Compte-Rendu COO/POO

SANCHEZ Jean-Christophe
DUMAZ Clément

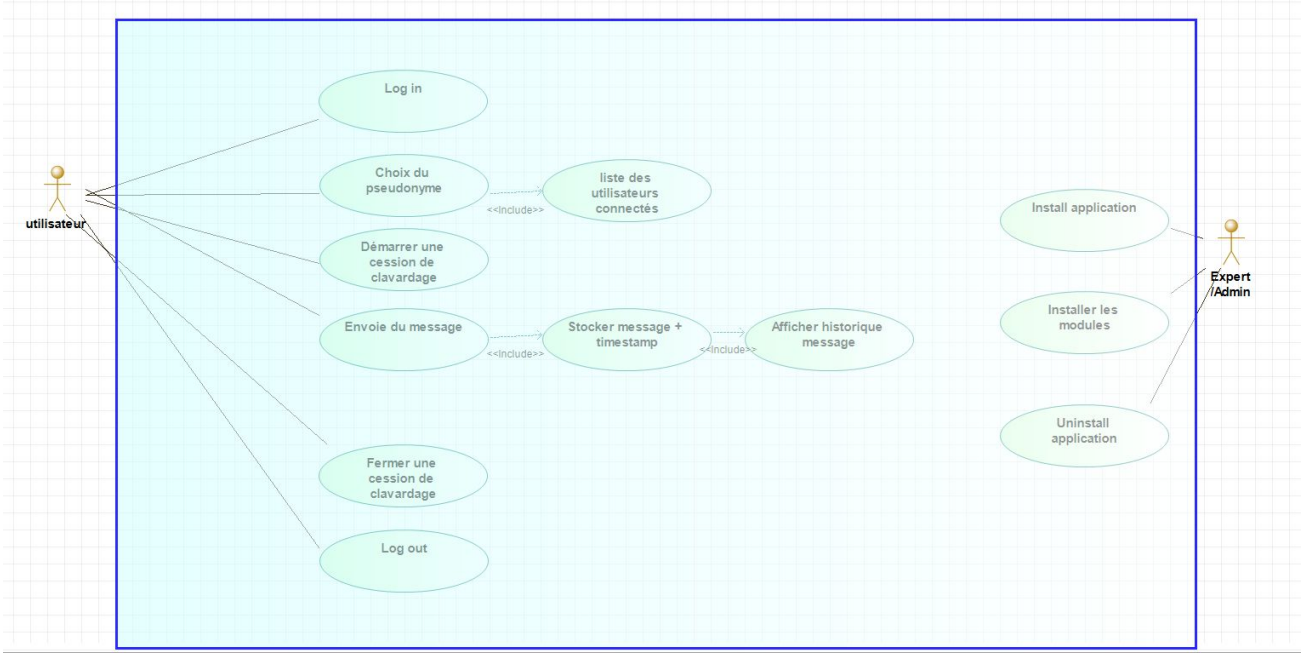
SYSTÈME DE CLAVARDAGE DISTRIBUE INTERACTIF MULTI-UTILISATEUR TEMPS RÉEL

I. Partie COO UML

Présentation des différents diagrammes UML : la première partie du projet consiste à élaborer différentes représentations UML du projet selon le cahier des charges et du produit attendu.

