Rapport de projet COO POO : Application de clavardage

Rédigé par Fabien Castilla et Mikhail Zakharov

INSA Toulouse, 4éme année

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Table des matières

Introduction

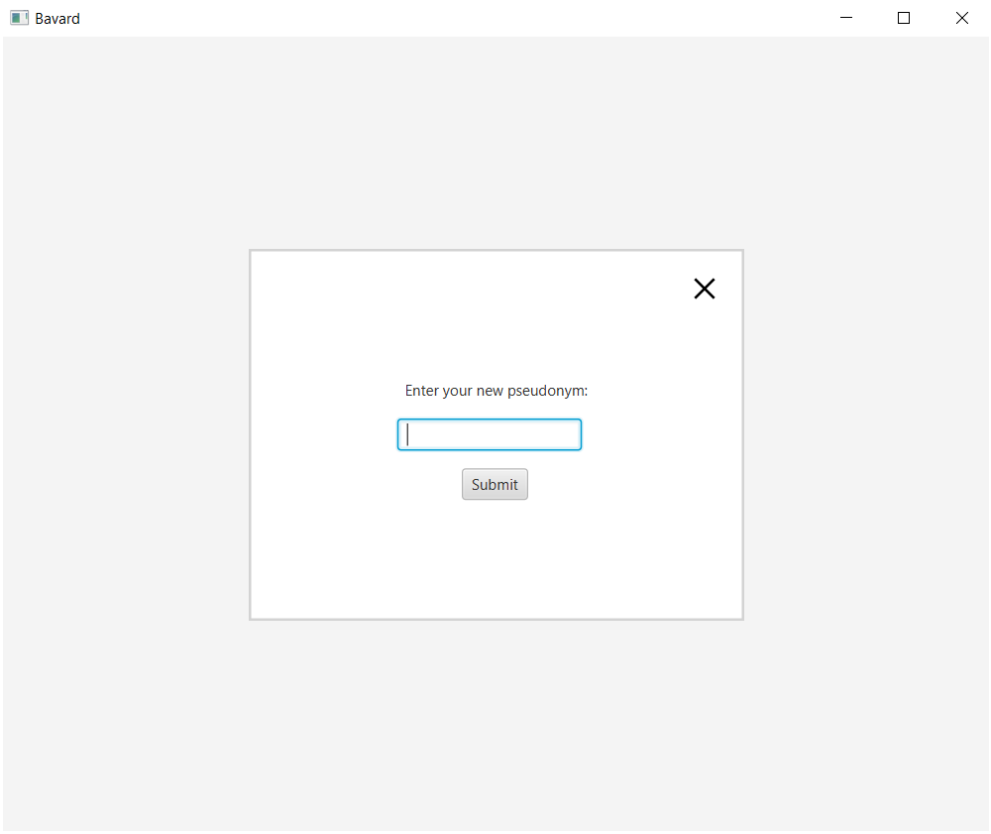
Bavard est une application de clavardage permettant à des utilisateurs distants d'échanger des messages. Elle fonctionne aussi bien sur Windows, Linux et MacOS à condition de disposer de la version 15 ou supérieurs de Java sur la plateforme utilisée. Cette application a été modélisé et développé par Fabien Castilla et Mikhail Zakharov de novembre 2020 à février 2021.

Vous trouverez dans ce rapport un manuel d'utilisation destiné à un utilisateur classique de l'application lui permettant d'appréhender simplement son utilisation. Vous aurez également toutes les informations nécessaires au déploiement de l'application. Enfin, une partie entière sera réservé à la conception de Bavard en ce qui concerne la base de données, les différents diagrammes de conception ou encore les choix stratégiques qui ont été choisis.

Manuel d’utilisation

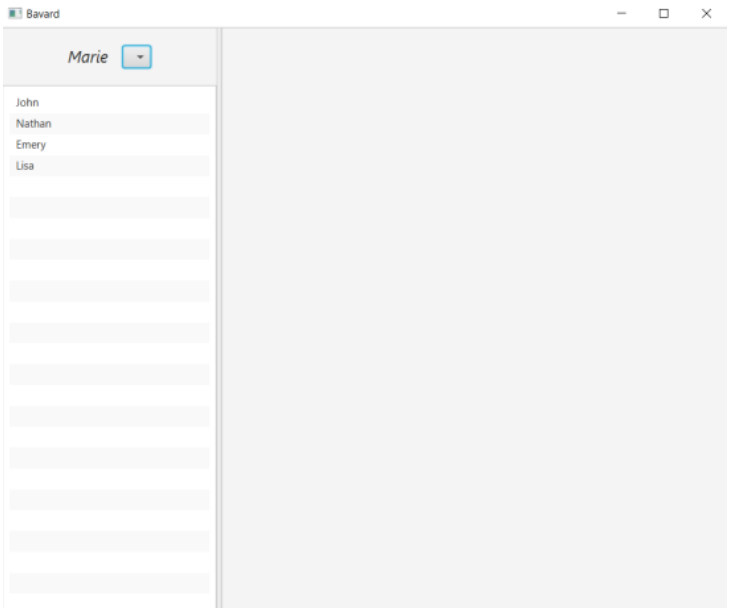
Démarrage de l’application

Quand vous démarrer l’application, il vous est demander de renseigner un pseudo. Ce pseudo permettra aux autres utilisateurs de vous identifier. Si le pseudo que vous saisissez est déjà utilisé par un autre utilisateur, il vous sera demander de saisir un nouveau pseudo. Néanmoins, vous pourrez par la suite changer de pseudo, si vous le désirez.



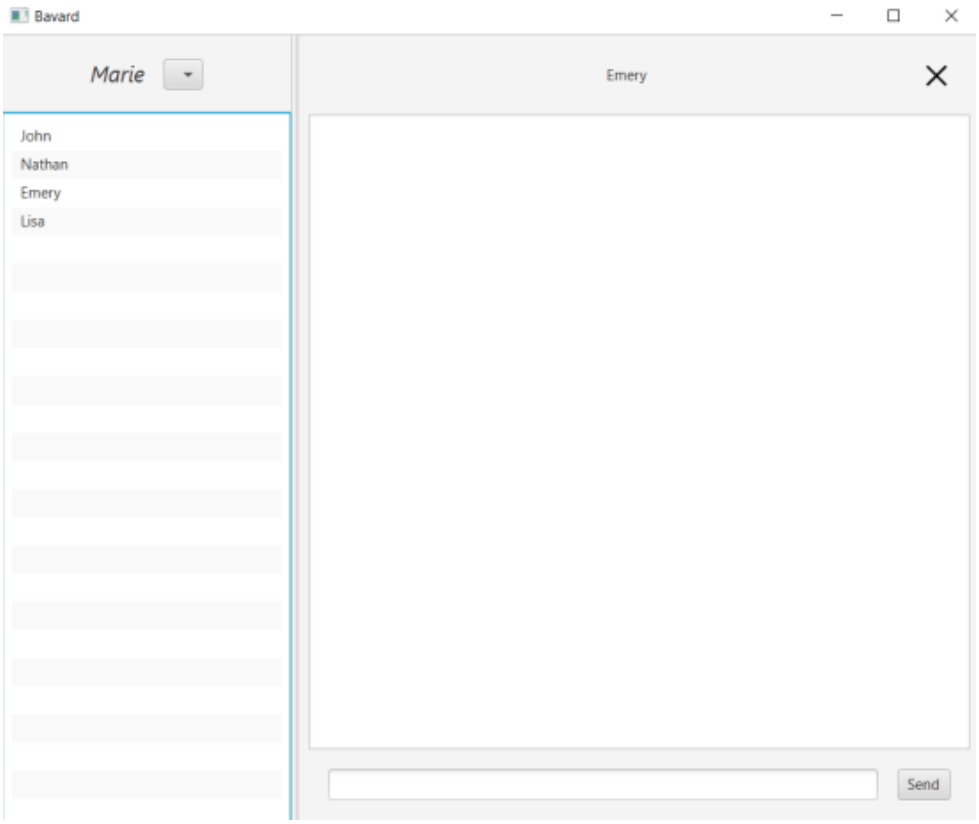
Voir les utilisateurs

Une fois vous êtes identifié, vous arriverez sur l’interface principale. Sur la gauche de cette interface vous trouverez la liste des utilisateurs connectés. Celle-ci se met automatiquement à jour lors de la connexion ou la déconnexion d’un utilisateur ou lors d’un changement de pseudo d’un utilisateur connecté.



Démarrer une conversation

Pour démarrer une conversation avec l'un des utilisateurs connectés, il vous suffit de cliquer sur l'utilisateur avec lequel vous souhaitez converser. Vous remarquerez l'apparition d'un espace de tchat sur la partie droite de l'interface.

