

# **Rapport de projet COO/POO**

## Système de clavardage

---

Perramond Florian  
Toutant Martin  
4 IR A2

# Sommaire

<b>I Introduction</b>	<b>2</b>
<b>II Manuel d'utilisation</b>	<b>3</b>
A - User guide	3
B - Administrator guide	6
<b>III Système de chat décentralisé</b>	<b>9</b>
A - Lancement de l'application	9
B - Architecture du projet	9
C - Découverte des utilisateurs connectés	10
D - Echange de messages	10
E - Renommage des pseudos et gestion des conflits	10
<b>IV Nouvelles fonctionnalités basées sur un serveur centralisé</b>	<b>13</b>
A - Attribution automatique d'identifiant utilisateur	13
B - Stockage et récupération de l'historique des messages	13
C - Utilisation du système de chat par des utilisateurs externes	13
<b>V Validation et tests</b>	<b>14</b>
<b>Annexes - Conception</b>	<b>15</b>
Use Case	15
Diagrammes de séquence	16
Diagrammes de classe	20
Package utils	20
Package network	21
Package database	21
Package application	22
Package application.client	23
Package application.server	26

# I Introduction

L'objectif de ce projet est, à l'aide de nos connaissances en conception et programmation orientée objet, de réaliser un système de clavardage (chat). Ce système de communication permet à différents utilisateurs de communiquer par grâce à une interface graphique simple d'utilisation, qu'ils soient sur le même réseau local ou non. Pour une utilisation en réseau local, le système repose sur une communication en "peer to peer" alors qu'en utilisation externe, les messages sont transmis par l'intermédiaire d'un serveur centralisé. Le langage de programmation utilisé est Java, et l'interface graphique a été conçue grâce à l'outil NetBeans.

En veillant à bien respecter le cahier des charges, nous avons émis quelques hypothèses quant à l'utilisation de l'application. Premièrement, nous considérons qu'un utilisateur n'utilise qu'un ordinateur et qu'un ordinateur ne sert qu'à un seul utilisateur. [Autres?]

Avec ce rapport, nous fournissons tout d'abord un manuel d'utilisation de notre système de chat, de l'installation aux premiers messages échangés. De plus, nous tentons de dépeindre le fonctionnement de nos programmes pour la gestion des connexions (interne ou externe), de la découverte des utilisateur connectés, des échanges de messages entre deux agents connectés et de la gestion de l'historique.

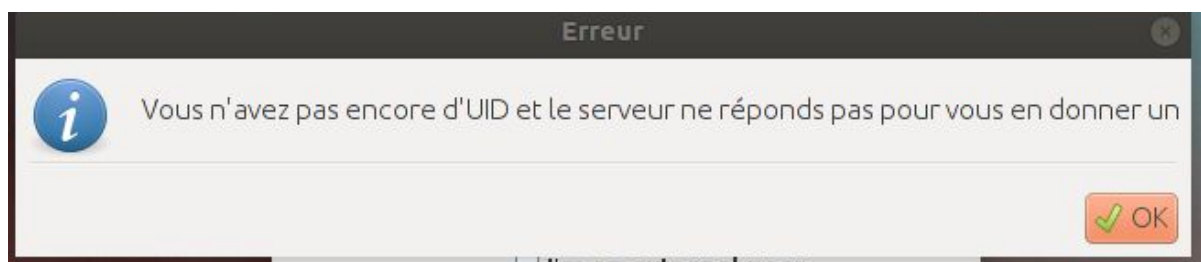
## II Manuel d'utilisation

### A - User guide

#### Utilisation générale de l'application

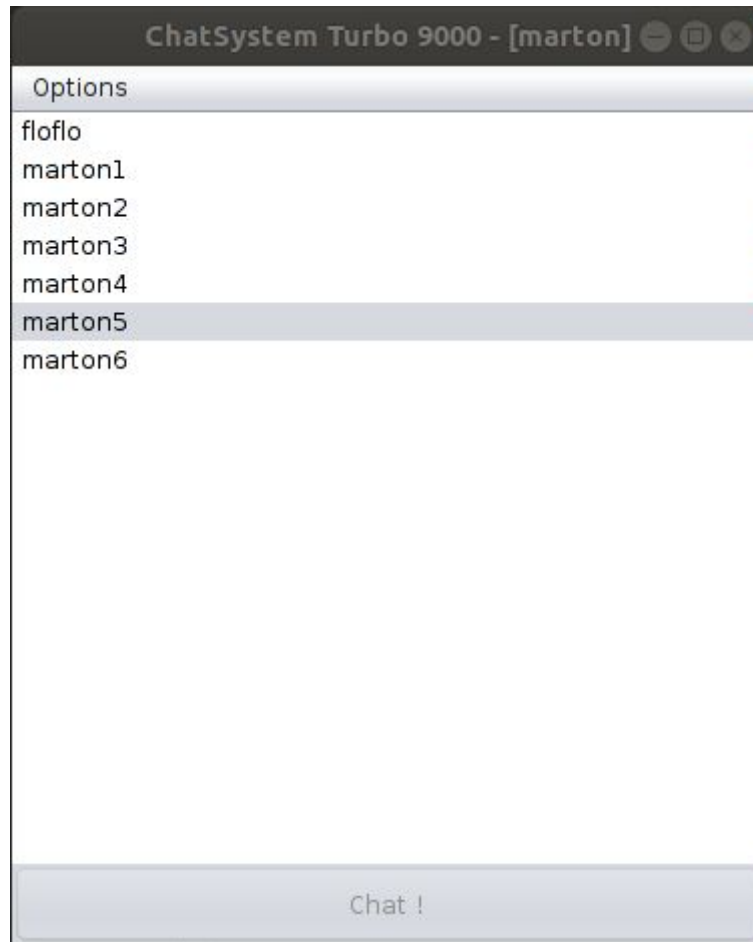
Au lancement de l'application, la fenêtre de login se lance et l'utilisateur est invité à saisir un pseudonyme qui servira à l'identifier sur le système de chat. Ce pseudonyme pourra être changé ensuite.

En cas d'affichage du message suivant:



- ❖ pour une utilisation avec un serveur centralisé vérifiez que celui-ci soit bien lancé et relancez l'agent;
- ❖ pour une utilisation en réseau local, il est possible que le fichier "userId.txt" soit inexistant dans le dossier turbo9000, reportez-vous au point "utilisation en réseau local" de la partie Administrator Guide.

Si le pseudonyme choisi est disponible, alors la fenêtre principale s'ouvre et la liste des utilisateurs connectés sur le réseau s'affiche :



Plusieurs opérations sont disponibles depuis cette interface.

Tout d'abord dans l'onglet "options", l'utilisateur peut changer de pseudo ("Rename") ou se déconnecter ("Disconnect"), qui entraînera un retour à la fenêtre de login. A noter que si l'utilisateur ferme la fenêtre sans appuyer sur le bouton de déconnexion, le comportement de l'application sera identique.

Ensuite, sur sélection d'un utilisateur connecté, le bouton "Chat!" s'active. A l'actionnement de celui-ci une fenêtre de clavardage s'ouvre (voir capture d'écran ci-dessous), avec le contact souhaité. Il s'affiche alors l'historique des messages entre les 2 utilisateurs (en mode avec serveur centralisé uniquement) et l'utilisateur peut envoyer un message à son correspondant dès qu'il le souhaite. A l'envoi du premier message, la connexion TCP est établie.



A la fermeture de la fenêtre, un message de confirmation apparaît pour savoir si l'utilisateur souhaite mettre fin à la session. Si oui, alors la connection TCP est fermée et sera rétablie à l'envoi, ou réception, d'un prochain message.

### Commandes du serveur

Il y a plusieurs commandes que l'on peut taper dans le flux entrant de l'application serveur afin de le manager. Voici la liste des commandes :

- exit: pour quitter le serveur
- db show users: afin d'afficher les numéro des utilisateurs inscrits
- db clear user [user\_id]: afin de supprimer un utilisateur
- db show messages [user\_id]: afin d'afficher les messages d'un utilisateur
- db clear messages [user\_id]: afin de supprimer les messages d'un utilisateur
- popup: affiche une popup temporaire sur les agents actifs
- help: affiche la liste des commandes

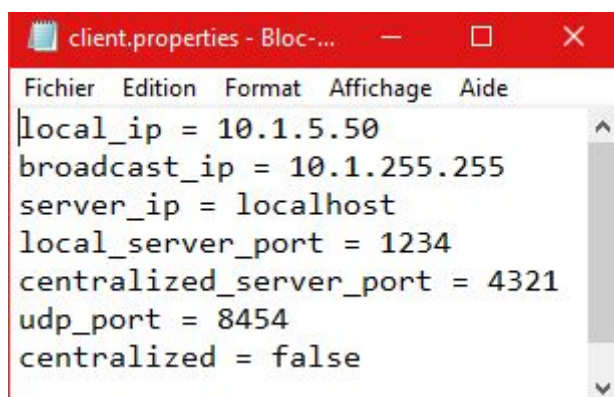
## B - Administrator guide

Important: le système est prévu pour un utilisateur unique par machine.

