Pierre CLOURY
Thomas PELOUS

4IR-SC

Rapport de projet COO-POO

SOMMAIRE

I - Introduction

Dans le cadre de notre 4ième année de formation à l'INSA (cursus informatique et réseaux), nous avons pour objectif de concevoir une application de chat en ligne sur réseau local. Le but de ce projet est de mettre en application nos connaissances et nos aptitudes sur le langage Java et la conception orienté objet abordée l'année dernière et en début de premier semestre cette année.

Contrairement aux années précédentes, aucune base de code n'est donnée et nous devons développer de A à Z, en total autonomie, une application fonctionnelle, en accord avec le cahier des charges spécifié. Nous avions donc une véritable liberté pour aborder ce projet, a condition bien sûr d'adopter des choix clairs et justifiables.

Ce rapport a justement pour but de présenter et d'argumenter sur les choix de conception effectués. Il présentera aussi l'application final et son ergonomie.

II - Cahier des charges

Voici les extraits du cahier des charges à respecter pour notre projet chat system :

Le système doit permettre un déploiement dans n'importe quelle région du monde. La présence d'un réseau local au sein de l'environnement de déploiement est requis. De la même façon, l'ensemble des participants éligibles au clavardage par l'intermédiaire de ce système devront avoir leur poste de travail situé sur le dit réseau local. D'autre part, le système a besoin d'établir une liaison avec le réseau de télécommunications afin de servir de relais de communication. Le système devra pouvoir être déployé rapidement et sa mise en route devra être rapide sans gêner les autres systèmes déjà existants déployés au sein de la structure.

Fonctionnalités d'utilisation de l'agent

[CdC-Bs-7] Le système doit permettre à l'utilisateur de choisir un pseudonyme avec lequel il sera reconnu dans ses interactions avec le système
[CdC-Bs-8] Le système doit permettre à l'utilisateur d'identifier simplement l'ensemble des utilisateurs pour lesquels l'agent est actif sur le réseau local
[CdC-Bs-9] Le système doit permettre à l'utilisateur de démarrer une session de clavardage avec un utilisateur du système qu'il choisira dans la liste des utilisateurs pour lesquels l'agent est actif sur le réseau local
[CdC-Bs-10] Le système doit garantir l'unicité du pseudonyme des utilisateurs pour lesquels l'agent est actif sur un même réseau local
[CdC-Bs-11] Tous les messages échangés au sein d'une session de clavardage seront horodatés

[CdC-Bs-12] L'horodatage de chacun des messages reçus par un utilisateur sera accessible à celui-ci de façon simple
[CdC-Bs-13] Un utilisateur peut mettre unilatéralement fin à une session de clavardage
[CdC-Bs-14] Lorsqu'un utilisateur démarre un nouvelle session de clavardage avec un utilisateur avec lequel il a préalablement échangé des données par l'intermédiaire du système, l'historique des messages s'affiche
[CdC-Bs-15] L'utilisateur peut réduire l'agent, dans ce cas, celui-ci se place discrètement dans la barre des tâches sous la forme d'une icône lorsque le système d'exploitation permet cette fonctionnalité
[CdC-Bs-16] Le système doit permettre à l'utilisateur de changer le pseudonyme qu'il utilise au sein du système de clavardage à tout moment
[CdC-Bs-17] Lorsqu'un utilisateur change de pseudonyme, l'ensemble des autres utilisateurs du système en sont informés
[CdC-Bs-18] Le changement de pseudonyme par un utilisateur ne doit pas entraîner la fin des sessions de clavardage en cours au moment du changement de pseudonyme

Remarque : nous avons ici répertorié qu'un extrait du cahier des charges. Ce sont les exigences correspondant aux fonctionnalités qui devront être disponible pour l'utilisateur de notre logiciel. Dans la suite de ce compte rendu, nous nous référerons à ces exigences avec les numéros ci-dessus (ex : exigence CdC14).

III - Choix de conceptions

1. Choix du protocole du transport

Pour l'envoi de nos messages entre deux utilisateurs, nous avons d'abord dû choisir entre TCP et UDP. Sachant que notre application est en grande majorité utilisé sur un réseau local, la probabilité de perte liée au fait d'utiliser le protocole UDP est faible voire inexistante. De plus, le protocol UDP est beaucoup plus simple et rapide à implémenter. Il nous permet de proposer une solution qui utilise seulement deux threads : un client, qui envoie les messages aux différents utilisateurs connectés, et un serveur, qui reçoit tous les messages provenant des différents utilisateurs. Le client gérant l'envoi de messages correspond à la classe Client_UDP, le serveur pour ces mêmes messages est Server_UDP. Nous avons aussi deux autres threads correspondant au client permettant d'envoyer un fichier et au serveur permettant de les recevoir. Ce sont respectivement les classes Client_File_Transfert et Server_File_Transfert fonctionnant tous deux en udp aussi.