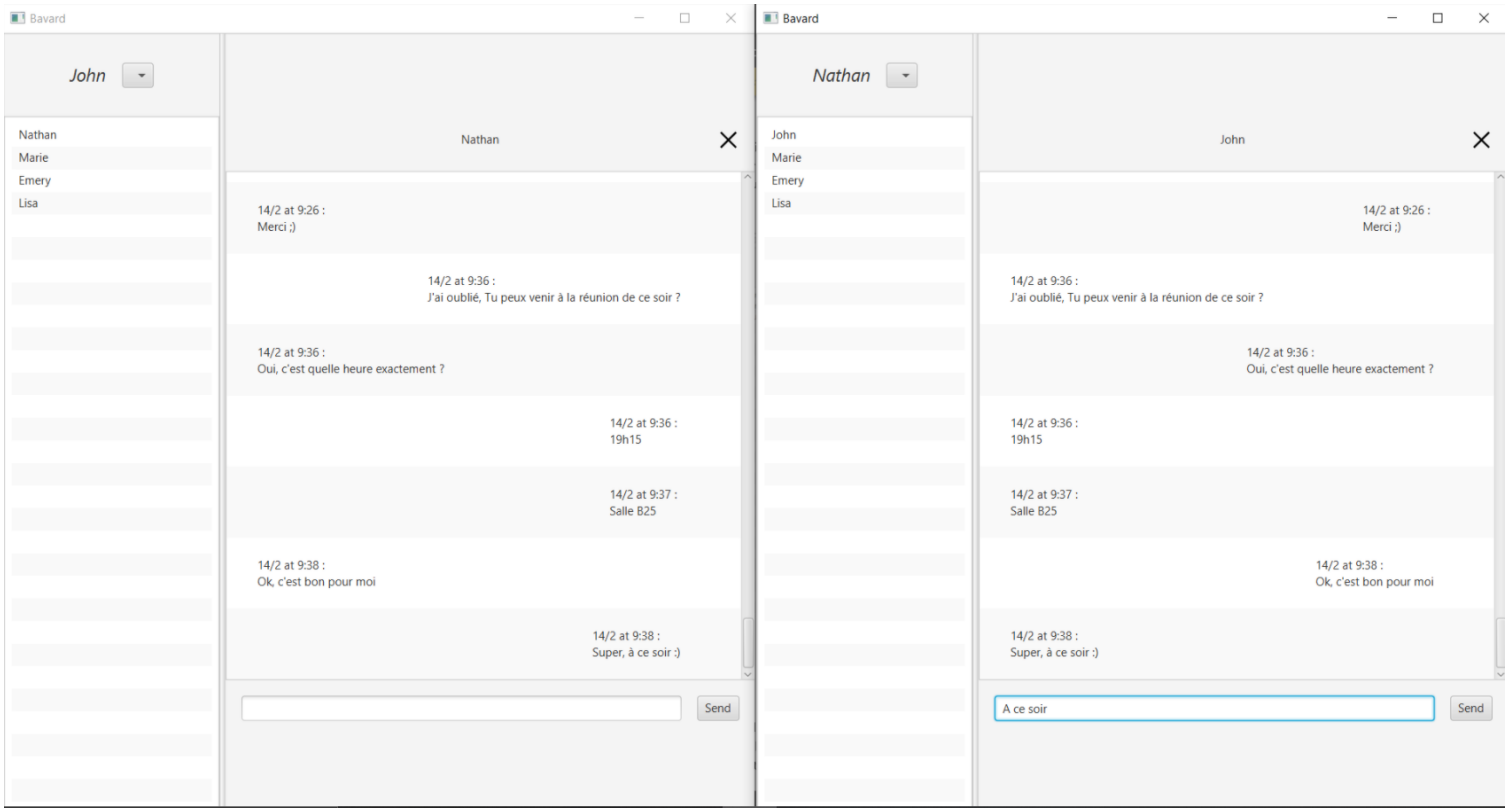


Consulter l’historique des messages

Après avoir sélectionné un utilisateur parmi la liste, vous pourrez accéder directement à l’historique des messages échangés avec cet utilisateurs en faisant défiler la liste des messages.

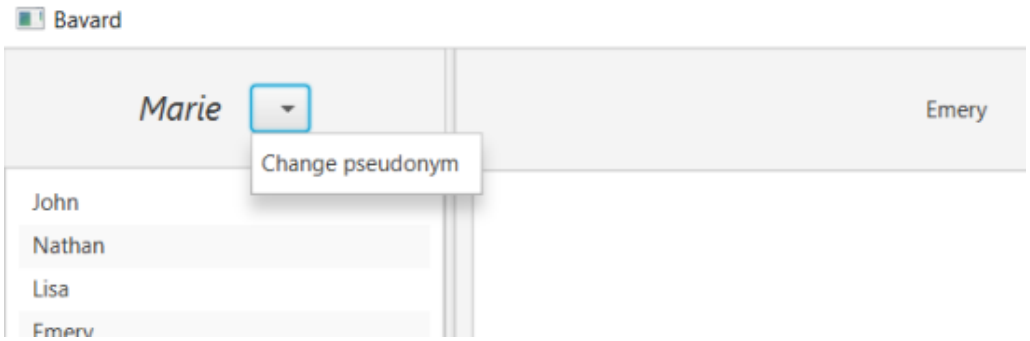
Echanger des messages

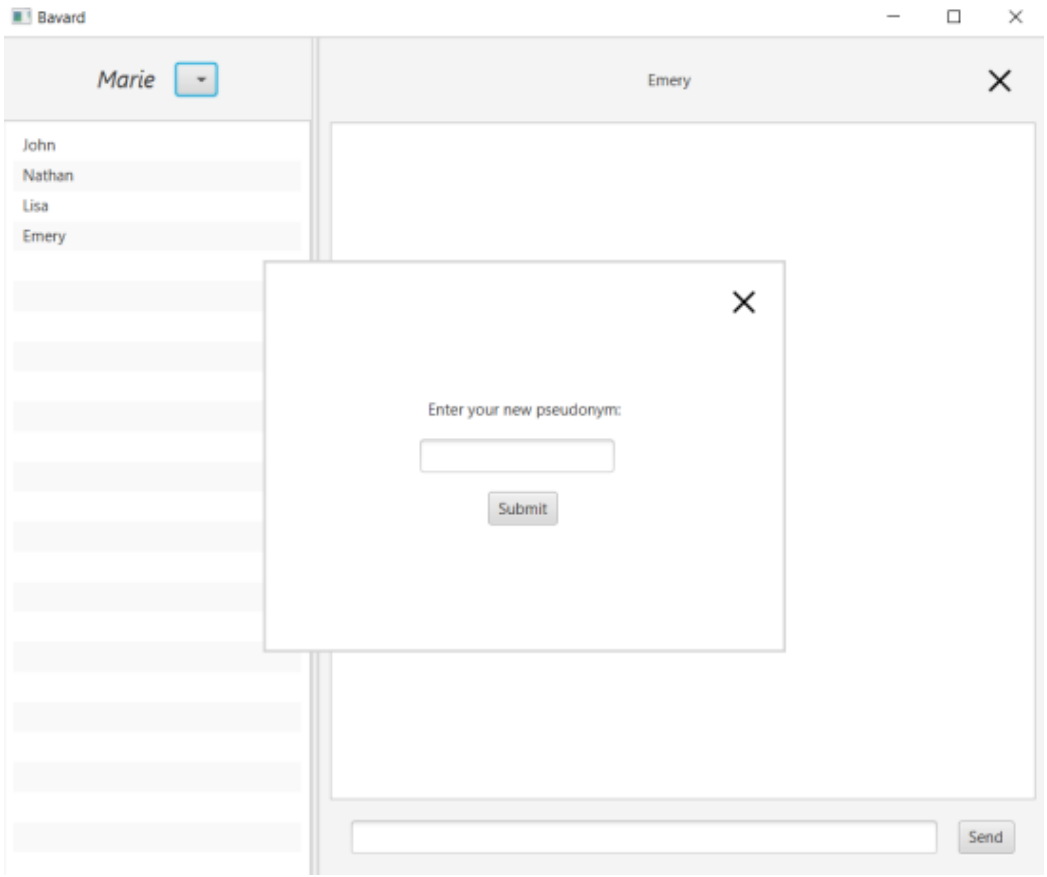
Vous pouvez envoyer un message en le saisissant au préalable dans la zone de texte prévu à cet effet et en appuyant sur le bouton **Send**. Votre message apparaitra alors à droite de la liste des messages tandis que les messages de votre interlocuteur se trouvent à gauche.



Changer de pseudo

En haut à gauche de l’interface, disposé à droite de votre pseudo actuel, se trouve une liste déroulante qui lors de la sélection vous permet de changer votre pseudo. Une zone de texte apparait et vous pourrez y saisir le pseudo désiré. Celui-ci doit être libre, car sinon il vous sera demander d’en saisir un autre.





Se déconnecter

Pour vous déconnecter. Il vous suffit simplement de quitter l'application en cliquant sur la croix en haut à droite de celle-ci.

Déploiement

Pour déployer l'application sur des postes de travail il suffit de télécharger le *app-all.jar* depuis notre dépôt git (disponible sur : github.com/FabienCst/INSA-COO-POO-Project) directement ou via git (avec git clone). Ensuite le copier sur chaque poste de travail ou elle sera utilisée. Pour le lancer, double-clique sur le jar et c'est parti. Pour le serveur proxy il faut télécharger le *proxy-all.jar* de même façon et le lancer sur la machine serveur. Si une interface graphique n'est pas disponible, le proxy peut être lancée avec la commande : *java -jar proxy-all.jar*.

Or toutes les fonctionnalités qui permet de déployer comme décrit au-dessus ne sont pas disponibles, nous avons fourni des *jar* prêts à tester sur votre machine. Ils se trouvent dans le dossier *demo* sur git. La suite explique comment tester notre application sur un seul ordinateur.

Coté serveur

Vous trouverez *proxy-all.jar* dans le dossier *demo*. Ouvrez un terminal dans ce dossier et tapez la commande : *java -jar proxy-all.jar* pour le lancer. Rien ne s'affichera sur l'écran, c'est prévu. Cela veut dire que le proxy est allumé. Pour l'éteindre il suffit de tapez *Ctrl + C* dans ce même terminal.

Coté utilisateurs

Dans le dossier *demo* vous trouverez aussi les dossiers *alpha, beta, gamma, radulph-external, sparklefin-external*. Chaque dossier contient un *app-all.jar* qui peut être considéré comme un utilisateur individuel. Donc pour tester sur une seule machine il suffit de lancer chaque *app-all.jar* (en le double-cliquant) que vous voulez. Par exemple, pour avoir deux utilisateurs internes et un utilisateur externe vous pouvez lancer le *app-all.jar* de *beta*, *gamma* et *sparklefin-external*. Les utilisateurs marqués *-external* ont la fonctionnalité de broadcast supprimée pour simuler le fait d'être hors le réseau interne. Pour que les utilisateurs externes soient bien reconnus, il faut que le proxy soit lancé avant toutes les applications utilisateur.

