


# Rapport projet logiciel de clavardage

28 janvier 2022  
Hugo BARRAL  
Oriane BERRY





# SOMMAIRE

<b>Introduction .....</b>	<b>1</b>
<b>Conception.....</b>	<b>1</b>
Diagramme des cas d'utilisation .....	1
Diagrammes de séquence .....	2
Diagrammes de classes.....	9
Diagramme de structure composite .....	14
Diagramme d'états .....	14
<b>Architecture et choix technologiques.....</b>	<b>15</b>
JavaFX et SceneBuilder .....	15
SQLite et cache.....	15
<b>Gestion de projet .....</b>	<b>16</b>
<b>Procédures de tests et déploiement .....</b>	<b>16</b>
<b>Installation.....</b>	<b>17</b>
<b>Manuel d'utilisation .....</b>	<b>17</b>
<b>Annexes .....</b>	<b>A</b>



## Introduction

Dans le cadre de l'UF POO/COO nous avons dû réaliser un logiciel de clavardage répondant à un cahier des charges précis.

Notre logiciel devrait permettre de pouvoir envoyer et réceptionner des messages entre deux machines connectées sur un même réseau.

Les messages échangés peuvent être de simples messages de type texte ou bien des fichiers de type PDF, PNG par exemple.

Notre logiciel fonctionne sur les trois systèmes d'exploitation Windows, Linux et OS X. Notre logiciel comporte également des fonctionnalités telles que l'horodatage des messages, l'affichage de l'historique des précédentes sessions de clavardage, l'unicité des pseudos.

## Conception

Cette première partie du rapport contient les diagrammes de conceptions débutés avant la phase de développement du projet et corrigés après celle-ci.

### Diagramme des cas d'utilisation

La *figure 1 : Diagramme des cas d'utilisation*, situé en page A des annexes, nous montre tous les cas d'utilisation couverts par notre logiciel. Ces cas d'utilisations répondent aux demandes fonctionnelles du cahier des charges.

# Diagrammes de séquence

## Connexion

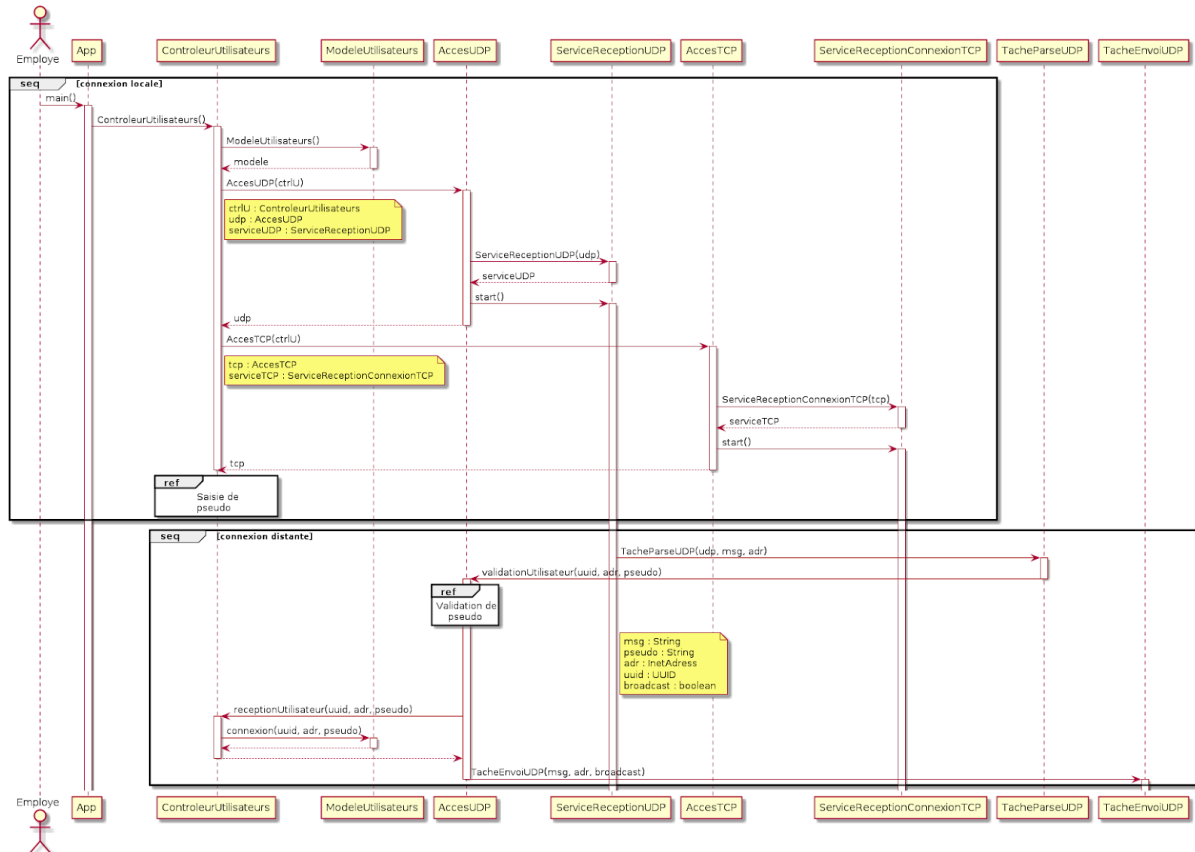


Diagramme de séquence de connexion

La connexion locale arrive dès le lancement de l'application, elle charge l'interface principale en attendant que l'utilisateur inscrive son pseudo. Le modèle de gestion des données utilisateurs, les services de réception de messages UDP et de connexion TCP sont initialisés puis l'interface de saisie de pseudo apparaît.

Lors d'une connexion distante et après la validation du pseudo, le service UDP notifie le contrôleur des utilisateurs qui indique la connexion au modèle.

## Saisie du pseudo

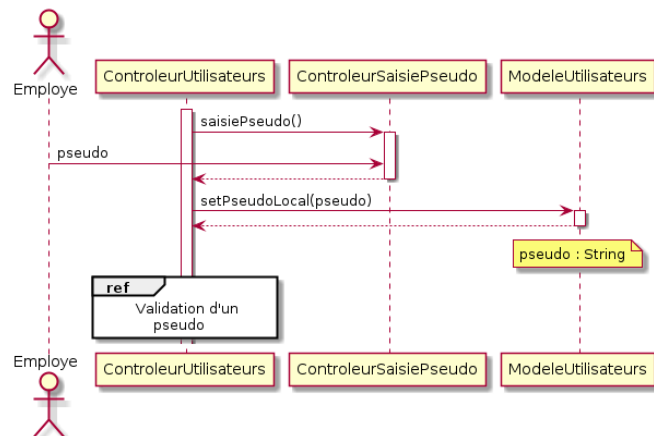


Diagramme de séquence de saisie du pseudo

Lorsque l'interface de saisie apparaît, l'utilisateur est invité à insérer un pseudo. Il est ensuite enregistré par le logiciel qui passe en phase de validation.