2. Gestion des utilisateurs connectés

Le fait de choisir une implémentation avec un client et un serveur unique nous a poussé à distinguer les utilisateurs sources (ceux qui envoient le message) au niveau applicatif. Pour cela, nous avons décidé d'envoyer via un socket udp des instances de la classe Message. Cette classe et les attributs que nous voulons envoyer héritent donc de la classe Serializable. Un attribut particulier nommé "type" (String), va permettre de différencier au niveau du serveur l'objectif du message envoyé. Il existe 4 types :

1. NORMAL : envoi d'un message contenant le texte envoyé d'un utilisateur vers le correspondant voulu.

2. BROADCAST : permet d'envoyer un message informant de notre présence aux autres utilisateurs, qui peuvent ainsi mettre à jour leur liste d'utilisateurs connectés. e message est envoyé dès la connexion pour prévenir les autres utilisateurs déjà connectés et connaître qui est connecté.

3. REP_BROADCAST : une machine connecté recevant un message broadcast va répondre, ce qui va permettre à la machine a l'origine de l'envoi en broadcast de savoir qui est connecté et va mettre à jour sa liste d'utilisateurs connectés. Il y a deux différentes listes d'utilisateurs : la liste de type ArrayList>User> contenat tous les utilisateurs connectés et leurs informations importantes (adresse IP etc…) et la JList permettant l'affichage graphique contenant uniquement les pseudos. Cette seconde liste permet de pouvoir sélectionner un utilisateur pour lui envoyer un message.

4. UPDATE : quand un utilisateur souhaite changer de pseudo, un envoi automatique de ce type en broadcast est émis, chaque utilisateur en ligne voit donc le pseudo mis à jour ainsi que sa liste des personnes connectées modifiées.

5. DISCONNECT : lorsqu'un utilisateur souhaite se déconnecter, l'application envoi un message en broadcast de type DISCONNECT. Ainsi, les autres machines connectés peuvent supprimer l'utilisateur automatiquement de la liste des utilisateurs connectés.

Classe Serveur_udp.java : gestion de la reception des messages en fonction de leur type.

```
//SI message normal et qu'un user parmis les users connectés est bien selectioné(=> on pourra afficher le message)
if (m.get_type().equals("NORMAL") && selected_user != null) {

    System.out.println("selected user");
    //Si le message reçu vient de la personne sélectionnée sur la conversation courante on l'affiche
    if (selected_user.get_login().equals(m.get_user_src().get_login())){
        display_zone.append("Message reçu : " + m.get_data() + "\n");
        display_zone.append(m.get_date() + "\n \n");
    }
```

Ci dessus, on vérifie que le message est de type NORMAL et qu'un utilisateur est sélectionné dans la liste des personnes connectées. Si aucun utilisateur n'est sélectionné ou bien si celui sélectionné ne correspond pas à l'émetteur du message alors on ne l'affiche pas.

```
    //Si on reçoit un BROADCAST, alors on répond que nous aussi on est connecté
}else if (m.get_type().equals("BROADCAST")){
    //user_src du message reçu correspond au destinataire du message a renvoyer
    if(m.get_user_src().get_login()!=ulocal.get_login()) {
        System.out.println("avant if BROADCAST ");
        //On prépare un message du type REP_BROADCAST
        Message mes_reponse = new Message("REP_BROADCAST",ulocal,m.get_user_src(),0);
        mes_reponse.set_data("Automatique");//inutile mais avait généré une erreur si pas de texte

        //On prépare le message de réponse à envoyer
        ByteArrayOutputStream baosBroadcast = new ByteArrayOutputStream();
        ObjectOutputStream oosBroadcast = new ObjectOutputStream(baosBroadcast);
        oosBroadcast.writeObject(mes_reponse);
        byte[] buffBroadcast = baosBroadcast.toByteArray();
        DatagramSocket client = new DatagramSocket();

        //On envoie la réponse uniquement à la personne ayant envoyé le broadcast
        InetAddress adresse = InetAddress.getByName(m.get_user_src().get_IP());
        DatagramPacket packetBroadcast = new DatagramPacket(buffBroadcast, buffBroadcast.length, adresse, m.get_user_src().get_port_ecoute());

        //le serveur envoie la réponse
        packet.setData(buffBroadcast);
        client.send(packetBroadcast);

        //Lignes suivantes uniquement pour tester
        System.out.println("avant if BROADCAST ");
        ulocal.display_List();

        //Si le User ayant envoyé le broadcats n'est pas dans la liste des users déjà connectés alors on le rajoute à la liste des Users
        if (ulocal.isInside(m.get_user_src().get_login())!= true && m.get_user_src().get_login()!=ulocal.get_login()) {
            // on rajoute l'utilisateur src du message dans la liste des utilisateurs connectés
            //On l'ajoute à l'arrayList des users connectés de l'utilisateur local
            ulocal.Connected_Users.add(m.get_user_src());
            System.out.println("mise a jour broadcast list 1");
            //Et on l'ajoute dans le model de la JList pour mettre à jour l'affichage
            this.DLM.addElement(m.get_user_src().get_pseudo());
        }
```

