INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

RAPPORT DE PROJET
CLAVARDAGE

Par Viktor Adrian Mouton et Aubry Dubois

4 IR 2

2021-2022

1. Introduction

Le projet clavardage que l'on a fait consistait à créer un système applicatif de communication entre deux machines sous un même réseau local. Un cahier des charges nous a été fourni, en décrivant en détail ce qui était attendu d'un point de vue de l'utilisation, mais également sur la configuration, la capacité du nombre d'utilisateurs maximum, la taille de l'application...

Avant de débuter la réalisation de ce projet, il faut savoir ce qu'il faut implémenter, du plus général au plus spécifique. Il nous faut donc concevoir pour bien organiser l'implémentation. On présentera dans un premier temps les différents diagrammes UML conçus pour le projet. Ils nous permettront de savoir où commencer.

Ensuite, nous présenterons les choix pris pour la réalisation de ce projet (technologies,librairies,solution à un problème...) et qu'est-ce que l'on a implémenté. Ce projet a été réalisé avec l'IDE Eclipse en Java, un langage orienté objet, utile pour la création d'application complexe.

Enfin, on expliquera nos choix concernant ce qui entoure le projet, comme la méthode d'organisation de l'implémentation, les tests automatiques créés et le déploiement de l'application.

2. Partie conception

a. Cas d'utilisation

Le diagramme de cas d'utilisation permet de retranscrire visuellement ce que le cahier des charges nous demande de faire, mais surtout d'avoir une idée de ce qu'un utilisateur lambda pourra faire avec l'application. Nous avons également représenté ce qu'un admin pouvait faire avec l'application en termes de configuration. Tout ce qu'un utilisateur peut faire est représenté ici, mais aussi ce que l'application fera en conséquence de ce que l'utilisateur choisis de faire (les cases pointées par des flèches <<includes>>). Cela nous permet d'avoir une idée de la dynamique de l'application, de ce que l'utilisateur ne verra pas.