## Validation d'un pseudo

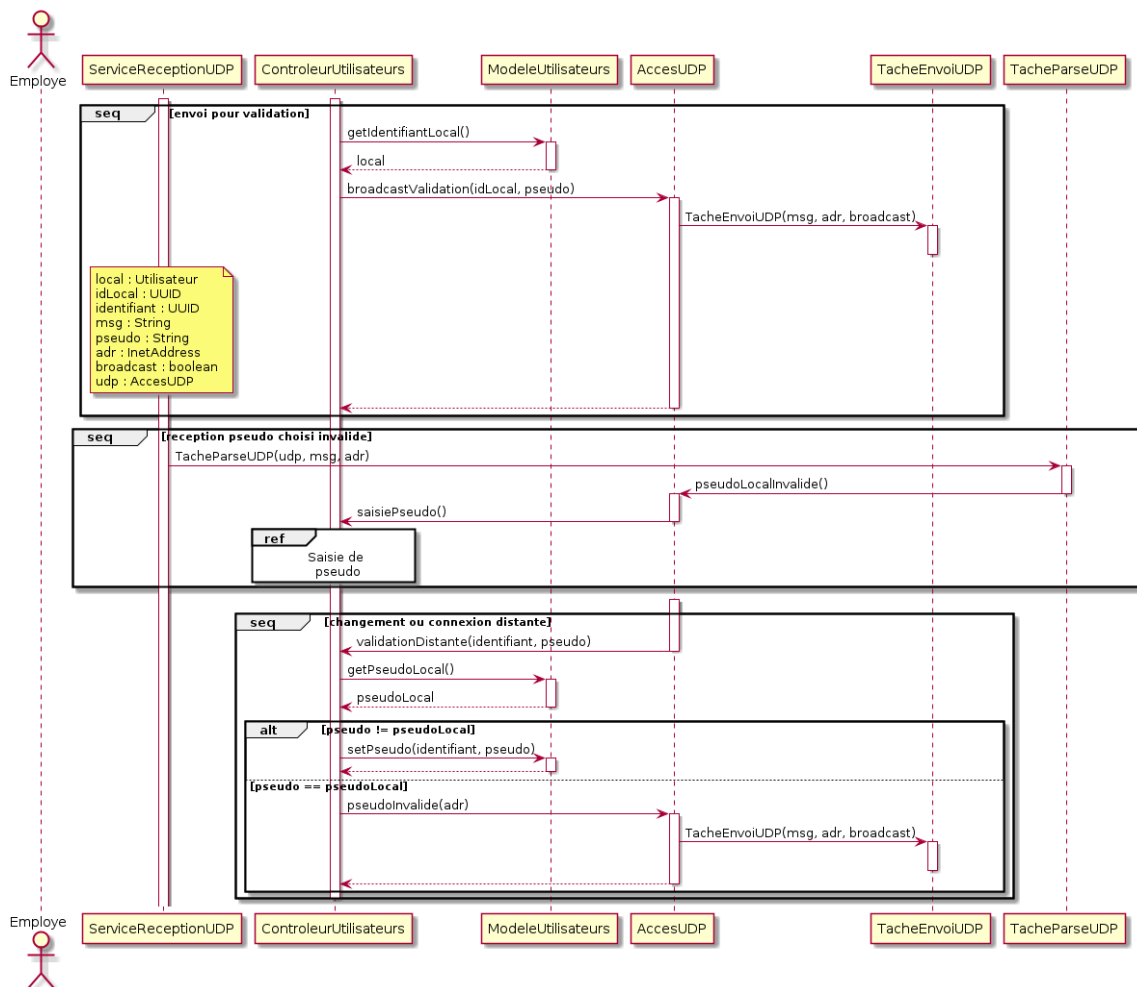


Diagramme de séquence de validation d'un pseudo

Après la saisie, le service UDP envoie à l'ensemble du réseau sa demande de validation du nouveau pseudo.

Si l'on reçoit un message d'invalidité de la part d'une machine sur le réseau, alors l'application demandera à l'utilisateur de saisir un pseudo différent.

Lorsque l'application reçoit une demande de validation de pseudo distante, elle va vérifier si celui-ci est bien différent de son pseudo local. Si ce n'est pas le cas, elle envoie donc un message d'invalidité à la machine concernée.

## Demande de session

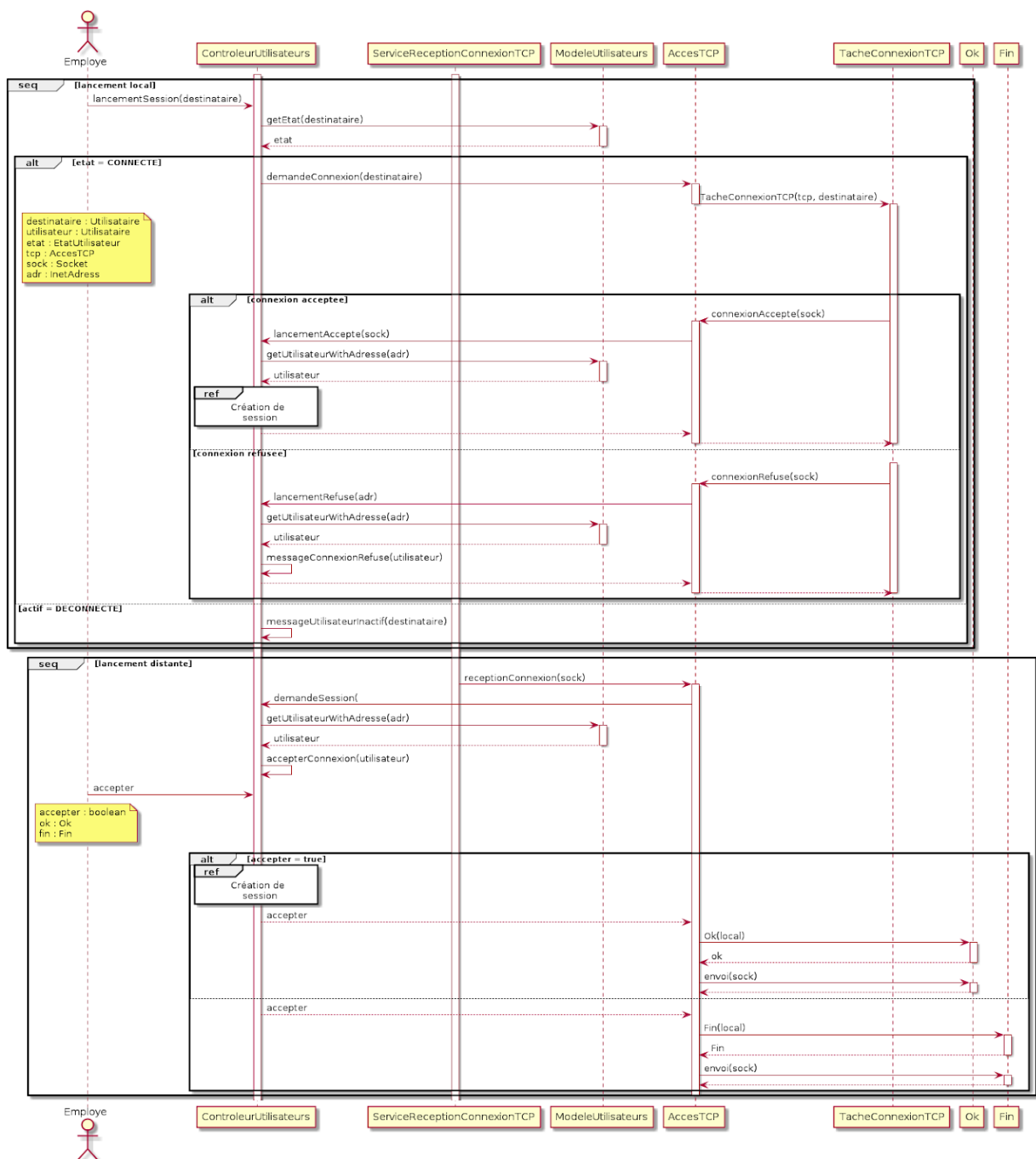


Diagramme de séquence de demande de session



Dans le cadre d'un lancement local, l'utilisateur sélectionne un destinataire et le logiciel vérifie si ce dernier est bien connecté. Si ce n'est pas le cas, l'utilisateur est notifié et le lancement s'arrête là. Sinon, le service TCP envoie une demande de connexion au destinataire choisi, selon sa réponse à la demande la session de discussion est créée ou l'utilisateur est notifié du refus du destinataire.

En cas de réception d'une demande distante, l'utilisateur est notifié de la demande par le logiciel. S'il accepte, la session est créée. S'il refuse, l'utilisateur distant est notifié du refus.

Les deux cas présentés sont valables pour des sessions de discussions, les sessions de consultation d'historique ne nécessitent pas de notifier un utilisateur distant.