1. Diagramme des cas d'utilisation

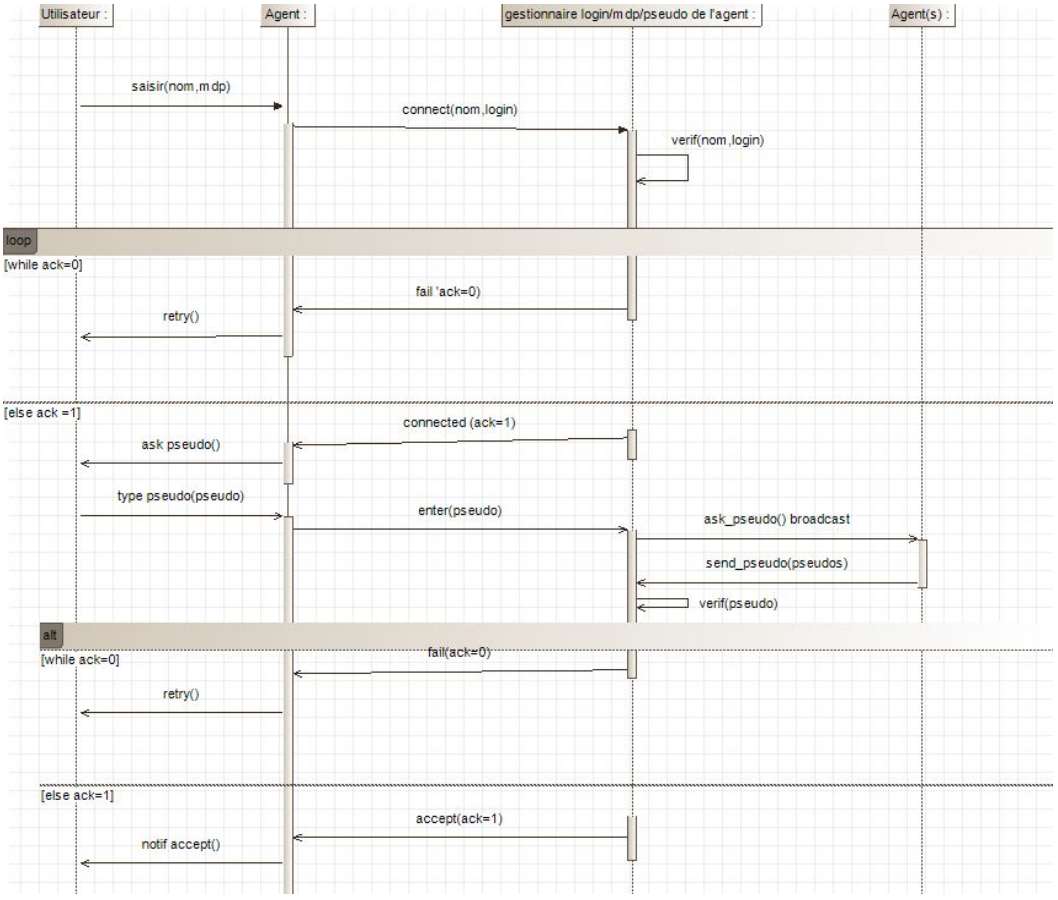
Ce premier diagramme consiste principalement à identifier les contours du système, les utilisateurs ainsi que les principales fonctionnalités.



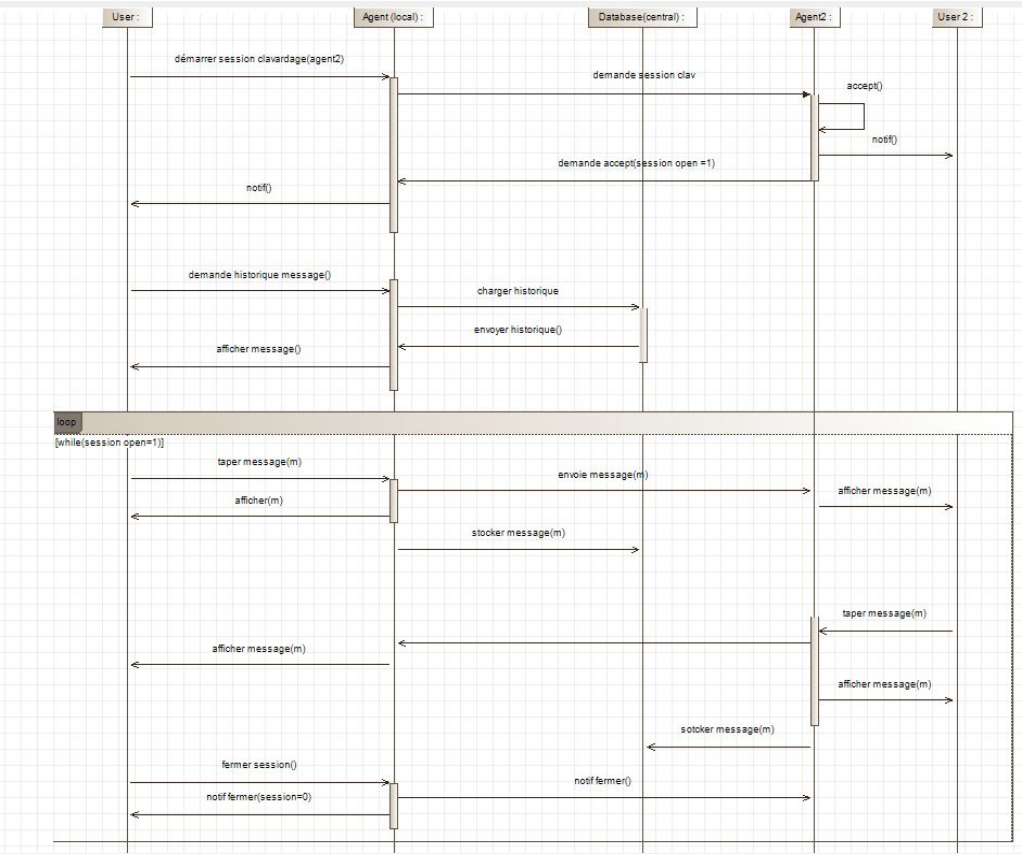
On peut distinguer deux entités externes au système : l'utilisateur classique (celui qui veut utiliser l'application pour communiquer avec d'autres utilisateurs) et un éventuel administrateur de l'entreprise qui se charge d'installer/désinstaller l'application. Ci dessus le diagramme illustrant les principales fonctionnalités du système.

