Rapport de projet

ChatSystem

Minot Eva
Gervais Paul

INSA Toulouse
2020-2021

Table des matières

Introduction

Lors de ce projet, nous avons créé une application de messagerie permettant d'envoyer des messages à des utilisateurs présents sur le même réseau physique. Tout d'abord, au niveau de l'architecture, nous avons choisi de suivre le modèle MVC (Model, View, Controller). Nous avons donc un package Model, contenant les classes User et Message, un package View qui contient toutes nos interfaces utilisateur, et enfin un package appelé Connection, qui contient toutes nos classes utilisées pour la connexion entre utilisateur (TCP et UDP). Comme vous pouvez le remarquer, nous ne l'avons pas appelé « Controller », simplement car le nom Connection nous semblait désigner plus directement à quoi servaient les classes du package, mais elles s'apparentent malgré tout à des controllers. Enfin nous avons un dernier package à part, le package Database qui contient toutes les classes utiles à la gestion de notre base de données.

Dans ce rapport, nous allons dans un premier temps expliquer tous les choix d'implémentation et d'architecture que nous avons effectué, puis nous allons présenter un manuel d'utilisation pour l'application de messagerie.

1. Choix d'implémentation

1.1 Connexion/Communication

L'un des aspects fondamentaux de ce projet est la communication : en effet, le but d'une application de messagerie est de pouvoir envoyer des messages à des destinataires distants. Pour permettre cette communication, il faut passer par de la connexion. Pour mettre en place cette connexion, nous avons choisi de séparer les deux sortes de connexions que nous utilisons : la connexion udp et la connexion tcp.

1.1.1 Connexion UDP

La connexion UDP est très importante au bon fonctionnement de notre application de messagerie : c'est elle qui gère le broadcast. Ainsi, nous avons créé une classe « UdpConnect » qui est une classe centrale : elle se comporte à la fois comme un serveur à l'écoute sur un port défini pour le broadcast (6666 dans notre code), et peut également envoyer des messages en broadcast. C'est elle qui traite les différents « signaux » reçus provenant des autres utilisateurs (signal de connexion d'un user, de déconnexion, de changement de login...). Cette classe doit être instanciée dès le lancement de l'application par l'utilisateur (avant même que son login ait été vérifié). Elle doit ensuite être « gardée » après la connexion de l'utilisateur, car elle sera utile tout le long de l'utilisation de l'application (à tout moment, un autre utilisateur connecté peut vouloir changer de pseudo... Le serveur UDP doit donc être en train de fonctionner pour traiter cette demande!). C'est une classe assez lourde, et nous en avons conscience (elle dispose de nombreuses méthodes, parfois similaires), cependant comme dit précédemment, c'est une classe centrale au fonctionnement de l'application, et il ne nous paraissait pas judicieux de la séparer en plusieurs petites classes.

1.1.2 Connexion TCP

La connexion TCP intervient lorsqu'un utilisateur souhaite démarrer une session de clavardage avec un autre utilisateur connecté. Pour implémenter la connexion TCP au sein de l'application, nous avons dû créer un serveur TCP multi-thread, qui se trouve dans la classe TCPConnect. Cette classe, liée à un utilisateur, agit comme un serveur capable d'établir plusieurs connexions TCP avec d'autres utilisateurs, et également comme un client capable d'envoyer des messages via des connexions pré-établies (même si ce n'est pas exactement cette classe qui envoie le message à proprement parler). Une instance de TCPConnect doit être créée au lancement de l'interface d'un utilisateur connecté. Elle se met alors en écoute, et à chaque nouvelle requête de connexion reçue, elle crée une instance de la classe « TCPThread », en lui fournissant un socket préalablement créé. Ainsi, la connexion TCP est établie entre l'utilisateur courant et l'utilisateur ayant effectué la requête. Comme son nom l'indique, TCPThread est une classe faisant tourner un thread permettant de récupérer les messages envoyés par l'utilisateur distant, et elle permet aussi d'envoyer des messages à l'utilisateur distant. Pour l'implémenter, nous avons eu besoin de 2 constructeurs : un avec un String indiquant le login de l'utilisateur distant, et un sans ce string, car lorsque l'on crée un TCPThread sur demande de l'utilisateur courant (via connectTo de TCPConnect), on connaît le nom de l'utilisateur distant, cependant lorsque c'est l'utilisateur distant qui a initié la connexion, il nous est momentanément impossible de récupérer son login lors de la création du TCPThread. Dans ce cas, le login manquant est récupéré plus tard grâce à l'envoi d'un message spécial par l'utilisateur distant indiquant son login. Tant que ce message n'est pas reçu, aucun autre message ne pourra être réceptionné (mais il est envoyé directement après le lancement du thread dans connectTo). Ceci permet entre autres de pouvoir indiquer l'utilisateur distant sur la chatWindow de l'utilisateur n'ayant pas lancé la discussion lui-même. D'ailleurs, une instance de ChatWindow est obligatoire dans le constructeur de TCPThread, car un TCPThread a besoin de pouvoir afficher les messages envoyés/reçus sur la chatwindow. Enfin, c'est dans la classe TCPThread que les messages envoyés sont ajoutés à l'historique.