### Création d'une session

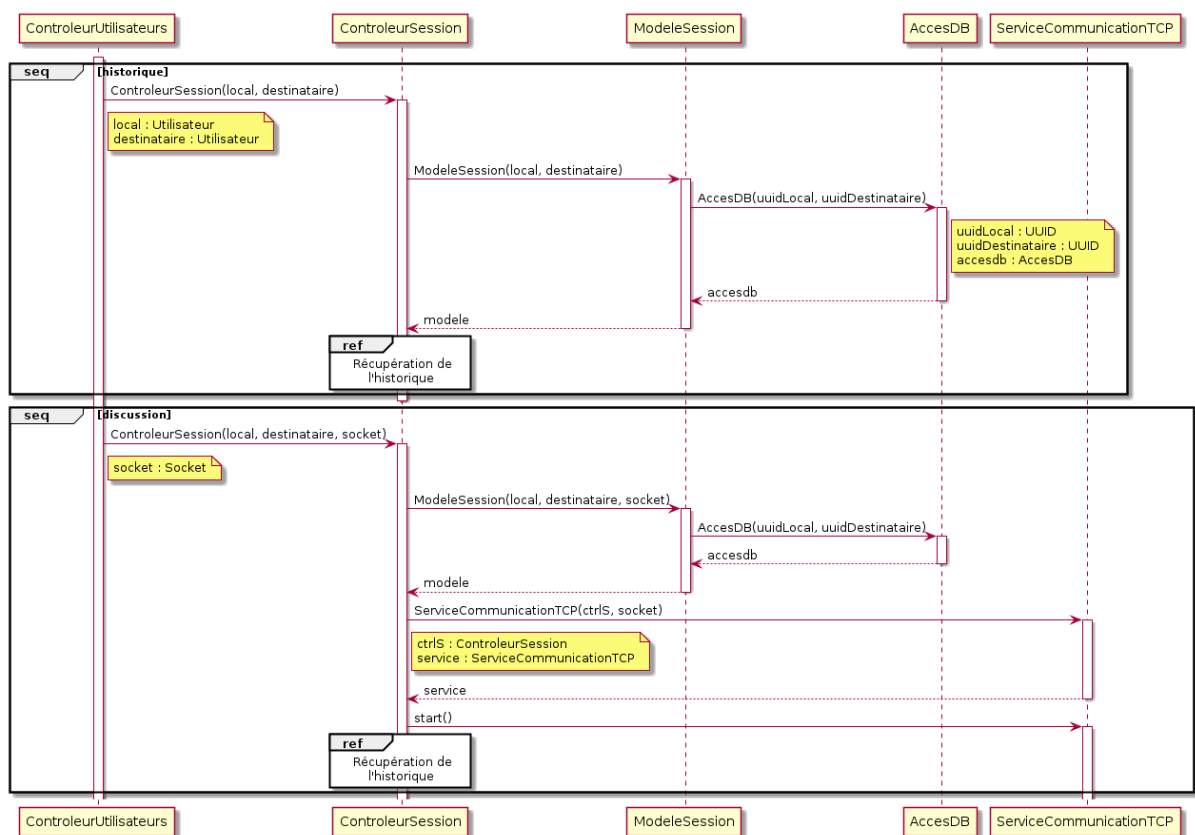


Diagramme de séquence de création d'une session

Il est possible de créer deux types de sessions, une session de consultation d'historique et une session de discussion. À la création de la première, on appelle seulement à l'initialisation du modèle et la récupération de l'historique à consulter. Pour la seconde, il convient également de lancer un service de réception des messages propre à la discussion.

## Récupération de l'historique

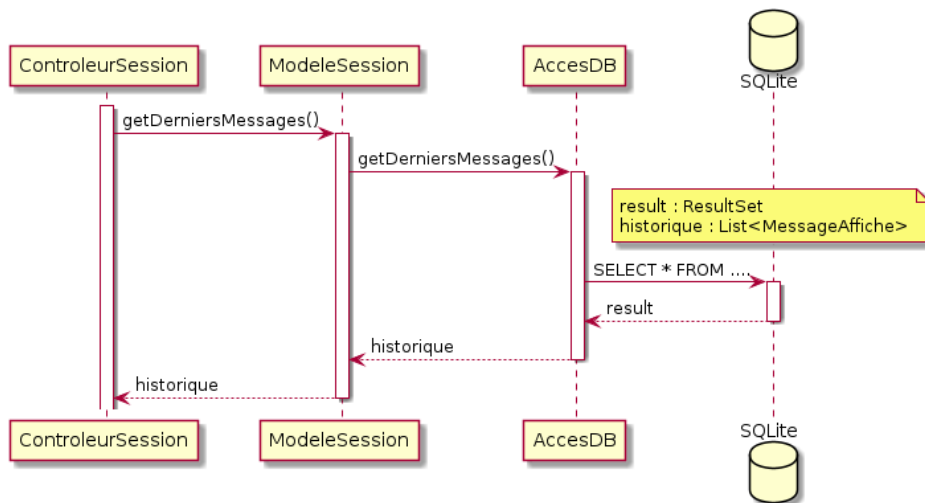


Diagramme de séquence de récupération de l'historique

La récupération de l'historique passe par la classe AccesDB qui envoie une requête à la base de données locale SQLite puis traduit sa réponse dans un format traitable par les autres classes de l'application.

## Envoi d'un message

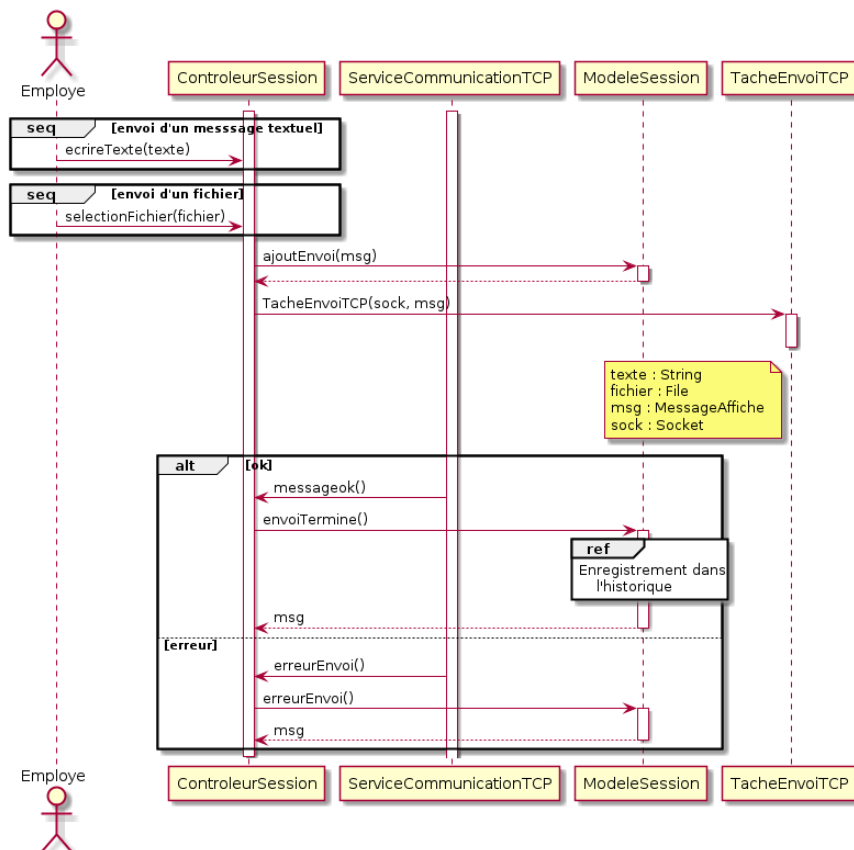


Diagramme de séquence d'envoi d'un message

Le message envoyé par l'utilisateur peut être du texte entré dans l'application ou un fichier. Lors de l'envoi, le message est mis en attente dans le modèle avant son affichage. Une fois que le destinataire a confirmé la bonne réception du message, il est retiré du modèle, enregistré dans l'historique, et affiché sur l'application.

Si le message n'a pas pu être transmis correctement, le message est retiré et un message d'erreur correspondant est affiché.

### Réception d'un message

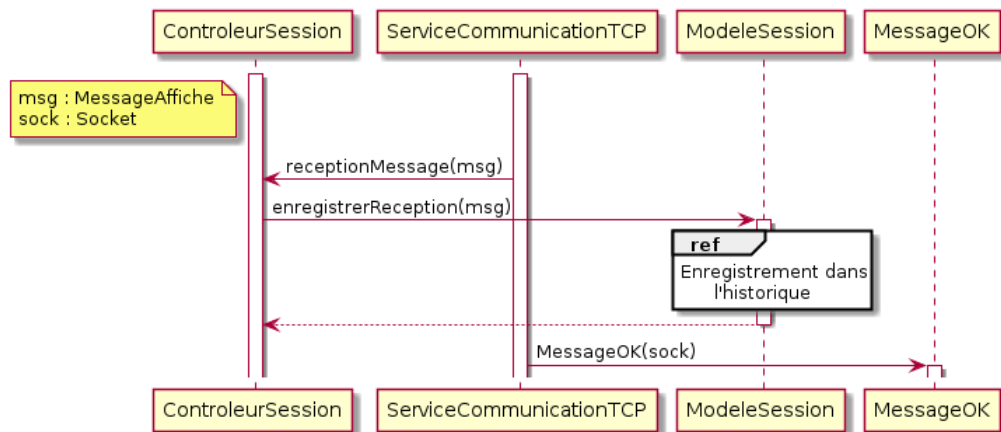


Diagramme de séquence de réception d'un message

Lors de la bonne réception d'un message, ce dernier est enregistré dans l'historique et affiché dans l'application. L'application envoie ensuite un acquittement à l'utilisateur distant.

### Enregistrement dans l'historique

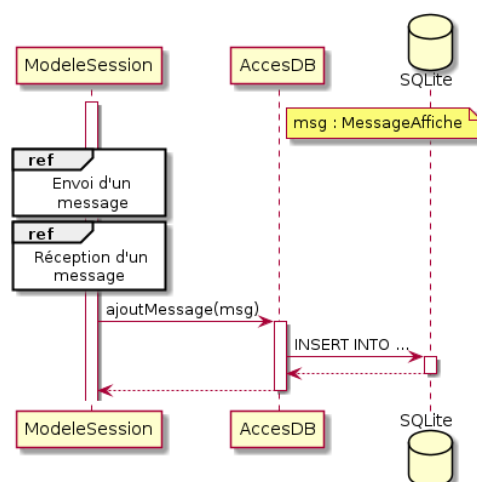


Diagramme de séquence d'enregistrement dans l'historique

Lors d'un envoi réussi ou d'une réception, un message est fourni à AccesDB qui en tire une requête SQL d'insertion dans la base de données locale SQLite.

