

Guide Administrateur

La partie POO de notre projet se situe dans le dossier Chat_System de notre git. Dans celui-ci, vous trouverez :

- Un fichier Chat_system.jar
- Le dossier Interface contenant :
 - src : Le code source séparé en 3 package : Network, Interface et Database
 - lib : Les librairies utilisées
 - out : dossier dans lequel les .jar sont générés

Librairies utilisées

Dans notre projet, nous avons utilisé la version 11.0.5 du JDK ainsi que la librairie mysql. Vous trouverez donc dans le dossier lib, le fichier mysql-connector.jar.

Exécuter l'application

Pour utiliser notre application, il faut télécharger le fichier Chat_System.jar et entrer dans un terminal la commande : `java -jar Chat_System.jar`.

L'utilisation de Linux est conseillé (environnement du GEI) puisque les tests n'ont pas encore été réalisés sur un système Windows. Les ports 3700 et 3600 seront utilisés par l'application.

Utilisation de la base de données

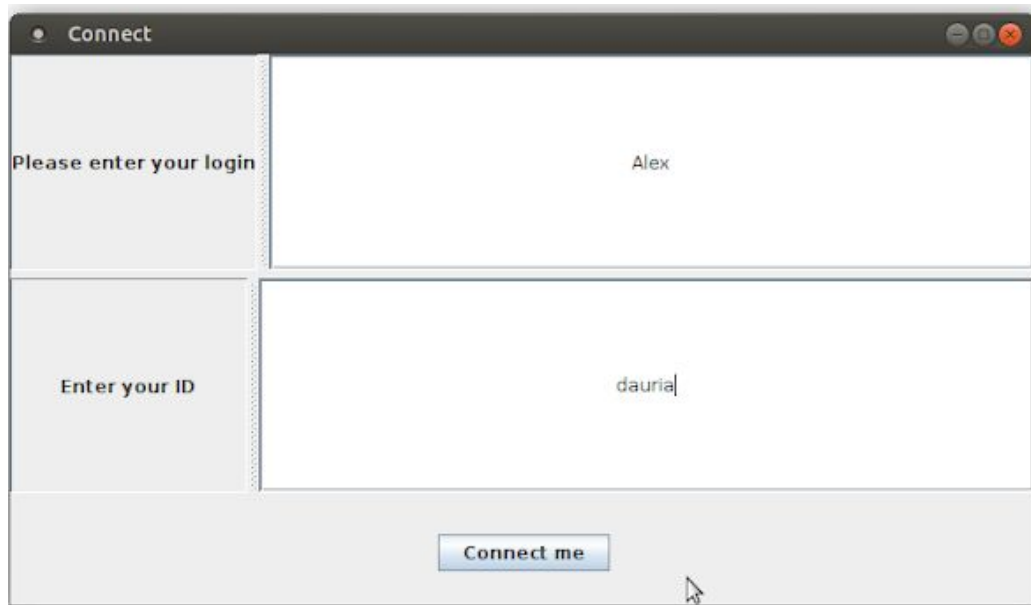
Pour ce projet, un serveur MySQL (v 5.7.28) a été déployé au sein du GEI. Il est possible de changer la base donnée utilisée en changeant dans notre classe BDD.java l'URI, l'identifiant et le mot de passe de la nouvelle base. Cette base doit respecter la structure que nous avons mis en place et décrite dans le dossier DOSSIER_COO/Diagram.

Création d'ID unique pour chaque utilisateurs

Dans notre base de données, la table *user* contient les ID uniques des utilisateurs(type : `varchar(6)`). Pour l'instant, 3 ID ont été rentré : dauria,batkoN et random.

Guide Utilisateur

Au lancement de l'application, une fenêtre s'ouvre. Afin d'accéder au service de communication, il faut rentrer dans cette première fenêtre votre ID unique et un login(=pseudo).

A screenshot of a window titled "Connect". The window has a dark title bar with standard window controls. The main area is divided into two sections. The top section has a label "Please enter your login" on the left and a text input field containing "Alex" on the right. The bottom section has a label "Enter your ID" on the left and a text input field containing "dauria" on the right. At the bottom center of the window is a button labeled "Connect me".

