INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Département de Génie Électrique & Informatique

Rapport Projet : Programmation/Conception Orientée Objet

Etudiants :

Ebersohl Gwenaël – ebersohl@etud.insa-toulouse.fr
Schäfer Susanna – schafer@etud.insa-toulouse.fr

Enseignants :

Sami Yangui
Marcel Mateescu

Table des matières

I. Introduction

Dans ce projet, entièrement consacré à la programmation à l'aide du langage Java, il nous a été demandé de créer un "Système de Clavardage distribué interactif multi-utilisateur temps réel".

Pour nous aider quant à la réalisation de ce projet, un cahier des charges représentant les exigences utilisateurs nous a été donné, comportant les exigences fonctionnelles, opérationnelles, performances…

Durant ce projet, nous avions dû concevoir ce système à partir de zéro, nous basant uniquement sur le cahier des charges. Nous avons donc dû réfléchir à notre conception, aux fonctionnalités présentes et nécessaires au bon fonctionnement du système, aux différentes étapes qui allaient nous permettre de réaliser ce projet sans accroc et de la meilleure des manières.

C'est pourquoi, nous allons détailler dans ce rapport, les choix qui ont été retenus pour la conception du système, en partant de l'origine du projet vers la conception finale, en rapportant chaque partie de notre conception aux points du cahier des charges qui y est associé.

II. Conception du Système

La première exigence liée au cahier des charges est celle du choix d'un pseudonyme pour un utilisateur se connectant au système. Dans un premier temps nous avons créé la classe 'Utilisateur', une classe permettant à un utilisateur quelconque d'obtenir : un port permettant de communiquer avec l'ensemble des autres utilisateurs connectés au système, un pseudo permettant de se faire reconnaître des autres utilisateurs, ainsi que son adresse IP, permettant de stocker dans une base de données les messages échangés lors d'une discussion avec un autre utilisateur

Rq: Dans ce projet, nous avons fait l'hypothèse qu'un utilisateur possède une seule et unique adresse IP dans l'ensemble de ses connexions, un utilisateur ne peut avoir plusieurs adresses IP distinctes, car nous considérons qu'un utilisateur ne se connecte uniquement qu'avec une seule et unique machine.

Maintenant que l'utilisateur est "créé", il a fallu trouver un moyen lui permettant de choisir un pseudonyme. Nous avons choisi comme solution, de créer une fenêtre (classe Window), à l'aide de JavaSwing, qui apparaîtrait pour chaque utilisateur se connectant de choisir un pseudonyme quelconque à l'aide d'une zone de texte, l'application récupère le texte de la zone et l'affecte à l'utilisateur. [CdC-Bs-7]

Après avoir choisi un pseudo, l'utilisateur doit pouvoir avoir une idée des personnes qui sont connectés en même temps que celui-ci à l'application. Nous avons fait le choix d'un broadcast UDP (entre les ports 1024 et 65535), transmis à l'ensemble des utilisateurs connectés qui permettrait de "repérer", et de stocker dans une table la liste de ces utilisateurs. Pour ceci, nous avons créé une classe UDPConnect qui permet d'avoir une liaison UDP avec chacun des utilisateurs connectés au système. Lorsqu'un utilisateur apparaît dans le système, même avant la sélection d'un pseudonyme, un thread UDP se lance, permettant de récolter chacun des messages UDP qui lui sont transmis. Lors du choix du nickname, cette connexion permet de stocker dans un dictionnaire, la totalité des ports utilisés par les utilisateurs, ainsi que les pseudos associés à ces derniers. Jusqu'à présent, deux utilisateurs peuvent avoir le même pseudo. [CdC-Bs-8]

Pour la communication entre deux utilisateurs, le service UDP semble légèrement obsolète, nous voulons que les messages envoyés soient réceptionnés à 100%, il ne faut aucune perte lors de la communication. C'est de cela qu'est venue la classe TCPConnect. Comme pour la classe UDPConnect, cette classe lance un thread qui va tourner indéfiniment jusqu'à la déconnexion de l'utilisateur, la différence

étant que pour celui-ci, le thread va se lancer une fois que le pseudonyme sera choisi par l'utilisateur, pour qu'une conversation ne puisse pas commencer alors que la phase de choix de pseudo n'a pas encore été passée. Une fois le pseudonyme choisi, l'utilisateur doit maintenant pouvoir communiquer avec un autre utilisateur connecté sur le système. Pour cela, nous avons fait le choix de créer une nouvelle fenêtre, WindowUserList, accessible depuis la classe Window qui, aurait une liste de tous les utilisateurs connectés, et, ces personnes là pourraient être choisies à l'aide de Checkbox, une liste d'utilisateurs associée à des case, pouvant être cochée, ou non et permettant de pouvoir ouvrir une connexion avec les personnes choisies. Cette partie-là nous ayant posé plus de problèmes que nous avions pu l'imaginer, nous avons décidé de dévier un peu par rapport à notre trajet initial, et nous consacrer à l'unicité des pseudonymes.