## Fermeture d'une session

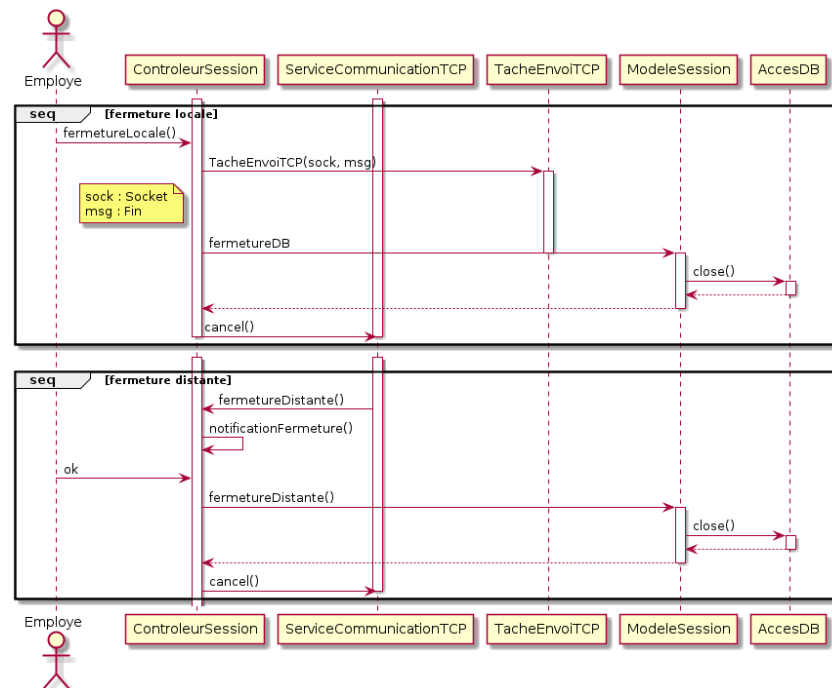


Diagramme de séquence de fermeture d'une session

La fermeture locale d'une session de discussion envoie à l'utilisateur une notification de fermeture, ferme l'accès à la base de données locale et arrête le service de réception TCP avant de détruire le contrôleur.

Les mêmes étapes se déroulent lors d'une fermeture distante hormis que cette fois l'utilisateur reçoit la notification et le contrôleur n'est pas détruit pour pouvoir continuer à consulter l'historique.

## Déconnexion

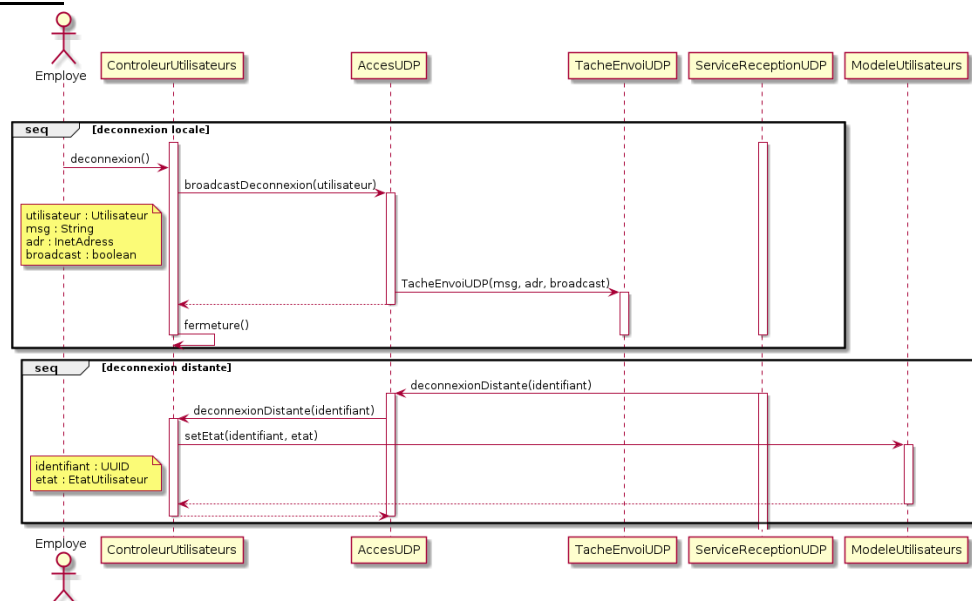


Diagramme de séquence de déconnexion

Une demande de déconnexion locale est d’abord “broadcastée” à l’ensemble du réseau avant d’engendrer la fermeture de l’application. Lors d’une déconnexion dans le réseau, le service UDP notifie le contrôleur qui demande ensuite au modèle de changer l’état de l’utilisateur en question.

## Diagrammes de classes

Par soucis de lisibilité, nous vous présenterons dans cette partie, notre diagramme de classes découpé en paquet. Cependant, vous pourrez retrouver le diagramme complet à la page C des annexes.

### Principal

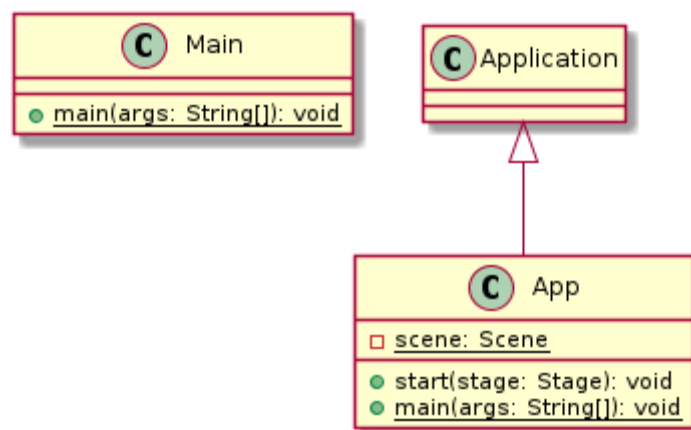


Diagramme des classes principales

Les classes principales sont à la racine des sources, elles servent à lancer le projet. La classe App hérite d’Application, une des classes fondamentales de JavaFX, et la classe Main est une surcouche pour le lancement depuis un .jar. La méthode main() de App est appelée lors d’un lancement de l’application par Maven et celle de Main (qui exécute celle de App) lors du lancement par le .jar déployé.

## Concurrent

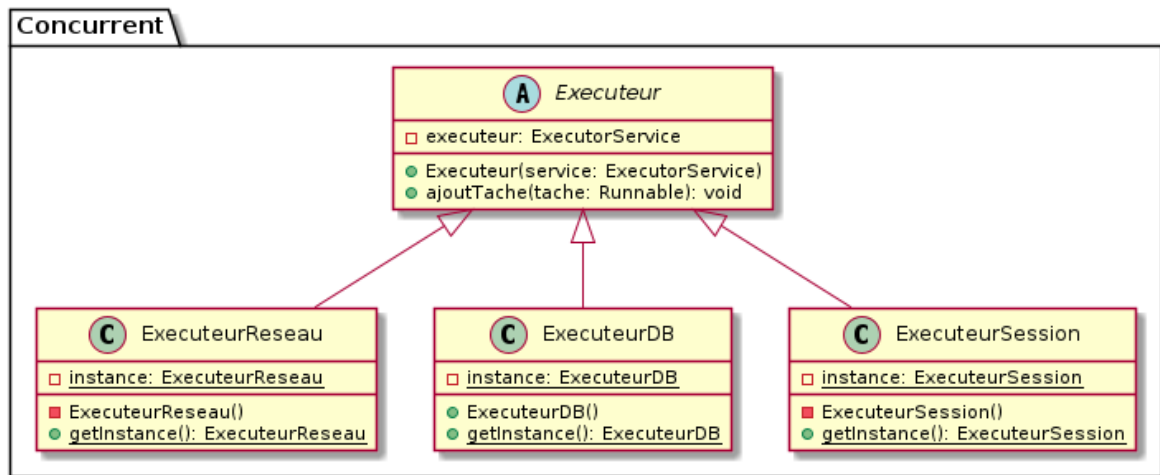


Diagramme de classes du paquet « concurrent »

Les classes de ce paquet sont les gestionnaires de threads, elles incluent un ExecutorService afin d'avoir une pool de thread plus ou moins grande selon les besoins.

Les trois classes instanciables sont également des singletons.

ExecuteurReseau est réservé aux services TCP et UDP, ExecuteurSession est utilisé par les session pour l'envoi de messages, ils contiennent une pool de thread en cache qui gagnent à être utilisés le plus possible.

ExecuteurDB est utilisé pour les accès à la base de données locale, son pool ne contient qu'un seul thread afin d'éviter les conflits d'écriture.