Tests

Nous n'avons pas écrit des tests unitaires ou autre à la fois à cause de la difficulté de tester les interfaces graphiques et de la logique du réseau, mais aussi à cause du temps. Il est, par contre, possible d'effectuer des tests d'usage avec les *jar* fournis. Voici quelques exemples :

- Deux (ou plus) utilisateurs internes peuvent se trouver et communiquer sans le proxy.
- Deux (ou plus) utilisateurs externes peuvent se trouver et communiquer entre eux via le proxy.
- Des utilisateurs internes et externes peuvent se trouver et communiquer via le proxy.
- Sans le proxy, un utilisateur interne ne trouvera pas un utilisateur externe, et inversement.

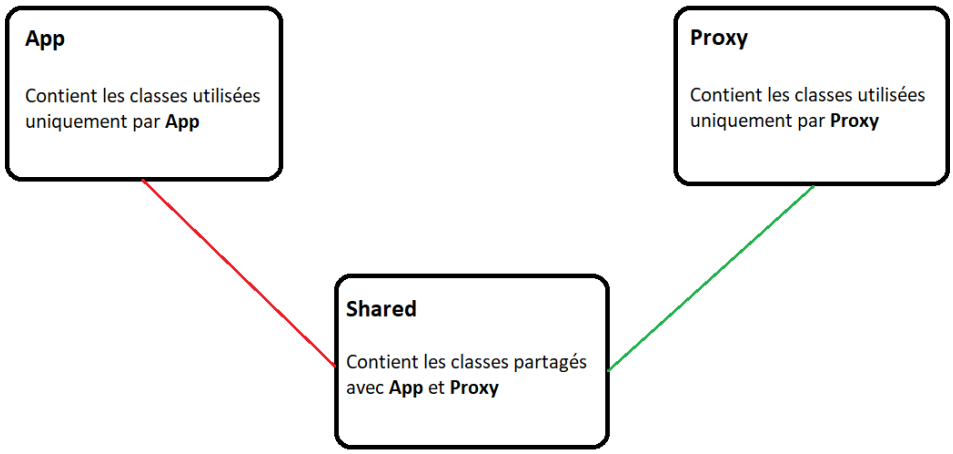
Conception

Choix techniques et technologiques

Nous avons programmé cette application sous Java 15 couplé avec l'utilisation de Gradle pour la compilation. Gradle permet de gérer simplement la gestion de dépendances et la gestion de sous projet.

En ce qui concerne le serveur, nous avons préféré une application Java classique plutôt qu'une application web utilisant des servlets et un serveur HTTP. Nous avons utilisé des sockets sur la base d'objets sérialisés pour échanger des données entre les utilisateurs d'un réseau locale. Il nous a semblé logique d'utiliser le même principe pour échanger des données entre les utilisateurs internes et les utilisateurs externes, et donc implicitement entre un utilisateur et le server. Autrement, cela nous aurait obligé à faire des transformations des données pour un serveur HTTP. Néanmoins, pour que cela fonctionne nous avons dû découper l'application en plusieurs sous projets.

En effet, ce projet est découpé en 3 sous-projets :



Cette technique était nécessaire pour le bon fonctionnement du partage des objets sérialisés via les sockets. Par exemple, l'objet *TextMessage* doit être le même pour *App* et *Proxy* afin de pouvoir être sérialisé depuis *App* puis désérialisé depuis *Proxy* et inversement.

Pour ce qui est de la partie utilisateur, nous avons créé l'interface à l'aide de JavaFx. Ce qui nous a permis de faire une interface simple et intuitive largement personnalisable par la suite selon le design désirer par l'entreprise.

JavaFx n'étant plus inclus dans le JDK nous avons dû l'importer ce qui à notamment augmenter la taille de l'application et qui justifie également le découpage du projet. App étant trop lourd nous ne pouvions pas l'ajouter aux dépendances de *Proxy* pour partager les classes communes car *Proxy* devenait logiquement trop lourd lui aussi et ne pouvait pas être héberger sur le serveur de l'INSA.

Base de données

En ce qui concerne le stockage des données, nous avons fait le choix d'utiliser des bases de données décentralisé au moyen de bases de données SQLite. Ainsi, dès le lancement de l'application, un fichier **store.db** est créé, si celui n'a jamais été lancé auparavant. Si un utilisateur change de poste, il lui suffit de prendre ce fichier et de le mettre à son emplacement initial sur le nouveau poste. Lorsque celui-ci lancera l'application sur le nouveau poste et il pourra récupérer l'historiques de toutes ces conversations.

Ce choix va de pair avec l'utilisation de **uuid** pour contrôler l'unicité des utilisateurs. Même si cette fonctionnalité n'est pour le moment pas opérationnelle, l'idée est de déterminer un id unique pour chaque utilisateur au premier démarrage d'une application. Cette id est ensuite enregistré dans la base donnée. Ainsi, lors du transfert de cette base vers un autre poste, l'utilisateur conserve son unicité aux yeux des autres utilisateurs.

La base de données que l'on a conçue, est composé d'une unique table intitulée *Message*.

On y trouve 5 attributs :

- **sender_uid** (text), cet attribut stocke l'identifiant de l'émetteur du message,

- **recipient_uid** (text), ce champ fait référence à l'identifiant du destinataire du message,

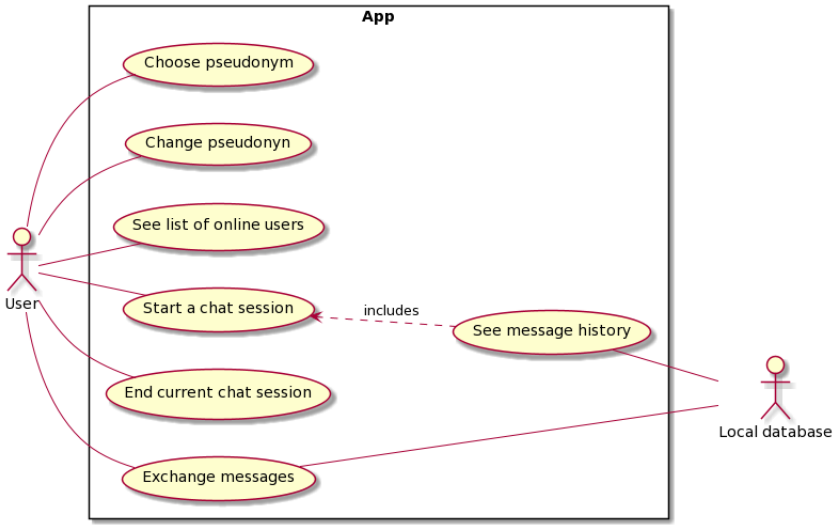
- **datetime** (text), ce champ stocke la date et l'heure de l'envoie du message,

- **type** (text), cet attribut indique le type du message, soit un texte, soit un document,

- **content** (text), cet attribut permet de stocker le texte d'un message ou encore le lien vers un document.

Diagrammes UML

Cas d'utilisation



Diagrammes de classes

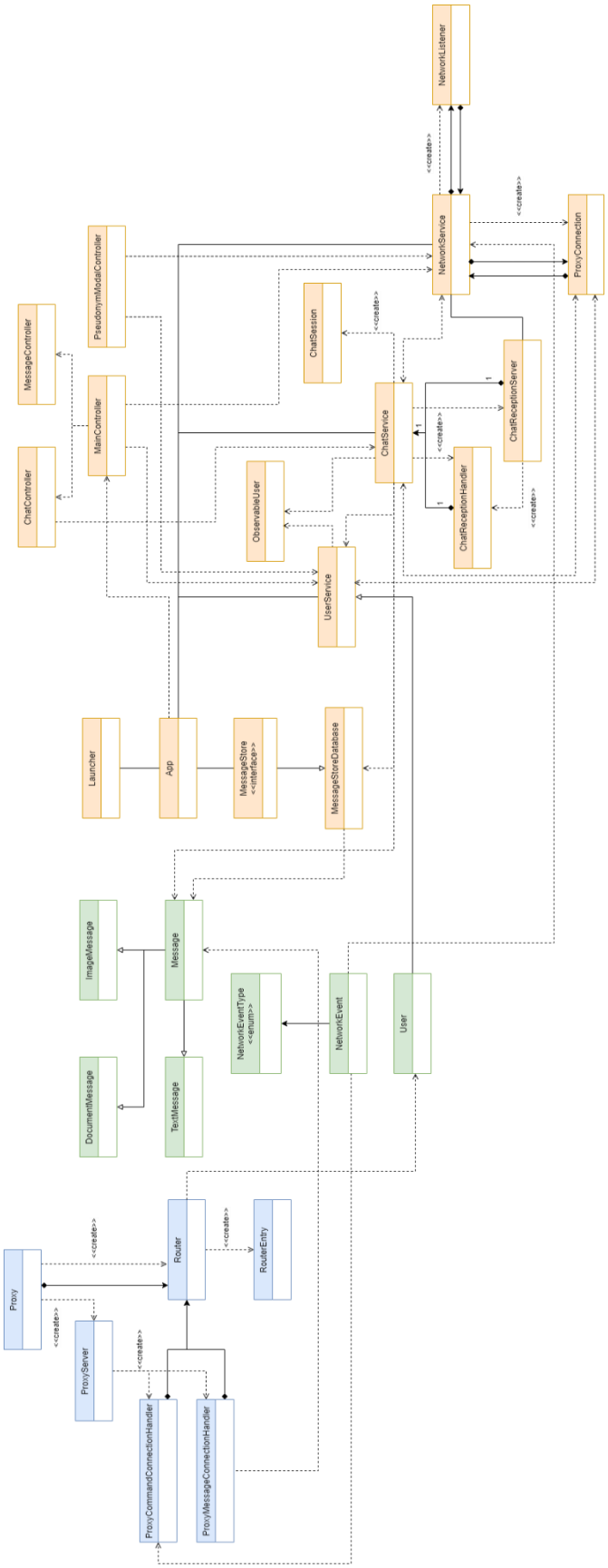
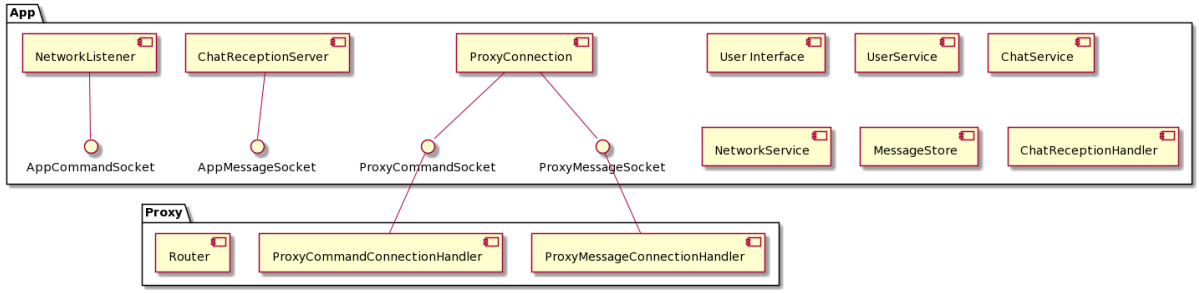
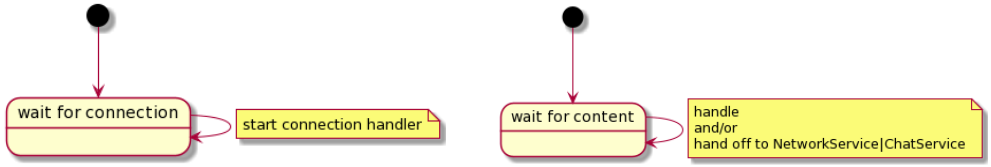