Connect

Please enter your login

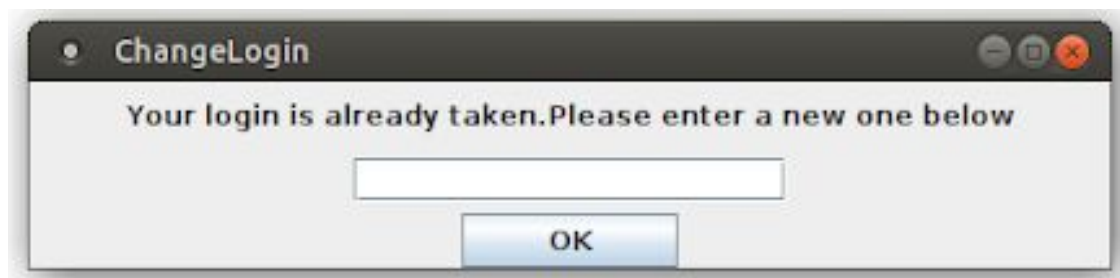
Alex

Enter your ID

dauria

Connect me

Si votre ID existe dans la base de données mais que votre login est déjà utilisé sur le réseau, une fenêtre s'ouvrira pour vous en demander un nouveau.

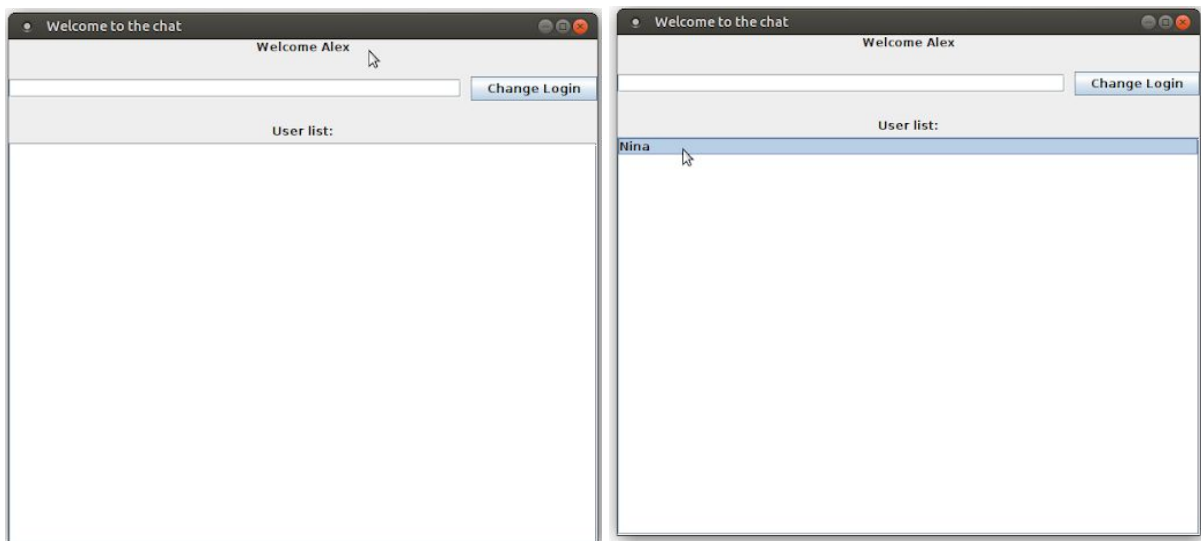
A screenshot of a window titled "ChangeLogin". The window has a dark title bar with standard window controls. The main area contains a message: "Your login is already taken. Please enter a new one below". Below the message is a text input field. At the bottom center of the window is a button labeled "OK".