## Contrôleurs

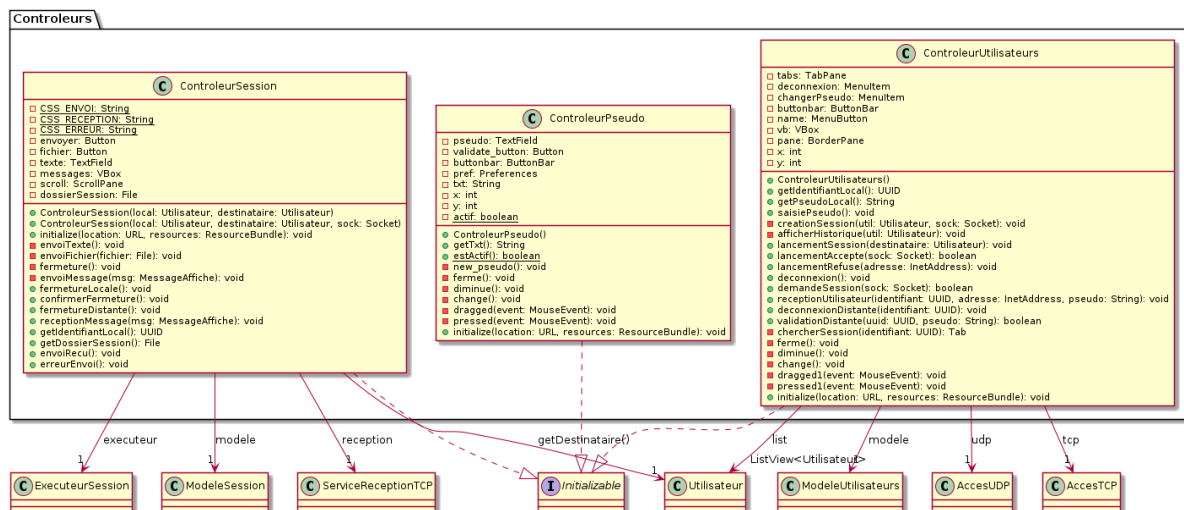


Diagramme de classes du paquet « contrôleurs »

Les classes contrôleurs sont à la fois les Controller et les Vue que l'on peut trouver dans l'architecture MVC.

Elles implémentent l'interface Initializable de JavaFX et sont associées à un fichier .fxml décrivant l'apparence de la fenêtre qu'elles manipulent.

Utilisateurs décrit l'interface principale contenant la liste des utilisateurs et les différentes options qu'elle contient (changement de pseudo, déconnexion, etc).

Pseudo manipule le pop-up de choix de pseudo pouvant être ouvert par l'utilisateur ou l'application.

Session décrit le comportement des onglets de session créés lors d'une discussion ou de la consultation d'un historique.

## Données

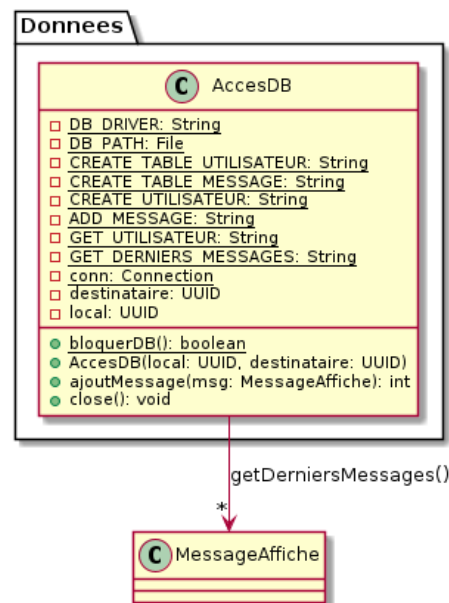


Diagramme de classes du paquet « données »

Ce paquet comporte la classe d'accès à la base de données.

AccesDB contient toutes les méthodes permettant de récupérer l'historique d'une conversation passée mais aussi d'ajouter des messages à celui d'une conversation en cours.

## Base de données

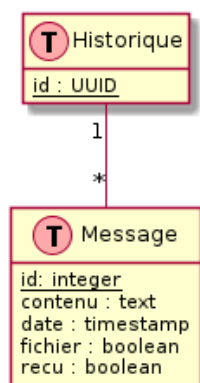
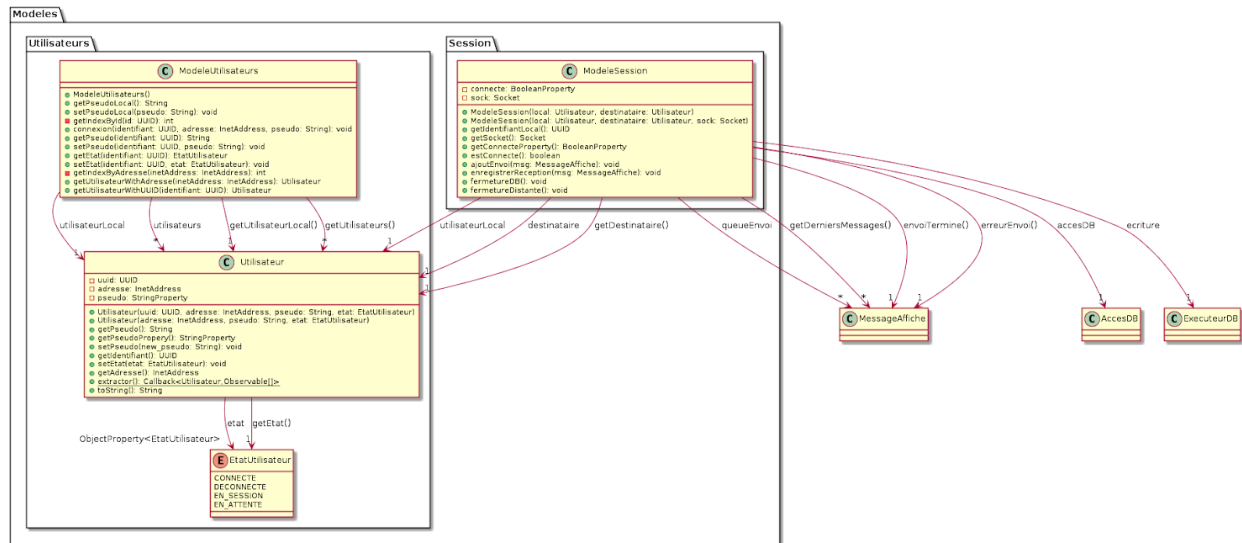


Diagramme de classes du paquet « base de données »

Les tables de la base de données se présentent sous cette forme, elles permettent un suivi des utilisateurs déjà mis en conversation et de leur lier des messages passés de tout type.

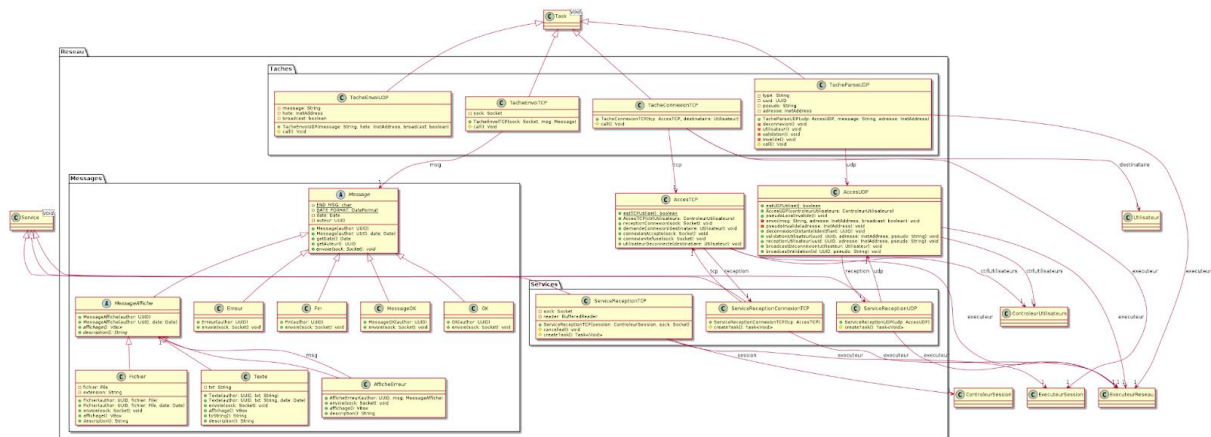
## Modèles



### Diagramme de classes du paquet « modèles »

Les classes Modèles contiennent et manipulent les données affichées par les Contrôleurs. On retrouve donc le `ModeleUtilisateurs` qui manipule la liste des utilisateurs et leurs états et le `ModeleSession` qui manipule les données d'une session en cours.

## Réseau



### Diagramme de classes du paquet « réseau »

Les classes de paquet Réseau sont scindées en plusieurs sous-groupes : les accès, les tâches, les services et les messages.