Rq: Les classes TCP et UDPConnect sont propres à un seul utilisateur, Cf. partie COO
Pour les communications UDP, nous allons utiliser les DatagramPacket fonctionnant avec les Socket.

Pour effectuer cela, nous allons utiliser le dictionnaire précédemment créé dans la classe utilisateur. En même temps, nous avons également remarqué que, un broadcast sur (65535 - 1024 = 64511) utilisateurs peut devenir très gourmand en termes d'utilisation des ressources d'une machine. C'est pourquoi, dans la classe UDPConnect, nous avons fait le choix d'implémenter un compteur, initialisé à 0, qui s'incrémente de 1 à chaque envoi d'un message sur le broadcast, et qui serait réinitialisé à chaque fois qu'une réponse d'un autre utilisateur serait reçu. Etant donné que les numéros de ports sont attribués de 1024 vers 65535 de manière croissante, et l'un après l'autre, nous avons pris comme pari que la limite de 100 serait suffisante pour le compteur, autrement dit, si après 100 envois, nous ne recevons aucune réponse, nous considérons qu'aucun utilisateur se trouve sur un port qui dépasserait ces 100 ports (Ex. Si à partir du port 2000 jusqu'à 2100 aucune réponse n'est reçue, le broadcast est coupé, car nous considérons qu'il n'y a aucun utilisateur connecté entre les ports entre 2100 et 65535).

Pour ce qui est de la fonction permettant de vérifier l'unicité des pseudonymes, cela s'effectue en local, sur la machine de l'utilisateur essayant de choisir un pseudo déjà en utilisation. Pour cela, une boucle va parcourir les valeurs associées au clés (port) du dictionnaire, et comparer chaque pseudonyme avec le pseudonyme actuellement choisi par l'utilisateur entrant actuellement son pseudo. Si le pseudo est unique, alors tout va bien, sinon, l'utilisateur possède une infinité de choix d'autres pseudo. [CdC-Bs-10]

Revenons maintenant aux discussions entre plusieurs utilisateurs. Après avoir appliqué la méthode du parcours du dictionnaire pour récolter la liste des pseudonymes de tous les utilisateurs, nous avons fait le choix d'utiliser un nouveau

parcours de liste pour voir quelles personnes ont été choisies par un utilisateur pour pouvoir ouvrir une discussion entre ces personnes. L'idée des Checkbox nous paraissait convaincante. Pour y arriver, nous avons associé l'utilisateur à la fenêtre WindowUserList, et avons ajouté un panel regroupant le pseudo des autres utilisateurs actuellement connecté sur le système de clavardage, après avoir fait ceci, nous pouvons maintenant choisir avec qui communiquer dans le système. [CdC-Bs-9]

Après avoir réussi cette étape, il nous manquait une autre partie pour que cela fonctionne, l'interface où nous allons pouvoir échanger des messages, sinon tout cela n'aura servi à rien. Vient-alors une nouvelle classe, WindowConversation, qui récupère un utilisateur (celui qui est du côté émetteur) ainsi que les coordonnées de l'utilisateur distant, pour pouvoir communiquer (ie. le numéro de port, le pseudo).

Jusqu'ici, la classe TCPConnect n'a pas encore trouvée une grande utilité, certes un thread est lancé pour écouter les messages reçus, mais actuellement aucune fonction ne nous permet d'envoyer des messages. Alors, cette nouvelle fenêtre va devenir notre seul moyen de pouvoir envoyer et recevoir des messages TCP, ceux qui vont être échangés entre plusieurs utilisateurs durant une discussion.