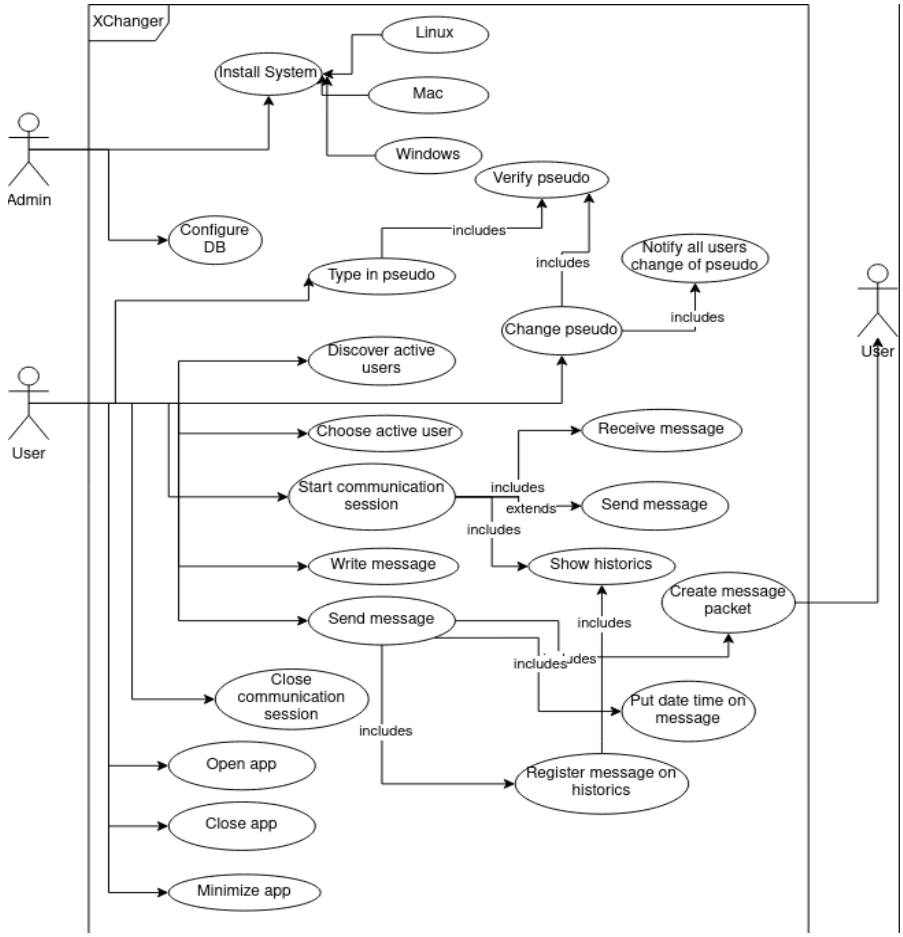


Diagramme de cas d'utilisation

b. Diagramme de séquence

Le diagramme de séquence ci-dessous retranscrit une action spécifique d'un cas d'utilisation, et représente les interactions entre les différents