2. Diagramme de Séquence

Le but ici est de détailler un peu plus les intéractions entre des composants internes du système : leur nom ici n'est pas nécessairement représentatif des classes qui seront présentes dans le code (celles-ci seront plus explicitement détaillées dans le diagramme de classes) mais permet de mettre en évidence des échanges de données, leur provenance et la manière dont le système doit réagir à leur réception.



Ce premier diagramme montre les intéractions entre l'utilisateur et l'agent qui représente ici le programme applicatif local sur la machine de l'utilisateur. Le sous-système de gestionnaire de login est mis en évidence pour gérer les données à émettre, recevoir et traiter (par exemple l'envoie en broadcast pour obtenir une liste d'utilisateurs connectés). L'agent lui transmet les commandes de l'utilisateur à ses sous-systèmes et le notifie des résultats (demande de pseudo, notification d'erreur etc).



Ce deuxième diagramme représente les intéractions concernant cette fois la création d'une session de clavardage ainsi que l'échange de message qui peut être réalisé une fois celle-ci en place. Un nouveau système est inclu : la base de données centralisée (donc unique pour tous les utilisateurs) qui stocke les conversations. Ce stockage se fait à la réception d'un message par un utilisateur.

3. Diagramme de Classes

Le diagramme de classe est disponible dans le projet sous le dossier *docUML> ClassDiagram*. L'organisation des classes, les fonctions qu'elles proposent et leurs intéractions seront explicitées dans la partie POO.

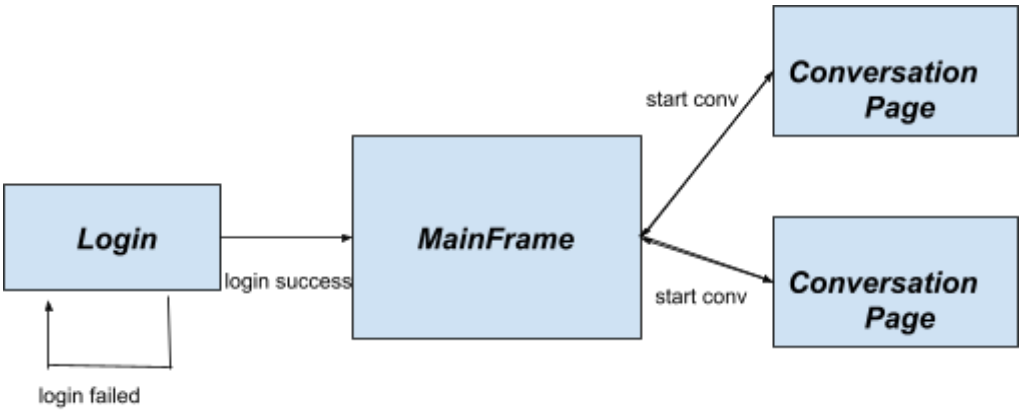
II. Partie POO

1. Description de l'architecture et choix technologiques

Le projet Java est composé de trois dossiers regroupant les différentes classes et fonctions de l'application : *GUI*, *clientLogin* et *clientClavardage*. *GUI* regroupe les classes principales en relations à l'affichage graphique, *clientLogin* celles permettant d'établir la première phase de connexion et d'accès à la fenêtre principale (la "MainFrame") et *clientClavardage* celles relatives à la discussion et l'échange de messages entre utilisateurs.

- *GUI* : description des différentes fenêtres

La technologie que nous avons utilisé est celle du Java Swing car c'était celle que nous avions vu en cours et donc étions plus familier avec. De plus, elle est relativement simple d'utilisation.
La première fenêtre est la fenêtre de connexion "*Login*" : elle demande un pseudo ainsi qu'un numéro de port (la fonctionnalité du numéro de port sera expliquée dans la partie *clientLogin*).
La seconde est la fenêtre principale "*MainFrame*" qui affiche les conversations déjà en place avec d'autres utilisateurs ainsi que la liste des utilisateurs connectés.
La troisième "ConversationPage" est la fenêtre de conversation avec les autres utilisateurs (une par utilisateur) d'où on peut envoyer des messages et visualiser la conversation.



