GM // Département génie mathématique

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN NORMANDIE

RAPPORT DU MINI-PROJET JAVA :

Implémentation d'un Chat à la

'TWITTER'

Membres du Groupe en GM4:

- **RAMAGE Lucas**
- **DRIGUEZ Claire**
- **BENDAHOU Sara**
- **ELMASLOUHE Zakaria**

Professeur Encadrant :

- **Laurent VERCOUTER**

21 Décembre 2017

RAPPORT DU MINI-PROJET JAVA :

PROGRAMMATION ORIENTEE-OBJET AVEC JAVA ET UML

INTRODUCTION :

Dans le cadre de notre mini-projet en Programmation Orientée Objet avec les langages Java et UML, on a réalisé une application Chat à la « Twitter ». Cette application est bien simplifiée par rapport aux différents fonctions de Twitter. A travers ce projet, on a pu mettre en exergue les principes fondamentaux de la programmation orientée-objet, c'est-à-dire l'héritage entre les classes, la gestion des exceptions, la communication sur un réseau, la gestion des interfaces, etc.

Dans une première partie, nous traiterons la modélisation UML de notre projet incluant les différents diagrammes UML vus en cours. Dans une autre partie, nous décrions l'implémentation de l'application en présentation quelques solutions des problèmes difficiles rencontrés basées sur des notions étudiées en cours. Enfin, nous présenterons la façon d'exécution du programme.

TWITTER...

C'est un large espace de conversation ou chacun dispose de 140 caractères pour diffuser ses messages appelés « tweets » qui sont publics. Il est utilisé en différents modes : Unilatéral, pour émettre des informations uniquement ou en recevoir. On peut aussi relayer les tweets d'autres membres. En fait, on peut choisir les membres de Twitter dont on veut suivre les publications et ces membres peuvent nous ajouter également en retour dans leur propre réseau. On peut enfin utiliser Twitter en mode conversation public.

Les utilisateurs de TWITTER utilisient des mots-clés précédés du signe # appelés ***hashtags*** *dans leurs publications ce qui permettra aux autres de voir leurs tweets en recherchant les hashtags utilisés.*

Table de matières :

I. Modélisation UML :

Après l'identification des différents besoins de l'application à développer, on s'est basé sur la modélisation UML « Unified Modeling Language » qui constitue un outil visuel permettant de donner une vision sur le projet grâce aux différents diagrammes qui le représentent : son implémentation, son fonctionnement, ses composants, sa mise en route, les méthodes et action effectués par l'application, etc.

1. Diagramme Cas d'utilisation :

Comme il est indiqué sur le diagramme ci-dessous, l'utilisateur a l'accès à toutes les fonctionnalités présentées ci-dessus, avec lesquelles il peut interagir. Certains cas d'utilisation imposent d'effectuer préalablement d'autre cas d'utilisation, représentée ici pour la flèche <<includes>> en pointillé. En effet, par exemple, la connexion à la plateforme inclut, lors de la première connexion, l'action de création de compte. Les différentes actions tel que "publier un compte" etc. ne peuvent être effectuées sans l'authentification, c'est-à-dire la connexion au serveur, d'où les flèches <<includes>> en pointillé convergeant vers l'action "se connecter". De même, pour suivre un utilisateur ou afficher les messages d'un utilisateur, il est nécessaire de sélectionner le nom de l'utilisateur s'affichant à gauche de l'écran, soit dans la liste des utilisateurs connectés, soit dans la liste des utilisateurs de la plateforme, après l'avoir recherché, pour que le menu avec les différentes actions s'affiche.