Ci dessus, lors de la réception d'un message de type BROADCAST, on renvoie un message de type REP_BROADCAST et on ajoute cet utilisateur dans la liste des personnes connectées (ArrayList et JList).

```
    //Lorsque l'on reçoit une réponse de broadcast, on ajoute l'utilisateur si il n'était pas déjà dans la liste
}else if (m.get_type().equals("REP_BROADCAST")){

    User user_src_RepBrdcst = m.get_user_src();

    //Quelques messages de tests
    System.out.println("Pseudo de la source du message reçu : " + m.get_user_src().get_pseudo());
    System.out.println("type du message : " + m.get_type());
    System.out.println("avant if REPBROADCAST ");
    ulocal.display_List();

    //Si il n'est pas déjà dans liste users connectés, on le rajoute
    if (ulocal.isInside(m.get_user_src().get_login())!= true && m.get_user_src().get_login()!=ulocal.get_login()) {
        ulocal.Connected_Users.add(m.get_user_src());
        this.DLM.addElement(m.get_user_src().get_pseudo());
        System.out.println("mise a jour broadcast list 2");
    }

    //Tests
    System.out.println("Rep broadcast recu \n");
    System.out.println("après if REPBROADCAST ");
    ulocal.display_List();
```

Dans l'image au dessus, si on reçoit un message de type REP_BROADCAST alors on ajoute cet utilisateur à la liste des personnes connectées.

```
    //Si on reçoit un message de type UPDATE, cela signifie qu'un user a changé son pseudo, donc on le met à jour
}else if(m.get_type().equals("UPDATE")) {
    //ON récupère l'index du user dont on vient de recevoir le message dans l'arrayList des uses connectés
    int index_connected_user = ulocal.index_of(m.get_user_src());
    //Grâce à cet index, on récupère l'ancien pseudo de ce user ayant changé de pseudo
    int index_model = DLM.indexOf(ulocal.Connected_Users.get(index_connected_user).get_pseudo());
    //Et on met à jour dans le model pour changer le pseudo dans la JList des Users conectés
    DLM.set(index_model,m.get_user_src().get_pseudo());
    //Puis on modifie le pseudo dans l'ArrayList des Users connectés propre au user local
    ulocal.Connected_Users.get(index_connected_user).set_pseudo(m.get_user_src().get_pseudo());

    //Test
    System.out.println("reçu nouveau pseudo pour :" + m.get_user_src().get_login() + " new pseudo = " + m.get_user_src().get_pseudo());
```

Lorsque l'on reçoit le message de type UPDATE, on modifie les listes des users connectées pour y mettre le bon pseudo. Ci-dessous, lorsqu'on reçoit un message DISCONNECT, on supprime l'utilisateur des listes des utilisateurs connectés.

```
    //Lorsque quelqu'un se déconnecte, on le supprime de l'arrayList et la JList des users connectés
}else if (m.get_type().equals("DISCONNECT")) {
    //Test
    System.out.println("User disconnected : " + m.get_user_src().get_pseudo());

    //On désélectionne le user au cas ou on serait en train de lui parler au moment ou il se déconnecte
    //permet d'éviter d'avoir la JList pointant sur un pseudo ayant disparu suite à une déconnexion
    JLUsers.setSelectedIndex(0);

    //on récupère l'index de la personne venant de se décinnecter dans l'arraylist du user_local
    int i = ulocal.index_of(m.get_user_src());
    //Test
    System.out.println("index : " + i);
    //On supprime cet utilisateur de l'arayList du user local
    ulocal.Connected_Users.remove(i);

    //On supprime aussi dans le model relié à la JList pour mettre à jour l'affichage des users connectés
    this.DLM.removeElement(m.get_user_src().get_pseudo());

    //Test
    ulocal.display_List();
}
```

Ainsi, lorsque qu'un utilisateur se connecte, l'Interface_Accueil lancée envoie un message de type BROADCAST via une instance de Client_UDP qui permet aux autres utilisateurs de mettre à jour leur liste d'utilisateurs connectés. Chaque machine ayant reçu un message de type BROADCAST renvoie un message de type REP_BROADCAST à la machine a l'origine de l'envoi du broadcast. Ainsi, celle-ci elle peut elle aussi mettre à jour sa liste d'utilisateurs connectés directement. Tout cela se fait automatiquement, l'utilisateur n'a rien à faire.