Cette fenêtre va donc comporter : un panel permettant d'afficher les messages, une zone de texte permettant de les saisir, ainsi qu'un bouton d'envoi.
Cette fenêtre, comme une fenêtre normale, peut se réduire et ainsi être disposée dans la barre de tâche sous forme d'une icône. [CdC-Bs-15]

Nous effectuons des premiers tests concernant les communications TCP, en associant les WindowConversation avec un user ainsi qu'avec TCPConnect, en utilisant la zone de texte disponible sur la fenêtre nous pouvons utiliser la méthode envoiMsg(), qui nous permet, à l'aide de System.out.println de voir si nos messages envoyés apparaissent ou non dans la console, ce qui nous permet de corriger le tir si l'une des méthodes ne fonctionne pas comme voulu.

Pour les communications TCP, nous allons utiliser les PrintWriter, ainsi que les méthodes getOutputStream(), getInputStream()...)

Le prochain objectif était de lier les réceptions de la classe TCPConnect à l'affichage des différentes WindowConversation. Pour pouvoir utiliser les méthodes de WindowConversation dans la classe TCPConnect, nous avons créé un dictionnaire dans la classe Utilisateur, permettant de lier un numéro de port avec une WindowConversation, ce qui nous permettait à la fois de : créer une WindowConversation à chaque nouvelle instance de conversation, et également de répertorier dans une même liste toutes les conversations en cours d'un utilisateur avec certains autres utilisateurs. Pour répondre à [CdC-Bs-13], nous avons ajouté un événement à WindowClosed(WindowEvent e) qui retire du dictionnaire de l'utilisateur le couple (Port, WindowConversation) et met ainsi fin à la conversation entre ces derniers. Dans TCPConnect, des fonctions s'occupent de créer des

WindowConversation du côté des destinatires des messages si ces conversations ne sont pas encore ouvertes, ce qui empêche de créer plusieurs fenêtres de conversation entre deux mêmes utilisateurs (problème que l'on a pu rencontrer avant d'implémenter ce système de "tri"). Après la création de deux méthodes envoi() et recevoir() dans windowConversation, qui affichent les messages reçus dans la fenêtre, deux utilisateurs peuvent maintenant communiquer correctement. [CdC-Bs-9bis].
De plus, nous avons utilisé des expressions régulières pour pouvoir stocker et/ou récupérer les messages, le numéro de port et l'adresse IP des utilisateurs nous envoyant des messages. En effet, nos messages envoyés en TCP sont de la forme : "PortDistant #forbidden# address_ipDistant #forbidden# message"
Ici, #forbidden# est ce qui va nous permettre de récupérer chaque partie du message à la réception, à l'aide de la méthode split(#forbidden#). Cela nous aidera pour la suite du projet, pour ce qui va concerner la base de données, qui sera utilisée pour l'historique des messages.

Pour avoir l'horodatage des messages, nous avons uniquement créé des formattedDate de la forme "dd-mm HH:mm" au niveau des méthodes envoi() et recevoir(), ce qui nous permet d'ajouter l'heure d'envoi du message avant le pseudo de l'utilisateur émetteur et du message. [CdC-Bs-11]
Ces horodatages apparaîtront en permanence avec les messages échangés, les utilisateurs y auront donc accès très facilement dans la fenêtre. [CdC-Bs-12]

Avant de s'attaquer à [CdC-Bs-14] et à la base de données, regardons d'abord [CdC-Bs-16], ce qui permet de changer de pseudonyme.
Dans la classe Window nous nous occupons actuellement de l'ouverture de la fenêtre WindowUserList ainsi que l'attribution du premier pseudonyme lors de la connexion, nous allons donc ajouter ici un bouton permettant de changer de pseudo durant l'utilisation du système.
Lors d'un clic sur ce bouton, une zone de texte s'affiche permettant d'entrer le nouveau pseudo. Comme pour le premier choix, un message d'erreur s'affiche si le nouveau pseudo choisi est déjà utilisé par un autre utilisateur. [CdC-Bs-16]
Après le changement du pseudo, un message doit être envoyé à l'ensemble des utilisateurs connectés, nous allons utiliser la méthode SendEcho(), présente dans la classe UDPConnect, qui permet de faire le broadcast à tous les utilisateurs connectés. Nous avons fait le choix d'utiliser le broadcast et non d'envoyer uniquement le message aux personnes que nous savons connectées, pour éviter qu'une perte UDP vienne perturber le système, et être sûr que tout le monde puisse être informé du changement de pseudo.
Après avoir implémenté cette fonction, nous avons commencé à discerner des cas lors de la réception des broadcast UDP. Nous avons discerné les cas suivant:

- “Connexion” et “Refresh”

  Permet d’actualiser la liste des utilisateurs connectés ainsi que d’avoir leurs numéros de port associé.