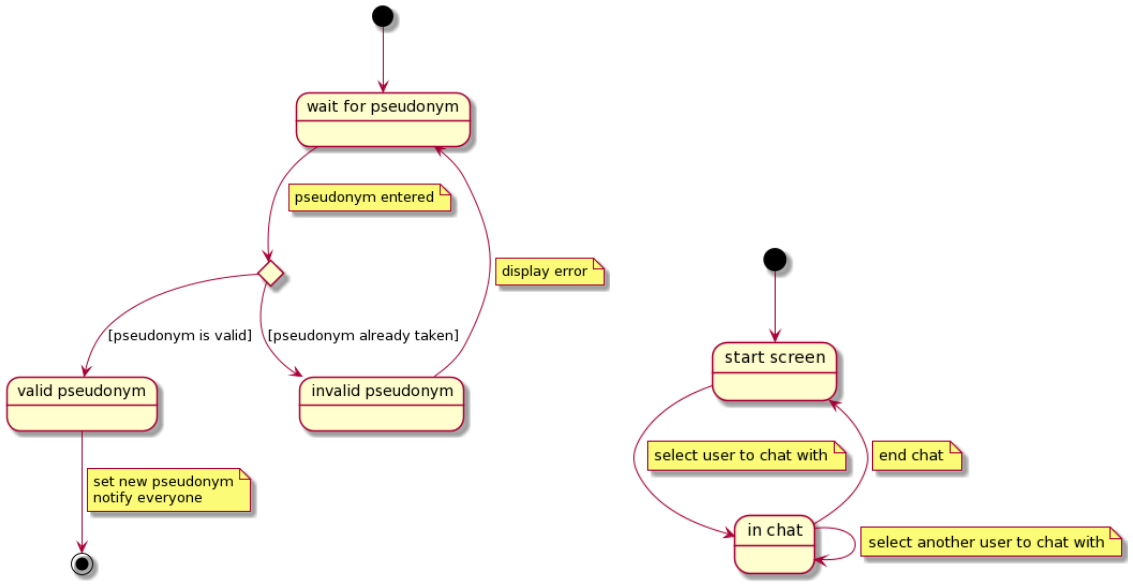
Diagramme de structure composite



Diagrammes d'états



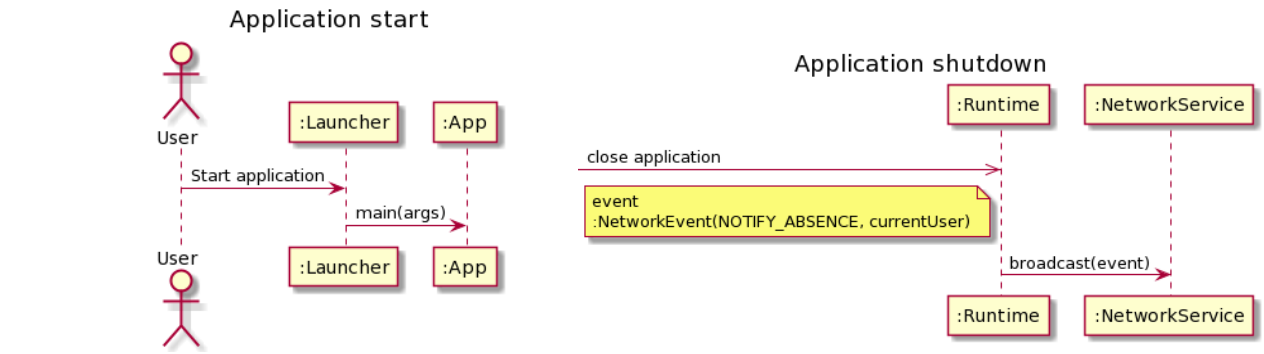
Gauche : attente de connexion par les serveurs.
Droite : attente de réception de contenu par un handler

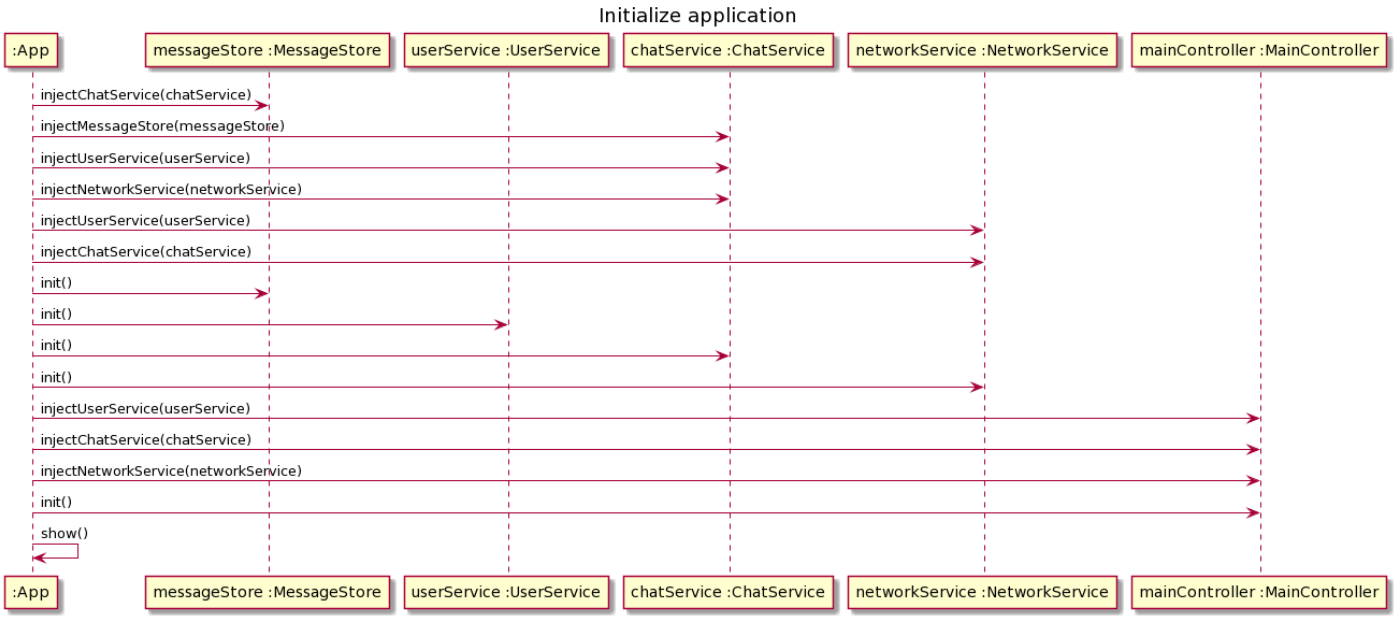


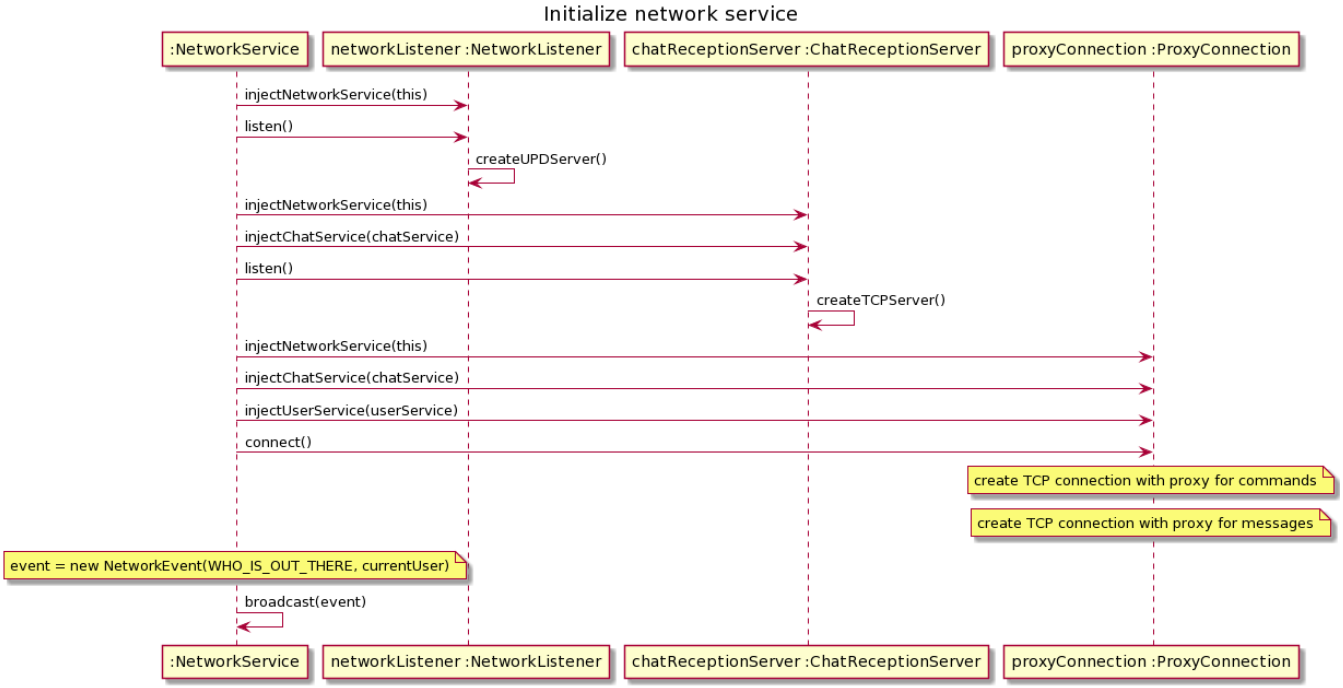
Gauche : validation du pseudonyme.
Droite : états de l'application principale.