ChangeLogin

Your login is already taken. Please enter a new one below

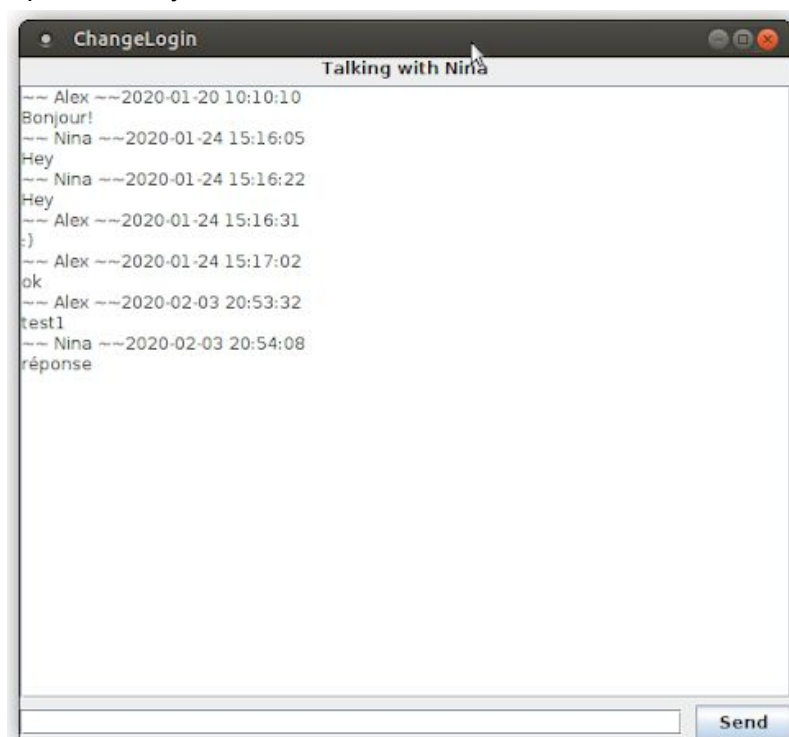
OK

Une fois votre login unique sur le réseau, vous accéderez à la fenêtre principale de l'application. Sur celle-ci, vous y trouverez la liste des utilisateurs présents sur votre réseau. Cette liste sera mise à jour au fur et à mesure que des utilisateurs se connectent ou se déconnectent du réseau.



Vous pouvez à tout moment changer de login via cette fenêtre si celui-ci n'est pas déjà utilisé sur le réseau. Les autres utilisateurs seront notifiés de ce changement.

Pour envoyer un message à un autre utilisateur, vous devez tout d'abord sélectionner un utilisateur dans la liste en double cliquant sur celui-ci. Une nouvelle fenêtre s'ouvrira et l'historique de conversation avec cet utilisateur sera affiché. Vous pouvez entrer votre message dans la zone de texte en bas de la fenêtre et cliquer sur le bouton SEND ou votre touche ENTREE pour l'envoyer.



Explication technique du projet

Vous trouverez dans notre dépôt git hub un dossier Diagram contenant notre diagramme des cas d'utilisation, notre diagramme de classe ainsi que plusieurs diagrammes de séquences.

Notre projet contient 3 packages permettant de séparer les 3 composantes majeures : la partie réseau, la partie interface graphique et la partie base de données. Ces packages sont reliés grâce à la classe *Controller* permettant de faire le lien entre tous les éléments.

Détection des utilisateurs sur un réseau local

Au lancement de l'application, la classe *Controller* va récupérer l'adresse IP de la machine et l'adresse de broadcast correspondant au réseau. Elle va ensuite créer un *ManagerNetwork*. Cette classe va s'occuper de toute la gestion du réseau.

Lors de la connexion, nous avons décidé de broadcast en UDP le message d'entrée dans le réseau. Ce message contient un header défini et le login de l'utilisateur. Tous les autres utilisateurs ayant une classe récupérant les messages UDP reçus, ils pourront alors les traiter, ajouter les nouveaux utilisateurs et leur envoyer un message UDP de retour en unicast vers l'adresse source qu'ils ont reçus.

Lors de la réception du message UDP, un thread est lancé pour le traiter. Nous vérifions si le login transmis existe déjà sur le réseau. Si oui, un message UDP en unicast est envoyé pour demander à l'utilisateur de changer son login. Si non (login unique), un objet *User* est créé permettant de garder en mémoire le login et l'adresse du nouvel utilisateur. Ce *User* est gardé dans une liste.

Lorsqu'un utilisateur se déconnecte, il envoie un message de déconnexion en broadcast UDP sur le réseau local.

L'échange de messages UDP se passe via le port 3700.

Tests : Pour tester la détection des utilisateurs locaux, nous avons utilisés 3 PC sur le même afin d'y connecter 3 utilisateurs. De plus nous avons ajouté un affichage de la liste des utilisateurs connus par chacun à chaque modification de celle-ci.

Scénarios testés:

- 2 utilisateurs avec 2 login différents : vérification de l'envoi du broadcast UDP, de la réception du message et de la réponse en unicast UDP

- 2 utilisateurs avec le même login : vérification l'unicité des login et de l'envoi du message UDP correspondant
- ajout d'un 3ème utilisateur avec un login différent : vérification de la réception de 2 réponses unicast simultanées
- ajout d'un 3ème utilisateur avec un login déjà existant : vérification que les 2 autres utilisateurs ne l'ajoutent pas à leur liste et lui renvoient un message unicast d'erreur de login
- déconnexion d'un utilisateur : vérification de l'envoi du message broadcast et du retrait de cet utilisateur dans la liste de ceux connectés

Gestion des logins

Nous avons vu ci-dessus comment l'unicité des login lors de la connexion est gérée. Mais il est aussi possible de changer son login à tout instant dans la fenêtre principale de notre application. Afin de vérifier si le nouveau login entré est utilisable, il est comparé à la liste des utilisateurs connus. Si aucun ne le possède, nous envoyons un nouveau broadcast UDP sur le réseau pour avertir du changement de login.