Pour déployer le chat system sur une machine, copiez le dossier "turbo9000" à l'emplacement souhaité. Ensuite vous devez configurer les fichiers de configuration de ce dossier. La suite de la configuration dépend du mode que vous allez utiliser.

### Préparation pour une utilisation en réseau local

Avant le lancement de l'agent, il faut configurer le fichier "client.properties" qui se trouve dans le dossier "turbo9000" en l'ouvrant dans un éditeur de texte.



```
client.properties - Bloc-...
Fichier  Edition  Format  Affichage  Aide
local_ip = 10.1.5.50
broadcast_ip = 10.1.255.255
server_ip = localhost
local_server_port = 1234
centralized_server_port = 4321
udp_port = 8454
centralized = false
```

Les champs suivants sont à compléter :

- ❖ local\_ip : l'adresse IP de votre machine sur le réseau local d'utilisation ciblée.
- ❖ broadcast\_ip : l'adresse IP de broadcast du même réseau local.
- ❖ server\_ip : non utilisé pour utilisation en local
- ❖ local\_server\_port : le port du serveur tcp qui sera ouvert sur cette machine, à laisser tel quel. En cas de problème lors de l'ouverture d'une session de clavardage, il est possible qu'il faille changer la valeur de ce champs avec une valeur entre 1024 et 65535. Tous les utilisateurs sur le même réseau doivent avoir la même valeur de local\_server\_port .
- ❖ centralized\_server\_port : non utilisé pour utilisation en local
- ❖ udp\_port : le port UDP utilisé pour le broadcast, à laisser tel quel. En cas de problème lors de la découverte des utilisateurs connectés, il est possible qu'il faille changer la valeur de ce champs avec une valeur entre 1024 et 65535. Tous les utilisateurs sur le même réseau doivent avoir la même valeur de udp\_port.
- ❖ centralized : false. Permet de préciser si l'utilisation de l'agent se fera sur un réseau local ou s'il est déployé sur un réseau possédant un serveur centralisé.

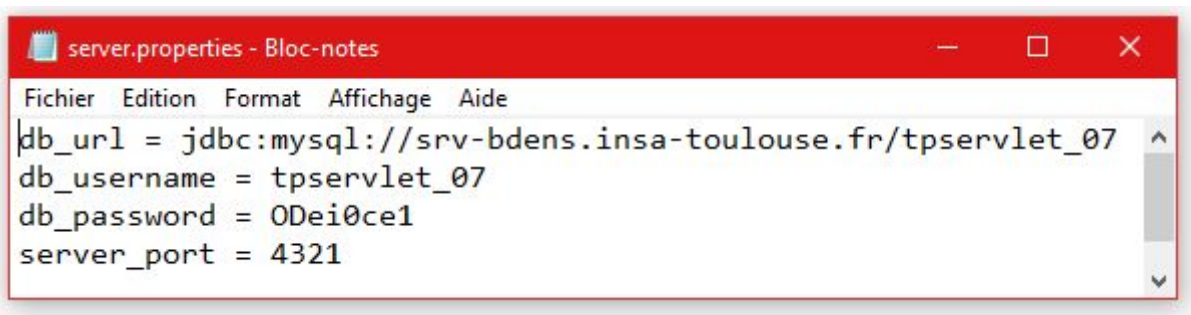
Le lancement de l'application se fait ensuite grâce au script run\_client.sh (pour une utilisation sous Linux) ou run\_client.bat (pour une utilisation sous Windows). Il faut que java soit installé sur la machine et soit dans le PATH. La version à utiliser est java 8 ou plus.

Si l'agent est utilisé pour la première fois sur un réseau local, alors il est nécessaire de créer un fichier nommé "userId.txt" dans le même dossier que les scripts de lancement. Ce fichier indique un numéro d'identification, qui est différent pour chaque agent du réseau. Si vous voulez opter pour la création automatique de ce fichier, il faut lancer l'agent pour la première fois avec le serveur centralisé qui se chargera de l'attribution, sinon il faut le faire en choisissant manuellement un numéro dans ce fichier.



### Préparation pour une utilisation avec un serveur centralisé

Pour une utilisation avec un serveur centralisé, il faut configurer le fichier server.properties.



Le remplissage des champs se fait de cette façon :

- ❖ db\_url : URL de la base de données qui sera utilisé pour stocker les correspondances noms d'utilisateur <-> id et l'historique des messages. La base de données doit être bien configuré avant d'être utilisé par le système de Chat. Par défaut, la valeur de l'URL est la base de données que nous avons configuré pendant les séance de TP.
- ❖ db\_username et db\_passwork : (respectivement) login et mot de passe pour accéder à la base de données.
- ❖ server\_port : port sur lequel sera ouvert un serveur tcp, choisi entre 1024 et 65535, qui doit être le même que le champs centralized\_server\_port des fichiers client.properties

Dans le fichier client.properties, les champs local\_ip et broadcast\_ip ainsi que udp\_port sont configurés de la même manière que lors d'une utilisation en local. Pour un utilisateur externe, le champ broadcast\_ip peut être ignoré car le broadcast n'est pas utilisé dans ce cas d'utilisation. Les autres champs sont configurés de la manière suivant :



- ❖ centralized\_server\_port : il s'agit du port sur lequel le serveur écoute les demandes, d'où la nécessité de configurer le fichier server.properties en accord avec les fichiers client.properties.
- ❖ centralized : true.

Le lancement du serveur se fait ensuite grâce au script run\_server.sh (pour une utilisation sous Linux) ou run\_server.bat (pour une utilisation sous Windows).

Si l'on veut recréer la table servant au serveur, on peut utiliser le script mysql suivant:

```
CREATE TABLE messages (  
id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,  
from_user_id INT UNSIGNED,  
to_user_id INT UNSIGNED,  
content TEXT,  
creation_timestamp LONG  
);
```

# III Système de chat décentralisé

## A - Lancement de l'application

On distingue l'application client et le socket client qui sont deux choses différentes. L'application client désigne le programme java lancé en mode client utilisateur. L'application serveur désigne le même programme, mais lancé en mode serveur centralisé (celui dont on parle dans la partie IV de ce document). On fait la distinction entre les deux mode de lancement grâce aux arguments de la ligne de commande. Une fois lancée, l'application va chercher les autres paramètres dont elle a besoin dans un fichier de configuration. L'application client correspond à l'agent mentionné dans le cahier des charges. Le socket client quand à lui désigne une abstraction pratique dans le code du package "network" représenté par la classe java TCPClient, elle même basé sur la classe déjà existante du JRE java.net.Socket.

## B - Architecture du projet

C'est le même projet java qui contient le code de l'application client et du serveur centralisé. La classe de démarrage est la même dans les deux cas, on fait la distinction avec l'argument de la ligne de commande. Si l'on voulait faire une version dépourvu du code du serveur pour les clients il suffirait de supprimer le package application.server (ou de ne pas l'exporter dans le JAR du client si l'on voulait faire deux JAR différents).

Le projet est découpé en 6 packages:

- application
- application.client
- application.server
- database
- network
- utils

Les packages database et utils sont chacun indépendant du reste du projet (pas du JRE). Le package network n'a besoin que du package utils. Le package database contient une classe représentant la base de donnée et des classes pratiques pour effectuer les requêtes. Le package network contient des classes qui ont été codées pour ce projet et dont le but est de simplifier l'utilisation de TCP et UDP, mais elles pourraient aussi être ré-utilisées car il n'y a rien de propre à cette application de clavardage dans ce package. Dans le package application.client il y a les classes propres à l'application client, dans le package application.server il y a les classes propres au serveur centralisé, et dans le package application il y a les classes communes aux deux côtés (qui servent par exemple pour la communication ou la représentation des données qui doivent souvent être la même des deux côtés).