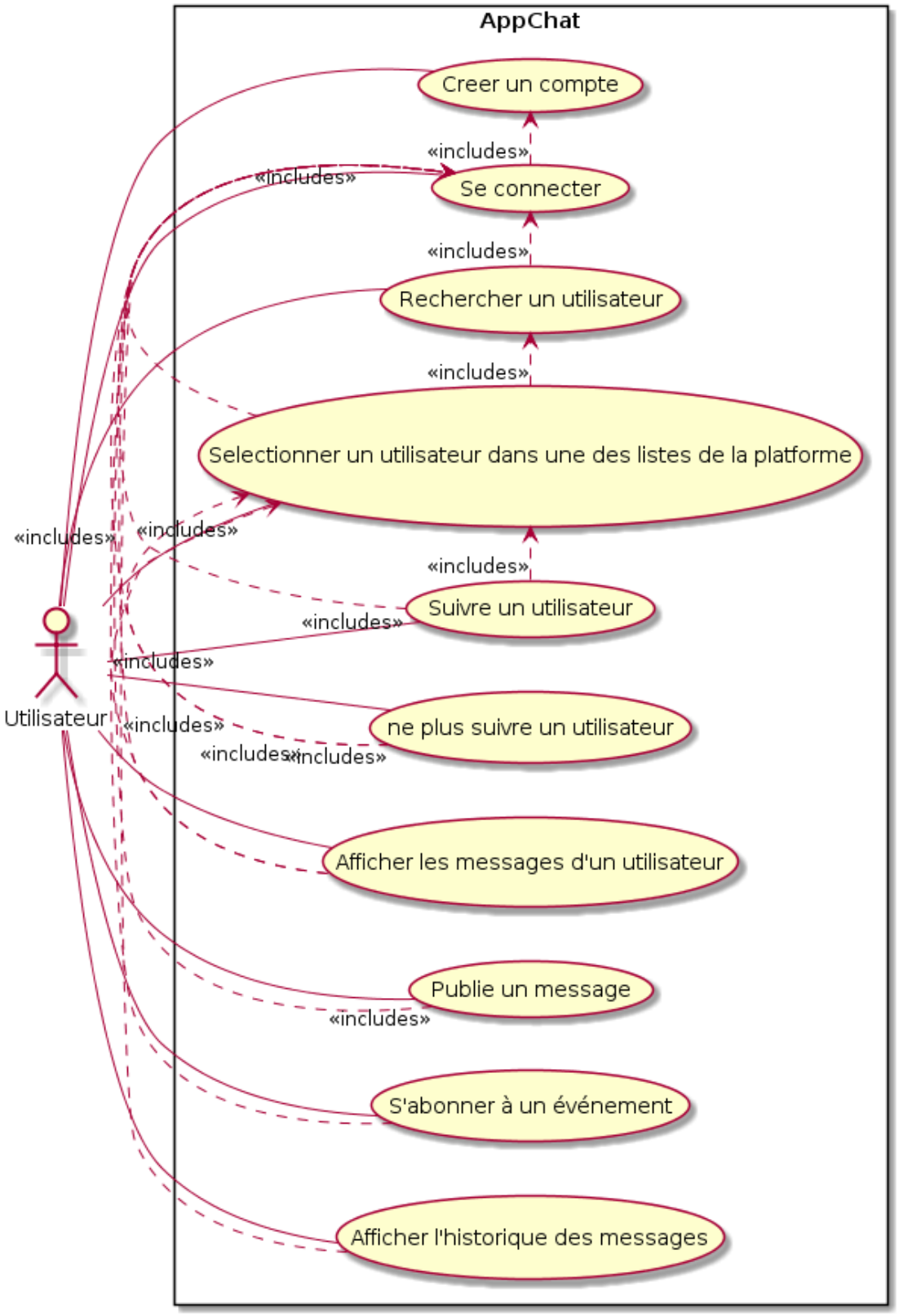


FIGURE 1 : DIAGRAMME DE CAS D'UILISATION

2. Diagrammes de séquences :

A. Création de compte :

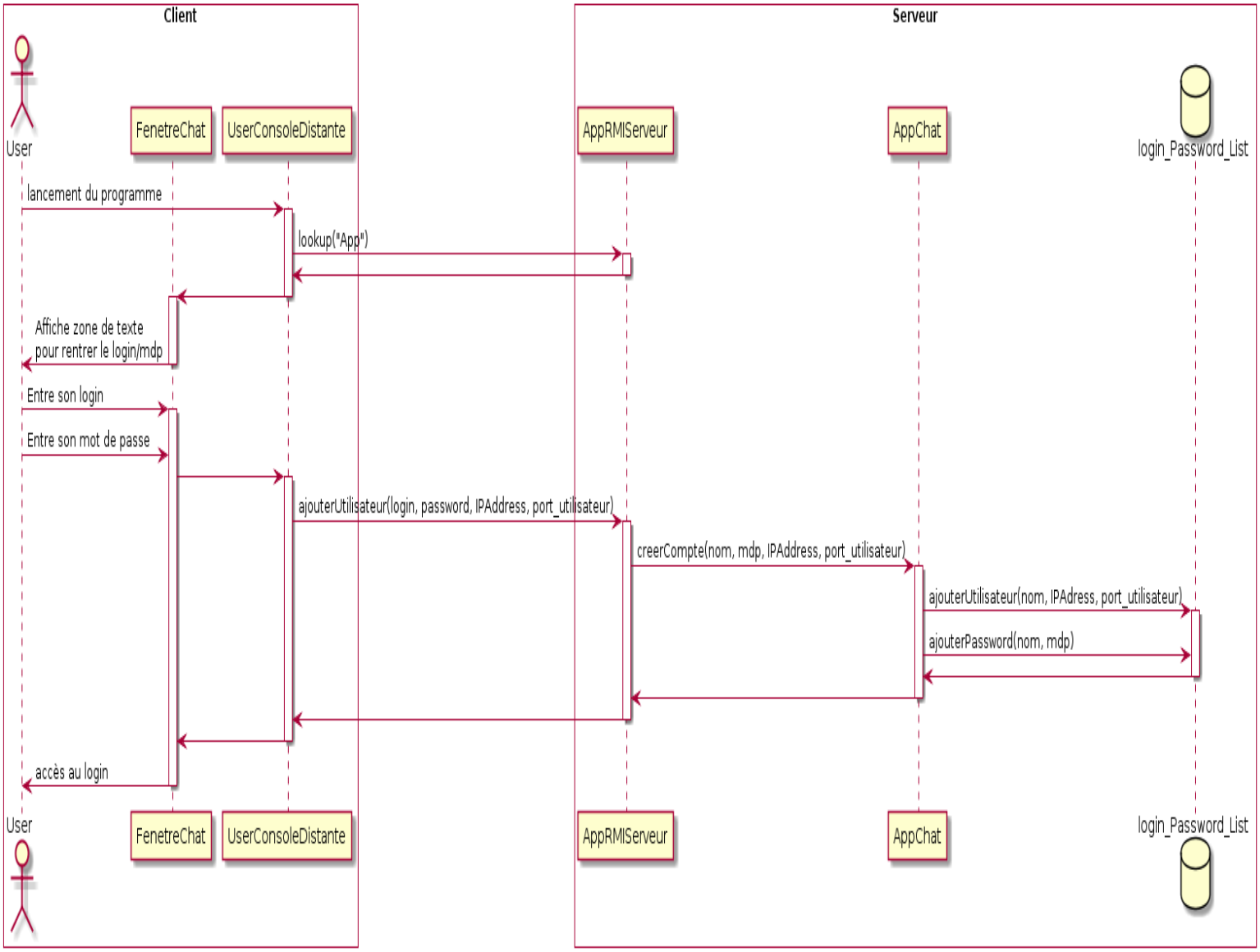


FIGURE 2 : DIAGRAMME DE SEQUENCE : CREATION DE COMPTE

B. S’identifier (Login) :

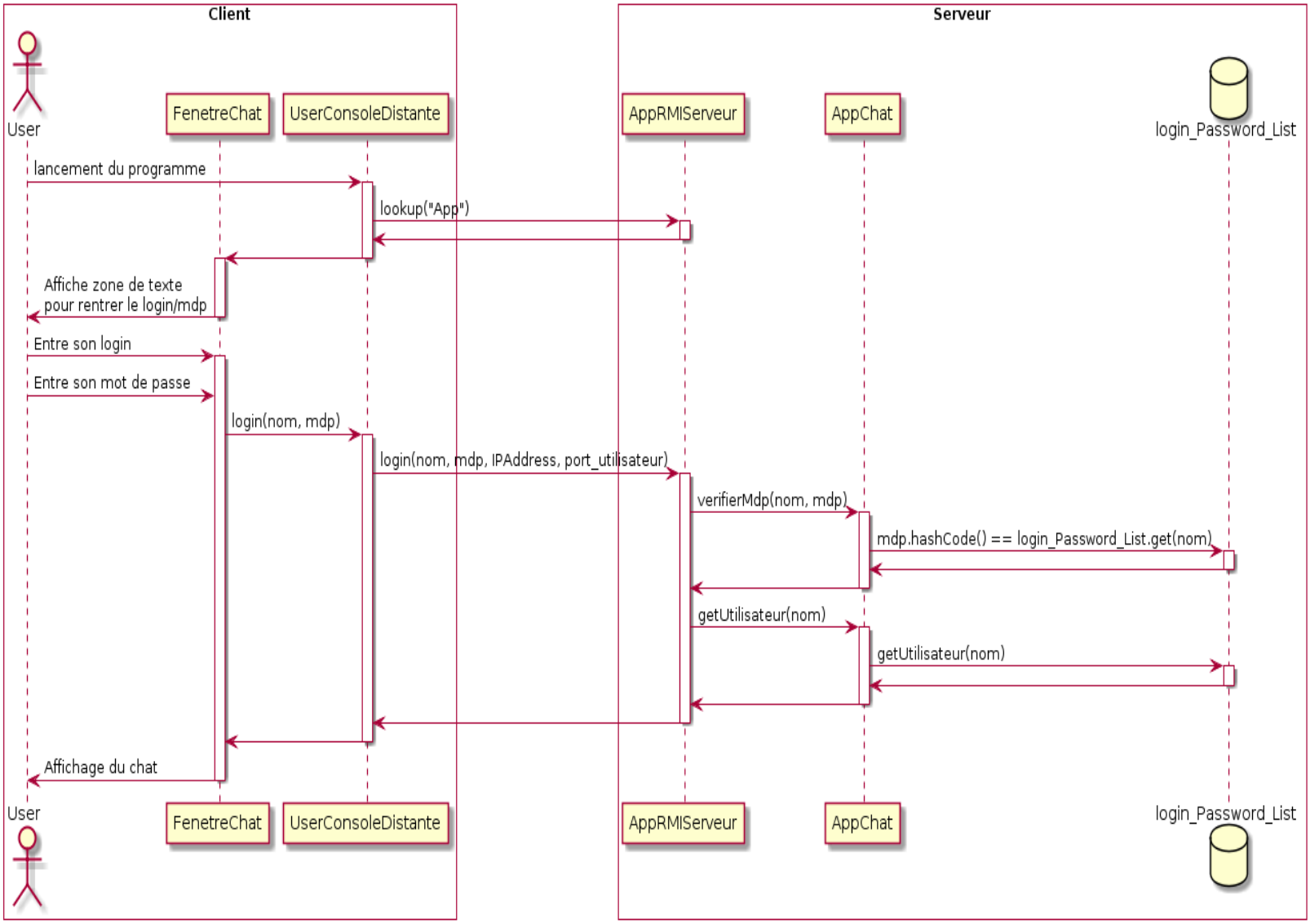


FIGURE 3 : DIAGRAMME DE SÉQUENCE: LOGIN

C. Envoie de message :

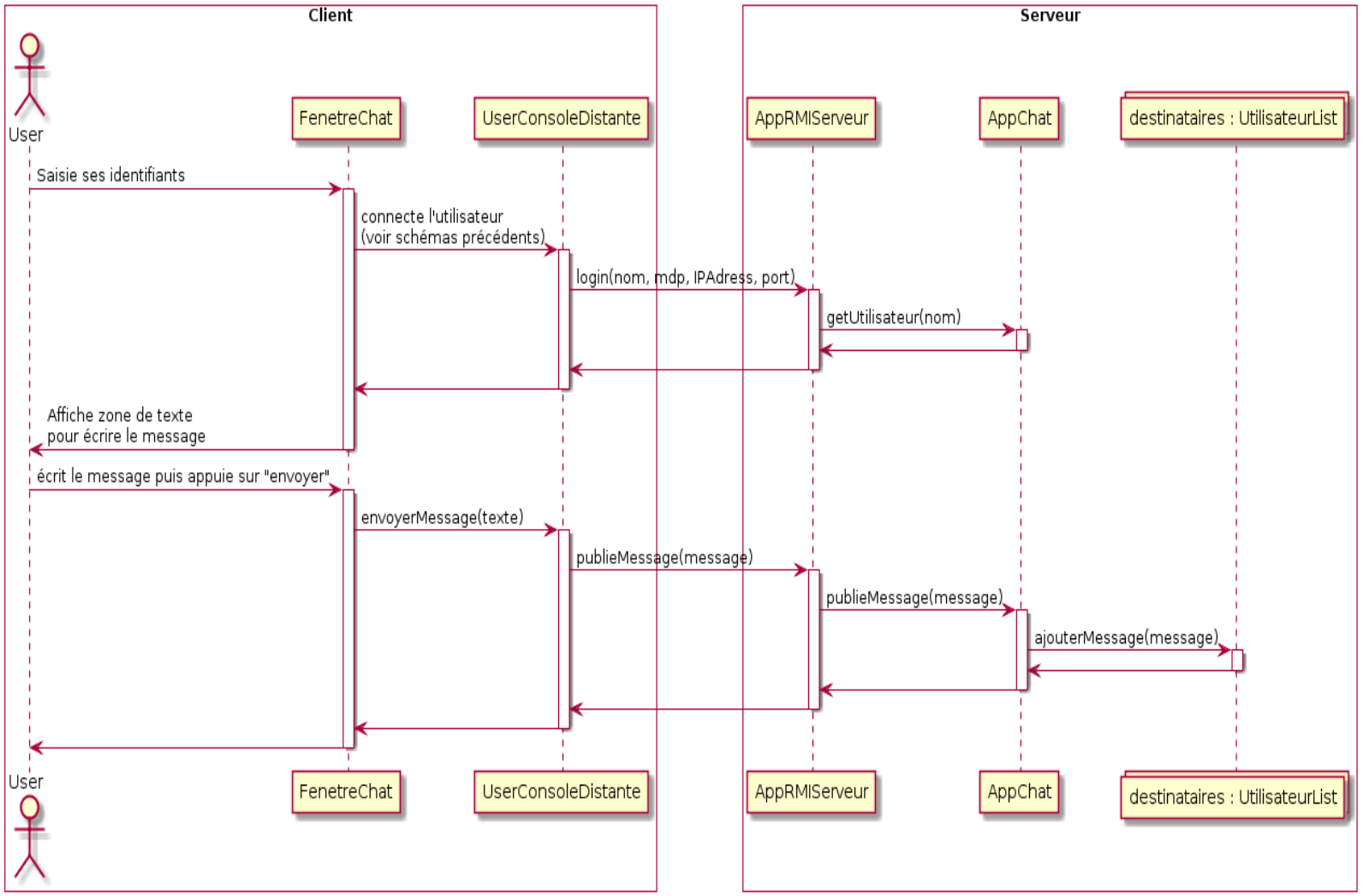


FIGURE 4 : DIAGRAMME DE SEQUENCE : ENVOIE DE MESSAGE

3. Diagrammes de Classes :

L'application qu'on a codée permet à un utilisateur « émetteur » de publier des messages automatiquement reçus par les utilisateurs « abonnés » à cet émetteur.

Notre code se décompose en 3 packages :

- Appchat : Ce package permettant de centraliser, et stocker les données.