De plus, lorsque qu'un utilisateur se déconnecte, l'application envoie en broadcast un message de type DISCONNECT qui permet aux autres utilisateurs de mettre à jour leur liste d'utilisateurs connectés.
Ce fonctionnement permet de respecter l'exigence du cahier des charges 8 :

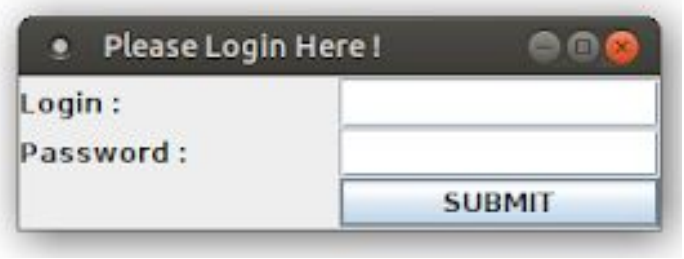
"Le système doit permettre à l'utilisateur d'identifier simplement l'ensemble des utilisateurs pour lesquels l'agent est actif sur le réseau local"

3. Gestion de l'affichage

La gestion de l'affichage est relativement simple. L'interface graphique a entièrement été réalisée en JavaSwing. Tout d'abord notre classe "Main" va lancer l'interface de connexion. Puis une fois la connexion réussit, cette même interface va directement lancer l'interface d'accueil (permettant de commencer le chat) ou bien l'interface Administrateur si l'on a rentré le login et le mot de passe à "Admin".
L'interface d'accueil lancera à son tour la fenêtre permettant de modifier son pseudo comme demandé dans le cahier des charges.

3.1 Interface de connexion - Connection_Interface

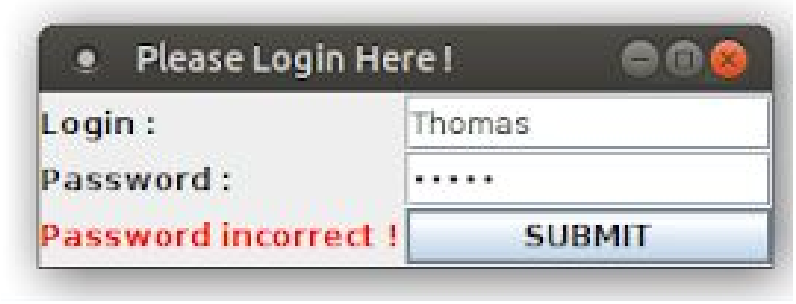
Cette classe étend la classe JFrame et l'interface ActionListener. Elle est principalement composé de JLabel, JtextField et un JBouton permettant d'afficher une fenêtre comme ceci :



Fenêtre de connexion

Le password est entré dans un JPassword permettant une confidentialité au moment de la connexion. L'utilisation de mot de passe n'était pas forcément demandé dans le cahier des charges mais nous trouvions intéressant de l'implémenter.

Si le mot de passe entré n'est pas le bon, un message prévient l'utilisateur comme montré ci-dessous:

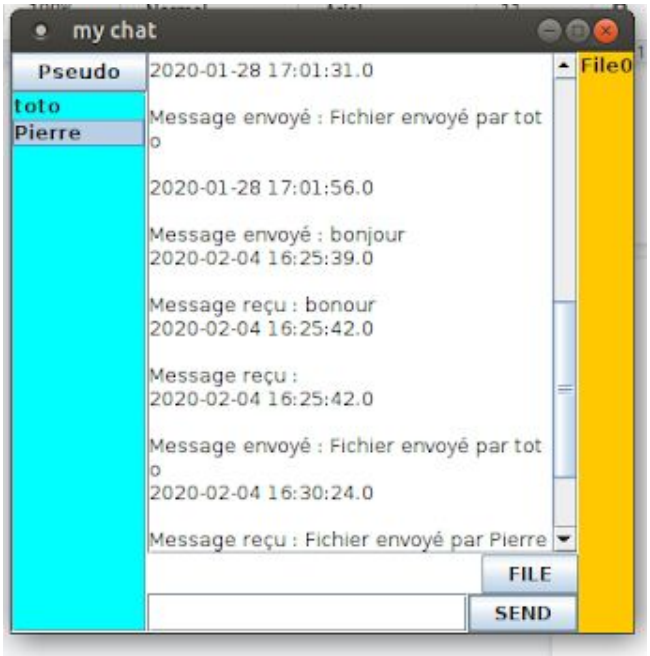


Exemple message connexion échouée

L'utilisateur doit donc retenter la connexion jusqu'à avoir le bon mot de passe.