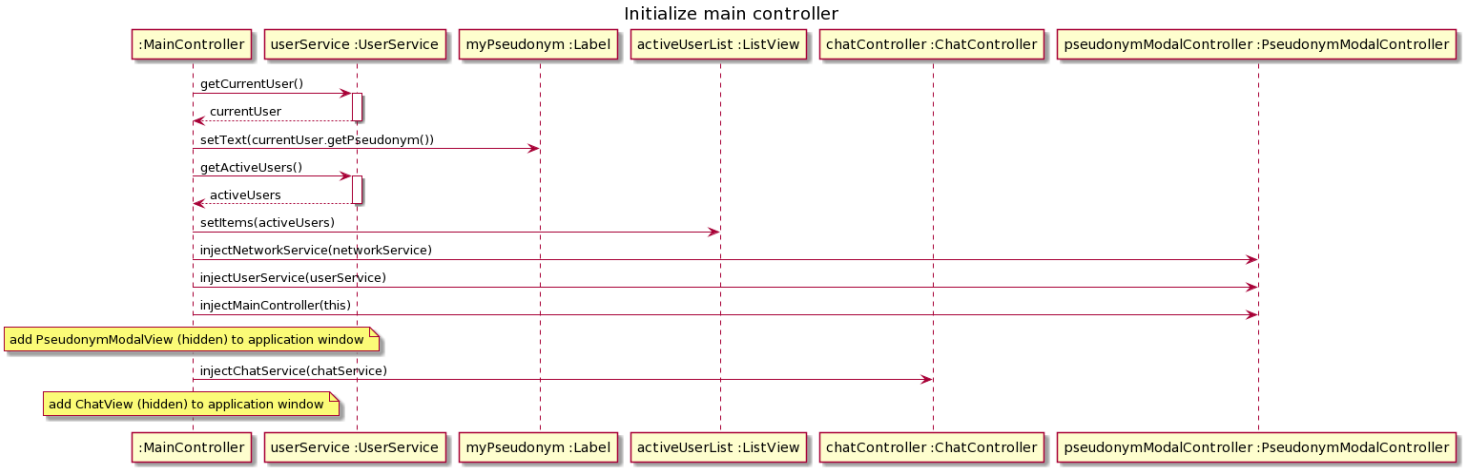
Diagrammes de séquence

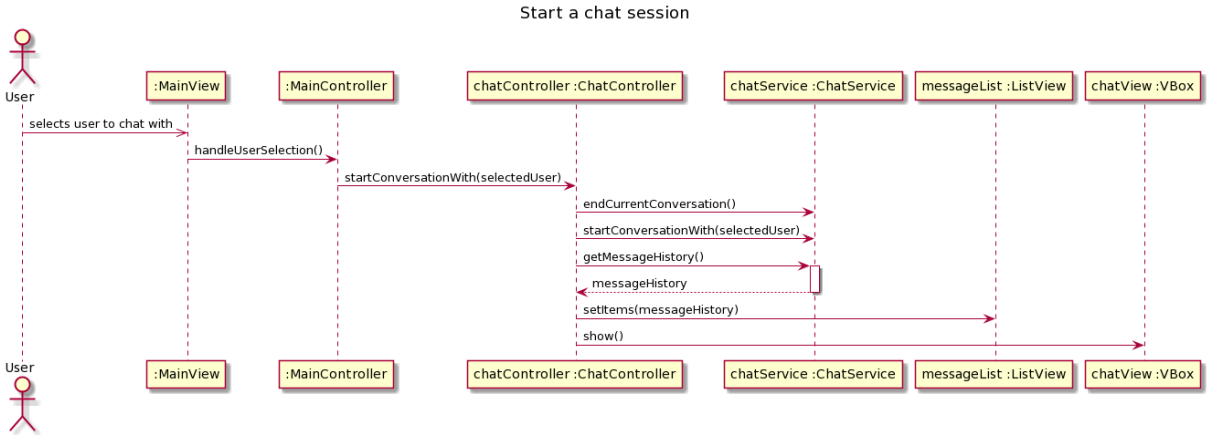
Les diagrammes de séquence expliquent en détaille le fonctionnement de notre application. D'abord par l'initialisation, on utilise l'injection des dépendances afin d'éviter des mauvaises pratiques.

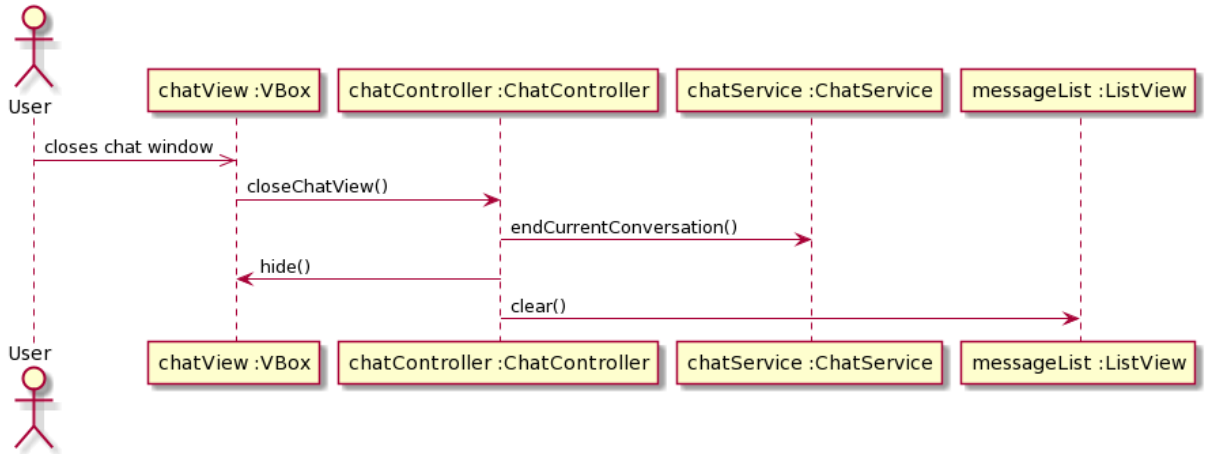

Application start
User
:Launcher
:App
Start application
main(args)
User
:Launcher
:App
Application shutdown
:Runtime
:NetworkService
close application
event
:NetworkEvent(NOTIFY_ABSENCE, currentUser)
broadcast(event)
:Runtime
:NetworkService

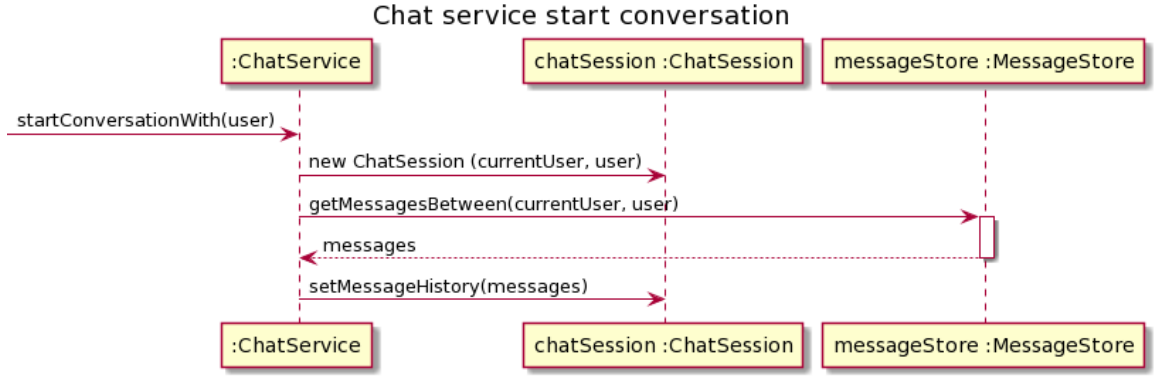

Initialize application
:App
messageStore :MessageStore
userService :UserService
chatService :ChatService
networkService :NetworkService
mainController :MainController
injectChatService(chatService)
injectMessageStore(messageStore)
injectUserService(userService)
injectNetworkService(networkService)
injectUserService(userService)
injectChatService(chatService)
init()
init()
init()
init()
injectUserService(userService)
injectChatService(chatService)
injectNetworkService(networkService)
init()
show()
:App
messageStore :MessageStore
userService :UserService
chatService :ChatService
networkService :NetworkService
mainController :MainController


Initialize network service
:NetworkService
networkListener :NetworkListener
chatReceptionServer :ChatReceptionServer
proxyConnection :ProxyConnection
injectNetworkService(this)
listen()
createUPDServer()
injectNetworkService(this)
injectChatService(chatService)
listen()
createTCPServer()
injectNetworkService(this)
injectChatService(chatService)
injectUserService(userService)
connect()
create TCP connection with proxy for commands
create TCP connection with proxy for messages
event = new NetworkEvent(WHO_IS_OUT_THERE, currentUser)
broadcast(event)
:NetworkService
networkListener :NetworkListener
chatReceptionServer :ChatReceptionServer
proxyConnection :ProxyConnection