- appChat.rmi : Ce package permettant d'échanger les données entre un client et un serveur.
- appChat.ihm : Ce package permettant de gérer l'interface graphisme du programme.

A. Package 1 : Appchat :

Les deux classes les plus importantes de ce package sont:

Class AppChat : permet de centraliser les données et de les stocker par l'intermédiaire de fichiers. Elle a comme attributs la liste des utilisateurs et la liste des mots de passe et sa fonction principale est ***UtilisateurList publieMessage(Message m)*** qui permet de récupérer l'auteur d'un message c'est à dire l'objet utilisateur propre au message pour ensuite utiliser les méthodes propres à l'utilisateur permettant d'ajouter ce message à la liste des messages ainsi qu'à celle des followers. Ajoutons à cela que si un hashtag est contenu dans le message, ce message est envoyé directement aux utilisateurs annotés dans le hashtag tout en renvoyant la liste des utilisateurs auxquels un message a été envoyé pour que ces utilisateurs puisse ensuite être informés qu'ils ont reçu un message.

Class Utilisateur : Cette classe permet de regrouper les informations propres à un utilisateur. On peut alors en récupérer l'ensemble de ses attributs : Liste des personnes suivies, liste des personnes qu'il suit, liste de ses messages, liste de ses messages récents, et liste des messages retweetés. Ceci grâce à ces différentes méthodes.

Le diagramme ci-dessous représente le diagramme de classes de ce package tout en montrant les différentes relations entre elles :