3.2 Interface de chat - Interface_Accueil

L'interface accueil est le point central de notre application car elle va lancer la partie graphique du chat mais aussi un Client et un Server udp pour l'envoi de texte et un autre Client et Serveur udp pour l'envoi de fichier. Voici son visuel une fois connecté:



Interface_Accueil

On distingue dans cet interface 3 principales zones, chacune étant un JPanel, le tout imbriqué dans un plus gros JPanel.

Panel des utilisateurs connectés :

Ce JPanel est situé sur la gauche. Il est constitué d'un JButton ainsi qu'une JListe.
La JListe permet l'affichage des utilisateurs actuellement connecté à l'application. On retrouve dans cette liste les **pseudos** des utilisateurs seulement (chaque utilisateur n'a pas à connaître le login des autres). Si on sélectionne un utilisateur, on accède aux messages que nous lui avions envoyé auparavant. Sur l'image, l'utilisateur Pierre est sélectionné.
Le JButton "Pseudo" permet d'ouvrir une fenêtre permettant de changer le pseudo de l'utilisateur. Nous détaillerons cette fenêtre plus tard.

Panel de conversation :

Ce JPanel est le Panel au milieu de l'écran. Il regroupe un JTextArea permettant d'afficher les messages, un autre JTextArea collé à un JButton pour l'envoi de fichiers ainsi qu'un JTextField avec un JButton permettant l'envoi de texte cette fois-ci.
Pour le JtextArea de l'envoi de fichiers, il suffit de Glisser-déposer le fichier et on obtient son "chemin" prêt à être envoyé grâce au bouton "FILE".
Le JTextField permet l'envoi de texte tout simplement pour dialoguer avec un utilisateur.

Panel des fichiers :

Ce Panel, situé sur la droite, affiche le fichier que nous venons de recevoir pour que nous puissions l'ouvrir (Attention fichier limité à **60ko**). Dans l'image ci-dessus, un seul fichier a été reçu. Un message texte est envoyé à l'utilisateur pour le prévenir aussi puisque le JTextArea affichant les messages ne permet pas l'affichage d'un fichier.

3.2 Interface de changement de pseudos - Change_Pseudo_Window

Cette fenêtre est inspiré de la fenêtre de connexion vu précédemment. Un message s'affiche si le pseudo est déjà pris. Voici le visuel :

Gauche : fenêtre pour changer le pseudo *Droite : Pseudo déjà utilisé*

Une fois le Button SEND sélectioné, on va vérifier dans la base de donnée que ce pseudo n'est pas déjà pris pour toujours avoir un **pseudo unique.**
Une fois le pseudo changé, un broadcast est envoyé et toutes les listes des utilisateurs connectés se mettent à jour.

3.3 Interface Administrateur - Admin_Interface

Pour accéder à cet interface, il faut se connecter avec le login et le mot de passe égaux à “Admin”. On accède ainsi à cet interface :

A gauche : Interface de l’administrateur *A droite: Tentative de création avec un login déjà pris*

Sur la gauche nous avons une JListe permettant l’affichage de tous les utilisateurs présents dans la base de donnée. Lorsque l’on en sélectionne un, ici Jean, on voit toutes ses informations. En haut de l’écran, on peut entrer un nouvel utilisateur à créer. La première case est le login, la seconde le password, la troisième le pseudo (nous aurions dû mettre des Label pour éviter toute erreur ais il s’agit du même ordre que l’affichage des informations dessous.
Le bouton en bas pour désactiver un utilisateur n’a pas été implémenté car ce n’était pas demandé et nous avons manqué de temps. Pour le faire fonctionner, il suffit de rajouter une colonne “Available/Disable” dans la base de donnée pour chaque utilisateur et suivant ce champ, autoriser l’utilisateur à se connecter ou non lors de la connexion (pourrait être utile lorsqu’un utilisateur quitte l’entreprise par exemple).
N.B : Pour changer le password, l’administrateur doit le faire dans la base de donnée directement.

4. Gestion de la base de donnée

Pour se connecter à la bdd, il faut entrer la commande dans un terminal
$ mysql -h srv-bdens.insa-toulouse.fr -u tpservlet_03 -p
avec comme mot de passe : phahNgo4 puis utiliser la bdd nommée tpservlet_03.