Ici c'est directement la classe Login qui appelle les fonctions nécessaires à la connexion lorsqu'un utilisateur essaye de se connecter une fois rempli les champs et agit en conséquence : lancement de la *mainFrame* (et fermeture de la fenêtre *Login*) ou maintien de la fenêtre *Login* avec notification à l'utilisateur.

MainFrame utilise les infos récupérées par Login pour permettre d'afficher les utilisateurs connectés et se charge de lancer de nouvelles fenêtres *ConversationPage* lorsqu'un utilisateur clique sur un utilisateur ou une conversation disponible : la *MainFrame* reste toujours ouverte permettant de lancer plusieurs conversations en même temps. Celle-ci fait également appel aux fonctions de *clientClavardage* pour pouvoir lancer ces demandes de conversations mais aussi pour en recevoir.

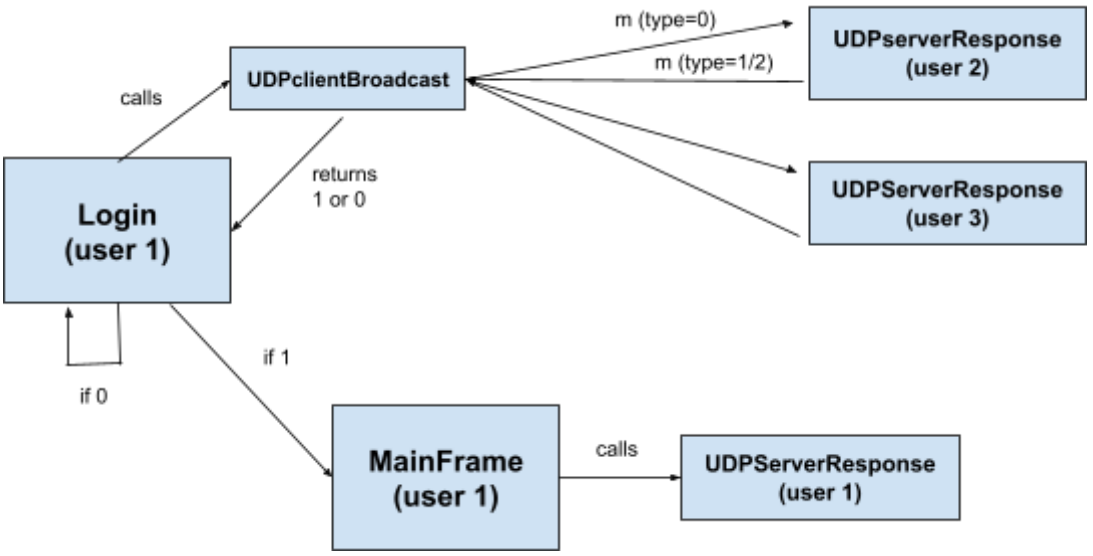
ConversationPage se voit ainsi passer un certain nombre d'arguments récupérés par la *MainFrame* lors d'une demande ou réception de conversation (noms des utilisateurs, sockets locaux créés etc…). Elle gère ensuite l'appel à des fonctions de clientClavardage pour envoyer et recevoir des messages.

- *clientLogin* : gérer la phase de connexion

Nous avons tout d'abord décidé de créer une classe *User* qui regroupe plusieurs caractéristiques nécessaires au bon fonctionnement de l'application. Un string *Login* correspondant au login entré dans la première fenêtre et permettant d'identifier l'utilisateur de manière temporaire, c'est-à-dire tant que celui-ci est connecté (le login peut changer d'une connexion à l'autre). Un Integer *UniqueId* permettant d'identifier l'utilisateur de manière statique à travers diverses connexions (on peut le voir comme étant lié à la machine utilisée), qui permet le stockage et la récupération d'informations stockées dans une base de données centralisée et permanente (voir partie sur les bases de données). On y retrouve également une adresse IP ainsi qu'un numéro de port permettant d'établir des sockets. Cette classe est très importante car les objets *User* qui sont créés sont passés à diverses classes de gestion des messages, sockets ou fenêtres graphiques en temps que attributs. Ainsi on peut donc accéder à toute sorte d'informations sur un utilisateur depuis plusieurs classes du programme, faisant ainsi plein usage de la programmation orienté objet.