1.2 Interfaces Utilisateur

Pour les interfaces utilisateur, nous avons assez logiquement décidé de créer une classe par interface existante, soit : une pour la fenêtre de connexion (LoginWindow), une pour la fenêtre principale de l'application (UserInterface) et une pour les fenêtres de Chat (ChatWindow).

1.2.1 Fenêtre de démarrage (LoginWindow)

Cette classe est, comme son nom l'indique, une fenêtre (interface utilisateur). Elle utilise donc swing pour créer notamment une Jframe. C'est la première fenêtre qui apparaît aux yeux de l'utilisateur lors du lancement de l'application. C'est cette fenêtre qui permet à l'utilisateur de se connecter en choisissant un pseudonyme (login). Il lui suffit pour cela d'entrer un nom dans une zone de texte, et de cliquer sur le bouton « Log in ». La classe va alors faire appel aux méthodes de la classe UDPConnect, dont une instance a été créée au préalable pour l'utilisateur courant, afin de vérifier si le login choisi est disponible ou pas. Si oui, alors l'utilisateur va être connecté, un pop up indique la connexion réussie, et entre temps un message en broadcast est envoyé aux autres utilisateurs pour indiquer qu'un nouvel utilisateur est en ligne. En revanche, si un autre utilisateur du réseau utilise déjà ce login, alors un autre pop up apparaît pour indiquer à l'utilisateur que ce login n'est pas disponible et qu'il faut en choisir un autre. Il peut alors changer son pseudonyme et retenter de se connecter.

1.2.2 Interface principale (UserInterface)

Cette classe est utilisée pour créer la fenêtre principale de l'application avec laquelle l'utilisateur va interagir. Elle comporte donc tous les éléments nécessaires à l'utilisation de notre application :

- une combobox permettant de choisir l'utilisateur avec qui on veut démarrer une conversation (elle se met à jour automatiquement sur réception par l'instance d'UDPConnect de l'utilisateur courant d'un signal de connexion ou de déconnexion). Elle comporte un élément « neutre » (Select User), cela peut paraître inutile mais nous en avons eu besoin pour éviter le lancement d'un chat avec le premier user de la liste dès le lancement de la fenêtre.
- Un espace pour pouvoir changer de login : à tout moment, l'utilisateur peut le changer en entrant le login désiré dans la zone de texte puis en cliquant sur le bouton « Change ». Ce nouveau login est vérifié, d'une manière similaire à ce qu'on a décrit pour la connexion.

1.2.3 Fenêtre de discussion (ChatWindow)

Pour cette classe, nous avons dû utiliser deux constructeurs, car il y a deux moments où on crée une chatwindow : soit en cliquant sur un utilisateur connecté dans l'interface utilisateur (auquel cas, on a à disposition toutes les variables utiles à la chatwindow, donc on utilise le premier constructeur et on utilise la méthode connectTo de TCPConnect pour lancer le thread), soit lorsqu'un utilisateur distant souhaite chatter avec nous (auquel cas, un des deux utilisateurs -celui ayant formulé la requête- est inconnu lors de la réception de la requête dans TCPConnect, mais on a quand même besoin de créer un chatwindow donc on utilise de deuxième constructeur). C'est également dans cette classe qu'on récupère l'historique de conversation (via la méthode « retrieveHistory »).