Le cahier des charges spécifie clairement que les utilisateurs doivent avoir accès à un historique pour chaque session de clavardage :

[CdC-Bs-14] Lorsqu’un utilisateur démarre un nouvelle session de clavardage avec un utilisateur avec lequel il a préalablement échangé des données par l’intermédiaire du système, l’historique des messages s’affiche

Pour satisfaire cette exigence, nous avons implémenter une base de donnée contenant deux tables : Messages et Users.

```
mysql> select * from Users
    -> ;
+---------+--------+----------+---------+
| id_user | login  | password | pseudo  |
+---------+--------+----------+---------+
|       1 | Thomas | Thomas   | toto    |
|       2 | Pierre | Pierre   | Pierre  |
|       3 | Jean   | Jean     | jeannot |
|       4 | Gerard | Gerard   | gege    |
|       5 | Admin  | Admin    | Admin   |
+---------+--------+----------+---------+
```

Ci-dessus : Table Users - Ci-dessous : Table Messages

```
mysql> select * from Messages
    -> ;
+--------+---------+---------+--------+-----------------------------------------+---------------------+
| id_msg | id_user | id_dest | type   | data                                    | date_message        |
+--------+---------+---------+--------+-----------------------------------------+---------------------+
|    117 |       2 |       2 | NORMAL | salut toi ?                             | 2020-01-17 12:00:28 |
|    118 |       1 |       1 | NORMAL | vrvr                                    | 2020-01-17 17:08:12 |
|    119 |       1 |       1 | NORMAL | file:///home/cloury/Bureau/palmiers.jpeg | 2020-01-17 17:11:33 |
|    120 |       1 |       2 | NORMAL | BOnjour                                 | 2020-01-28 14:08:14 |
|    121 |       2 |       1 | NORMAL | ah                                      | 2020-01-28 14:08:25 |
|    122 |       1 |       1 | NORMAL |                                         | 2020-01-28 15:12:07 |
|    123 |       1 |       2 | NORMAL | bonjour                                 | 2020-01-28 15:56:58 |
|    124 |       2 |       1 | NORMAL |                                         | 2020-01-28 15:59:01 |
|    125 |       2 |       1 | NORMAL |                                         | 2020-01-28 15:59:05 |
|    126 |       2 |       1 | NORMAL | cououc                                  | 2020-01-28 15:59:16 |
|    127 |       2 |       1 | NORMAL | rare                                    | 2020-01-28 16:30:59 |
|    128 |       2 |       1 | NORMAL | salu toto  a marche                     | 2020-01-28 16:31:12 |
|    129 |       1 |       2 | NORMAL | Fichier envoyé par toto
                | 2020-01-28 17:01:31 |
|    130 |       1 |       2 | NORMAL | Fichier envoyé par toto
                | 2020-01-28 17:01:56 |
+--------+---------+---------+--------+-----------------------------------------+---------------------+
14 rows in set (0.01 sec)
```

Ainsi, un message est stocké grâce à l'ID de l'utilisateur émetteur (id_user ci-dessus), et à l'ID de l'utilisateur destinataire. Chaque fois qu'un message de type NORMAL est envoyé, le message est stocké sur la base de donnée grâce à une méthode de la classe ConnectionDB (classe qui gère toutes les interactions avec la base de donnée).

```java
//méthode permettant d'insérer le message dans la base de donnée
public boolean Insert_messBDD(Message message) throws SQLException {
    // dans la bdd on ne stocke que l'ID du User source et dest dans la table message ainsi que la dte et la data (texte important constituant le message)
    int id_src = message.get_user_src().get_id();
    int id_dest = message.get_user_dst().get_id();
    String type = message.get_type();
    String data = message.get_data();
    String date = message.get_date();
    String sql="INSERT INTO Messages (id_user, id_dest, type, data, date_message) VALUES ("  + id_src + ", " + id_dest + ", \"" + type + "\", \"" + data + "\", " +"\""+ date + "\");";
    Statement smt = con.createStatement();
    smt.executeUpdate(sql);
    return true;
}
```

Méthode permettant l'insertion d'un message dans la base de donnée.

De plus, lorsque l'utilisateur U1 clique sur l'utilisateur U2 avec lequel il souhaite communiquer, la méthode *get_historique* permet de rapatrier sous forme d'un String l'ensemble des messages dont l'utilisateur source est U1, est l'utilisateur destinataire est U2 ou inversement (automatiquement trié par date).

```java
//methode permettant de charger l'historique de la conversation entre le user local et le user dest dest
public String get_historique(User user_local, User user_dest) throws SQLException {

    String resultat ="";
    //On sélectionne tous les messages ayant les bons destinataires/sources et on créé un String contenant tous ces messages
    String sql="select * from Messages where (id_user = "  + user_local.get_id() + " and id_dest = " + user_dest.get_id() + ") OR (id_user = "  + user_dest.get_id() + " and id_dest = " + user_local.get_id() + ");";
    Statement smt = con.createStatement();
    ResultSet rs = smt.executeQuery(sql);
    while (rs.next()) {
        String data = rs.getString("data");
        String date = rs.getString("date_message");
        if(Integer.parseInt(rs.getString("id_user"))==user_local.get_id()) {
            resultat = resultat + "Message envoyé : " + data + "\n" + date + "\n \n";
        }else {
            resultat = resultat + "Message reçu : " + data + "\n" + date+ "\n \n";
        }
    }
    return resultat;
```