- “Disconnect”

  Ceci aura pour effet de retirer l’utilisateur qui vient de se déconnecter des différents dictionnaires ou il apparaissait

- Autre message

  Tous les autres messages viendront de nouveaux utilisateurs connectés, qui vont envoyer leur nickname, ce qui aura pour effet de les ajouter dans les dictionnaires à la suite des utilisateurs déjà connectés, ou alors, si le numéro de port est déjà dans le dictionnaire, cela aura pour effet de remplacer la valeur associée à la clé dans le dictionnaire de tous les utilisateurs connectés. [CdC-Bs-17]

Un message contenant “Ok” sera renvoyé à l’émetteur lors des envois “Connexion”, “Refresh” et les autre messages, pour notifier à l’émetteur que son message a bien été reçu, et ainsi remettre le compteur permettant de vérifier si les envois ne finissent pas dans le vide à 0.

Conception de la database

Pour cette dernière partie, nous avons fait le choix de se baser sur une base de données disponible sur le serveur de l’INSA. Les éléments importants dans cette base de données sont :

- Être sûr que chaque message soit redistribué aux bons utilisateurs selon leur adresse IP, que les messages enregistrés viennent également des bons utilisateurs
- Que cette base de donnée soit accessible à tous les utilisateurs, sans problème de connexion
- Que l’horodatage soit respecté et que l’intégrité des messages ne soit pas affectée.

Voici la table que nous avons choisi pour cette base de donnée :

Field	Type	Null	Key	Default	Extra
ip_adress1	varchar(15)	NO		NULL	
ip_adress2	varchar(15)	NO		NULL	
message	varchar(200)	NO		NULL	
date	timestamp	NO		CURRENT_TIMESTAMP	

Les adresses IP 1 et 2 sont des varchar de longueurs 15 car la taille maximale pour une adresse de format IPv4 est de 15 (255.255.255.255).
Nous avons choisi comme longueur maximale par message 200 caractères, ce qui sera peut-être amené à changer dans le futur selon les critères de l'administrateur.
Pour la date, nous avons choisi de prendre l'horodatage actuel de la base de données pour chaque message, cette proposition pose un seul petit bémol, dans le cas où un utilisateur se trouverait dans un autre fuseau horaire que celui de la base de données.
De plus, nous voulons qu'aucun de ces champs soient "Null" car il y a forcément deux personnes qui communiquent, un message envoyé ainsi qu'une date, il ne peut pas manquer une information lors de l'ajout d'une donnée dans la BDD.

Cette base de données sera uniquement utilisée à l'intérieur de la classe WindowConversation, car la seule utilité de cette BDD est de stocker les messages, et de les restaurer.
Ainsi, dès qu'une conversation est ouverte, la base de données restaure les messages qui ont été envoyés dans une ancienne conversation entre les deux utilisateurs. [CdC-Bs-14]

Nous venons de traiter toutes les exigences qui concernaient les fonctionnalités de l'agent, ce qui met fin à notre essai sur ce sujet qui était le développement d'un système de clavardage temps réel. Nous n'avions pas eu le temps d'explorer la partie concernant les servlets, car tout n'était pas encore opérationnel au niveau de la partie concernant la communication indoor. Nous pencher sur la partie outdoor alors que le système n'était pas encore développé comme nous le voyons nous aurait laissé un sentiment d'incomplet, de travail mal fait, d'où ce choix.

Pour conclure ce rapport, qui fut très enrichissant pour tout ce qui touche au développement d'une application de A à Z, et nous a permis de nous projeter dans le futur, en traversant toutes les étapes de réalisation d'un projet à partir d'un cahier des charges. N'ayant jamais eu ce genre d'expérience, cela fut très instructif, bien que certaines parties du projets nous aient passé au dessus.
La première difficulté du projet était de devoir imaginer une application de A à Z, sans avoir de structure initiale ou quoi que ce soit, il y a tellement de manière d'imaginer une application, mais une fois lancés, nous nous sommes bien rendus compte que tout était bien plus difficile que ce qui était prévu.