composants de l'application lors de cette action, composants qui seront précisés dans le diagramme de classe (la classe "Anonymous" correspond à un thread).

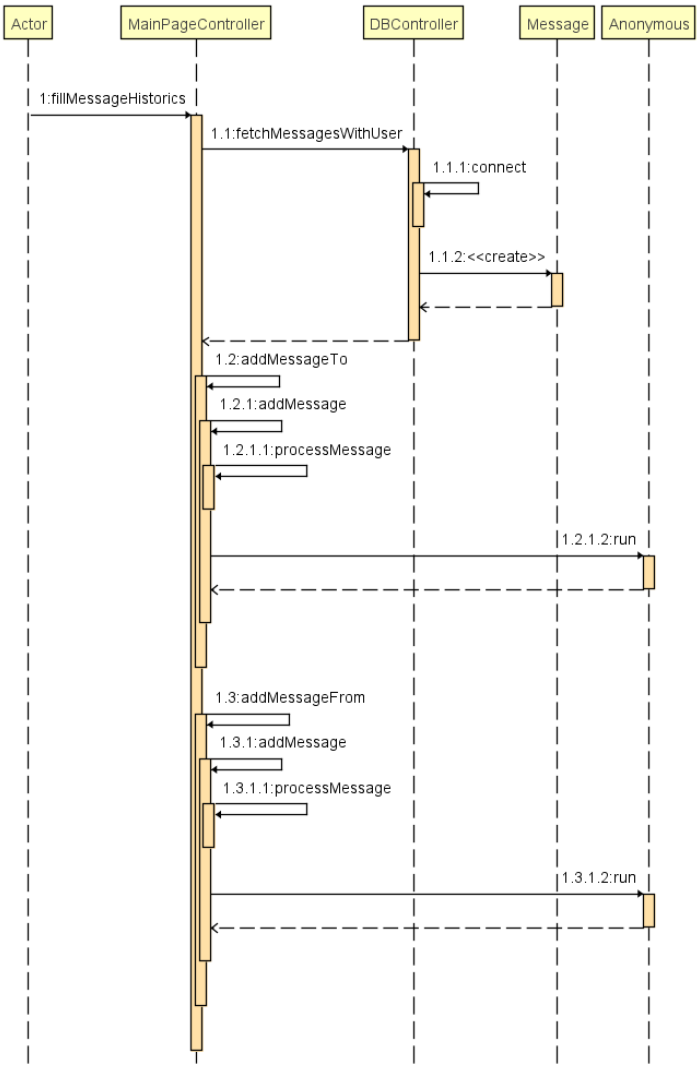
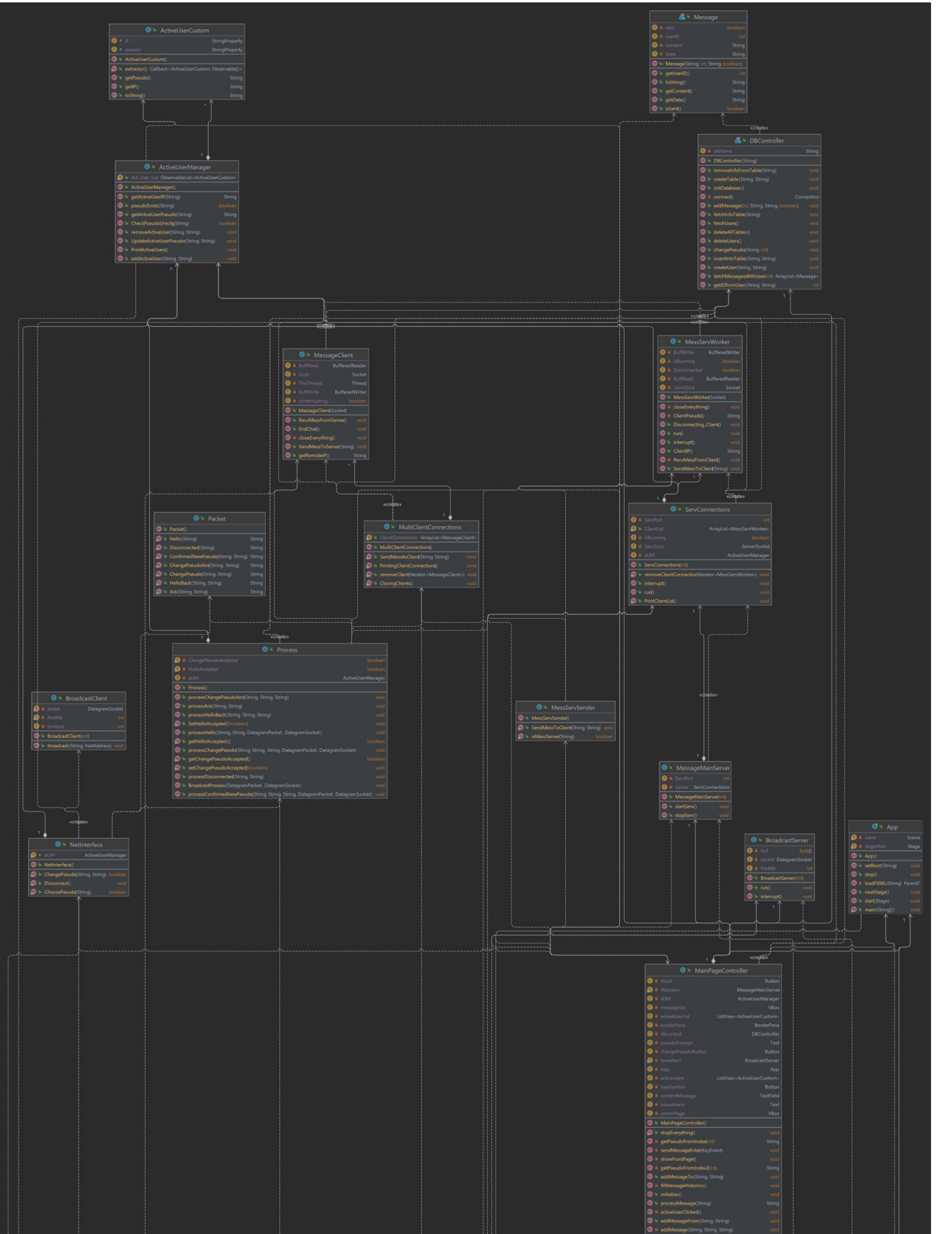