Une autre classe toute autant utilisée est celle *Message*. Au début nous ne voulions qu'envoyer un seul String via les différents sockets créés mais nous nous sommes vite rendu compte que nous aurions besoin d'informations supplémentaires à échanger. Donc au lieu de tout convertir en String puis d'effectuer une analyse dans les fonctions de réception (ce qui aurait aussi impliqué de gérer l'ordre des messages) nous avons préféré envoyé tout un objet Message comportant différents attributs (pour envoyer l'objet via un socket nous avions utilisé les fonctions *Serializable* ainsi que *ObjectStream*). L'objet contient donc un String (qui le message à envoyer ou recevoir), deux User (un pour celui qui envoie le message, un autre pour le destinataire), une date (pour pouvoir afficher la date dans la fenêtre de conversation) ainsi qu'un entier *Type*. Ce dernier est important car il permet d'identifier le "type" de message par exemple une demande de connexion, une demande de conversation ou le renvoie d'un message d'erreur.

Lors d'une tentative de connexion depuis la fenêtre *Login*, celle-ci fait appel à la classe *UDPBroacast* en lançant un thread *UDPclientBroadcast* qui se charge d'envoyer un message (avec type=0) à tous les utilisateurs. Ceux-ci enverront un message en vérifiant si le login de l'émetteur est identique au leur (type=1 si ok sinon type=2). L'émetteur attend les réponses pendant un petit moment et si aucun message n'a le type=2 alors la connexion est acceptée : de plus il peut récupérer la liste des utilisateurs connectés grâce au champ User des messages reçus. La MainFrame est lancée et celle-ci appelle à son tour un serveur UDP chargé de recevoir des demandes de connexion en Broadcast. Si un message possède le type=2 alors la connexion échoue et il est demandé d'utiliser un autre login.



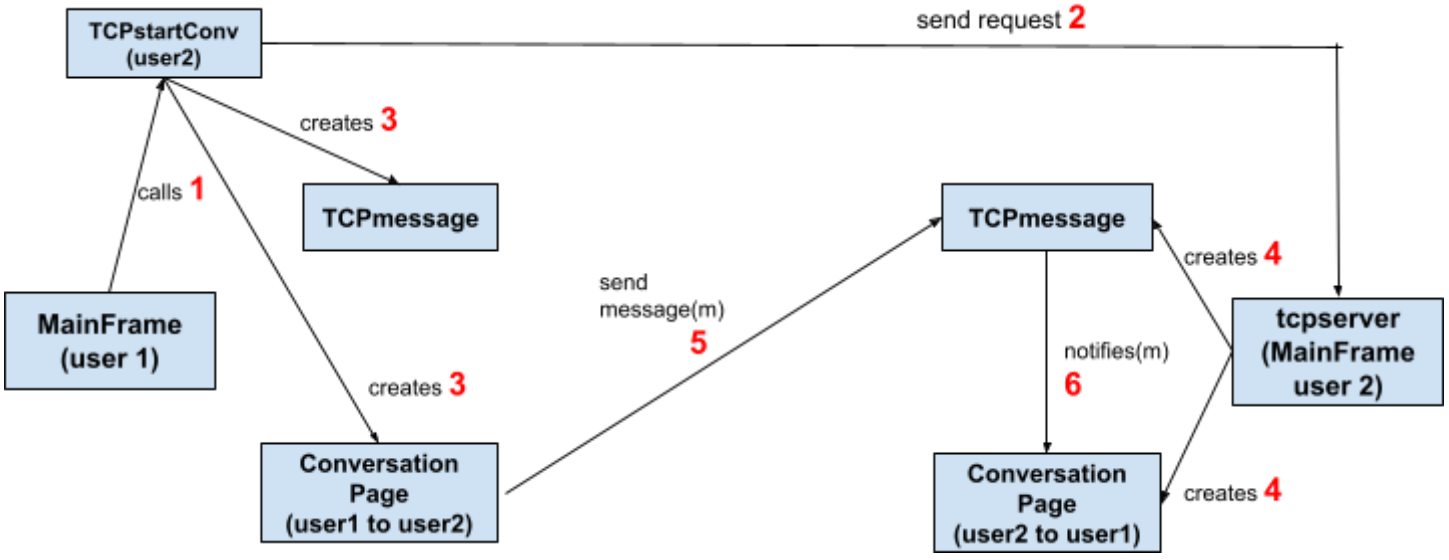
Pour la phase de login le protocole UDP est préféré car nous n'avons pas besoin de maintenir une connexion avec un utilisateur en particulier. De plus, le serveur qui gère l'arrivée de nouveaux utilisateurs n'a pas non plus besoin de créer une connexion ni d'échanger plusieurs messages car toutes les informations nécessaires sont contenues dès le premier message.