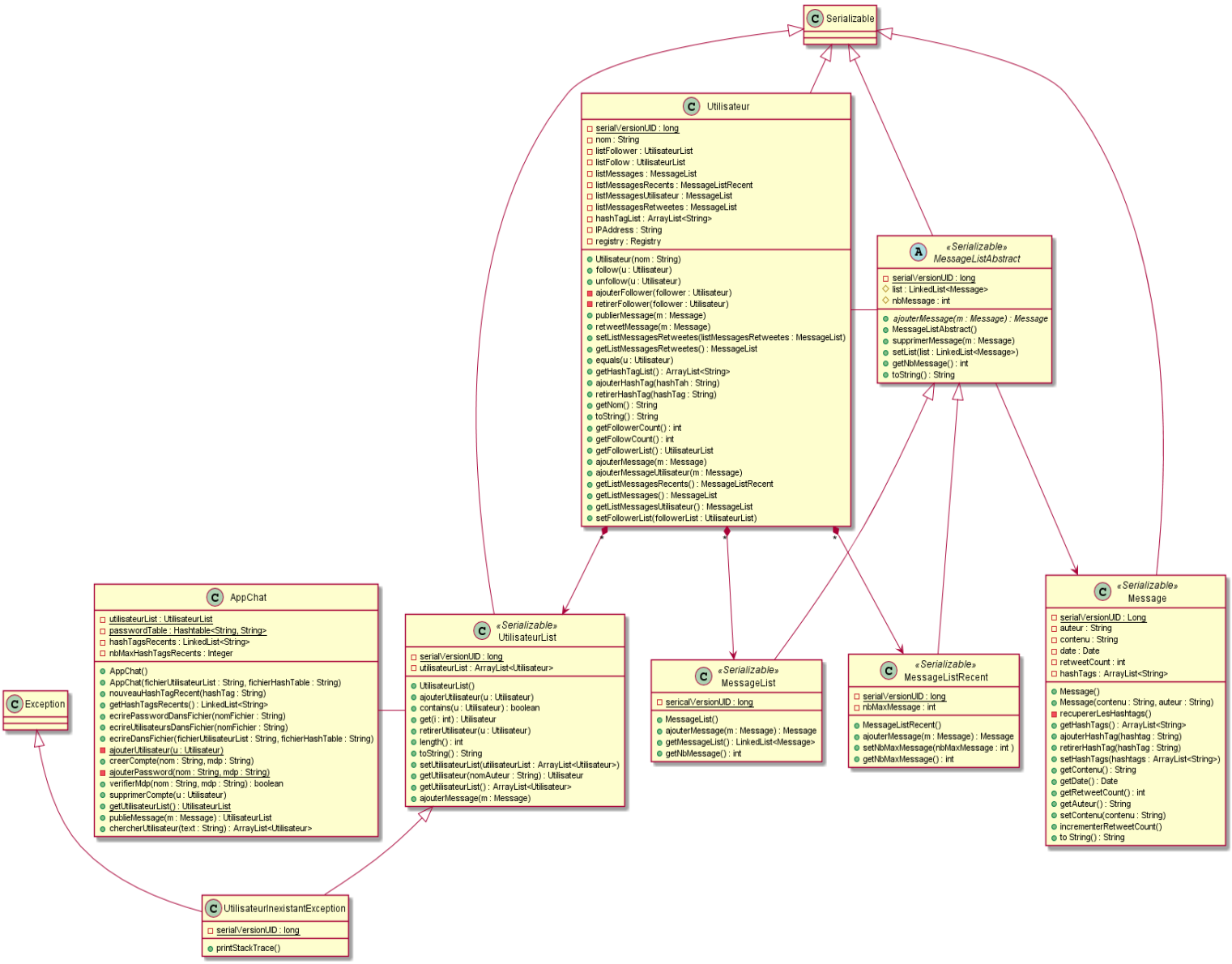
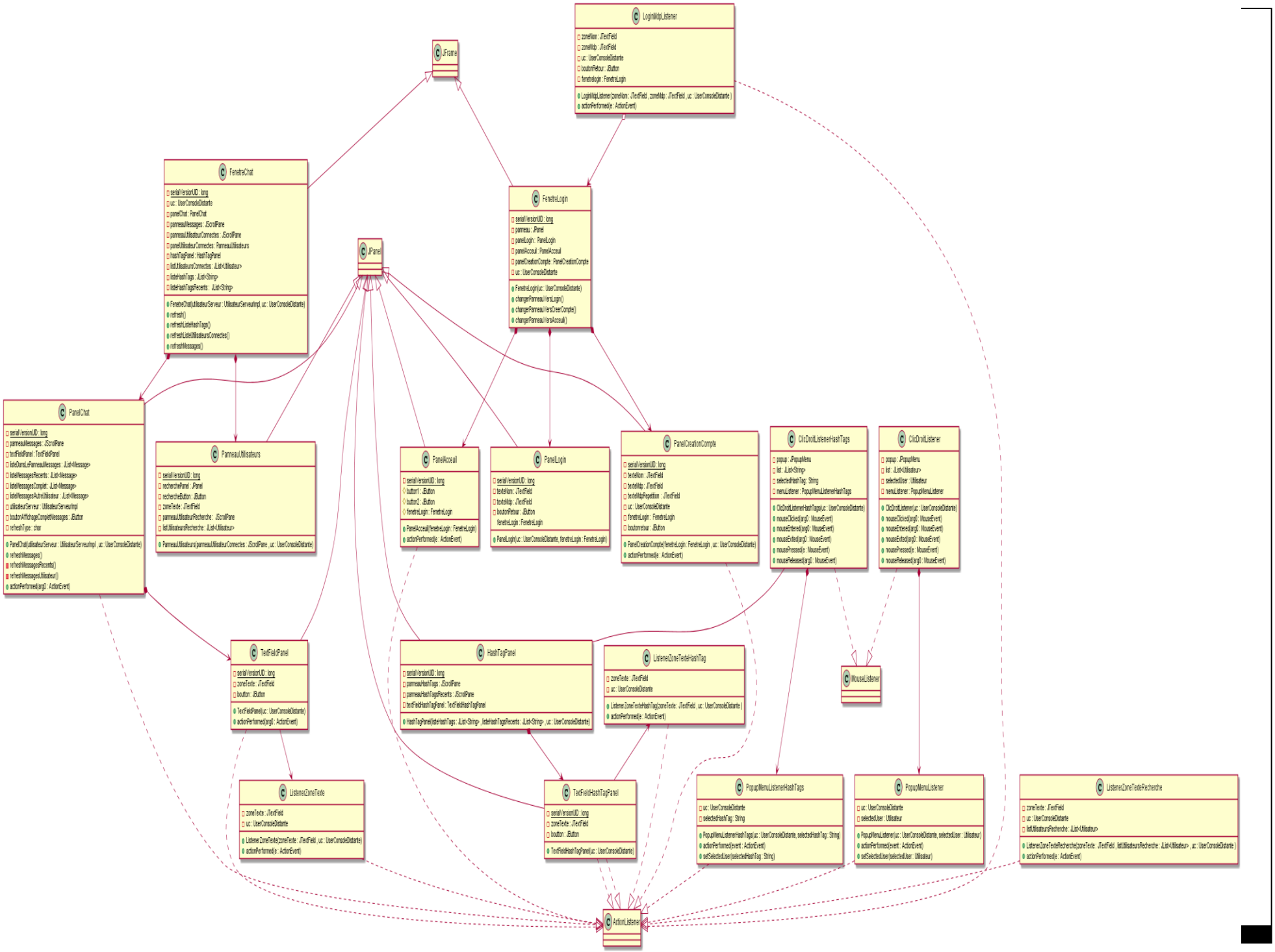