1.3 Base de données

Pour la base de données, au début nous voulions utiliser la base de données de l'insa. Cependant, un problème assez contraignant s'est dressé devant nous : le VPN. En effet, pour utiliser cette base de données, nous avions obligatoirement besoin de nous connecter au VPN de l'INSA. Or, en faisant cela, le broadcast devenait impossible... Notre application étant grandement basée sur le broadcast, nous avons donc dû choisir une autre solution pour la base de données. Nous avons donc décidé de créer une base de données distante gratuite en ligne. De plus, notre base de données ne nous sert que pour la gestion de l'historique des messages. Ensuite, nous avons choisi de séparer la configuration de la base de données de la gestion de la base de données. La configuration se trouve donc dans la classe Database_Config, et qui est instanciée et dont la méthode configureDatabase() est appelée lors de la création de la LoginWindow, et la gestion de la database de messages est dans la classe Database_Message, où on peut ajouter des messages à l'historique entre 2 utilisateurs ou récupérer l'historique entre 2 utilisateurs. Nous utilisons les adresses ip des utilisateurs comme « id » pour reconnaître par qui a été envoyé chaque message, et à destination de qui. Nous avons fait ce choix plutôt que d'utiliser les login, car ceux-ci ne garantiraient pas de pouvoir récupérer son historique de messages si un utilisateur ne choisissait pas le même login entre deux connexions, étant donné que nous n'avons pas créé de base de données pour les utilisateurs. Ainsi, même en changeant de login, il reste toujours possible de récupérer son historique avec un autre utilisateur, tant que la machine utilisée est la même (autrement, l'adresse ip n'est pas la même). C'est un léger défaut, car si l'utilisateur change de machine, son historique n'est plus accessible, mais cela reste plus que satisfaisant. Par ailleurs, nous avons choisi d'effectuer l'ajout de messages à l'historique lors de l'envoi d'un message, et pas lors de sa réception car ainsi, même si l'un des deux utilisateurs a fermé la fenêtre de discussion, tant qu'il reste connecté, si l'autre utilisateur envoie un message alors le premier utilisateur pourra visualiser ces messages s'il ré-ouvre une fenêtre de chat avec l'autre utilisateur.

2. Tests effectués

Pour vérifier le bon fonctionnement de l'application, il a fallu réaliser de nombreux tests pour chaque fonctionnalité.

La première fonctionnalité que nous avons dû tester est le broadcast, avec la classe UDPConnect. Nous avons donc vérifié que, si sur deux machines différentes, deux utilisateurs se connectent, les envois et réceptions de messages en broadcast fonctionnent. Pour tester, nous avons donc, sur deux machines différentes, lancé l'application. L'utilisateur Paul se connecte en premier, puis Eva se connecte également.

Voyez ci-dessous les tests affichés sur la console lors des connexions des deux utilisateurs :

Machine de Paul :

```
Une session a été créée pour l'utilisateur paul
paul is sending this: Verify,paul,23315
paul is sending this: Connected,paul,23315
user interface set to views.UserInterface@24f05f79
Message recu: Verify,eva,56996
paul is sending this: ToVerify,paul,23315 to address /192.168.1.19 on port 6666
Message recu: Connected,eva,56996
```

Machine de Eva :

```
Une session a été créée pour l'utilisateur eva
eva is sending this: Verify,eva,56996
Message recu: ToVerify,paul,23315
Login valid!
eva is sending this: Connected,eva,56996
Message recu: ConnectedToo,paul,23315
```

Comme vous pouvez l'observer, on vérifie d'abord que des sessions ont bien été créées pour les utilisateurs, à leur nom (login). Ensuite, Paul envoie son premier message en broadcast afin de vérifier la disponibilité de son login : Eva n'étant pas encore connectée, il n'obtient pas de réponse, il envoie donc le message indiquant qu'il se connecte. Puis, c'est au tour de Eva de se connecter : elle envoie donc le message de vérification de login, que Paul reçoit. Il répond alors avec un « ToVerify » pour que Eva puisse vérifier son login. Eva reçoit bien ce message de Paul, et constate que son login est valide. Elle peut donc se connecter : elle envoie donc un message en broadcast pour indiquer qu'elle est désormais connectée (Connected). Paul reçoit ce message, et Eva reçoit un ConnectedToo de Paul, indiquant qu'il est également connecté.