- *clientClavardage* : création de conversations et échange de messages

Afin de garder la trace des messages envoyés et des utilisateurs respectifs nous avons implémenté une classe *Conversation*. Cela permet également de rendre plus facile l'affichage graphique depuis la *MainFrame* des différentes conversations et le stockage dans la base de données centralisée. Celle-ci contient donc deux *User*, une liste de *Message*, et un *id* unique (permettant de la retrouver dans la base de données).

Une fois la MainFrame lancée celle-ci fait appel à la classe *TCPconvInit* et lance le thread *tcpserver* qui est exécuté tant que l'utilisateur est actif : celui-ci est chargé de recevoir les demandes de connexions d'autres utilisateurs et d'ouvrir un socket dédié à cette connexion. Respectivement lorsque l'utilisateur veut lancer une conversation le thread *TCPstartconv* est appelé qui effectue plusieurs opérations : il crée un socket vers le port correspondant de l'User destinataire et envoie directement un premier message qui permettra au destinataire de savoir qu'une demande est formulée. De plus une fenêtre *ConversationPage* est créée recevant le socket en argument, ainsi que le thread *TCPmessage* recevant le socket et la fenêtre. *ConversationPage* se charge d'afficher les messages reçus et envoyés, ainsi que d'appeler la fonction d'envoie de message lorsque l'utilisateur écrit un message. *TCPmessage* lui gère la réception de message et le notifie à la fenêtre.

Il est important de remarquer que lorsque le socket d'accueil *tcpserver* reçoit une demande de connexion, une fenêtre *ConversationPage* est également créée (côté réception) ainsi qu'un thread *TCPmessage*. Donc si un utilisateur reçoit une demande de conversation, une fenêtre de conversation apparaîtra de son côté également, à la manière d'une notification, et pourra tout de suite dialoguer dedans.



Contrairement à la phase de login, la demande de conversation utilise le protocole TCP : afin de maintenir la page de conversation opérationnelle tant qu'elle n'est pas fermée il faut pouvoir maintenir une connexion entre deux sockets, un sur chaque machine. Ainsi de chaque côté les classes *TCPmessage* et *ConversationPage* se partagent les fonctionnalités (envoie, réception, affichage) sur le même socket : les deux sockets sont créés et reliés dès qu'un utilisateur envoie une demande de conversation ou en reçoit une.

L'objet *Conversation* mentionné au début fonctionne de manière similaire à la *ConversationPage* : lors d'une demande de conversation depuis la *MainFrame* un objet *Conversation* est créé, de même lors de la réception. L'objet est également passé à la *ConversationPage* qui peut ainsi afficher correctement les utilisateurs concernés (User 1 vers User 2 et vice-versa).

Une fois les deux sockets reliés, les deux utilisateurs peuvent envoyer des messages librement tout en pouvant ouvrir d'autres conversations en parallèle parmi les utilisateurs qui s'affichent sur leur *MainFrame*. Le socket est détruit et le lien coupé lorsqu'un des utilisateurs décide de se déconnecter.

2. Base de données

- Base de données centralisée

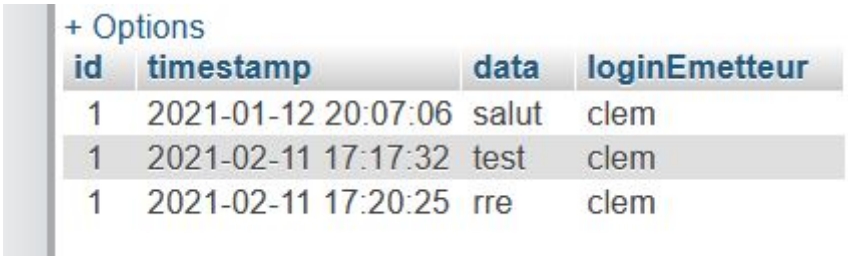
Cette base de données est commune à tous les utilisateurs de notre application. Chaque client accède à cette dernière en permanence pour stocker les messages, créer et accéder à des conversations ou bien connaître l'historique des messages. Cette base de données contient 2 tables: une table pour stocker les conversations et une autre pour stocker tous les messages.

+ Options

	idConv	User1	User2
Éditer Copier Supprimer	1	clem	jc
Éditer Copier Supprimer	28	clem	test

Table des conversations de la base de donnée centralisée

Chaque conversation comporte un identifiant unique ainsi que le login des deux utilisateurs participants à la conversation.