## C - Découverte des utilisateurs connectés

Pendant que l'application client fonctionne, elle écoute les paquets UDP qui arrivent afin de connaître les utilisateurs connectés sur le même réseau local (le port d'écoute UDP est configurable). Quand un utilisateur réussit la phase de login, l'application client passe en mode "logged" et elle va envoyer régulièrement un paquet UDP en mode broadcast contenant l'id unique de l'utilisateur, son pseudo ainsi que l'adresse ip locale sur laquelle on peut le contacter. Quand une application client reçoit un de ces paquets, elle démarre un timer (et reset l'ancien timer si il y en avait un associé à cet utilisateur). Quand ce timer arrive à 0, cela signifie qu'on peut considérer l'utilisateur comme déconnecté et l'application client agit en conséquence afin de mettre à jour la liste des utilisateurs connectés dans la fenêtre graphique. Tant qu'un utilisateur reste connecté, les autres applications client recevront un signe de sa présence avant que le timer arrive à 0, mais s'il se déconnecte (ou qu'il reste connecté mais qu'il devient injoignable à cause de n'importe quel problème au niveau du réseau), alors les autres clients en seront informés dès que le timer sera arrivé à 0. Ceci permet de maintenir une liste des utilisateurs qui sont vraiment connectés (qui sont joignables). Nous avons fait ce choix car nous préférons cette solution à une solution qui enverrait un paquet au moment de la déconnexion afin de mettre à jour les listes des autres utilisateurs du système, cette dernière n'étant pas résistante aux problèmes réseau.

## D - Echange de messages

Chaque application client lance un socket serveur (le port d'écoute est configurable), afin de réceptionner les demandes de clavardage venant d'autres utilisateurs. Quand un utilisateur A ouvre une fenêtre de chat afin de parler à un utilisateur B et qu'il lui écrit un message, l'application client de A ouvre un socket client et le connecte sur le socket serveur de l'application client de B. Si, après cela, B écrit à A, il ne créera pas un nouveau canal mais utilisera celui déjà existant qui correspond à la session de clavardage entre A et B. Tous les messages écrits entre A et B transiteront par ce canal propre à ce couple d'utilisateurs. Les messages de clavardage envoyés et reçus sont conservés dans la RAM jusqu'à ce que l'application client soit fermée, ce qui permet de retrouver ces messages lorsque on ferme puis ré-ouvre une session de clavardage sans avoir besoin d'un système d'historique tel que celui présenté dans la partie IV. Bien sur les messages sont perdus lorsque qu'on ferme l'application client.

## E - Renommage des pseudos et gestion des conflits

Les pseudos (ou logins) ne sont pas associés à un mot de passe, car n'ayant pas de serveur centralisé dans cette partie, il serait inutilement compliqué de concevoir un système distribué préservant l'authenticité des utilisateurs car d'après le cahier des charges ce n'est pas le but de ce système de clavardage. Dans la suite on utilise le terme pseudo pour désigner la chaîne de caractère que choisit l'utilisateur au moment de se connecter, et on utilise le

terme login pour désigner l'action de se connecter au système de clavardage avec un certain pseudo. La phase de login doit garantir l'unicité du pseudo parmi les utilisateurs qui sont logués au sein d'un même réseau local. Lors d'un renommage, l'unicité du nouveau pseudo doit aussi être garantie. Au niveau de l'implémentation nous avons considéré chaque tentative de renommage comme une déconnexion suivie d'une tentative de login avec le nouveau pseudo, et en cas d'échec la phase de renommage se terminera par la reconnexion avec l'ancien pseudo. L'utilisateur ne verra pas de différence car cela est automatique. Ça a l'avantage de se ramener à l'implémentation d'une seule fonctionnalité gérant les conflits: la phase de login.

La phase de login d'un client A se déroule comme ceci:

- ❖ Au temps  $t_0$  il envoie une demande de login (un paquet qui contient le pseudo voulu ainsi qu'un discriminant est émis en broadcast).
- ❖ Une seconde plus tard au temps  $t_1$ , il fait une validation de la phase de login qui peut se dérouler des 3 façons suivantes:
  - Pour cela, il vérifie n'avoir reçu aucune demande de login conflictuelle (une demande avec le même pseudo) et qu'aucun autre utilisateur n'est connecté avec le pseudo que voulait l'utilisateur A. Si c'est le cas le client A se considère comme logged et commence à émettre régulièrement afin de montrer sa présence aux autres clients.
  - S'il a reçu des demandes conflictuelles, il procède à une résolution des conflits grâce au discriminant contenu dans chaque demande de login. Il n'y a qu'un seul vainqueur par résolution de conflits portant sur le même pseudo. S'il est le vainqueur il se considère comme logged et commence à émettre régulièrement afin de montrer sa présence aux autres clients.
  - Si la résolution le désigne comme perdant ou qu'un autre utilisateur ayant le même pseudo est déjà connecté, il affiche un message à l'utilisateur afin de l'informer qu'un autre utilisateur avec le même pseudo est déjà connecté (et se reconnecte avec l'ancien pseudo si c'était une tentative de renommage).

Si le client A se retrouve dans le premier ou le deuxième des 3 cas possibles au niveau de la validation de la phase de login et qu'il se considère donc comme connecté, il peut être sûr qu'aucun autre client du même réseau local ne se considèrera aussi comme connecté avec le même pseudo pour les raisons suivantes:

On se place dans le cas suivant: le client A au moment de la validation ne voit aucun autre utilisateurs connecté avec le pseudo qu'il voulait. Le client A émet la demande au temps  $t_0$  et procède à la validation au temps  $t_1$  une seconde plus tard. On suppose que le délais de transmission d'un paquet entre deux machines du réseau local n'est jamais supérieur à  $D_{max}$ . Le système est censé garantir l'unicité des pseudos des utilisateurs connectés lorsque  $D_{max}$  est strictement inférieur à 500ms. Le client A va traiter les demandes de login conflictuelles qu'il a vu avant sa phase de validation, les demandes de login conflictuelles qu'il ne voit pas car elles arrivent trop tard (après sa phase de validation) ne sont pas un problème pour la raison suivante.

Si d'autres clients ont émis des demandes de login conflictuelles qui arrivent après le temps  $t_1$  chez le client A ( $t_1$  est le temps auquel la phase de validation se déroule), elles auraient été nécessairement émises après le temps  $t_2 = t_1 - D_{\max}$  (à cause du délai de transmission), ce qui veut dire que leur phase de validation se serait déroulée après le temps  $t_3 = t_2 + 1s$ . Au moment où le client A se considère connecté (à l'instant  $t_1$  de la validation) il envoie le premier paquet de ceux visant à montrer sa présence aux autres clients, ce premier paquet arrive chez les autres clients après le temps  $t_4 = t_1 + D_{\max}$ . Or  $t_3 - t_4 = (t_2 + 1s) - (t_1 + D_{\max}) = (t_1 - D_{\max} + 1s) - (t_1 + D_{\max}) = t_1 - D_{\max} + 1s - t_1 - D_{\max} = 1s - 2 * D_{\max}$ , et comme  $D_{\max} < 0.5s$ , la différence  $t_3 - t_4$  est strictement positive ce qui veut dire que la phase de validation des autres clients se produit forcément après avoir vu la présence du client A comme connecté, ce qui permet à la validation de ces clients d'échouer. Ça signifie que chaque demande conflictuelle que le client A ne voit pas car elle arrive trop tard n'est pas un problème car l'utilisateur correspondant ne pourra pas être connecté. Il faut par contre qu'il soit capable de régler les conflits qu'il voit.

Pour assurer qu'un seul client parmi les demandes conflictuelles aura le droit de se connecter, on utilise le discriminant associé à chaque demande de login qui est un entier sur 64 bits généré aléatoirement grâce à la classe du JRE `java.security.SecureRandom`. Un client se considère gagnant si son discriminant est plus petit que tous les discriminants des demandes conflictuelles. Si deux utilisateurs tentent de se connecter à peu près au même moment avec le même pseudo est que le discriminant généré est par malchance identique, il se peut qu'il y ait un problème, mais ce cas a une chance sur plusieurs milliards de milliards de se produire, et le risque que l'implémentation ne soit pas parfaite et génère d'autres bugs est bien plus probable. Le risque est par conséquent peu grave dans cette situation, surtout qu'un système plus compliqué que celui mais qui ne présente pas en théorie ce défaut, aurait été plus compliqué à implémenter augmentant aussi la potentielle présence de bug non vus.

Le seul cas où le client A aurait tort de se considérer comme gagnant est le cas où un autre client B (voulant se connecter avec le même pseudo) se considérerait aussi comme gagnant, ce qui voudrait dire qu'il n'a pas vu la demande de login conflictuelle du client A (car sinon il aurait fait la même conclusion que le client A à partir des discriminants, c'est à dire que le client A gagne et que le client B perd), or on a déjà vu que dans ce cas ça signifie que la demande conflictuelle que le client B n'a pas vu ne pose pas de problème (seules les demandes conflictuelles qui sont vues sont potentiellement problématiques, mais dans ce cas elles seront vues par chaque client impliqué dans le conflit ce qui permet d'avoir une résolution identique sur chaque client). Pour que ce qui précède reste vrai, il faut que chaque client commence par écouter les paquets qui arrivent en broadcast pendant 1 seconde, et c'est la raison pour laquelle le bouton login dans l'interface graphique ne s'active qu'au bout de 1 seconde. On a donc en théorie jamais 2 utilisateurs connectés avec le même pseudo.