Ensuite, beaucoup de fonctionnalités que nous avions imaginé au début du projet, se sont révélées bien plus compliquées que ce que nous aurions pu imaginer, comme par exemple une meilleure interface utilisateur, un meilleur système de conversation…
Un dernier challenge à été celui de la distance, avec un accompagnement physique, le projet n'aurait pas été aussi dur que dans le contexte actuel, les procédures de tests ont souvent été compliquées et entre chaque groupe d'étudiants, les tuteurs avaient souvent du mal à jongler entre les différentes architectures proposées par l'ensemble des binômes du projet.

Malgré tout, ce projet a été une belle expérience, et nous donne hâte de découvrir les futurs projets en situation réelle.

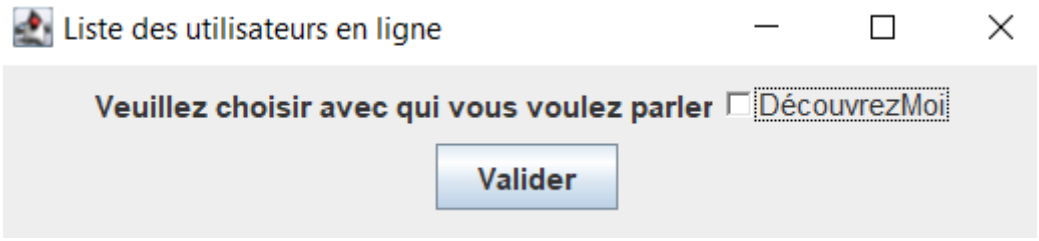
III. Différents tests effectués

Voyons maintenant certains tests concernant les exigences du cahier des charges.

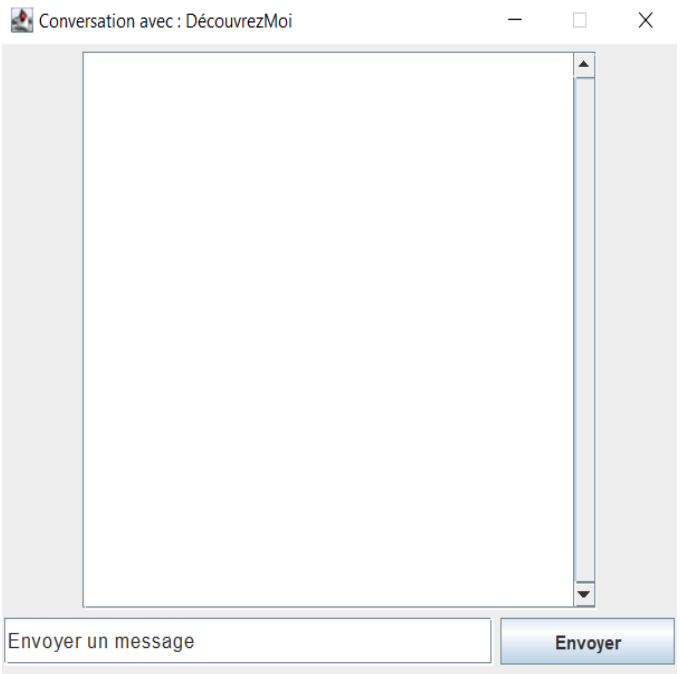
Est-ce qu'un utilisateur peut-être reconnu à l'aide de son pseudonyme?

- Création de deux utilisateurs, dont l'un des deux qui doit découvrir l'autre utilisateur