Méthode get_historique permettant de rapatrier l'utilisateur entre deux utilisateurs

La classe ConnectionDB permet aussi de gérer la base de donnée de façon plus générale : changer le pseudo, le login ou le mot de passe d'un utilisateur, créer un utilisateur, …

Remarque : *le fichier utilisé pour créer la base de donnée est donné en annexe.* Il y a quelques différences avec ce que nous avons car il avait été implémenté avant que nous ayons accès à la base de donnée.

IV - Manuel d'utilisation

Voici un copier-coller du README.txt présent dans le github qui explique comment utiliser l'application, aussi bien pour un administrateur que pour un utilisateur lambda :

Déploiement de l'application

Notre application n'a pas pu être mise sous format .jar à cause de l'utilisation d'une base de donnée centralisée. Pour lancer notre application il suffit donc de faire la commande suivante dans un terminal depuis le dossier racine de notre arborescence (COO-POO) :

- ./KitChat
- ou simplement double cliquer sur le fichier KitChat puis sur démarrer si votre système d'exploitation le propose.

Une fois la seconde commande lancée, on se situe sur l'écran de connexion à l'application.

-

Une fois la seconde commande lancée, on se situe sur l'écran de connexion à l'application.

Connexion en tant qu'administrateur

Entrer Login : Admin Mot de passe : Admin

Ainsi vous pouvez rajouter un utilisateur ou bien voir les informations sur un utilisateur en cliquant sur la liste de gauche présentanrt l'ensemble des utilisateurs (le bouton de désactivation d'un utilisateur n'est pas activé).

Connexion en tant qu'utilisateur

Entrer un des login/mot de passe présent dans la base de donnée à savoir : Thomas/Thomas, Pierre/Pierre, Gerard/Gerard, Jean/Jean.

Possibilité aussi de créer un User depuis l'interface admin et ensuite se connecter avec celui-ci.

Utilisation de la parti clavardage

L'interface de chat est très simple :

- Sur la gauche, on voit la liste des users connectés. Un click sur l'un d'eux affiche la conversation que nous avons eu la dernière fois avec cet utilisateur et on peut lui parler.
- Au milieu on a 3 zones; la fenêtre de chat, la zone de DragNDrop de fichier et le bouton pour envoyer le fichier puis en dessous la zone pour entrer le texte a envoyé et le bouton d'envoi du texte.
- Sur la droite, si aucun fichier n'est reçu il n'y a rien mais quand un fichier est reçu, il s'affiche dans une liste (ressemblante avec celle des utilisateurs connectés) et on peut cliquer dessus pour l'ouvrir (fichier limités à 60Ko).