## IV Nouvelles fonctionnalités basées sur un serveur centralisé

### A - Attribution automatique d'identifiant utilisateur

Quand une application client se connecte pour la première fois au serveur centralisé, celui-ci lui attribue un identifiant unique, et l'application client va le stocker dans un fichier nommé `userId.txt`. Cela permet de simplifier la configuration qui était jusque là manuelle. Pour cela le serveur centralisé va stocker cet identifiant dans une table de la base de donnée auquel il est connecté afin de savoir que cet identifiant est déjà utilisé et de ne pas le prendre lors de l'attribution du prochain identifiant utilisateur.

### B - Stockage et récupération de l'historique des messages

Lorsqu'on lance l'application client en mode centralisé et qu'on ouvre une session de clavardage avec quelqu'un, l'application client va demander au serveur centralisé l'historique des messages déjà échangés avec cet utilisateur, et le serveur va lui répondre après avoir été chercher ces informations dans la table de la base de donnée (c'est stocké dans la même table que précédemment car ça ne gêne pas l'attribution des ids).

En mode centralisé, les applications client internes à l'entreprise envoient les messages qu'elles échangent à leur destinataire, mais également au serveur centralisé ce qui lui permet de sauvegarder l'historique des messages dans la base de donnée.

On peut aussi effectuer des actions sur le serveur centralisé lorsque celui-ci est lancé, comme par exemple `"db show messages all"` qui affichera la liste des messages stockés dans la base de donnée.

### C - Utilisation du système de chat par des utilisateurs externes

En mode centralisé, les utilisateurs externes à l'entreprise peuvent quand même utiliser le système. Quand un utilisateur se connecte le serveur centralisé va lui envoyer la liste des utilisateurs connecté, car les applications client des utilisateurs externes n'émettent ni ne reçoivent aucun paquet broadcast, c'est donc de cette manière qu'ils ont accès à cette information. Le serveur centralisé, quand à lui, maintient cette liste à jour grâce au fait que chaque application cliente (qu'elle soit interne ou externe) va informer le serveur lors de sa connection ainsi que de sa déconnection, et le serveur centralisé va aussitôt informer tous les autres applications clientes de cette mise à jour afin que chaque utilisateur voit la liste des utilisateurs connectés.

Afin de mieux séparer le réseau local de l'extérieur de l'entreprise, nous avons fait le choix suivant: les communications avec les utilisateurs externes passeront par le serveur centralisé, autrement dit applications client des utilisateurs externes ne se connecteront pas

directement à l'application cliente avec laquelle elle souhaite communiquer, sinon ça nécessiterait que chaque utilisateur se trouvant derrière un NAT fasse une redirection de port afin de pouvoir être joint par d'autres sans passer par le serveur centralisé. Au niveau des échanges avec l'extérieur de l'entreprise, il y a donc uniquement la machine sur lequel le serveur centralisé se trouve qu'il faudra gérer. Pour cela, quand un utilisateur ouvre une session avec un autre utilisateur, si l'un des deux au moins est externe, alors l'application cliente va créer un "tunnel" à dont la destination est l'autre utilisateur. Ensuite elle va utiliser ce tunnel afin d'échanger tous les paquets habituels qui seront encapsulé dans des paquets propre au tunnel jusqu'au serveur centralisé qui se chargera de les retransmettre au destinataire qui le décapsulera et le traitera ensuite comme n'importe quel autre paquet.

## V Validation et tests

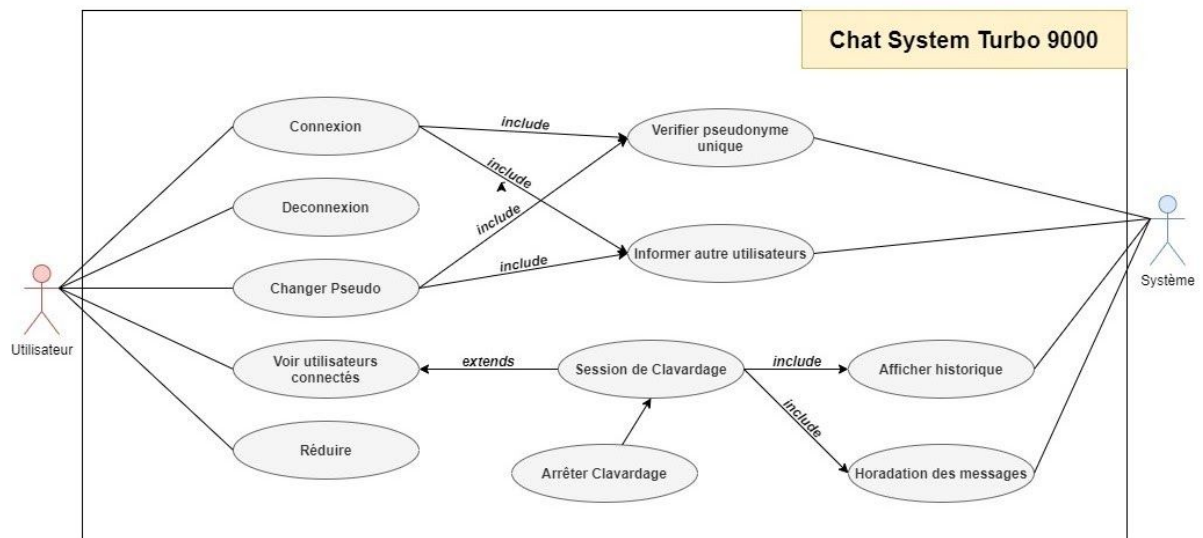
Pour tester nous devons utiliser deux machines différentes. Il y a plusieurs raisons à cela: le port d'écoute udp ainsi que le port du serveur local tcp que doivent ouvrir chaque agent entrent en conflit si l'on veut lancer plusieurs agents sur la même machine. Il aurait pour cela fallu une configuration plus compliqué. Il y a aussi le fait que chaque agent va lire dans le fichier `userId.txt` son numéro d'utilisateur, qui sera donc le même si l'on lance deux fois l'agent sur la même machine.

Les tests se sont donc déroulés principalement en ouvrant la même session utilisateur insa sur deux ou trois machines (pour accéder aux mêmes fichiers dès que l'on fait une modification dans le code). Une seule instance d'eclipse peut cependant être lancée à la fois, donc sur les autres machines nous utilisons des scripts qui passaient des options supplémentaires afin d'indiquer dans quel fichier trouver l'identifiant de l'utilisateur (qui normalement se trouve dans `userId.txt`, mais qui là se trouvait dans `userId1.txt` pour la machine 1, et `userId2.txt` pour la machine 2). Cela permettait d'avoir un user id différent pour chaque machine et de tester l'application facilement sans avoir à push les fichiers sur git puis pull sur une autre session puis compiler afin d'avoir le même programme sur plusieurs machines afin de le tester, ce qui aurait été bien plus long. Après une modification dans le code, il ne fallait donc que quelques secondes pour lancer les applications sur les 2 machines, ce qui a pas mal faciliter le débogage.

Il aurait peut être été possible de faire des tests junit, mais ça aurait été plus compliqué que de faire des tests junit sur une application fonctionnant sur une seule machine.