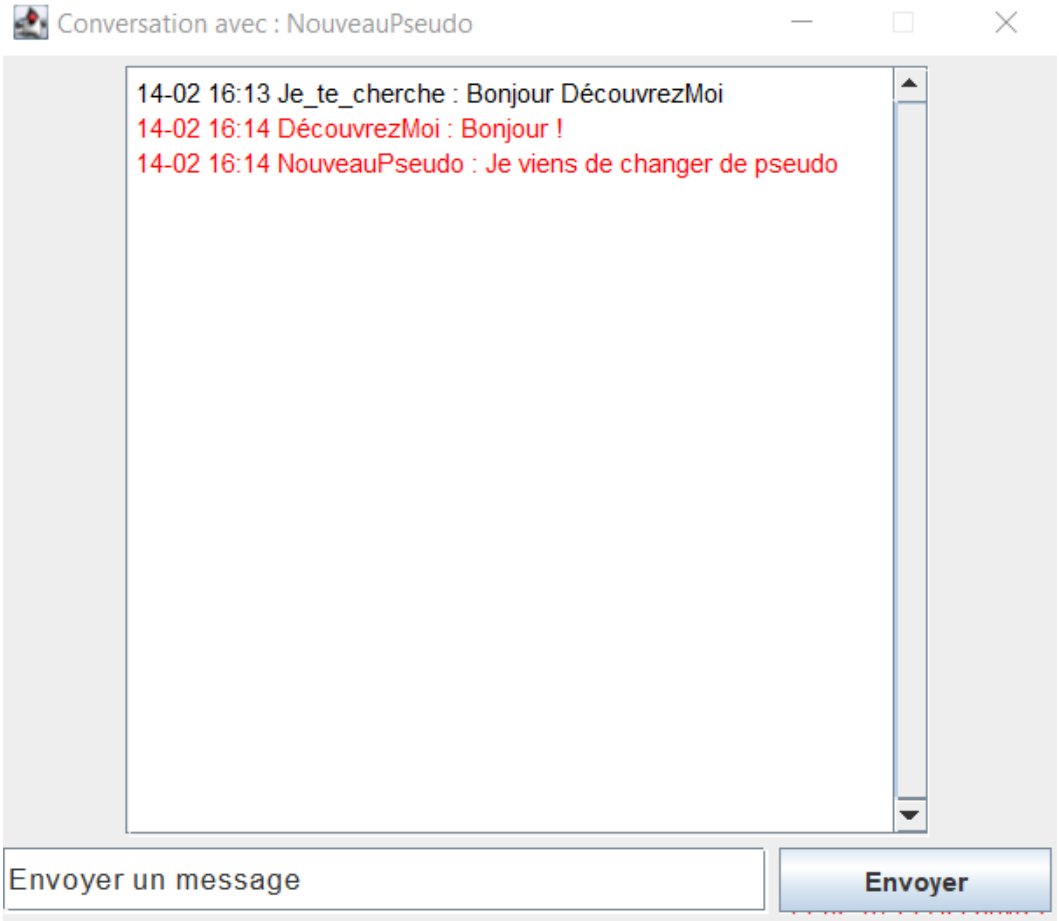
Dans la fenêtre WindowUserList, nous voyons que nous arrivons à découvrir l'utilisateur à distance :



En même temps, la CheckBox devant *DécouvrezMoi* est cochée, il est possible de communiquer avec ce dernier :



Est-il possible de changer de pseudo, de se faire reconnaître dès qu'on change de pseudo, et sans que cela ne ferme les conversations déjà en cours?



Cela est possible, nous voyons qu'entre les deux réponses, l'utilisateur distant n'a pas le même pseudo, cela ne coupe pas la discussion, et le changement est même affiché dans le titre de la fenêtre "Conversation avec: …"
De plus, nous nous apercevons que l'horodatage est présent, et très facile d'accès.

Nous pouvons d'ailleurs observer les messages UDP échangé de DécouvrezMoi vers Je_te_cherche depuis l'ouverture de l'application :

```
Je_te_cherche : L'utilisateur au port 1027 m'a  envoyé le message : DécouvrezMoi

Je_te_cherche : L'utilisateur au port 1027 m'a  envoyé le message : Ok
Je_te_cherche : L'utilisateur au port 1027 m'a  envoyé le message : Connexion
Je_te_cherche : L'utilisateur au port 1027 m'a  envoyé le message : Ok
Je_te_cherche : L'utilisateur au port 1027 m'a  envoyé le message : NouveauPseudo
```

Je_te_cherche est la personne recevant les messages
L'utilisateur au port 1027 est vraisemblablement DécouvrezMoi.
Tout d'abord, il lui envoie son premier pseudo, avant la connexion.

Un “ok” est envoyé à Je_te_cherche pour notifier qu’un de ses messages a été réceptionné.
DécouvrezMoi envoie Connexion, ce qui va lui permettre de mettre à jour sa table de numéro de port et de pseudo, pour la suite des évènements
Ensuite, ce dernier envoie un message après le choix de son nouveau pseudo.