FIGURE 5 : DIAGRAMME DE CLASSE DU PACKAGE APPCHAT

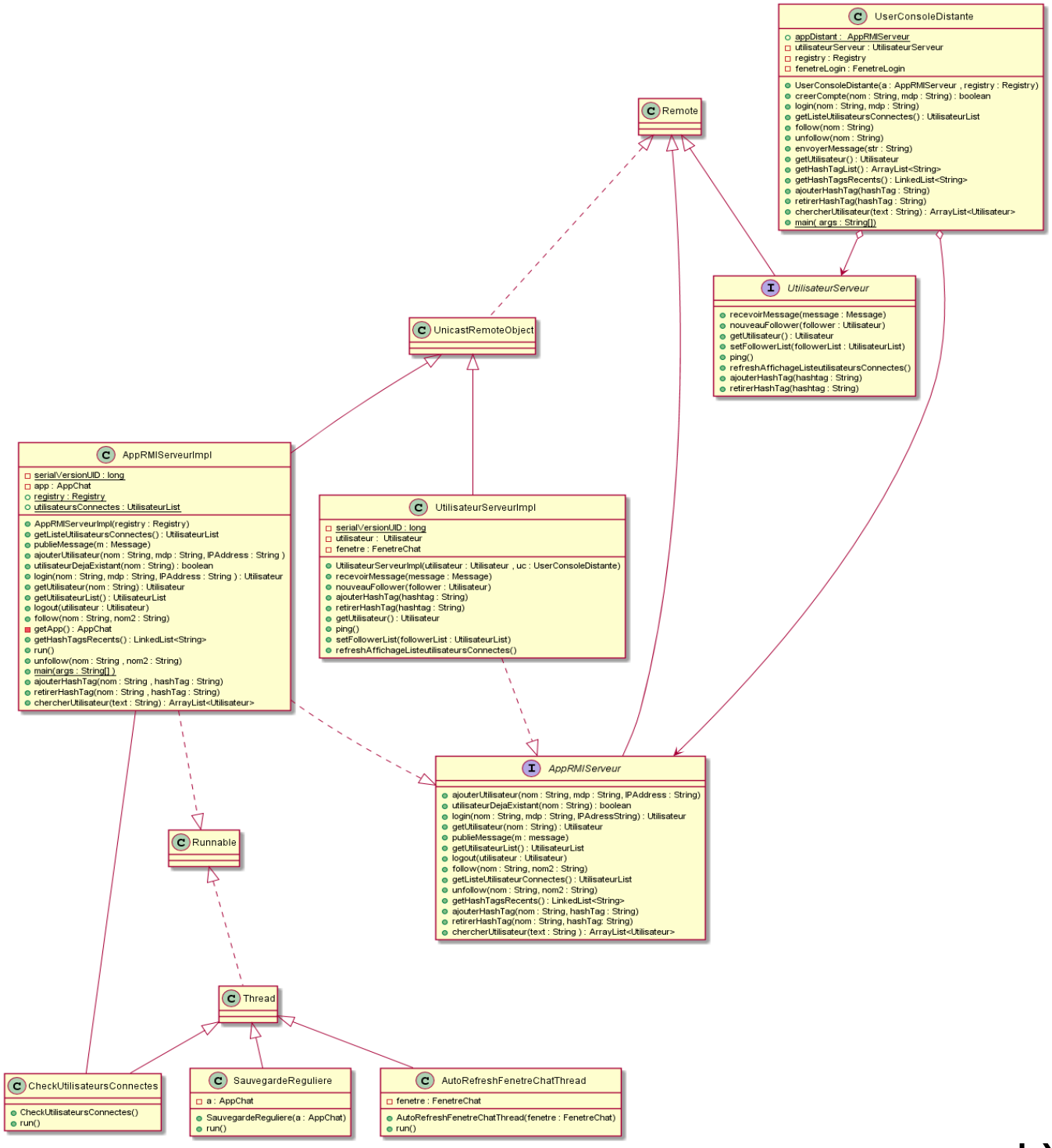
B. Package 2 : '*Appchat.ihm*' :

Ce package permet de générer plusieurs fenêtres, dont le panel est modifié en fonction de l'action de l'utilisateur. Il y a deux fenêtres principales qui ne sont jamais affichées simultanément : la fenêtre de connexion (***FenetreLogin***) au compte et la fenêtre d'utilisation du programme (***FenetreChat***).