Les premières permettent de faire le pont entre le réseau et l'application, elles sont fondamentales car elles interagissent directement entre les services et le contrôleur des utilisateurs.

Les tâches sont des actions d'arrière-plan pouvant être déclenchées spontanément par l'application, elles peuvent permettre d'envoyer un message, demander une connexion ou "parser" un message reçu.



Les services sont des threads quasi permanents qui manipulent les sockets d'attente de connexion ou de messages.

Les messages sont des classes permettant de faciliter l'envoi de paquets TCP, ils peuvent être des messages systèmes indiquant une bonne réception, une erreur ou une fin de connexion ou bien ils peuvent être du contenu à afficher dans une session de discussion comme du texte ou un fichier.

## Utilitaires

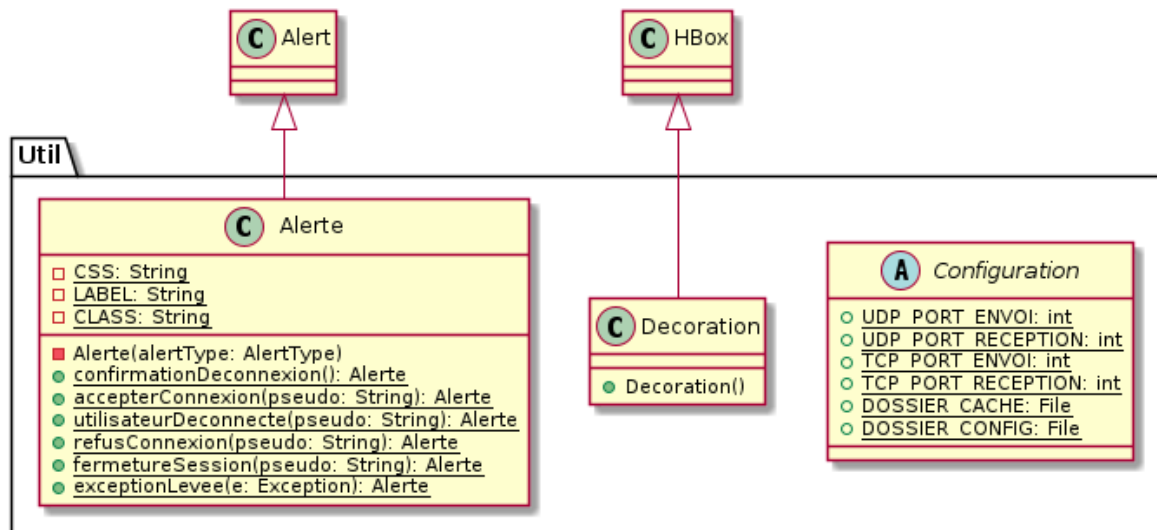
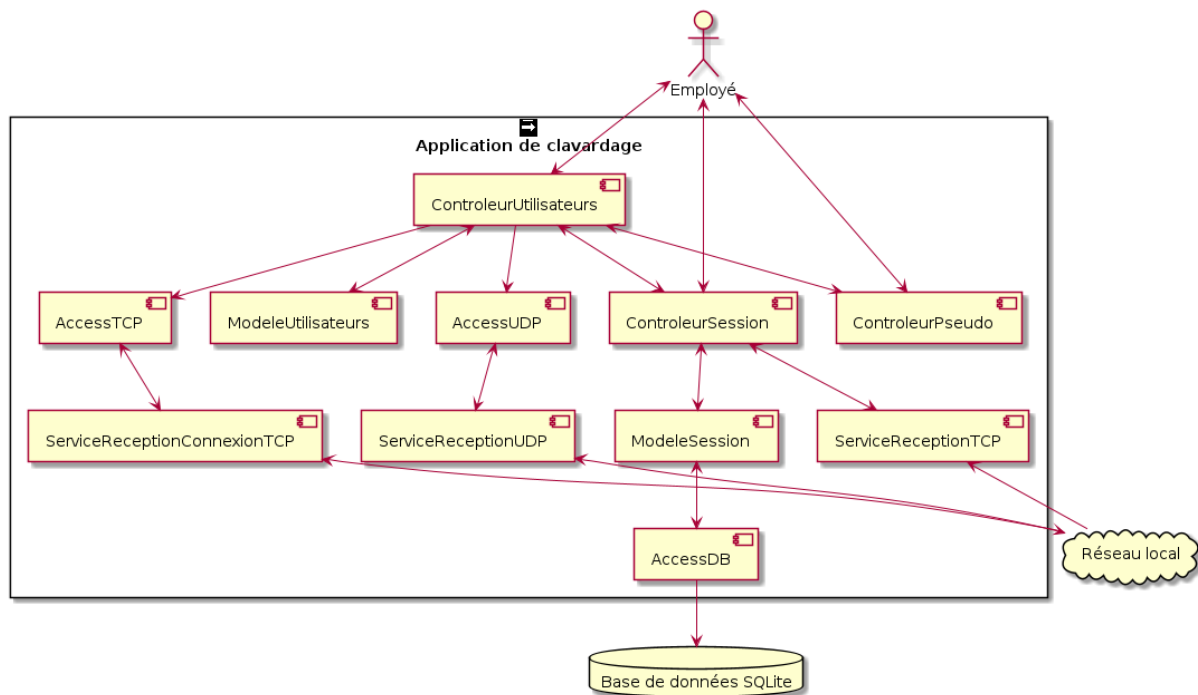


Diagramme de classes du paquet « utilitaires »

Les classes utilitaires sont des outils pouvant être utilisés un peu partout dans le projet. **Alerte** fournit des modèles de notifications inspirés de la classe **Alert** de JavaFX, **Decoration** permet de remplacer la décoration initiale des fenêtres et **Configuration** contient les données de configuration générale de l'application.

## Diagramme de structure composite



Le diagramme de structure composite nous permet de bien comprendre comment est structuré le code de notre application, en faisant apparaître ses structures majeures et leurs relations.

## Diagramme d'états

Le diagramme d'états, en page B des annexes, nous montre dans quels différents états peut se trouver notre application.

Il y'en a essentiellement trois :

- la VuePseudo qui est appelée en premier lors du démarrage mais qui peut être rappelée en cas de changement souhaité.
- la VueApplication qui concerne l'interface principale des utilisateurs et l'accès aux options.
- la VueSession qui concerne les conversations ou les historiques que l'utilisateur a avec des utilisateurs distants.

## Architecture et choix technologiques

Pour développer notre application nous avons fait le choix d'utiliser un certain nombre de technologies et d'outils de développement afin de réussir à atteindre les objectifs du cahier des charges.

### JavaFX et SceneBuilder

JavaFX fut notre premier choix de technologie à utiliser car cette librairie nous attirait beaucoup. Il s'agit d'un framework d'interface pour application de bureau très riche et documenté, comportant de nombreux avantages par rapport à Swing par exemple. En plus de la gestion graphique, cette librairie contient de nombreuses classes permettant de simplifier la gestion de threads. Elle comporte également un outil nommé SceneBuilder, permettant de réaliser des maquettes graphiques d'une interface qui peut ensuite être exportée pour l'intégrer au code sous forme de fichier.

### SQLite et cache

Nous avons choisi d'utiliser la base de données SQLite pour archiver les historiques de conversation car elle est légère, simple et parfaitement compatible avec Java et JDBC. Dans une optique de localité et de légèreté, la base de donnée ne contient que les utilisateurs distants avec lesquels l'utilisateur local a eu une conversation. Les fichiers transmis par l'application sont archivés dans un dossier de cache et peuvent ainsi être récupérés lorsque l'on ouvre à nouveau la conversation. Si le dossier cache est vidé, ils sont remplacés par un simple message contenant le nom du fichier transmis.

### Organisation des threads

Nos threads s'appuient soit sur la librairie de base de Java avec ses pools de threads soit sur JavaFX avec les classes Platform, Service ou Task.

L'organisation des threads se justifie autour de leurs utilité, nous avons plusieurs catégories de threads :

- Le thread application de JavaFX qui manipule l'interface et ne doit être sollicité que par de courtes opérations pour rester disponible. (Platform)
- les threads de services fixes UDP et TCP toujours en attente de réception de nouveaux messages. (Service)
- Les threads de réception de session qui sont créés ou supprimés à la volée des nouvelles sessions de discussion. (Service)
- Les threads en cache des pool de réseau et session qui peuvent être sollicités à tout moment pour l'envoi de message ou le "parse" d'une réception. (ExecutorService et Task)
- Le thread unique de base de données qui reçoit des requêtes d'écriture et lecture traitées les unes après les autres. (ExecutorService)

## Gestion de projet

Concernant la mise en place de ce projet, nous avons utilisé Jira et mis en place 5 sprints. Cette mise en place a été une réussite puisque nous avons bien respecté les dates limites de nos sprints en exécutant toutes les tâches prévues dans chaque sprint. Le premier sprint que nous avons mis en place était principalement basé sur l'instauration d'une connexion TCP. C'est aussi pendant ce sprint que nous avons codé l'unicité des messages. Lors du second sprint, nous avons codé l'envoi de messages par TCP, ainsi que la visibilité des utilisateurs connectés ou déconnectés. Nous avons également créé notre interface d'une session de discussion. Durant le troisième sprint, nous avons développé l'aspect fermeture des discussions avec la fermeture du socket de communication et la notification de fermeture de discussion notamment. Le sprint quatre nous a permis de coder la base de données et la réception de messages en TCP. Enfin lors du sprint cinq, nous avons terminé l'interface de notre application, nous avons également coder l'envoi et la réception de fichier, et finalement nous nous sommes occupés de tout ce qui était lié au déploiement de notre application.