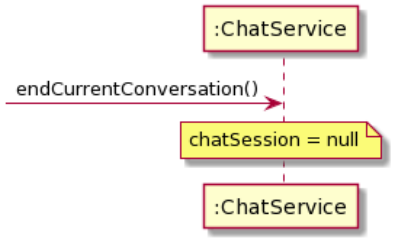

Initialize main controller
:MainController
userService :UserService
myPseudonym :Label
activeUserList :ListView
chatController :ChatController
pseudonymModalController :PseudonymModalController
getCurrentUser()
currentUser
setText(currentUser.getPseudonym())
getActiveUsers()
activeUsers
setItems(activeUsers)
injectNetworkService(networkService)
injectUserService(userService)
injectMainController(this)
add PseudonymModalView (hidden) to application window
injectChatService(chatService)
add ChatView (hidden) to application window


Start a chat session
User
:MainView
:MainController
chatController :ChatController
chatService :ChatService
messageList :ListView
chatView :VBox
selects user to chat with
handleUserSelection()
startConversationWith(selectedUser)
endCurrentConversation()
startConversationWith(selectedUser)
getMessageHistory()
messageHistory
setItems(messageHistory)
show()

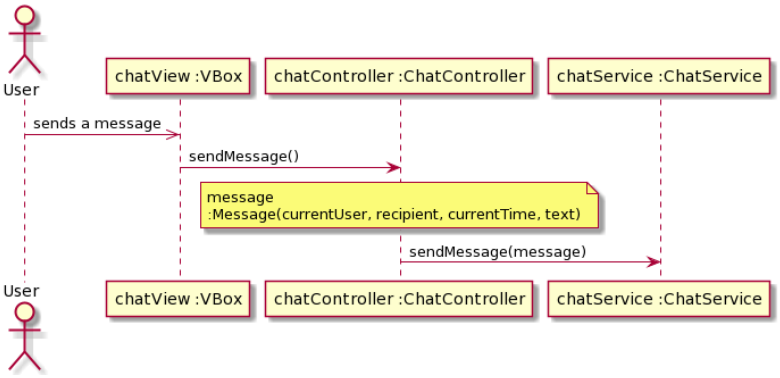

End a chat session
User
chatView :VBox
chatController :ChatController
chatService :ChatService
messageList :ListView
closes chat window
closeChatView()
endCurrentConversation()
hide()
clear()


Chat service start conversation
:ChatService
chatSession :ChatSession
messageStore :MessageStore
startConversationWith(user)
new ChatSession (currentUser, user)
getMessagesBetween(currentUser, user)
messages
setMessageHistory(messages)

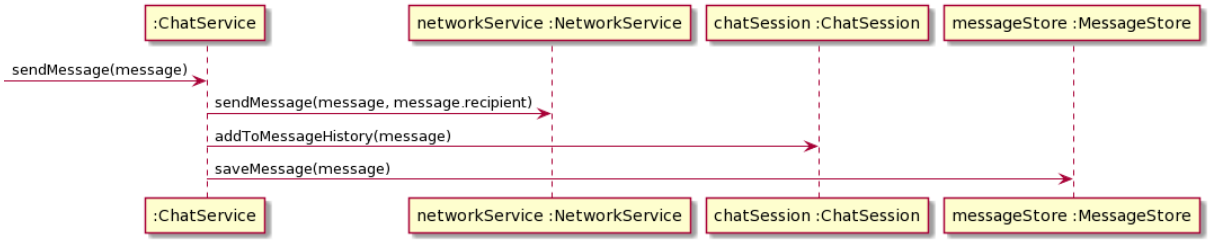
Chat service end conversation



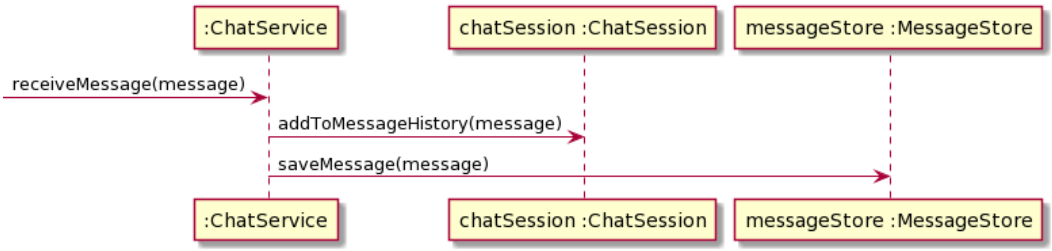
Send a message



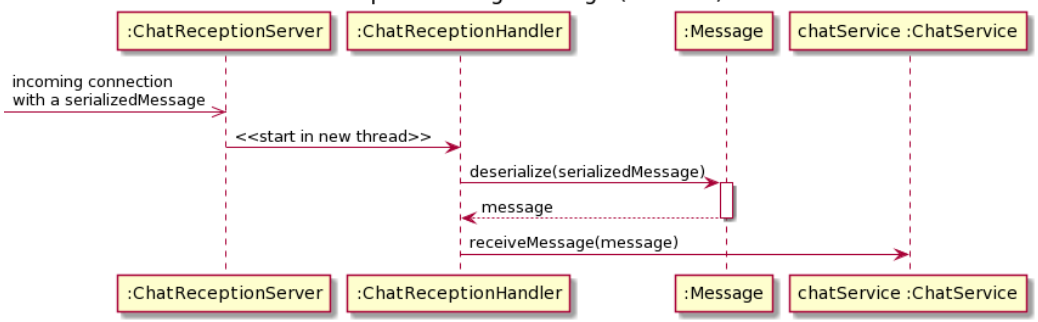
Chat service send message



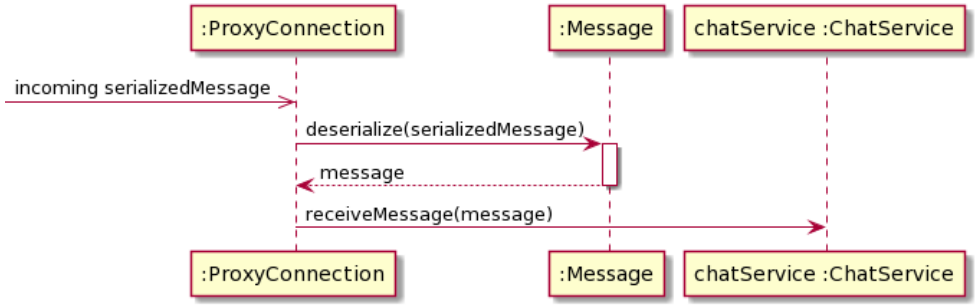
Chat service receive message



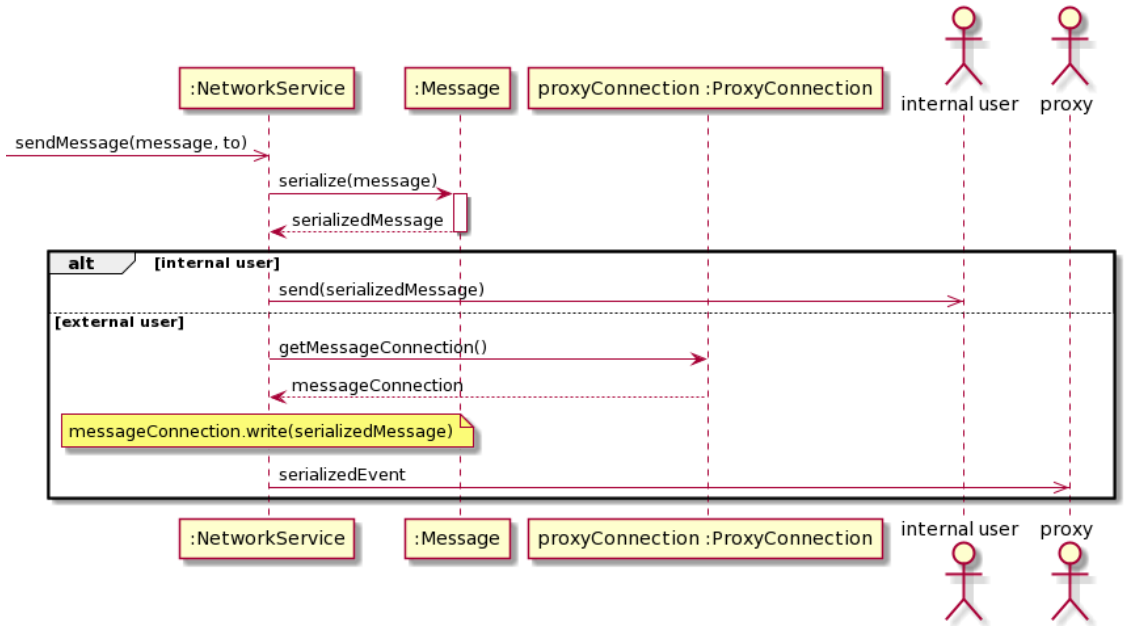
Accept incoming message (internal)



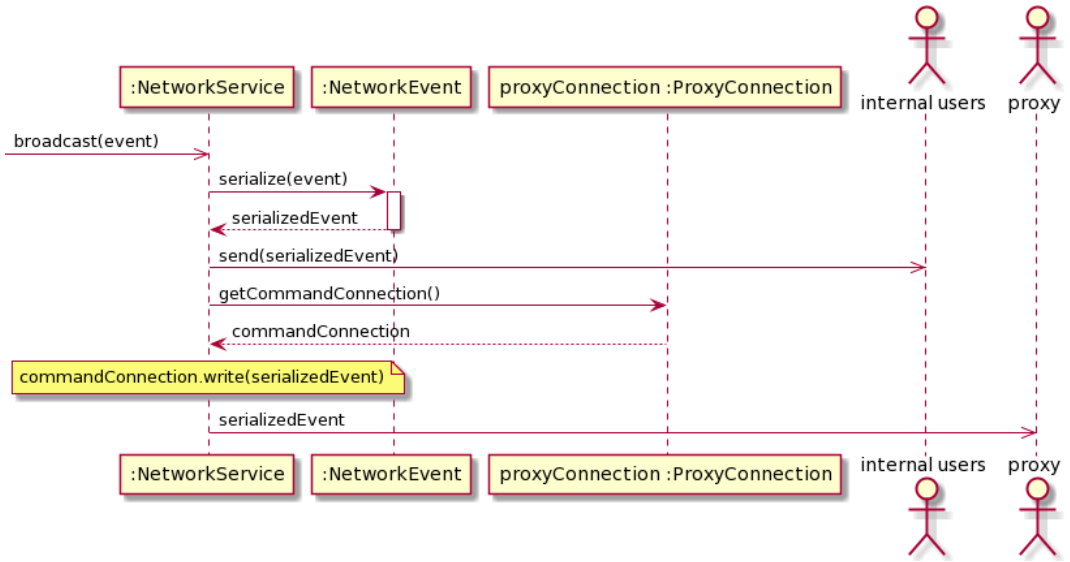
Accept incoming message (external)