Avec ces tests, nous avons vérifié que deux utilisateurs sont capables d'envoyer et de recevoir et traiter des messages en broadcast. Par ailleurs, ces tests montrent aussi le bon fonctionnement des tests et des signaux envoyés lors de la connexion d'un utilisateur.

Ensuite, il a fallu vérifier le bon fonctionnement de la connexion TCP : un utilisateur doit être capable d'envoyer et de recevoir des messages par TCP. Ici, pour le test, Eva envoie le message « salut » à Paul.

Machine de Eva :

```
Envoi du message salut
Recherche du thread correspondant
Thread trouvé
Message envoyé: salut
Ajout du message salut à l'historique
```

Machine de Paul :

```
message recu : salut
```

On peut voir que lorsque Eva souhaite envoyer son message à Paul, il faut d'abord retrouver dans TCPConnect le thread TCP (instance de TCPThread) associé à Paul et elle. Il y a donc une phase de recherche de thread, et ce thread est bien retrouvé. On peut donc envoyer le message, son envoi est correctement effectué (notification comme quoi le message a effectivement été envoyé). Dans la machine de Paul, une notification du message reçu est affichée : il s'agit bien du message envoyé par Eva. Par ailleurs, on teste par la même occasion que lorsqu'un message est envoyé, il est immédiatement ajouté à l'historique entre les deux utilisateurs.

Enfin, le bon fonctionnement des vues peut être testé directement sur l'application, en cliquant sur les bouttons etc.

3. Manuel d'utilisation

- Pour lancer l'application, double-cliquez sur le jar exécutable.
- Une fenêtre de connexion apparaît (peut prendre un peu de temps dû à la connexion à la base de données en ligne parfois un peu lente).
- WARNING: parfois, les composants swing n'apparaissent pas, il faut passer dessus avec la souris (bug apparu sur une des machines utilisé pour les tests, pas de fix trouvé).
- Entrez le login désiré dans la zone de texte, et cliquez sur le bouton « Log in » (si le login est déjà utilisé, cliquer sur « ok » sur le pop up et réitérer avec un autre login).
- Cliquez sur « ok » dans le pop up qui apparaît.
- L'interface principale apparaît. Si vous souhaitez :

  - envoyer un message : sélectionnez l'utilisateur avec qui vous voulez chatter dans la liste déroulante, cela lancera automatiquement un chat (WARNING: s'il n'y a qu'un autre utilisateur connecté, il est parfois nécéssaire de cliquer d'abord sur "Select User" dans la liste).

  - changer de login : entrez le nouveau login dans la zone de texte prévue à cet effet, puis cliquer sur le bouton « Change ». Un pop up apparaît pour vous indiquer la réussite ou non du changement.
- Si vous avez lancé un chat avec un autre utilisateur, OU si un utilisateur distant a lancé un chat avec vous, une fenêtre de chat apparaît. L'historique de vos messages avec cet utilisateur s'affiche, et vous pouvez ensuite envoyer des messages textuels en entrant le texte dans la zone de texte, et en cliquant sur le bouton « envoyer » .
- Pour fermer l'application, il suffit de cliquer sur la croix en haut à droite de l'interface principale (fermer une fenêtre de chat ne fermera pas l'application).

3. Conclusions et réflexions

Ce projet nous a semblé très intéressant, cependant comme vous avez pu le remarquer, nous n'avons pas pu réaliser la partie « outdoor » du projet. Cela s'explique de plusieurs manières : tout d'abord, la difficulté que nous avons eu à nous répartir correctement le travail et à tester notre code, à cause du manque de matériel dont nous disposions. Ensuite, le retard conséquent pris au début à cause de problèmes de conception dû à une mauvaise compréhension du sujet (propres à notre groupe de tp). Finalement, nous n'avons pas eu le temps de nous pencher sur la partie outdoor, préférant finaliser proprement la partie indoor. C'est un choix que nous assumons, bien que nous regrettons de ne pas avoir eu la chance de réaliser la partie outdoor qui semblait intéressante.