# Annexes - Conception

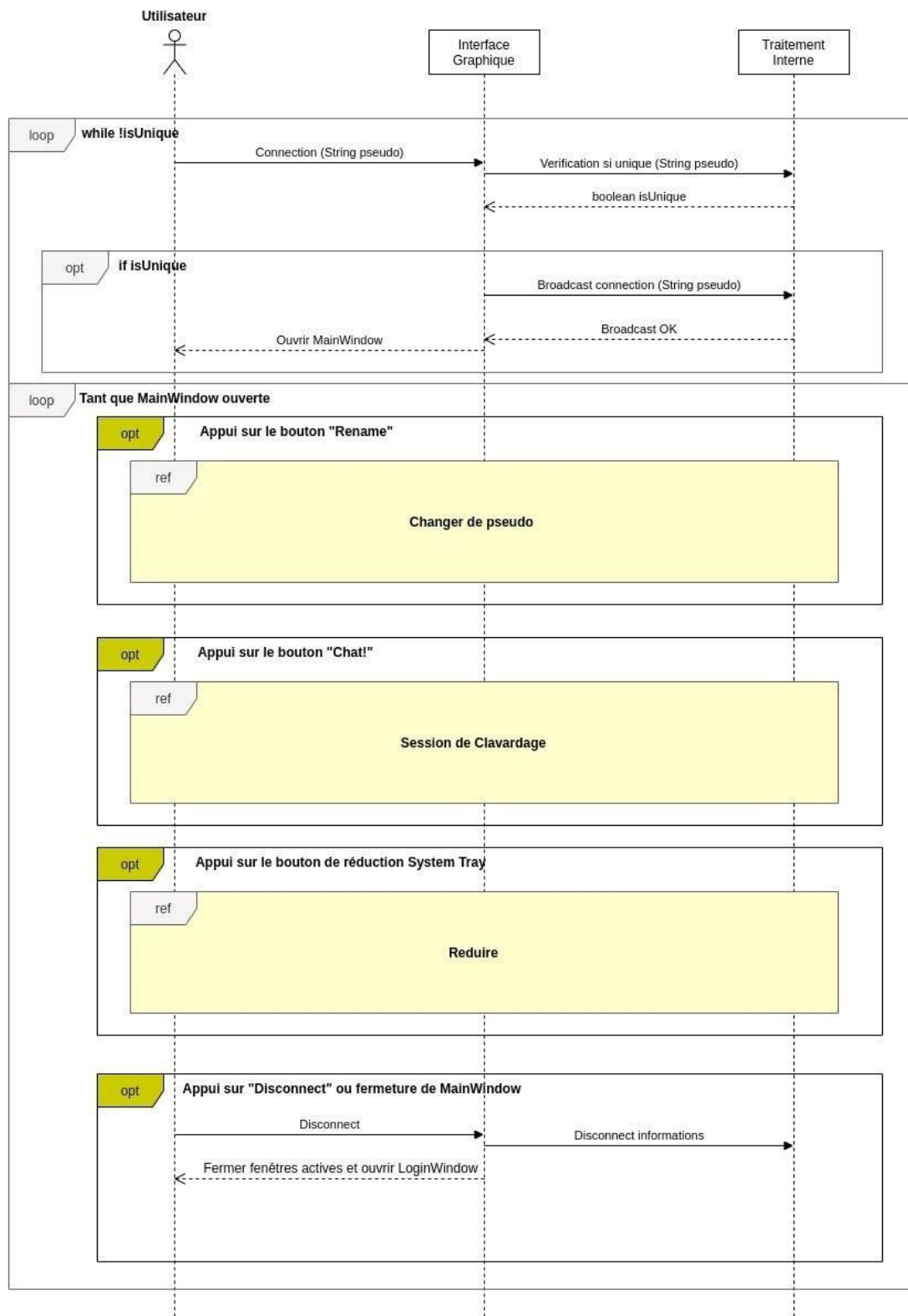
## Use Case



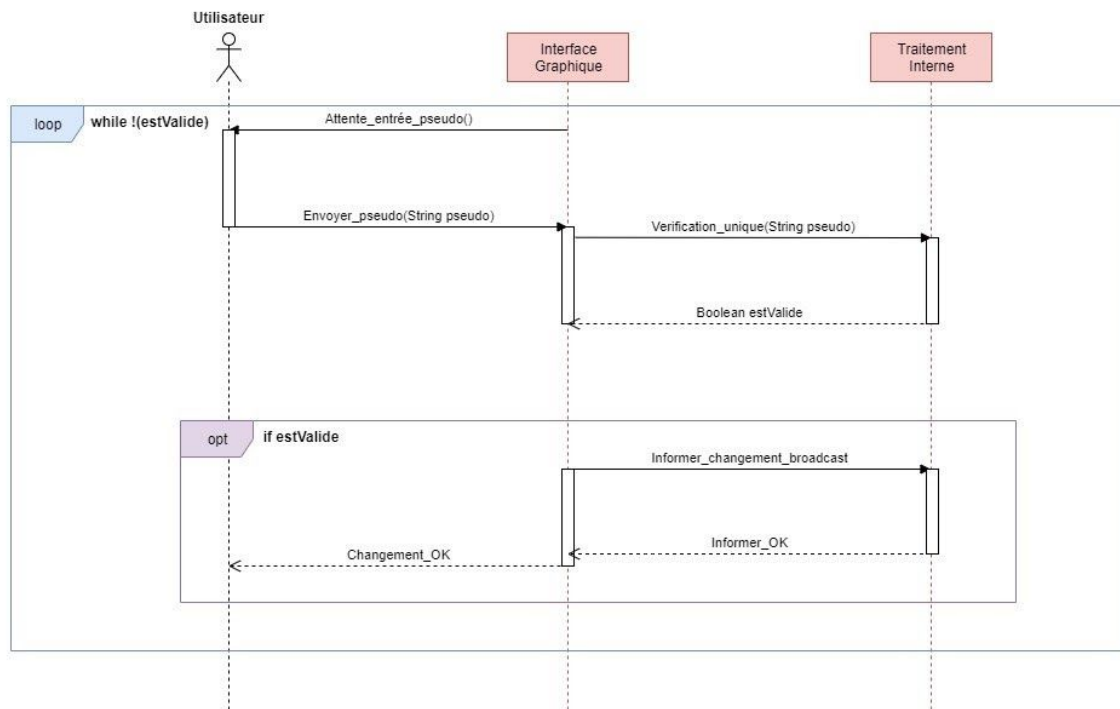


# Diagrammes de séquence

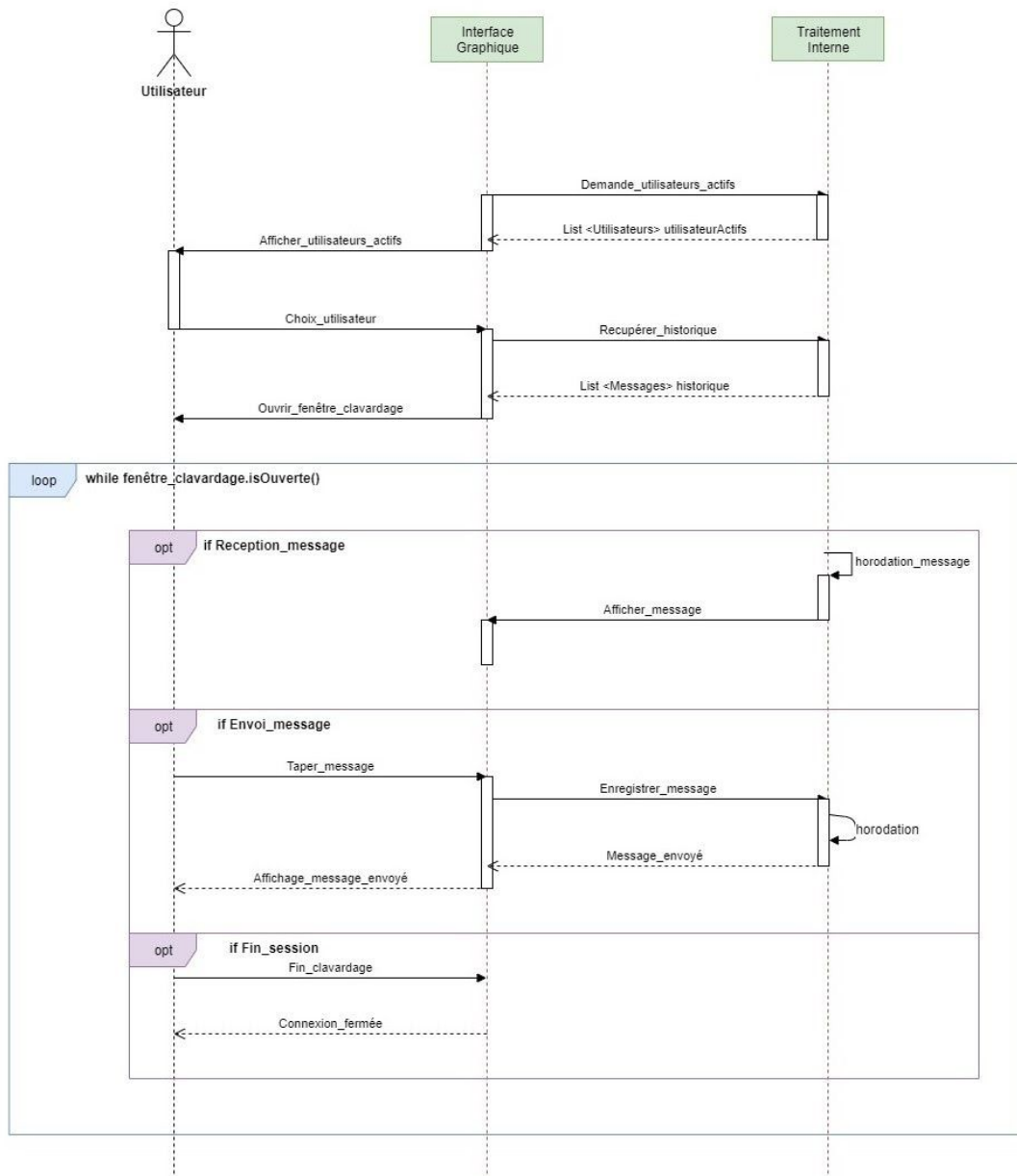