A la réception, nous modifions dans la liste des utilisateurs connectés celui dont l'adresse IP correspond à l'adresse source du broadcast.

Tests : 3 utilisateurs connectés sur le réseau et chacun change son login. Nous avons vérifié les listes de chaque utilisateur et si les modifications ont bien été prises en compte.

Authentification

Au lancement de l'application, l'utilisateur doit entrer un ID. Nous avons fait le choix d'identifier chaque personne par un ID de 6 caractères unique. Ces ID sont entrés dans la table *user* de notre base de données avec le login actuel de l'utilisateur. Si celui-ci est déconnecté, le champ contient NULL.

Tests : Nous nous sommes connectés à la base de données via un terminal et nous avons ajoutés 3 ID dans notre table user. Nous avons connectés 3 utilisateur prenant chacun un ID différent. Nous avons vérifié qu'après la connection les login sont bien actualisés dans la table.

Améliorations possibles : Ajout d'un mot de passe pour améliorer la sécurité des comptes. Ajout d'une vérification si l'ID est déjà utilisé sur le réseau.

Envoi/Réception des messages entre utilisateurs

Pour l'échange de message, nous avons décidé d'utiliser TCP. Ainsi après la phase de connection, un thread *Server* est créé pour la réception de message. Il va créer un serveur socket sur le port 3600 qui va attendre une connection. Pour chaque connection demandée, il va créer un thread *TCPListener* qui va traiter le message reçu. Ce message sera transmis au *ManagerNetwork* puis au *Controller*. Si la fenêtre de discussion avec cet utilisateur était ouverte, le message sera affiché.

De plus, à l'ouverture de l'application, le *ManagerNetwork* va créer un objet *Client*. Lorsque l'utilisateur souhaite envoyer un message, le *ManagerNetwork* va récupérer l'adresse IP correspondante et la transmettre avec le message au *Client* pour qu'il l'envoie.

Test : 3 utilisateurs connectés sur le réseau s'échangent des messages en envoyant plusieurs messages à une personne puis en alternant entre les deux personnes. Les messages reçus sont affichés.

Idée d'amélioration : Afin d'éviter de faire plusieurs fois l'échange de SYN/SYNACK avec l'utilisateur distant, il serait peut être intéressant de créer une liste des socket client pour chaque utilisateur distant et d'envoyer les messages via ceux-ci au lieu de rouvrir une connection.

Gestion de l'historique

Pour l'historique comme pour la gestion des ID, nous avons choisi d'utiliser une base de donnée centralisée. Celle-ci est sur un server MySQL du GEI. La connection à celle-ci se fait via notre classe *BDD* dans laquelle l'URI, le port, le login et le mot de passe sont renseignés.

Pour la gestion des messages, nous utilisons la table *message* de notre BDD. Chaque ligne de celle-ci contient : un id unique, l'id de l'envoyeur, l'id du receveur, la date de réception et le contenu du message.

Lorsqu'un utilisateur reçoit un message, il déduit de l'adresse source, l'utilisateur source. Le *Controller* va alors le transmettre à notre classe *BDD* pour que celle-ci ajoute le message à la base de données , même si la fenêtre de discussion avec l'utilisateur source n'était pas ouverte. Le message a été reçu, stocké et sera affiché à la prochaine ouverture de la fenêtre de discussion.

Lorsqu'un utilisateur ouvre une fenêtre de discussion avec un autre utilisateur, une requête est envoyée à notre BDD pour recevoir tous les messages existants entre ces deux utilisateurs rangés par ordre chronologique (liste de *MessageHistory*). La classe

ChatSession s'occupe de l'affichage de ceux-ci pour que chaque message soit précédé de l'expéditeur et de la date de réception.

Tests : Plusieurs messages ont été ajoutés manuellement dans notre BDD sans ordre chronologique et en alternant les utilisateurs parmi les 3 de la table *user*. Nous avons ensuite ouvert la fenêtre de discussion avec un des utilisateurs et vérifié l'ordre des messages affichés.

Améliorations possibles

Afin de respecter au mieux le cahier des charges de notre système, nous devons ajouter la possibilité d'envoyer des fichiers ou des images en plus des messages textuels. Nous pensons que cela est possible en utilisant une classe sérialisable pour nos messages.

Nous pouvons aussi ajouter la détection des utilisateurs à l'extérieur du réseau local. Pour cela, nous pouvons utiliser un serveur de présence tomcat avec un servlet.

Pour que le système soit facilement administré par les techniciens, nous pensons qu'il serait important de créer des ID spécifiques admin qui auraient plus d'accès qu'un simple utilisateur. Ils pourraient par exemple administrer la base de données : ajouter des ID utilisateur, effacer certains messages...