+ Options

id	timestamp	data	loginEmetteur
1	2021-01-12 20:07:06	salut	clem
1	2021-02-11 17:17:32	test	clem
1	2021-02-11 17:20:25	rre	clem

Table des messages de la base de donnée centralisée

Dans cette table, l'id joue le rôle de clé étrangère et réfère à l'id de la conversation auquel le message est attaché. Nous avons aussi besoin du login de l'émetteur pour nous permettre de faire un affichage harmonieux.

- Base de donnée locale sur chaque client

Cette base de données est installée sur chaque machine de chaque utilisateur. De par les contraintes présentes dans le cahier des charges, nous n'avions pas eu le choix quant à la centralisation de cette base de données. Lors du lancement de l'application un broadcast est effectué chaque réponse à ce broadcast est enregistrée dans cette base de données. De plus, un envoi de broadcast est enregistré par toutes les machines recevant celui-ci qui indique qu'un utilisateur vient de se connecter. Elle comprend uniquement une seule table, avec un id, un login et un numéro de port. Cette base de données est vidée à chaque fois que l'utilisateur se déconnecte.

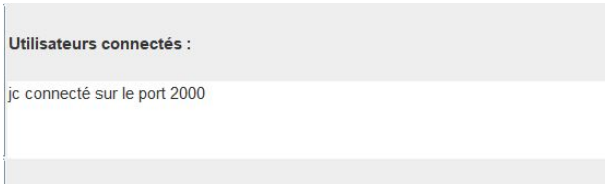
3. Tests de l'application en local

Puisque nous n'avions pas à disposition un environnement de réseau local, nous avons décidé de tester notre application en local sur la même machine. On peut ainsi lancer plusieurs fois l'application depuis la même machine. Néanmoins les parties login et réseaux de l'application sont développées en considérant cet aspect. En effet on utilise les numéros de ports en tant que référence local pour un utilisateur au lieu d'une adresse IP dans un réseau local.
Ainsi au lieu de réaliser un broadcast sur la couche réseau pour l'étape de connexion, on réalise un broadcast en utilisant l'adresse locale de la machine et en parcourant un certain nombre de ports prédéfinis : l'application par défaut peut ouvrir 3 clients différents sur les ports 2000,3000 et 4000. A noter qu'il est tout à fait possible de lancer des clients additionnels si on ajoute un envoi de paquet vers d'autres ports prédéfinis dans la fonction *UDPclientBroadcast2* du dossier *UDPBroadcast* (par exemple sur les ports 5000,6000…).

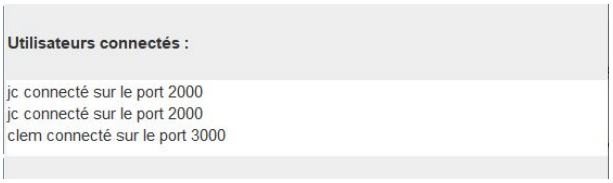
```
InetAddress adresse = InetAddress.getLocalHost();
DatagramPacket outpacket = new DatagramPacket(buffer, buffer.length, adresse, 2000);
DatagramPacket outpacket2 = new DatagramPacket(buffer, buffer.length, adresse, 3000);
DatagramPacket outpacket3 = new DatagramPacket(buffer, buffer.length, adresse, 4000);
//On lui affecte les données à envoyer
outpacket.setData(buffer);

System.out.println("Envoie du broadcast");
//On envoie au(x) serveur(s)
client.send(outpacket);
client.send(outpacket2);
client.send(outpacket3);
```

On envoie un paquet sur chaque port susceptible d'accueillir un client

Cela permet donc de lancer 3 utilisateurs respectivement sur les ports 2000,3000 et 4000 : chaque utilisateur reçoit correctement les réponses des autres utilisateurs connectés (aucune réponse si c'est le premier utilisateurs). De plus, les bases de données locales sont correctement mises à jour lors de la connexion d'autres utilisateurs et ces derniers sont observable depuis la fenêtre "*NewConversation*".



Liste des utilisateurs depuis "JC" lorsqu'il est le seul connecté



Liste des utilisateurs depuis "JC" après la connection d'un utilisateur "Clem"

NB : Puisque l'on expérimente sur la même machine cela signifie que la base de données est partagée donc certains utilisateurs apparaissent plusieurs fois dans la liste.