LoginMdpListener
- zoneNom : JTextField
- zoneMdp : JTextField
- uc : UserConsoleDistante
- boutonRetour : JButton
- fenetreLogin : FenetreLogin
+ LoginMdpListener(zoneNom : JTextField , zoneMdp : JTextField, uc : UserConsoleDistante)
+ actionPerformed(e : ActionEvent)
JFrame
FenetreChat
- serialVersionUID : long
- uc : UserConsoleDistante
- panelChat : PanelChat
- panneauMessages : JScrollPane
- panneauUtilisateurConnectes : JScrollPane
- panelUtilisateurConnectes : PanneauUtilisateurs
- hashTagPanel : HashTagPanel
- listUtilisateursConnectes : JList<Utilisateur>
- listeHashTags : JList<String>
- listeHashTagsRecents : JList<String>
+ FenetreChat(utilisateurServeur : UtilisateurServeurImpl, uc : UserConsoleDistante)
+ refresh()
+ refreshListeHashTags()
+ refreshListeUtilisateursConnectes()
+ refreshMessages()
FenetreLogin
- serialVersionUID : long
- panneau : JPanel
- paneLogin : PaneLogin
- panelAcceuil : PanelAcceuil
- panelDreationCompte : PanelCreationCompte
- uc : UserConsoleDistante
+ FenetreLogin(uc : UserConsoleDistante)
+ changerPanneauVersLogin()
+ changerPanneauVersCreerCompte()
+ changerPanneauVersAcceuil()
JPanel
PanelChat
- serialVersionUID : long
- panneauMessages : JScrollPane
- textFieldPanel : TextFieldPanel
- listeDansLePanneauMessages : JList<Message>
- listeMessagesRecents : JList<Message>
- listeMessagesComplet : JList<Message>
- listeMessagesAutreUtilisateur : JList<Message>
- utilisateurServeur : UtilisateurServeurImpl
- boutonAffichageCompletMessages : JButton
- refreshType : char
+ PanelChat(utilisateurServeur : UtilisateurServeurImpl , uc : UserConsoleDistante)
+ refreshMessages()
+ refreshMessagesRecents()
+ refreshMessagesUtilisateur()
+ actionPerformed(arg0 : ActionEvent)
PanneauUtilisateurs
- serialVersionUID : long
- rechercheePanel : JPanel
- rechercheButton : JButton
- zoneTexte : JTextField
- panneauUtilisateurRecherche : JScrollPane
- listUtilisateursRecherche : JList<Utilisateur>
+ PanneauUtilisateurs(panneauUtilisateurConnectes : JScrollPane , uc : UserConsoleDistante)
PanelAcceuil
- serialVersionUID : long
- button1 : JButton
- button2 : JButton
- fenetreLogin : FenetreLogin
+ PanelAcceuil(fenetreLogin : FenetreLogin)
+ actionPerformed(e : ActionEvent)
PaneLogin
- serialVersionUID : long
- texteNom : JTextField
- texteMdp : JTextField
- boutonRetour : JButton
- fenetreLogin : FenetreLogin
+ PaneLogin(uc : UserConsoleDistante, fenetreLogin : FenetreLogin)
PanelCreationCompte
- serialVersionUID : long
- texteNom : JTextField
- texteMdp : JTextField
- texteMdpRepetition : JTextField
- uc : UserConsoleDistante
- fenetreLogin : FenetreLogin
- boutonretour : JButton
+ PanelCreationCompte(fenetreLogin : FenetreLogin , uc : UserConsoleDistante)
+ actionPerformed(e : ActionEvent)
ClicDroitListenerHashTags
- popup : JPopupMenu
- list : JList<String>
- selectedHashTag : String
- menuListener : PopupMenuListenerHashTags
+ ClicDroitListenerHashTags(uc : UserConsoleDistante)
+ mouseClicked(arg0 : MouseEvent)
+ mouseEntered(arg0 : MouseEvent)
+ mouseExited(arg0 : MouseEvent)
+ mousePressed(e : MouseEvent)
+ mouseReleased(arg0 : MouseEvent)
ClicDroitListener
- popup : JPopupMenu
- list : JList<Utilisateur>
- selectedUser : Utilisateur
- menuListener : PopupMenuListener
+ ClicDroitListener(uc : UserConsoleDistante)
+ mouseClicked(arg0 : MouseEvent)
+ mouseEntered(arg0 : MouseEvent)
+ mouseExited(arg0 : MouseEvent)
+ mousePressed(e : MouseEvent)
+ mouseReleased(arg0 : MouseEvent)
TextFieldPanel
- serialVersionUID : long
- zoneTexte : JTextField
- bouton : JButton
+ TextFieldPanel(uc : UserConsoleDistante)
+ actionPerformed(arg0 : ActionEvent)
HashTagPanel
- serialVersionUID : long
- panneauHashTags : JScrollPane
- panneauHashTagsRecents : JScrollPane
- textFieldHashTagPanel : TextFieldHashTagPanel
+ HashTagPanel(listeHashTags : JList<String> , listeHashTagsRecents : JList<String> , uc : UserConsoleDistante)
ListenerZoneTexteHashTag
- zoneTexte : JTextField
- uc : UserConsoleDistante
+ ListenerZoneTexteHashTag(zoneTexte : JTextField , uc : UserConsoleDistante)
+ actionPerformed(e : ActionEvent)
MouseListener
ListenerZoneTexte
- zoneTexte : JTextField
- uc : UserConsoleDistante
+ ListenerZoneTexte(zoneTexte : JTextField , uc : UserConsoleDistante)
+ actionPerformed(e : ActionEvent)
TextFieldHashTagPanel
- serialVersionUID : long
- zoneTexte : JTextField
- bouton : JButton
+ TextFieldHashTagPanel(uc : UserConsoleDistante)
PopupMenuListenerHashTags
- uc : UserConsoleDistante
- selectedHashTag : String
+ PopupMenuListenerHashTags(uc : UserConsoleDistante, selectedHashTag : String)
+ actionPerformed(event : ActionEvent)
+ setSelectedUser(selectedHashTag : String)
PopupMenuListener
- uc : UserConsoleDistante
- selectedUser : Utilisateur
+ PopupMenuListener(uc : UserConsoleDistante, selectedUser : Utilisateur)
+ actionPerformed(event : ActionEvent)
+ setSelectedUser(selectedUser : Utilisateur)
ListenerZoneTexteRecherche
- zoneTexte : JTextField
- uc : UserConsoleDistante
- listUtilisateursRecherche : JList<Utilisateur>
+ ListenerZoneTexteRecherche(zoneTexte : JTextField , listUtilisateursRecherche : JList<Utilisateur> , uc : UserConsoleDistante)
+ actionPerformed(e : ActionEvent)
ActionListener