L'utilisation de Jira nous a aidé dans l'organisation et le déroulement de ce projet. Nous avons une vision claire de ce que nous faisons et de ce que nous devons faire, ainsi que le temps qu'il nous restait pour exécuter nos tâches.

Nous avons également créé un répertoire Git pour notre projet avec quatre branches, une pour chacun des membres du binôme, une autre pour fusionner nos codes, et enfin une dernière utilisée pour le rendu final.

Cette utilisation combinée de Jira et Git nous a permis de bien avancer en binôme sans craindre de travailler sur les mêmes choses ou d'oublier de coder des caractéristiques.

## Procédures de tests et déploiement

Les procédures de tests automatiques n'ont pas pu être écrites durant le projet, elles étaient complexes à retranscrire dans un code et nous avons manqué de temps.

Nos tests de fonctionnalités furent donc des scénarios d'utilisation que nous organisions "à la main" en lançant l'application sur deux postes d'un même réseau ou sur deux machines virtuelles interconnectées.

Ils pouvaient être très sommaires, comme vérifier qu'un nouvel utilisateur apparaît bien, que le pseudo change, qu'il soit refusé en cas de similarité, etc.

Ou plus complexe, comme vérifier qu'une conversation est bien enregistrée en base de données ou qu'une image est transmise entièrement d'un utilisateur à l'autre.

Le déploiement de notre application ne s'est pas effectué avec l'outil Jenkins vu en cours mais grâce à Github Actions qui a rempli le même rôle pour nous.

Nous avons deux actions distinctes :

- La première lançait les procédures de test lorsque quelqu'un effectuait un push sur **dev** ou fusionnait sa branche avec elle
- La seconde effectuait un packaging de notre application et publiait une release lorsque l'on poussait des changements sur **main**.

Les détails de nos pipelines sont disponibles dans les fichiers .yml du dossier [.github/workflows](#) de notre dépôt.

## Installation

L'installation de l'application se fait simplement en téléchargeant la dernière archive .jar disponible dans les [releases](#) du dépôt Github de notre projet.

Une fois en possession de l'archive, il vous suffit d'ouvrir un terminal dans le même dossier et de taper **java -jar nom\_d\_archive.jar** pour provoquer son démarrage.

L'archive est compatible avec les systèmes Windows, Mac OS et Linux.

## Manuel d'utilisation

Lors du lancement d'application, une fenêtre pour vous connecter, identique à celle-ci-dessous, va s'ouvrir :



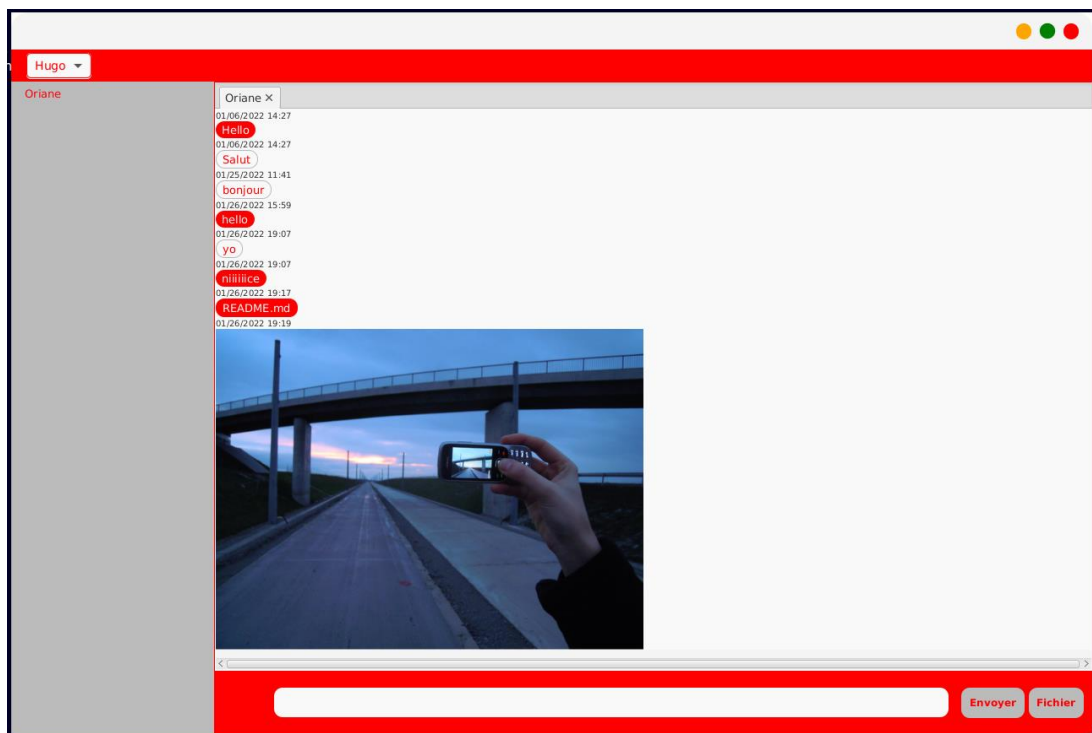
Fenêtre Pseudo

Il vous suffira de rentrer un pseudo pour accéder à la fenêtre principale de l'application. Celle-ci est illustrée en page suivante



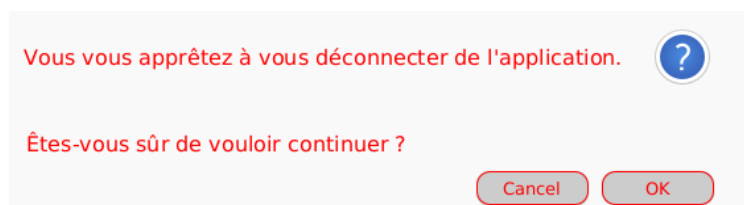
Fenêtre application sans session de discussion ouverte

Dans le cadre à gauche s’affiche la liste des personnes connectées, ou déconnectées après votre arrivée. Pour démarrer une session avec une des personnes présente sur la liste, il vous suffit d’appuyer sur le pseudo de la personne que vous voulez joindre. De là une demande sera envoyée à cette personne, et si elle accepte votre demande de connexion alors un nouvel onglet va s’ouvrir sur la partie à droite de la liste des utilisateurs. Sur cet onglet vous verrez s’afficher l’historique de vos conversations et vous pourrez envoyer des messages à cette personne :



Fenêtre application avec session de discussion ouverte

En haut à gauche, vous pouvez voir un bouton avec votre pseudo dessus, il s'agit d'un menu déroulant. Il comporte deux fonctionnalités : « Se déconnecter » et « Changer de pseudo ». Pour vous déconnecter vous pouvez donc appuyer sur le bouton « Se déconnecter », mais vous pouvez également appuyer sur le bouton rouge de la barre de la fenêtre. Dans les deux cas, un pop-up s'affiche pour que vous puissiez confirmer que vous voulez bien vous déconnecter :