Diagramme de séquence qui retranscrit l'affichage de l'historique de messages d'une conversation

c. Diagramme de classe

Le diagramme de classe permet en détail de visualiser les interactions entre les différentes composantes de l'application. On peut distinguer plusieurs grands sous-ensembles de l'application. Tout d'abords nous avons la partie "Frontend" avec les controllers (MainPageController et FrontPageController), la base de données avec DBController et ses classes périphériques, puis la partie réseau avec une sous-partie s'occupant de la partie broadcast de l'application, notamment lors de la connexion de l'application et le changement de pseudo, la partie s'occupant de la communication entre deux individus, avec l'aspect client/server.


ActiveUserCustom
IP StringProperty
pseudo StringProperty
ActiveUserCustom()
extractor() Callback<ActiveUserCustom, Observable[]>
getPseudo() String
getIP() String
toString() String
Message
sent boolean
userID int
content String
Date String
Message(String, int, String, boolean)
getUserID() int
toString() String
getContent() String
getDate() String
isSent() boolean
«create»
ActiveUserManager
Act_User_List ObservableList<ActiveUserCustom>
ActiveUserManager()
getActiveUserIP(String) String
pseudoExists(String) boolean
getActiveUserPseudo(String) String
CheckPseudoUnicity(String) boolean
removeActiveUser(String, String) void
UpdateActiveUserPseudo(String, String) void
PrintActiveUsers() void
addActiveUser(String, String) void
DBController
dbName String
DBController(String)
removeInfoFromTable(String) void
createTable(String, String) void
initDatabase() void
connect() Connection
addMessage(int, String, String, boolean) void
fetchInfoTable(String) void
fetchUsers() void
deleteAllTables() void
deleteUsers() void
changePseudo(String, int) void
insertIntoTable(String, String) void
createUser(String, String) void
fetchMessagesWithUser(int) ArrayList<Message>
getIDfromUser(String, String) int
«create»
MessageClient
BuffRead BufferedReader
Sock Socket
ThisThread Thread
BuffWrite BufferedWriter
isInterrupting boolean
MessageClient(Socket)
RecvMessFromServer() void
EndChat() void
closeEverything() void
SendMessToServer(String) void
getRemoteIP() String
MessServWorker
BuffWrite BufferedWriter
isRunning boolean
Disconnected boolean
BuffRead BufferedReader
clientSock Socket
MessServWorker(Socket)
closeEverything() void
ClientPseudo() String
Disconnecting_Client() void
run() void
interrupt() void
ClientIP() String
RecvMessFromClient() void
SendMessToClient(String) void
«create»
Packet
Packet()
Hello(String) String
Disconnected(String) String
ConfirmedNewPseudo(String, String) String
ChangePseudoAns(String, String) String
ChangePseudo(String, String) String
HelloBack(String, String) String
Ack(String, String) String
«create»
MultiClientConnections
ClientConnections ArrayList<MessageClient>
MultiClientConnections()
SendMessAsClient(String, String) void
PrintingClientConnections() void
removeClient(Iterator<MessageClient>) void
ClosingClients() void
ServConnections
ServPort int
ClientList ArrayList<MessServWorker>
isRunning boolean
ServSock ServerSocket
aUM ActiveUserManager
ServConnections(int)
removeClientConnection(Iterator<MessServWorker>) void
interrupt() void
run() void
PrintClientList() void
«create»
Process
ChangePseudoAccepted boolean
HelloAccepted boolean
aUM ActiveUserManager
Process()
processChangePseudoAns(String, String, String) void
processAck(String, String) void
processHelloBack(String, String, String) void
SetHelloAccepted(boolean) void
processHello(String, String, DatagramPacket, DatagramSocket) void
getHelloAccepted() boolean
processChangePseudo(String, String, String, DatagramPacket, DatagramSocket) void
getChangePseudoAccepted() boolean
setChangePseudoAccepted(boolean) void
processDisconnected(String, String) void
BroadcastProcess(DatagramPacket, DatagramSocket) void
processConfirmedNewPseudo(String, String, String, DatagramPacket, DatagramSocket) void
BroadcastClient
socket DatagramSocket
PortNb int
timeout int
BroadcastClient(int)
broadcast(String, InetAddress) void
«create»
MessServSender
MessServSender()
SendMessToClient(String, String) void
isMessServer(String) boolean
«create»
MessageMainServer
ServPort int
Server ServConnections
MessageMainServer(int)
startServ() void
stopServ() void
«create»
NetInterface
aUM ActiveUserManager
NetInterface()
ChangePseudo(String, String) boolean
Disconnect() void
ChoosePseudo(String) boolean
BroadcastServer
buf byte[]
socket DatagramSocket
PortNb int
BroadcastServer(int)
run() void
interrupt() void
App
scene Scene
stageMod Stage
App()
setRoot(String) void
stop() void
loadFXML(String) Parent?
nextStage() void
start(Stage) void
main(String[]) void
«create»
MainPageController
flopit Button
MainServ MessageMainServer
aUM ActiveUserManager
messagelist VBox
activeUserList ListView<ActiveUserCustom>
borderPane BorderPane
dbcontrol DBController
pseudoForeign Text
changePseudoButton Button
broadserv BroadcastServer
App App
activeusers ListView<ActiveUserCustom>
backbutton Button
contentMessage TextField
pseudotext Text
centerPage VBox
MainPageController()
stopEverything() void
getPseudoFromIndex(int) String
sendMessageEnter(KeyEvent) void
showFrontPage() void
getPseudoFromIndex2(int) String
addMessageTo(String, String) void
fillMessageHistorics() void
initialize() void
processMessage(String) String
activeUserClicked() void
addMessageFrom(String, String) void
addMessage(String, String, String) void