Déconnexion

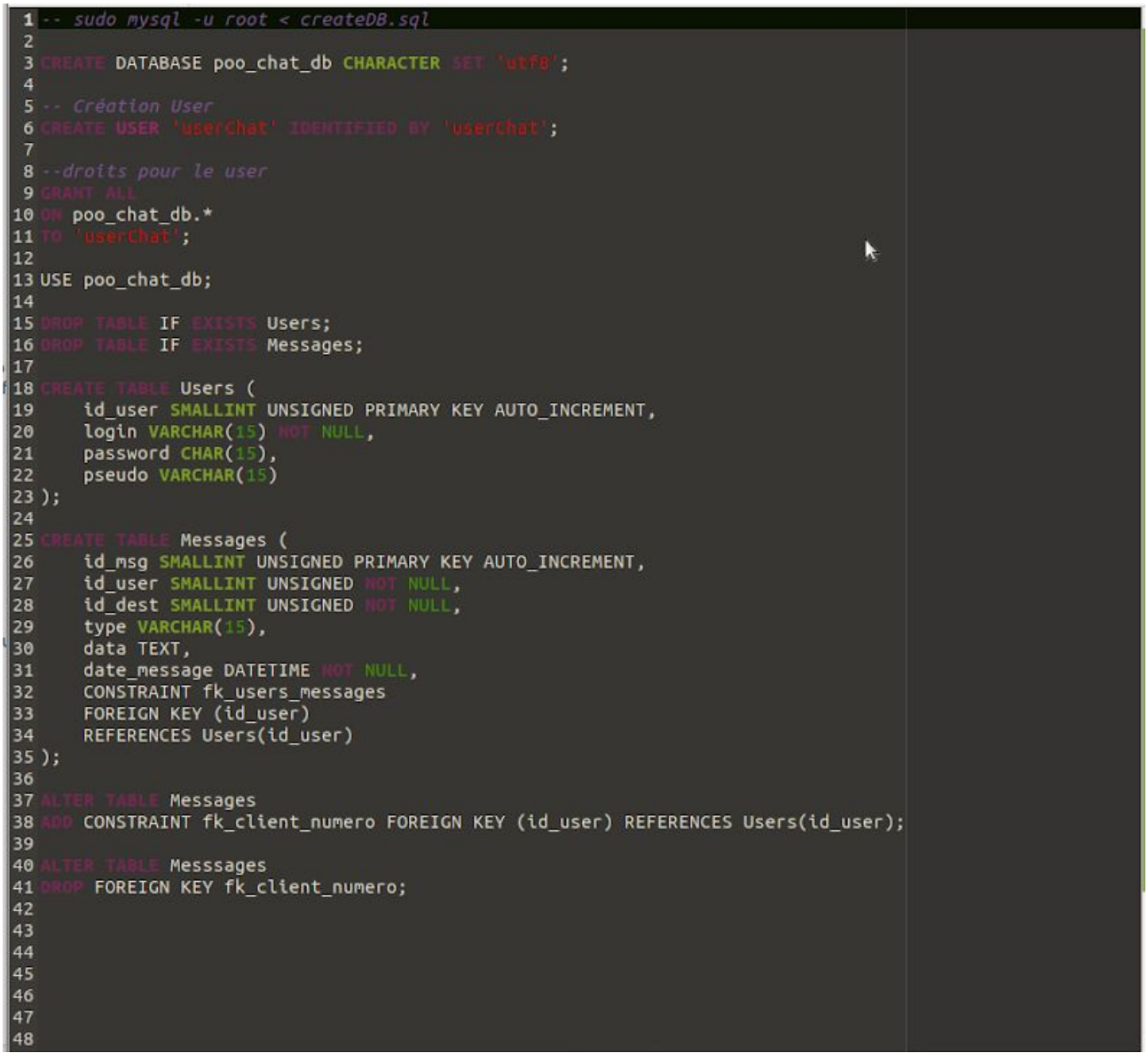
La déconnexion se fait automatiquement en quittant l'application en cliquant sur la petite croix de fermeture de fenêtre.

V - Conclusion

Ce projet nous a permis de mettre en application nos connaissances en programmation et conception orienté objet. Ce fut la première fois que nous avions a mener un projet aussi complexe en autonomie, de A à Z. Nous avons pu découvrir toutes les notions abordées lors du cours Java du premier semestre telles que l'envoi de messages, la gestion d'une interface graphique ainsi que la liaison entre une application Java et une base de donnée MySQL. Si nous avions eu plus de temps, nous aurions amélioré l'interface graphique et implémenté le servlet pour détecter les utilisateurs connectés via l'internet. Nous aurions aussi continué à creuser l'envoi de fichier pour des fichiers plus gros qu'actuellement.

VI - Annexes

Fichier ayant servi pour créer la base de donnée. Cela avait été fait pour notre base de donnée sur nos ordinateurs donc les identifiants de connexion à la base de donnée ne sont pas bons mais l'idée générale de la bdd actuelle est identique.



```sql
-- sudo mysql -u root < createDB.sql

CREATE DATABASE poo_chat_db CHARACTER SET 'utf8';

-- Création User
CREATE USER 'userChat' IDENTIFIED BY 'userChat';

--droits pour le user
GRANT ALL
ON poo_chat_db.*
TO 'userChat';

USE poo_chat_db;

DROP TABLE IF EXISTS Users;
DROP TABLE IF EXISTS Messages;

CREATE TABLE Users (
    id_user SMALLINT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    login VARCHAR(15) NOT NULL,
    password CHAR(15),
    pseudo VARCHAR(15)
);

CREATE TABLE Messages (
    id_msg SMALLINT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    id_user SMALLINT UNSIGNED NOT NULL,
    id_dest SMALLINT UNSIGNED NOT NULL,
    type VARCHAR(15),
    data TEXT,
    date_message DATETIME NOT NULL,
    CONSTRAINT fk_users_messages
    FOREIGN KEY (id_user)
    REFERENCES Users(id_user)
);

ALTER TABLE Messages
ADD CONSTRAINT fk_client_numero FOREIGN KEY (id_user) REFERENCES Users(id_user);

ALTER TABLE Messsages
DROP FOREIGN KEY fk_client_numero;
```