Pop-up de déconnexion

## Annexes

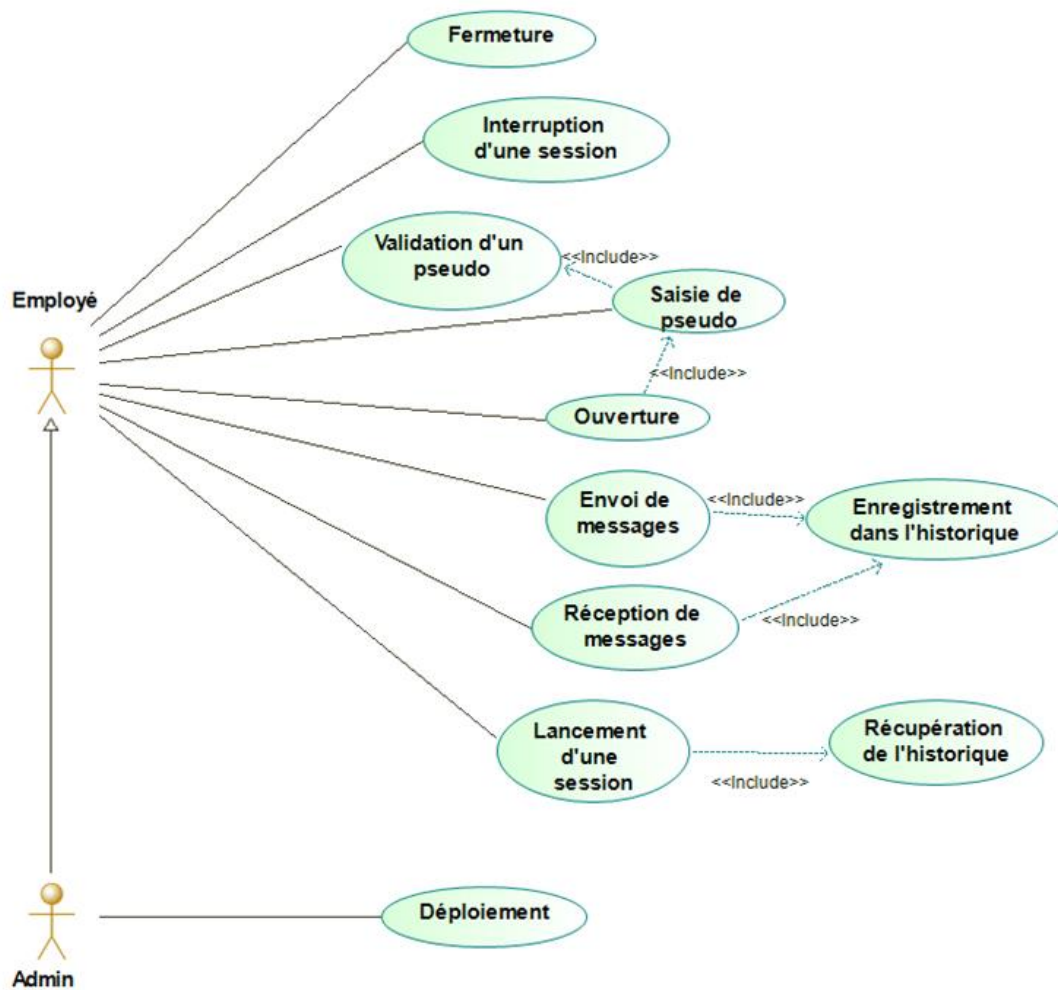


Figure 1 : Diagramme des cas d'utilisation



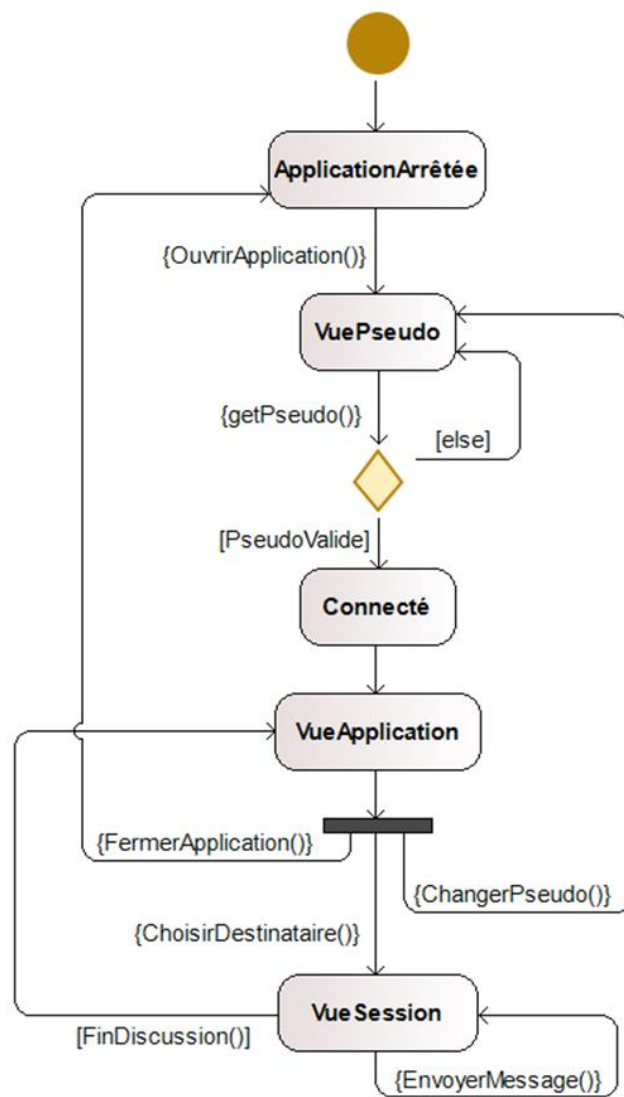
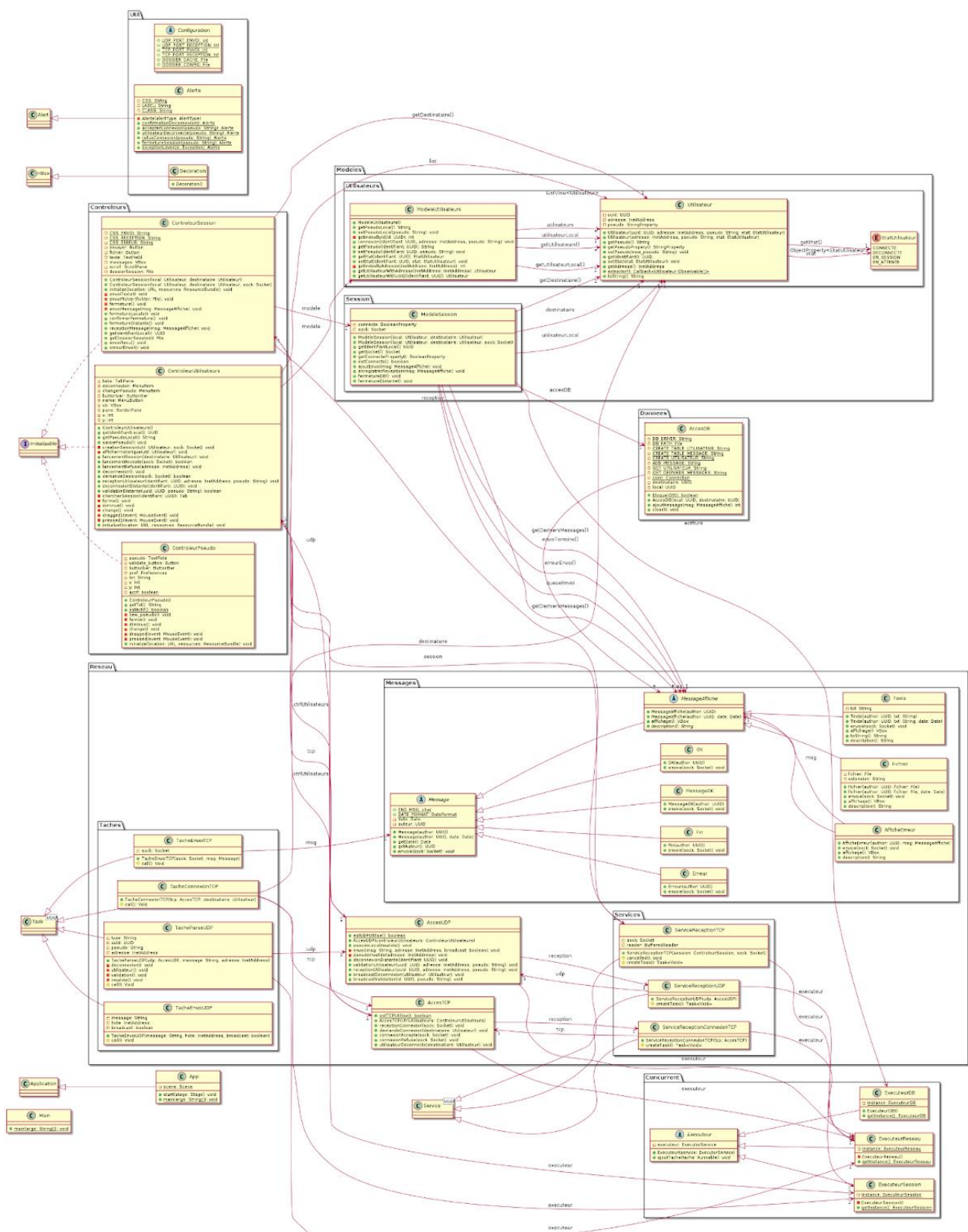


Figure 2 : Diagramme d'états





## **INSA Toulouse**

135, avenue de Rangueil  
31077 Toulouse Cedex 4 - France  
**[www.insa-toulouse.fr](http://www.insa-toulouse.fr)**



MINISTÈRE  
DE L'ÉDUCATION NATIONALE,  
DE L'ENSEIGNEMENT SUPÉRIEUR  
ET DE LA RECHERCHE