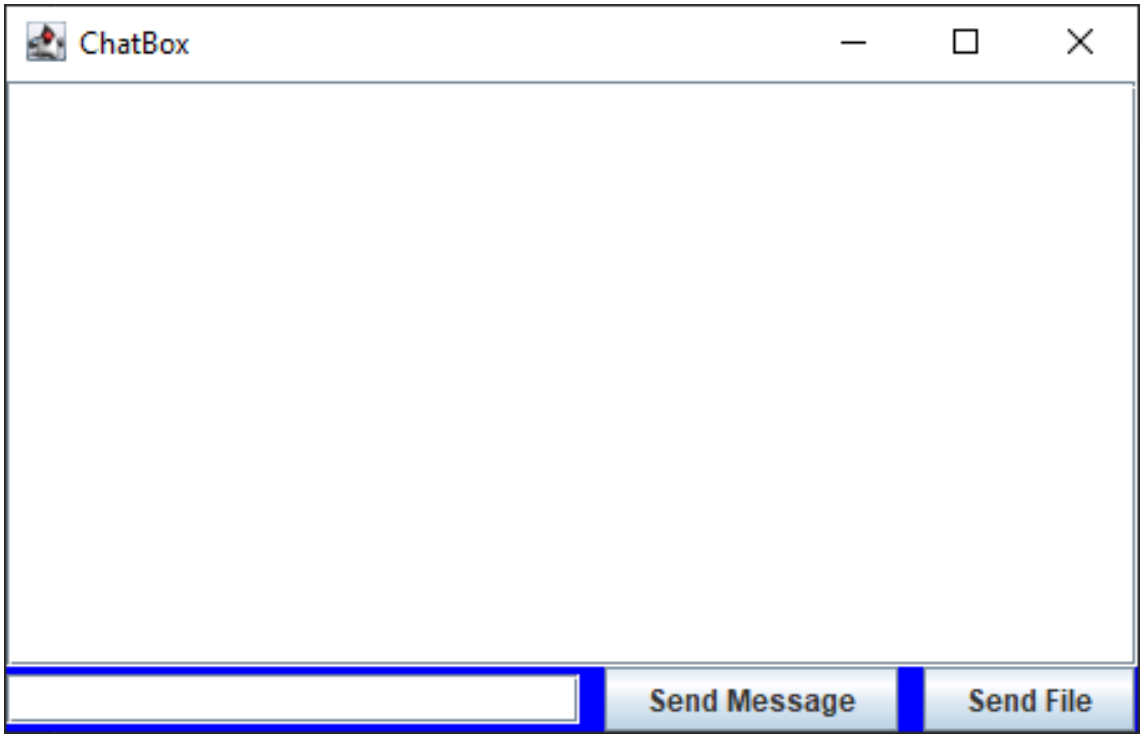

FIGURE 2.6 – Apparence de la fenêtre de messagerie

l'affichage du message dans la chatBox. Elle implémente l'interface Runnable, ce qui fait d'elle un thread. Sa méthode abstraite *run()* permet l'affichage de l'émetteur suivi du message daté.

2.3.8 Envoi et affichage de fichier à destination d'un autre utilisateur

Pour envoyer un fichier, il faut cliquer sur le Jbutton *sendFile*. Un listener a été positionné sur ce dernier. Sur occurrence d'un évènement, un explorateur de fichier s'ouvre. Il suffit de sélectionner un fichier et de l'ouvrir, et le nom de fichier ainsi que la date s'affichent dans la *chatBox*. En plus de cela la méthode *sendFile()* est invoquée ce qui permet d'envoyer le fichier ouvert sous forme de *Message* à destination du *destinationUser*.

2.3.9 Affichage et lecture de fichier venant d'un autre utilisateur

L'affichage des fichiers se fait de la même manière que celui des messages.

3

Tests effectués

Les tests ont été effectués entre deux machines virtuelles configurées en réseau local pour simuler deux ordinateurs distincts. Nous utilisons le système d'exploitation Ubuntu mais avons également réalisé des tests sur un ordinateur Windows afin de vérifier la portabilité de notre application.

3.1 Test sur le changement de pseudonyme

Le but de ce test est de vérifier le bon fonctionnement de la méthode *setPseudonym()*. Pour cela, le test doit être lancé sur deux utilisateurs différents. Dans un premier temps, l'utilisateur 1 lance le test. Le résultat attendu de ce test est une modification réussie du pseudonyme car aucun autre utilisateur n'a choisi le pseudonyme donné par le test. Dans un second temps, l'utilisateur 2 lance le test. Le résultat attendu est une erreur car le pseudonyme donné par le test pour l'utilisateur 2 est le même que celui donné par le test à l'utilisateur 1.

3.2 Test IP

Ce test a pour but d'afficher l'adresse IP de l'agent ainsi que l'adresse de broadcast correspondant au réseau sur lequel il est connecté.

3.3 Test sur l'envoi de message

Le but de ce test est d'observer l'envoi et la réception de messages. Il faut deux utilisateurs différents. Après la phase d'initialisation (choix d'un pseudonyme aléatoire afin de ne pas en avoir deux identiques), on envoie un message à l'autre utilisateur (que l'on connaît dans la liste *activeUsers* de la classe Pseudonym).

3.4 Test toString

Ce test est utilisé pour pour la méthode *toString* de la classe Message.

3.5 Test BDD

Dans ce test, on insère 10 messages "basiques" dans la base de données à l'aide de la méthode *insert()* puis on la teste également avec un message nul puis un message vide. Enfin, on affiche l'historique des messages récupéré avec la méthode *fetch()*.

Conclusion

Nous avons développé une application permettant la communication textuelle et l'échange de fichiers sur un même réseau local, typiquement une entreprise. Nous avons respecté le premier cahier des charges. La mise en place d'un environnement de test ayant été compliquée, nous n'avons pas eu le temps d'implémenter la partie "Outdoor users" ainsi que le serveur de présence. La vue d'ensemble de notre système est présente en annexe A.

Annexes

A : Diagramme de classes