3. Partie réalisation

a. L'interface utilisateur

Beaucoup de choix s'offrait à nous concernant les librairies que l'on pouvait utiliser pour faire une application fonctionnelle sur machine (Swing, Javafx...). On pouvait faire le choix de faire un site sur internet pour la communication. Le choix a été pris de faire une application à installer sur chaque machine, et pour faire cela on a utilisé JavaFX, un framework open-source efficace. Avec ce framework, on pouvait utiliser en complément un logiciel de conception de l'interface utilisateur qui est Scene Builder. Ce logiciel permet sans code de créer des vues facilement, et avec une grande liberté d'affichage. Cette vue génère ensuite un document sous un format particulier, le fxml, l'équivalent de html en JavaFX, qui peut ensuite être utilisé par notre code du projet sans problème. Il est simple de lier par exemple une action d'un utilisateur sur la vue (par ex. cliquer sur un bouton) par une fonction dans notre code.

Par ailleurs, il est simple de lier une scène (ou un document fxml) à une classe contrôleur java de la page. C'est dans cette classe que l'on pourra lier chaque fonction à une action d'une page. On aura besoin de seulement deux pages pour l'application, une où l'on choisira un pseudo, et la deuxième où l'on pourra choisir un utilisateur actif qu'on a envie de communiquer avec dans une liste, et lui envoyer des messages. On aura alors besoin que de deux contrôleurs.

JavaFX nous permet en plus de changer une partie de la vue sans réinitialiser toute celle-ci. Ainsi, lorsqu'on choisit un utilisateur actif, on change la vue pour montrer l'historique avec celui-ci.

b. La base de données

Le cahier des charges nous informe qu'il nous faudra montrer l'historique des messages envoyés et reçus d'un utilisateur lorsque l'on voudra communiquer avec lui. Cela veut dire qu'on devra sauvegarder de l'information même quand l'application ne tourne pas. Il nous faut donc concevoir une base de données. Encore une fois, beaucoup de choix s'offrait à nous sur quelle technologie il fallait utiliser (PostGreSQL, MySQL, SQLite...). Nous avons pensé dans un premier temps faire une base de données centralisées, nous permettant d'avoir un contrôle sur tous les messages envoyés de tous les utilisateurs. Or cela demanderait d'avoir un serveur qui tournera tout le temps quelque part, d'autant plus qu'il pourrait y avoir beaucoup trop de requêtes inutiles.

On s'est donc décidé de faire une base de données local pour chaque machine qui installera l'application. Il restait à voir quelle technologie utiliser. Une option était de faire tourner un serveur local sur la machine pour héberger la base de données, or cela aurait demandé de faire tourner un serveur en permanence, pour sauvegarder principalement les historiques de conversations de chaque utilisateur qu'on est entré en contact. On a donc choisi d'utiliser SQLite, qui permet de convertir un fichier en une base de données, où l'on peut stocker les informations comme dans une BDD conventionnelle. Il suffira seulement de faire des requêtes envers ce fichier, et non à un serveur.

Cependant, il y avait des limites avec SQLite : certaines structures de données n'était pas reconnu comparé à des BDD conventionnelles (par exemple la structure Boolean, le Temps, les GUID...), structures qui auraient pu être utiles notamment pour l'horodatage des messages. Cette base de données contient seulement deux classes : une classe "Utilisateurs", qui stock tous les utilisateurs qu'on est entré en contact par message, en stockant son pseudo et son adresse IP. Une deuxième classe "Messages" stock tous les messages envoyés ou reçus par tous les utilisateurs de la première classe, et chaque message est liée à quelqu'un dans "Utilisateurs".

c. La gestion du réseau

Cette application de chat peer to peer sur un réseau local d'une entreprise doit comprendre une section permettant de manifester sa présence, de communiquer de manière fiable avec les autres utilisateurs et offrir la possibilité de changer de pseudo à tout moment. Pour répondre à ces problématiques, nous avons séparé la communication sur le réseau en 2 grandes sections : une partie broadcast via UDP l'autre gérant les échanges de messages via TCP. Ces 2 parties sont elles-mêmes subdivisées en 2 sous-parties : un côté client et un côté serveur afin d'obtenir un système parfaitement réparti et permettre à chaque utilisateur de réaliser toutes les fonctionnalités,

i. Le Broadcast

Le client broadcast permet d'émettre des trames en broadcast vers tous les utilisateurs via une adresse fournie par le responsable du déploiement de l'application. Ces trames ayant des rôles différents en fonction de la situation, nous avons mis en place des messages prédéfinis pour chaque possibilité. Après l'envoi de ce paquet, le client attend les réponses des différents utilisateurs puis traite chacune en parallèle dans un thread dédié.