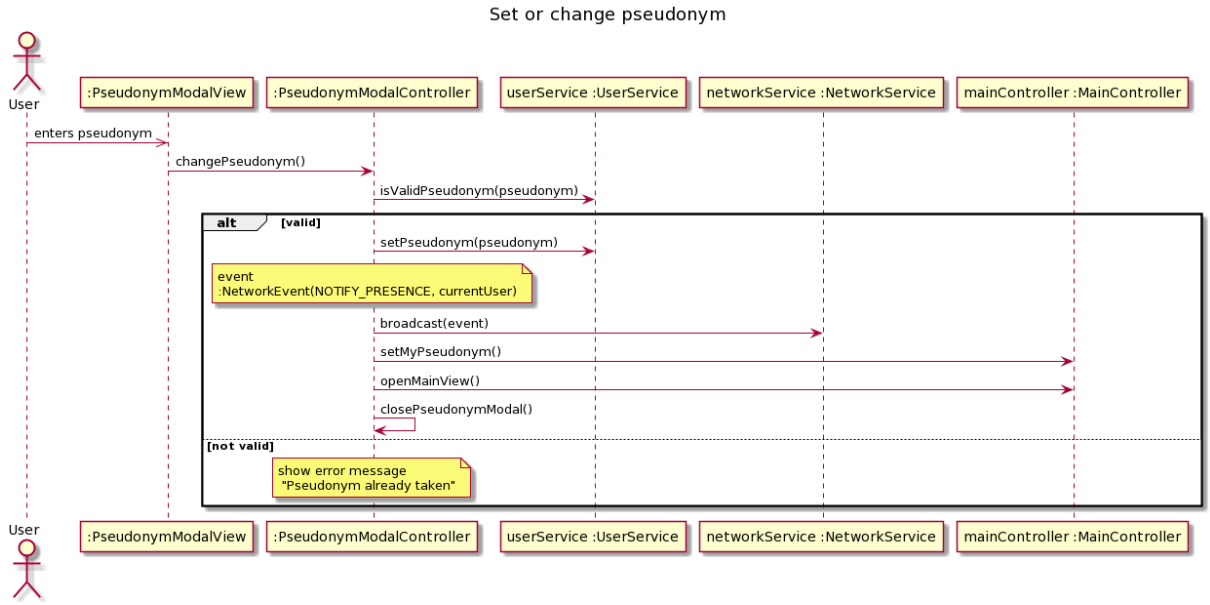


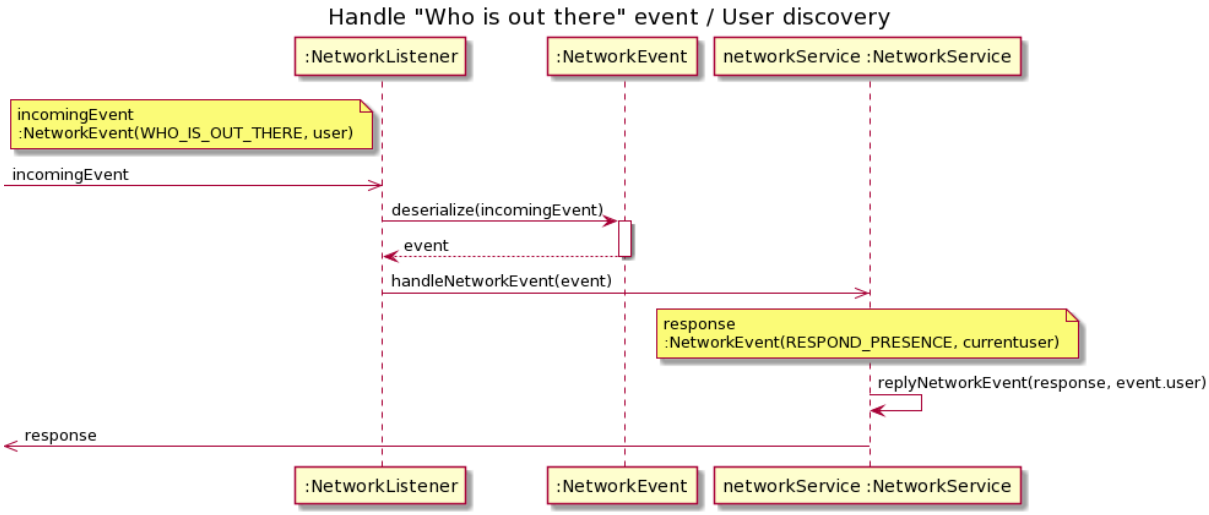
Network service sendMessage

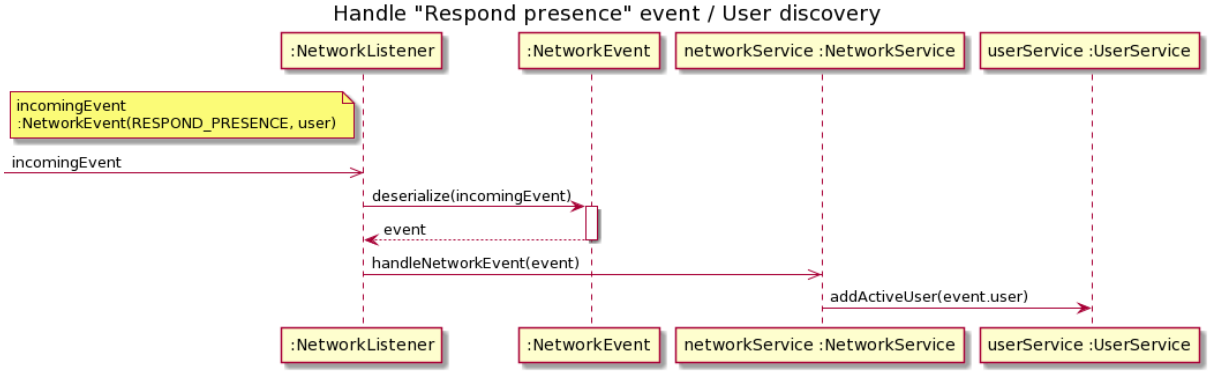


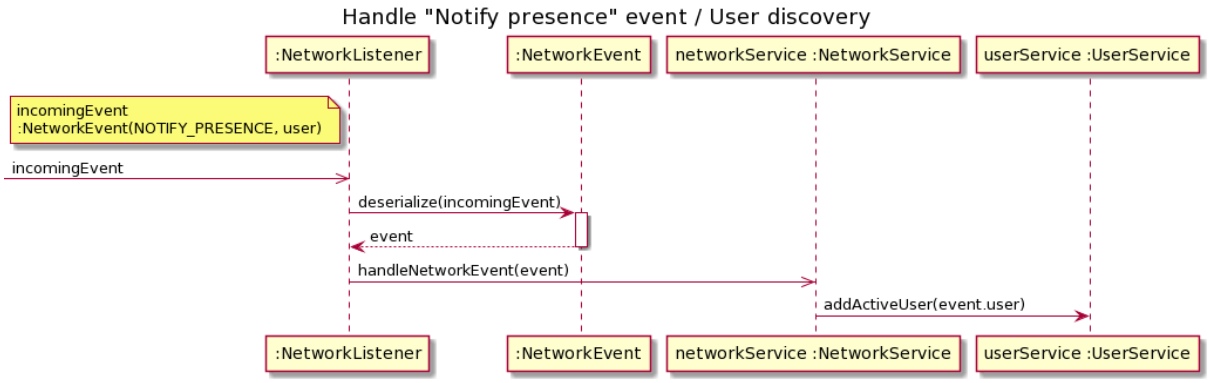
Network service broadcast

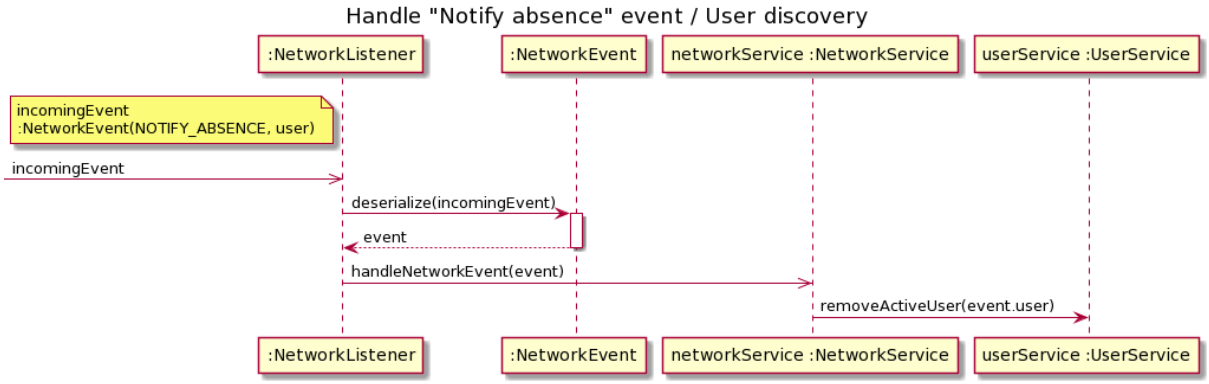



Set or change pseudonym
User
:PseudonymModalView
:PseudonymModalController
userService :UserService
networkService :NetworkService
mainController :MainController
enters pseudonym
changePseudonym()
isValidPseudonym(pseudonym)
alt
[valid]
setPseudonym(pseudonym)
event
:NetworkEvent(NOTIFY_PRESENCE, currentUser)
broadcast(event)
setMyPseudonym()
openMainView()
closePseudonymModal()
[not valid]
show error message
"Pseudonym already taken"
User
:PseudonymModalView
:PseudonymModalController
userService :UserService
networkService :NetworkService
mainController :MainController