## Utilisation générale



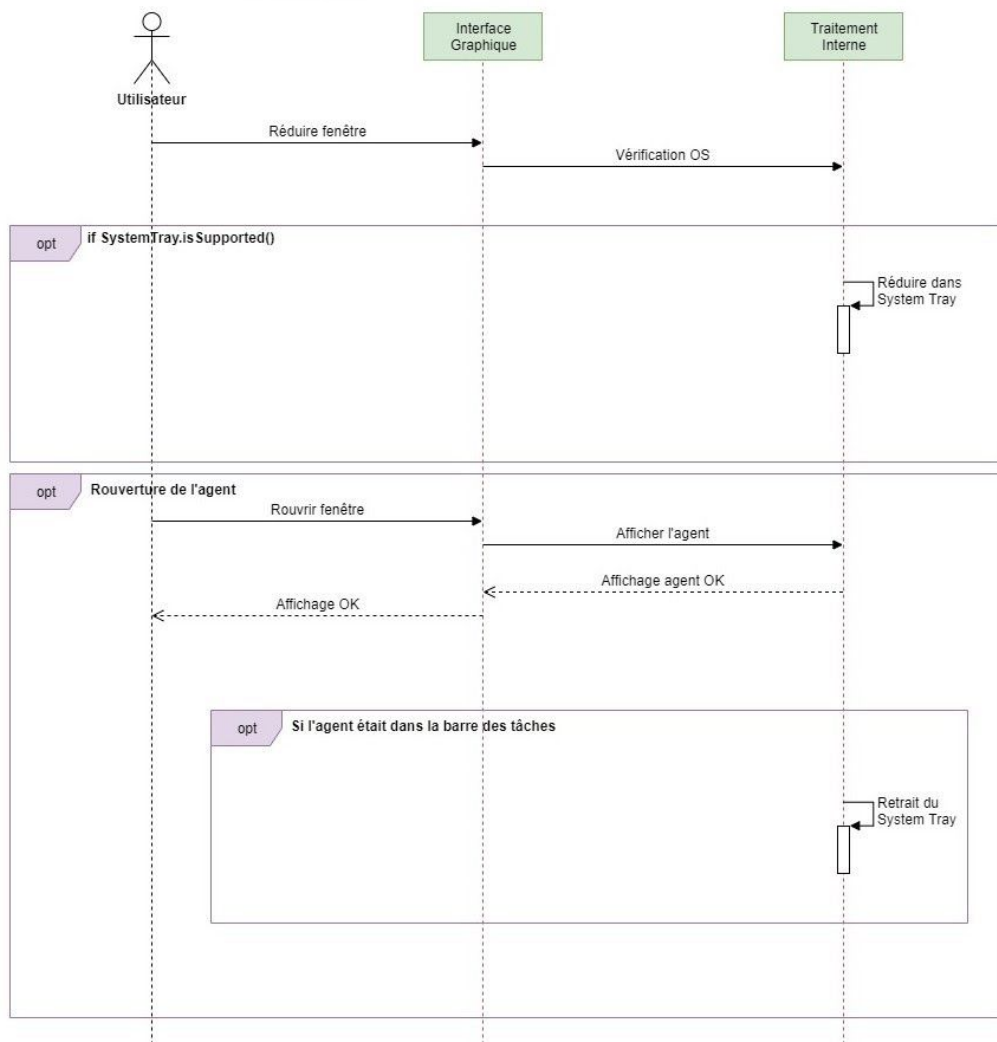
## Changer de pseudo



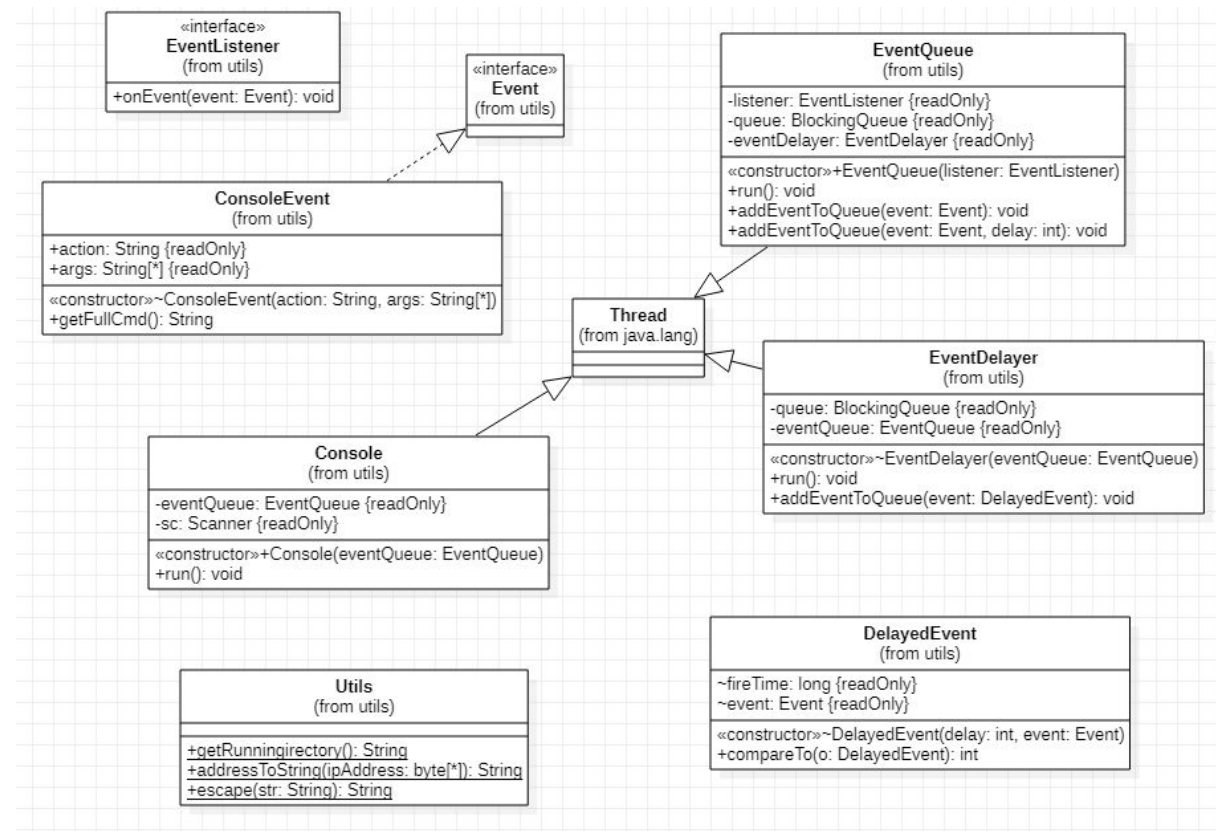
## Session de Clavardage



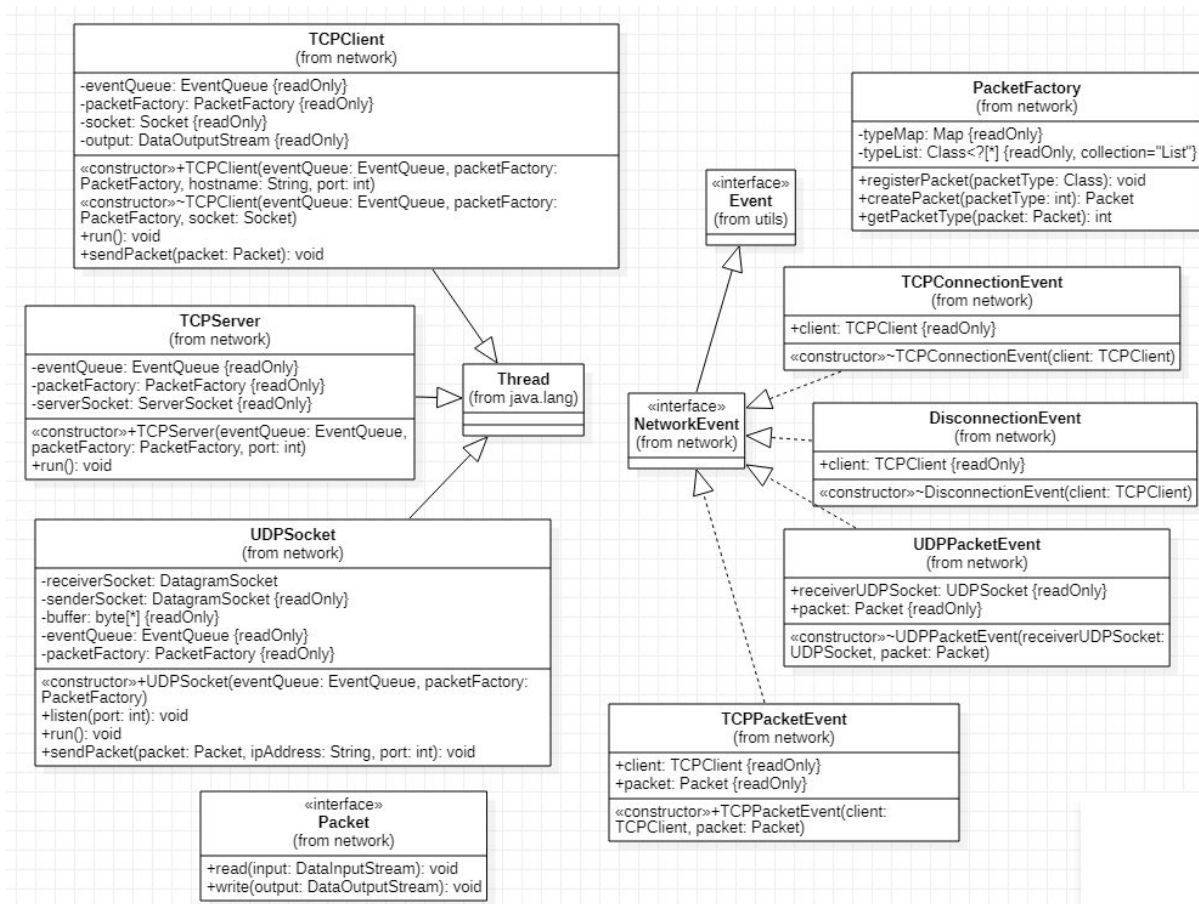