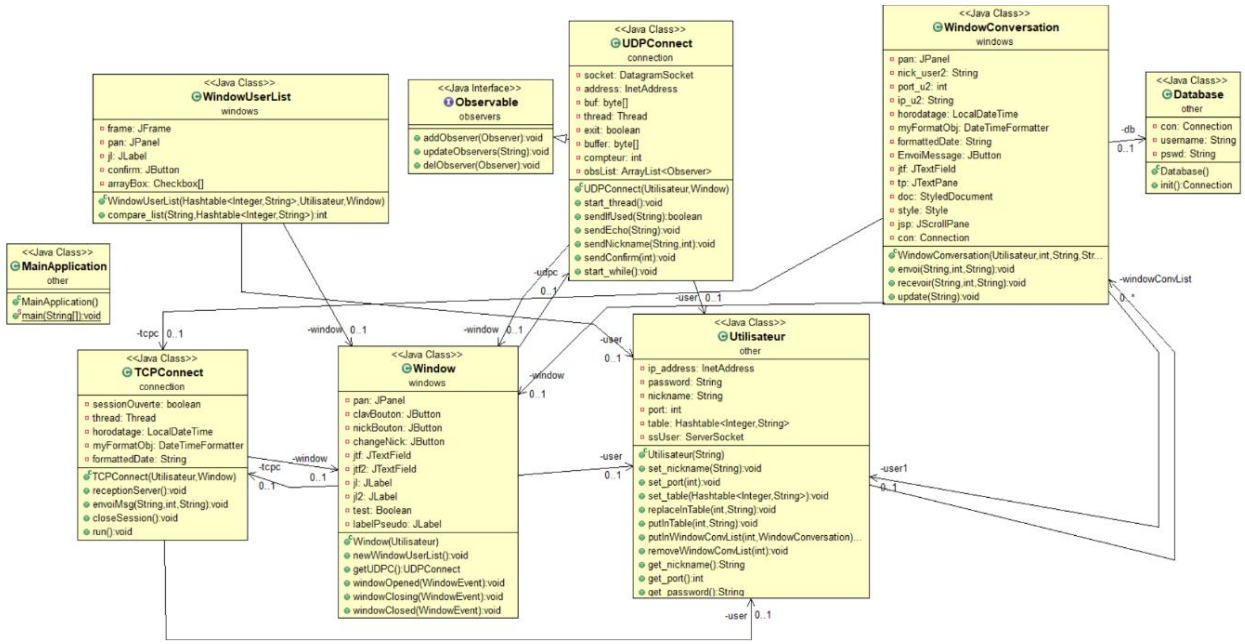
Regardons maintenant l’utilité de la base de données, là où les choses se compliquent un petit peu…
En effet, ayant fait le projet à distance, nous nous sommes très rapidement retrouvés limités par les équipements que nous possédions. Lors d’une conversation sur nos machines, l’adresse IP de l’émetteur était la même que celle du récepteur, rendant les choses compliquées pour réattribuer les messages au bon utilisateur :

```
mysql> select * from Message;
+-------------+-------------+----------------------+---------------------+
| ip_adress1  | ip_adress2  | message              | date                |
+-------------+-------------+----------------------+---------------------+
| 10.29.40.39 | 10.29.40.39 | Test envoi IP1 = IP2 | 2021-02-14 18:15:43 |
+-------------+-------------+----------------------+---------------------+
1 row in set (0.00 sec)
```

Après avoir manipulé un peu les if, else if présent dans la classe WindowConversation, nous avons remarqué que, en effet, les messages étaient bien restitués lors de l’ouverture d’une conversation, mais nous avons soit que des messages en rouge (reçus), soit des messages en noir (envoyés), nous n’avons pas l’alternance attendue rouge/noir pour différencier les messages que nous avons envoyés des messages reçu, et cela car les adresses ip émettrices et réceptrices sont les même.

IV. Conception Orientée Objet

La partie conception orientée objet aurait pu être très intéressante et utile, mais, malheureusement, nous n'avons pas pu l'utiliser lors de la partie programmation de notre projet.
En effet, nous ne disposions pas de connaissances et d'une maîtrise suffisante des outils Java nécessaires à l'élaboration du code pour pouvoir planifier et organiser à l'avance ce code. Le diagramme de classes que nous avons fait au début est très loin du diagramme de classes correspondant à notre code obtenu en fin de projet. Nous avons donc imaginé et créé les classes du projet au fur et à mesure que nous avancions dans le code, au lieu de suivre un diagramme de classes.



Cependant, en analysant le diagramme de classe associé à notre code (image ci-dessus), nous nous sommes rendus compte que l'on aurait pu produire un code plus "propre" en ajoutant plus d'observers notamment.
Cette "erreur" aurait pu être évitée si on avait pu construire un diagramme correct avant de passer au code, ce qui nous a tout de même permis de nous rendre compte à quel point le diagramme aurait dû être utile à la conception de notre projet.