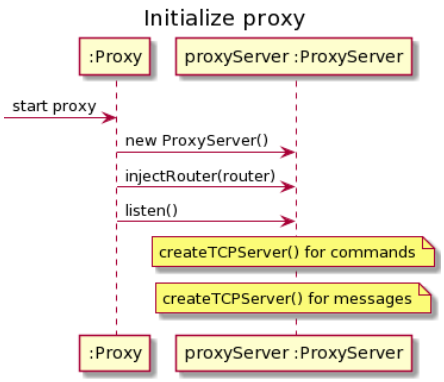

Handle "Who is out there" event / User discovery
:NetworkListener
:NetworkEvent
networkService :NetworkService
incomingEvent
:NetworkEvent(WHO_IS_OUT_THERE, user)
incomingEvent
deserialize(incomingEvent)
event
handleNetworkEvent(event)
response
:NetworkEvent(RESPOND_PRESENCE, currentuser)
replyNetworkEvent(response, event.user)
response
:NetworkListener
:NetworkEvent
networkService :NetworkService

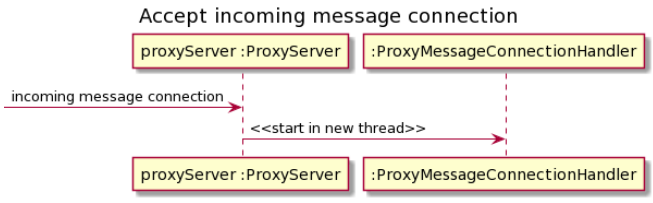

Handle "Respond presence" event / User discovery
:NetworkListener
:NetworkEvent
networkService :NetworkService
userService :UserService
incomingEvent
:NetworkEvent(RESPOND_PRESENCE, user)
incomingEvent
deserialize(incomingEvent)
event
handleNetworkEvent(event)
addActiveUser(event.user)
:NetworkListener
:NetworkEvent
networkService :NetworkService
userService :UserService

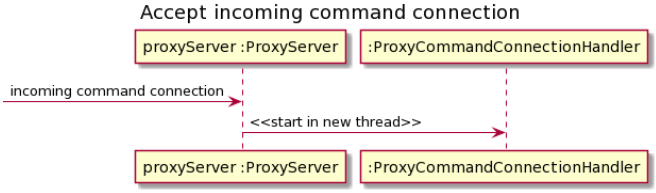


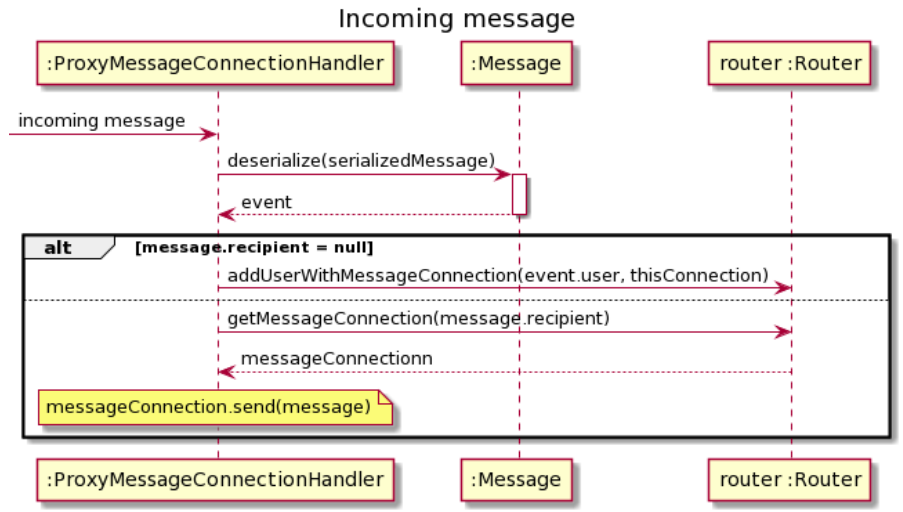


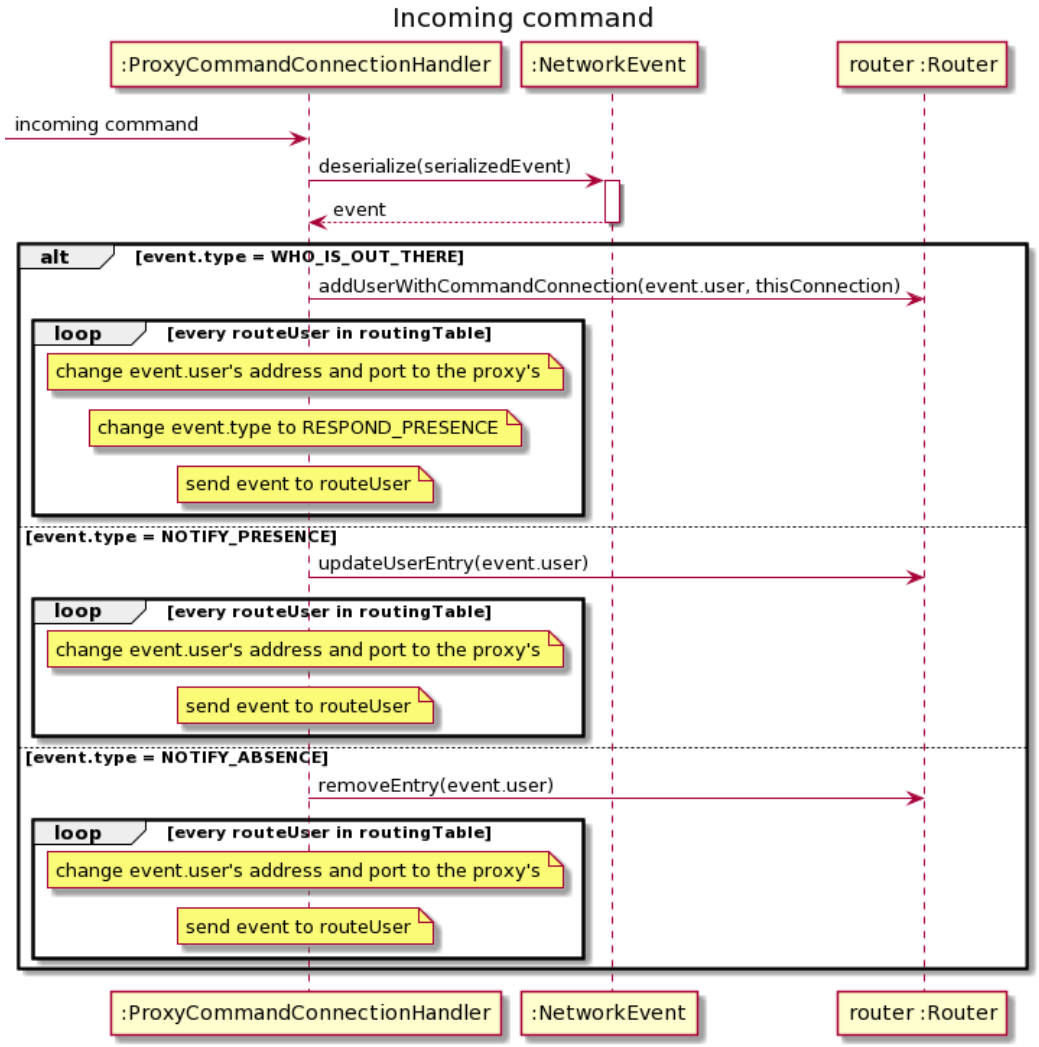
Pour la partie proxy :








Incoming message
:ProxyMessageConnectionHandler
:Message
router :Router
incoming message
deserialize(serializedMessage)
event
alt
[message.recipient = null]
addUserWithMessageConnection(event.user, thisConnection)
getMessageConnection(message.recipient)
messageConnectionn
messageConnection.send(message)
:ProxyMessageConnectionHandler
:Message
router :Router


Incoming command
:ProxyCommandConnectionHandler
:NetworkEvent
router :Router
incoming command
deserialize(serializedEvent)
event
alt
[event.type = WHO_IS_OUT_THERE]
addUserWithCommandConnection(event.user, thisConnection)
loop
[every routeUser in routingTable]
change event.user's address and port to the proxy's
change event.type to RESPOND_PRESENCE
send event to routeUser
[event.type = NOTIFY_PRESENCE]
updateUserEntry(event.user)
loop
[every routeUser in routingTable]
change event.user's address and port to the proxy's
send event to routeUser
[event.type = NOTIFY_ABSENCE]
removeEntry(event.user)
loop
[every routeUser in routingTable]
change event.user's address and port to the proxy's
send event to routeUser
:ProxyCommandConnectionHandler
:NetworkEvent
router :Router

Conclusion

Ce projet a été très enrichissant. Il nous a permis d'utiliser Java et la programmation orientée objet sur différentes technologies et sous différents aspects, en passant par l'utilisation de base de données, de sockets, de serveurs et de communications TCP et UDP.

Les fonctionnalités principales sont présentes. Néanmoins, certaines sont manquantes telles que la fonctionnalité relative à l'envoie de document entre les utilisateurs et la fonctionnalité permettant d'identifier de manière unique les utilisateurs. Cette dernière, explique notamment pourquoi plusieurs .jar sont nécessaires pour simuler des utilisateurs uniques dans la démonstration de notre application.

Également, le design de l'interface est brut mais prêt à évoluer et un style peut être appliqué facilement selon les désirs du client grâce à la souplesse des classes relatives aux éléments de l'interface.