## Reduction



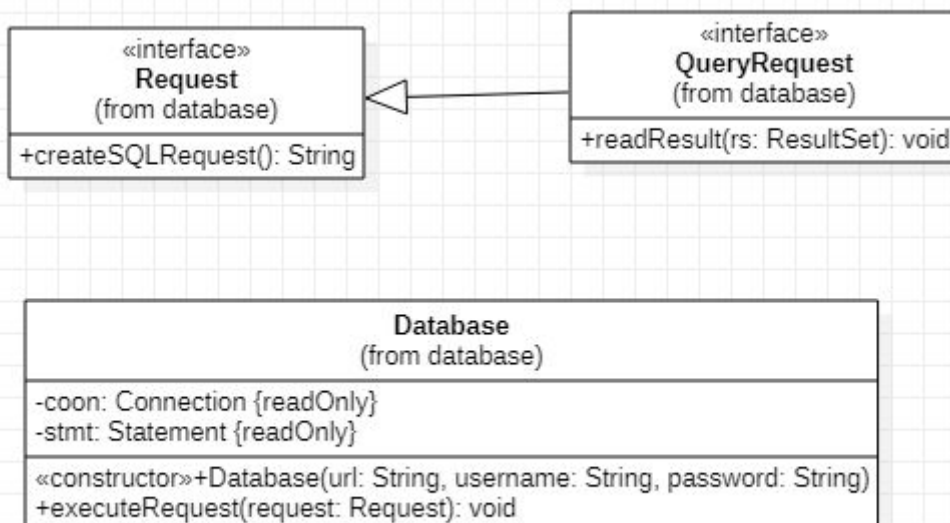
## Package utils



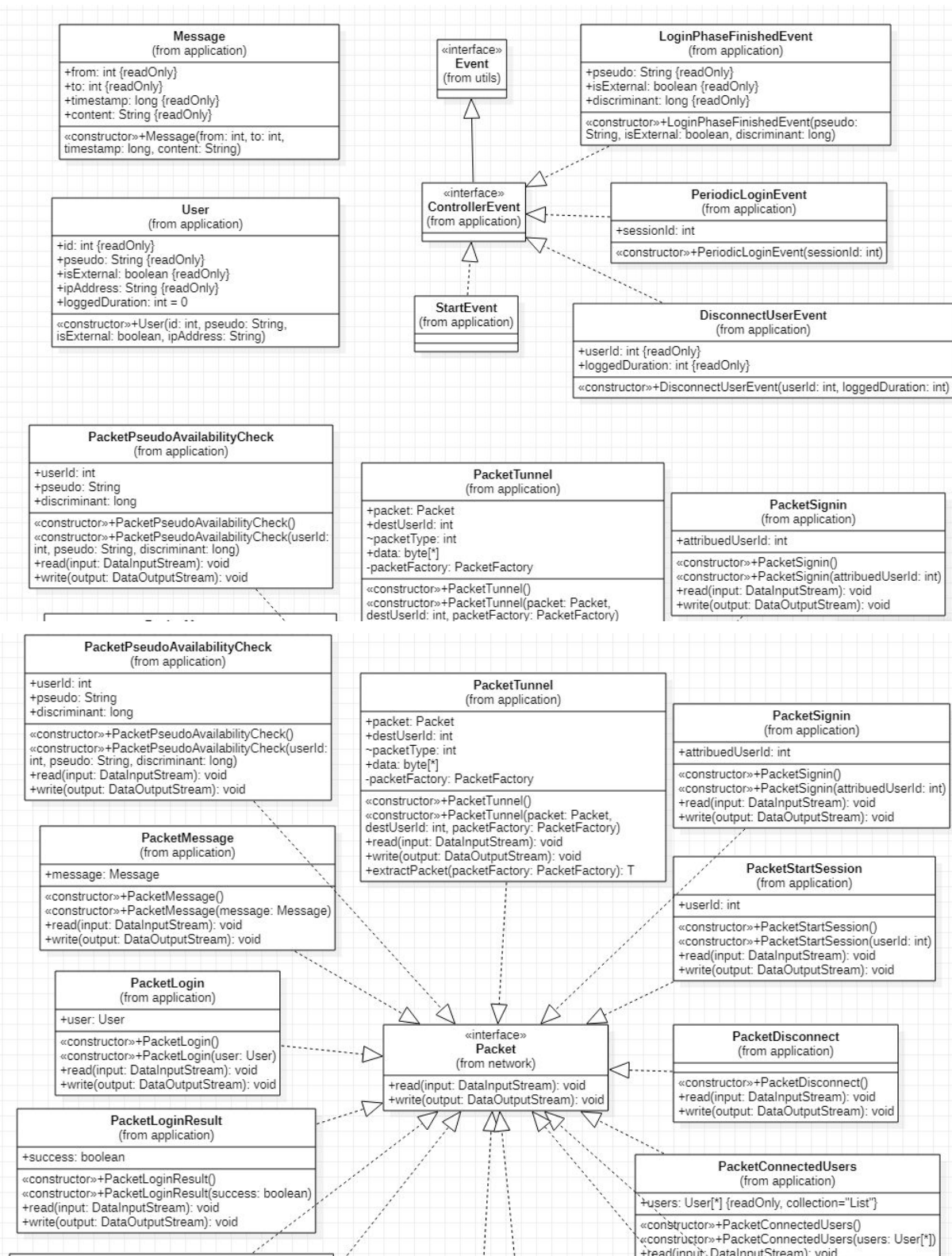
## Package network

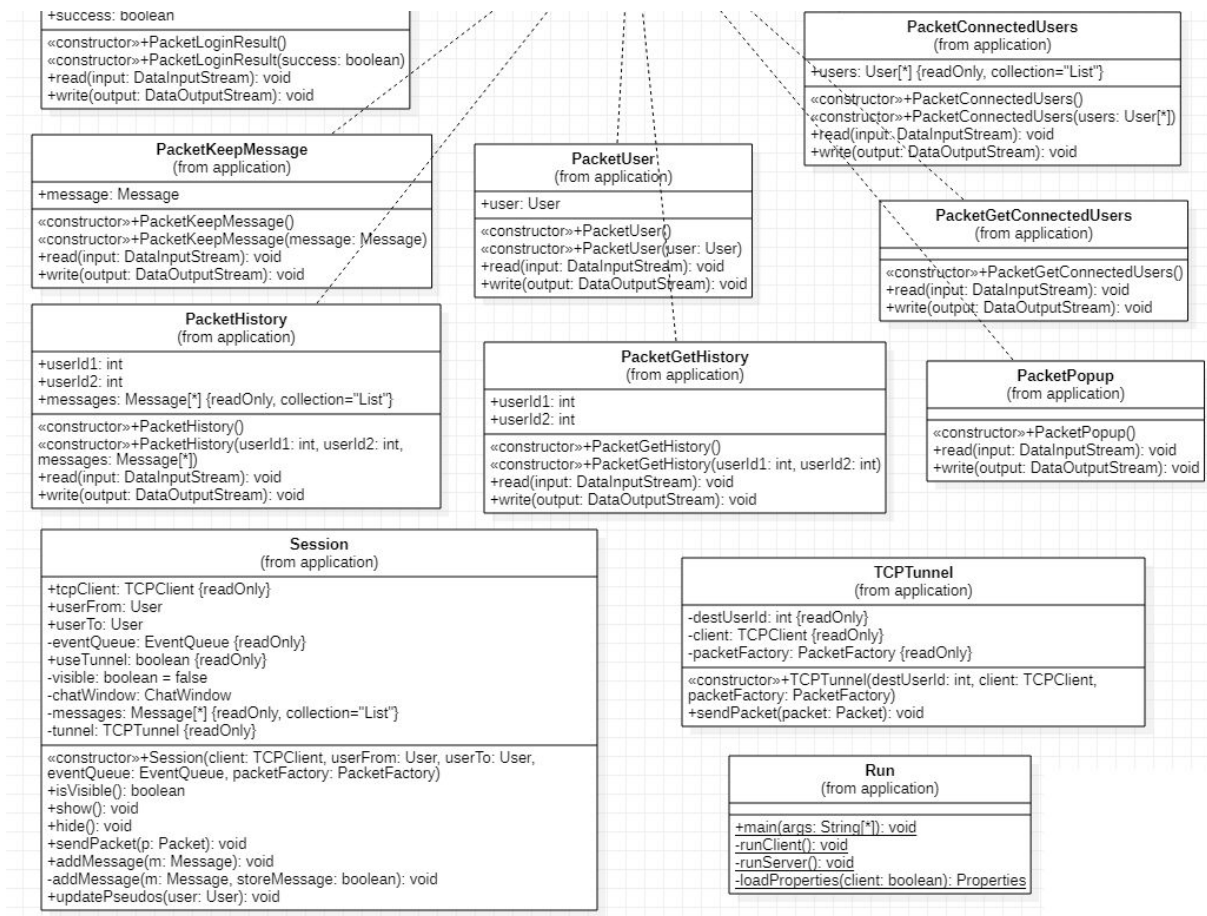