Lors de la connexion si on utilise “*Refresh Conversation list*” les conversations tenues lors de connexions précédentes apparaissent correctement. Si on veut en créer une nouvelle, on utilise “*NewConversation*”, on tape les informations puis “*Back to Main Frame*” et à nouveau “*Refresh Conversation*” pour la faire apparaître.
Pour commencer à discuter il faut cliquer directement sur la ligne de la conversation : la fenêtre de clavardage apparaîtra (ainsi qu’une deuxième correspondant à l’autre utilisateur puisque l’on est sur la même machine).
Si l’utilisateur d’une conversation précédente n’est pas disponible alors la cliquer sur la conversation n’ouvrira aucune fenêtre chat, indiquant que l’utilisateur est hors-ligne.

Une fois les fenêtres de discussion ouvertes les deux utilisateurs peuvent s’envoyer des messages librement : les messages envoyés s’affichent à gauche et ceux reçus à droite. On peut fermer la conversation en appuyant sur le bouton “*Back to Main Frame*”. Si un des deux utilisateurs d’une conversation quitte la fenêtre de discussion, l’autre pourra continuer à en envoyer tant qu’il ne quitte pas la fenêtre lui-même ! Les messages alors envoyés seront quand même enregistrés dans la base de données centralisés et pourront être visualisés lorsqu’ils voudront continuer leur conversation lors d’une prochaine connexion.

4. Manuel d'utilisation

Lors du lancement de l'application une fenêtre de login apparaît, une base de donnée en local est créée. L'utilisateur doit entrer un nom d'utilisateur ainsi qu'un numéro de port.

Pour effectuer des tests en local (sur la même machine) : vous pouvez lancer jusqu'à 3 clients mais il faut utiliser les ports 2000,3000 et 4000 respectivement !

Si le login n'est pas utilisé on est dirigé vers la page principale sinon un nouveau login est demandé à l'utilisateur. Sur la page principale, l'utilisateur peut se déconnecter, afficher la liste des conversations auxquelles l'utilisateur connecté a participé, ou bien créer une nouvelle conversation. Lorsque l'utilisateur arrive sur la page de création d'une nouvelle conversation, on lui propose la liste des utilisateurs connectés (ceux qui ont répondu au broadcast UDP) ensuite nous avons encore une fois besoin du numéro de port du destinataire.

Pour commencer à dialoguer, l'utilisateur doit afficher les conversations auxquelles il prend part, pour cela il suffit de cliquer sur le bouton "refresh conversation list". Il lui suffit ensuite de cliquer sur la conversation qu'il souhaite ouvrir. La page de conversation s'ouvre aussi sur le client du destinataire. Vous pouvez maintenant dialoguer au sein de cette conversation en tapant du texte dans la section "enter new message" et cliquer sur envoyer. Pour quitter la conversation, l'utilisateur doit cliquer sur le bouton "back to main frame".

Si on a lancé plusieurs clients en local alors lorsqu'un d'eux demande une conversation, 2 fenêtres de dialogue vont s'ouvrir : 1 côté du demandeur (ex: user1 vers user2), 1 autre côté réception (ex: user2 vers user1) afin de notifier l'user ciblé (à la manière d'une infobulle).

5. Installation

Il vous faudra importer les deux bases de données nécessaires au fonctionnement de l'application pour effectuer des tests. Les deux fichiers correspondants sont : LAUNCH/BDD_SETUP/login.sql et LAUNCH/BDD_SETUP/conv_mess.sql. Il vous faudra donc un serveur local mySQL qui tourne sur votre machine (ex : WAMP). Les paramètres de connexions à mySQL sont par défaut (login : root, mdp :), si vous voulez les changer pour les adapter à vos paramètres de connexions , il faut les changer ligne 22 du fichier POO\src\clientLogin\DatabaseLogin.java et ligne 30 du fichier POO\src\clientClavardage\DatabaseConv_mess.java. Une fois connecté à phpMyAdmin il faut sélectionner "nouvelle base de donnée", entrer un nom dans le champ "nom base de données"(de préférence le même que le fichier .sql correspondant), "créer". Puis "importer" dans la base de données qui vient d'être créée, et sélectionner un fichier .sql précédemment cité, faire la même chose pour l'autre fichier .sql.

Afin de lancer le client, il suffit de double cliquer sur le fichier .jar : LAUNCH/DUMAZ_SANCHEZ_poo_project.jar Lors de l'exécution du programme, il vous sera demandé d'autoriser l'application à utiliser des fonctionnalités bloquées par le pare-feu. Ces fonctionnalités sont nécessaires à la mise en place de tcp (utilisation des ports) et à la connexion à la base de données.