La partie serveur réceptionnant les paquets broadcastés tourne tout au long de la durée d'activité de l'application dans un thread à part afin de ne pas bloquer les autres fonctionnalités. Chaque requête réceptionnée est transmise à un worker pour être traitée dans un thread dédié puis le serveur retourne immédiatement en attente de nouveaux paquets.

ii. L'échange de message en TCP

En tant qu'utilisateur on peut initier la communication avec un autre utilisateur ou l'inverse. Ainsi nous pouvons être soit un client soit un server lors de la discussion avec un utilisateur actif.

En tant qu'utilisateur, on doit pouvoir initier plusieurs discussions si je le souhaite, ainsi il faut pouvoir créer plusieurs instances de clients tournant en parallèle pour échanger dans chacune de ces discussions. Nous traquons chacune de ces instances via un ArrayList. Ceci nous permet de sélectionner au gré de nos discussions avec quel serveur échanger des messages et de pouvoir interrompre chaque communication sans impacter les autres. De plus notre classe client permet de stocker chaque message reçu d'un serveur dans notre base de données et d'afficher le message sur le GUI si nous discutons activement avec ce serveur.

De multiples utilisateurs peuvent démarrer une discussion avec nous, ainsi notre serveur (TCP) de messages, tournant dans un thread dédié, permet de recevoir les multiples connections d'utilisateurs initiant un chat et les transfère dans une instance de MessServWorker tournant en parallèle gérant la réception et l'émission de messages. Chaque instance de MessServWorker est stockée dans un ArrayList afin de communiquer ou interrompre n'importe quelle discussion avec un client. Comme pour le RecvMessFromServer du client TCP, à la réception d'un message d'un client nous le stockons immédiatement dans notre base de données et si l'on discute activement avec ce client, le message est affiché sur notre fenêtre de chat. MessServSender permet de parcourir tous nos clients et de charger la bonne instance de communication de l'envoie du message vers le client souhaité.

d. Intégration entre le “Frontend” et le “Backend”

Ayant travaillé chacun une partie différente, il fallait tout au long du projet intégrer les deux parties “frontend” (GUI) et “backend” (BDD, gestion du réseau...) pour s’assurer que l’application marchait bien avant d’avancer à un autre élément. Cependant, il fallait souvent revenir en arrière dû à des découvertes de bugs qui n’ont pas pu être repéré avant. C’était les classes controllers qui commandaient toute la dynamique de l’application. C’était ces classes-là qui allaient commencer les serveurs de broadcast, d’appeler d’autre classe pour s’occuper de la gestion du réseau en arrière-plan, et de gérer la BDD (via la classe DBController). La classe NetInterface serait la classe utilisée par les controllers pour la partie réseau de l’application, ainsi que la classe Classe MultiClientConnection et MessServSender. Lorsque l’on envoie un message, ce message est envoyé avec les classes d’interface réseau, puis le contenu du message est stocké dans la BDD, et enfin une partie de la vue change pour montrer le nouveau message envoyé. Lorsque l’on reçoit un message, ce seront les classes ServConnections (si on est en tant que serveur) ou MultiClientConnections (si on est en tant que client) qui vont dans un premier temps recevoir le message, le stocker dans la BDD, et selon les cas, mettre à jour la vue de l’application.

Pour afficher les utilisateurs actifs, la classe activeUserManager s’occupe d’une liste qu’elle met à jour si un utilisateur se connecte, change de pseudo ou se déconnecte. Cette liste, manipulé par les classes du réseau, est lié à une composante de la vue qui liste les utilisateurs actifs. Lorsque la liste est mise à jour, cette composante de la vue se met à jour automatiquement aussi, ce qui facilite la gestion de la vue, et dynamise l’application.

4. Partie organisation et déploiement

a. Utilisation de Git

Étant donné un projet où il fallait travailler à deux sur un projet Java, il nous fallait un moyen de bien harmoniser nos codes. La solution d’utiliser une personne qui code et une autre qui dicte ce qu’elle doit écrire aurait pris beaucoup trop de temps. Il fallait tous les deux coder.