## Package database

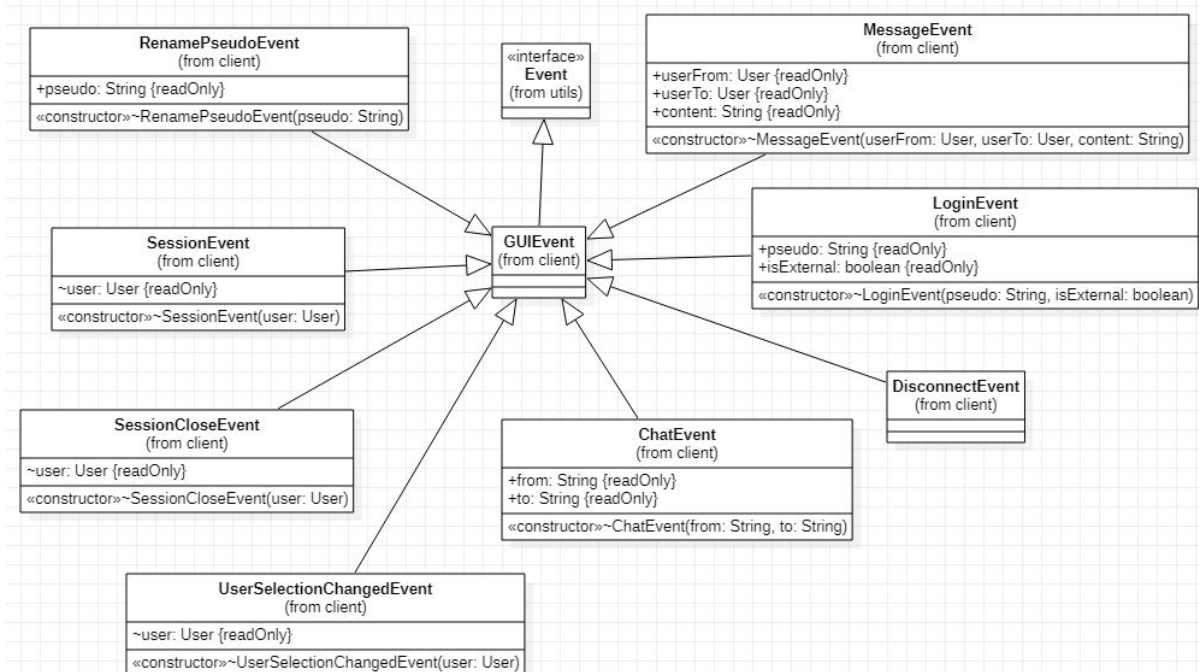


## Package application

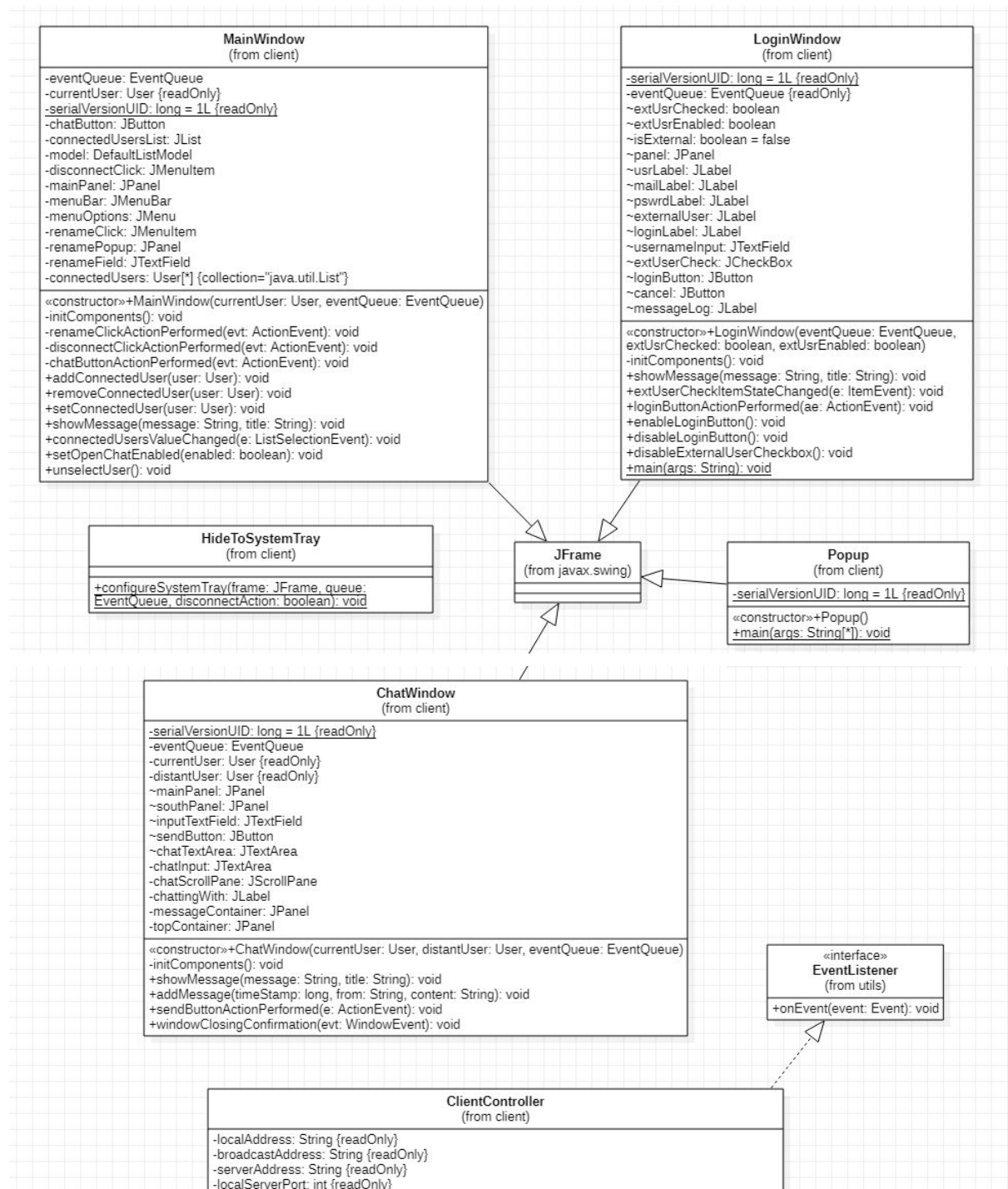




## Package application.client









## Package application.server