On s’était mis d’accord qu’une personne devait s’occuper de la partie “frontend”, c’est-à-dire la gestion de la vue de l’application et la dynamique de celle-ci, et de la partie base de données (même si considérée comme plutôt “backend” en général). L’autre personne allait s’occuper de la partie réseau et communication de l’application, c’est-à-dire la partie en s’occupant de la gestion des messages broadcast, et les parties des communications individuelles. Cette répartition nous aura évité trop de conflits sur les changements dans nos codes et dans les fichiers. Avec Git, il était ainsi simple de pouvoir mettre à jour son travail, dans sa propre branche, sans nuire au travail de l’autre. Et il était également plus simple de pouvoir recevoir les changements de l’autre (git fetch), et de les fusionner dans sa branche.

On s’était mis d’accord dès le début d’utiliser pour chacun sa propre branche git, et de fusionner au fur et à mesure les changements de chacun. La branche master/main servirait de branche auxiliaire pour pouvoir “merge” (fusionner) les changements de l’autre, mais aussi pour être sûr que ce que l’on met dans la branche main soit toujours opérationnel, si jamais l’on devait revenir sur des changements sur notre branche. On a utilisé ce qu’on appelle des “pulls requests” pour fusionner avec la branche main, qui permettait avant de fusionner de pouvoir voir toutes les modifications d’un commit avant la fusion, et de revenir sur ces modifications si l’on remarque une erreur.

b. Utilisation de Jenkins

Jenkins a été utile pendant le développement lorsqu’on voulait être sûr que les changements faits n’allaient pas faire des erreurs de déploiement de l’application. En créant des pipelines, on pouvait tester automatiquement notre application avant son "build” pour s’assurer que les changements n’allaient pas casser ceux d’avant aussi. Si les tests réussissaient, on pouvait tester le build, et si cela passait aussi, on pouvait tester le déploiement. Des Junit tests ont été fait pour la BDD, mais aussi pour la partie réseau. Notre pipeline avait donc 3 tâches : le premier consistait à tester avec les différents Junit tests l’application. Le second test le build de l’application, et le dernier test le déploiement de celui-ci.

c. Utilisation de Jira

Pour ne pas se lancer dans le code directement sans savoir ce qu'il faut faire après, et quelles étaient les tâches prioritaires, il fallait organiser le développement avec la méthode agile. On a d'abords définis tous les users stories qui pouvait survenir, les tâches qu'il fallait sûrement faire pour ces stories, à qui ils étaient destinés, leur "temps" estimés, et leurs priorités. On pouvait alors définir les sprints avec les stories associés. Le premier sprint correspondait à un sprint d'initialisation des outils. Pendant le développement cependant, on s'est rendu compte que pour ce sprint, les tâches à faire étaient bien plus longues que prévues, et que l'on avait mal estimé les temps. De plus, les tâches concernant la partie réseau "backend" étaient ceux qui étaient les plus longues à réaliser, il était du coup difficile de bien estimer leurs temps. Des tâches ont dû alors passer au prochain sprint en plus de ceux déjà définis.

d. Deploiement

En raison de problème technique lié à des problèmes de différence de système d'exploitation, l'application n'a pas pu être converti en un fichier executable. Notamment, pour une raison quelconque, la BDD ne fonctionne plus dans le fichier executable, alors que les drivers étaient bien installés. De plus, pour une même OS (mac), le fichier exécutable marchait sur l'une des machine (mac Air), mais pas sur l'autre (Imac), pour une raison qui nous échappe. Nous vous prions de nous excuser pour ce désagréement, mais annonçons que l'application marche très bien sur Eclipse, et ces problèmes nous ont fortement surpris.

5. Conclusion

Malgré les difficultés qui ont été rencontrés lors de la réalisation de ce projet, qui ont pu nous bloquer pendant beaucoup trop de temps, l'application est fonctionnelle, et respecte le cahier des charges. On peut choisir un utilisateur actif, avoir une conversation avec lui, changer de pseudo mais en ayant les mêmes conversations commencées avec le pseudo d'avant...

On est revenu sur certaines décisions prises, notamment sur l'affichage des utilisateurs actifs, la façon dont le programme procède avec les threads au niveau des envois de messages et de réceptions de ceux-là. Ce projet a été intéressant à réaliser, il nous a demandé pas mal d'effort, mais on a appris beaucoup de chose sur la réalisation d'un projet concret, de la conception jusqu'à son déploiement.