C. Package 3 : '*Appchat.rmi*' :

Ce package regroupe les Classes permettant d'établir une connection via des serveurs RMI. On retrouve la classe principale et exécutable pour le serveur, ***AppRMIServeurImpl***, ainsi que le RMI lié au client. Le client possède la classe exécutable ***UserConsoleDistante*** qui va chercher le serveur dans le registre RMI à la bonne addresse IP et ***UtilisateurServeurImpl*** qui permet à l'utilisateur d'instancier son propre serveur afin de recevoir des notifications par exemple. Ci-dessous, le diagramme de classe de ce package :



II. Problèmes rencontrés et solutions :

Parmi les problèmes rencontrés lors de l'implémentation de l'applicatio, on y trouve la difficulté de **la récupération des messages entrants en réseau local.**

En effet, un utilisateur du programme doit être capable d'envoyer des messages à d'autres utilisateurs mais aussi en recevoir. **Il doit donc être notifié lors de l'arrivée d'un message lui étant destiné afin de pouvoir l'afficher et le lire.**

Nous avons donc mis en place un serveur du côté du client afin que le serveur puisse notifier le client. La classe abstraite *UtilisateurServeur* et son implémentation *UtilisateurServeurImpl* remplissent ce rôle. Nous avons donc définit une méthode **recevoirMessage(Message m)** qui sera appelée par le serveur lors de l'arrivée d'un message.

Cette solution fonctionne très bien sur un même ordinateur. Cependant nous ne sommes pas parvenus à la faire fonctionner dans le cadre d'un réseau local avec différentes machines : le serveur ne parvient pas à se connecter au serveur du côté du client. Aussi, malgré de nombreux essais concernant les ports ou les adresses IP, cette solution restait inopérationnelle.

Nous avons donc décidé de changer de stratégie et de laisser le client choisir quand il mettrait à jour sa liste de messages. Le client peut ainsi, à intervalles réguliers, récupérer la liste des messages reçus auprès du serveur. Plus cet intervalle est court et plus l'apparition d'un nouveau message après son envoie sera rapide.

Pour implémenter cette solution nous avons utilisé une ***Thread*** **qui fera une requête auprès du serveur à intervalle régulier.** L'intérêt d'utiliser une Thread est de rendre ce processus transparent pour l'utilisateur qui peut continuer d'envoyer des messages, de visionner des messages publiés par d'autres utilisateurs, etc… sans que la recherche de nouveau message ne mette en pause le programme.

Cette solution étant fonctionnelle nous l'avons appliquée pour d'autres cas similaires comme l'affichage des hashtags récemment utilisés.

Ce problème que nous avons rencontré nous a permis de trouver une solution de remplacement qui fonctionne et présente des avantages mais aussi des inconvénients par rapport à la réponse adaptée que nous n'avons pas réussi à mettre en oeuvre. En effet, le fait que ce soit le client qui choisisse quand rafraichir ses messages entrant, ou d'autres paramètres, lui donne la liberté d'accélérer ce processus (pour un ordinateur avec un haut débit) ou de ralentir le rafraichissement de son choix. Un utilisateur ayant un débit assez bas pourrait par exemple choisir de privilégier l'arrivée de messages en sacrifiant la mise à jour des hashtags par exemple. Cependant, cette solution consomme tout de même plus de bande passante puisque l'on met à jour alors qu'il n'y peut être pas eu de changement.

III. Lancement de l'application :

Le programme se compose d'une partie serveur et d'une partie client.

Dans manipulations qui vont suivre on se placera dans le dossier *appChat/bin/* dans lequel se trouve le dossier *appChat* correspondant au package du même nom.

Pour le faire fonctionner, il faut d'abord exécuter le serveur sur un ordinateur. Pour cela on utilise la méthode main de AppRMIServeurImpl avec la commande :

java appChat.rmi.AppRMIServeurImpl

Le serveur est alors exécuté et une addresse IP est visible.

On exécute ensuite un client sur une autre machine. On utilise la commande :

java appChat.rmi.UserConsoleDistante xxxx.xxxx.xxxx.xxxx

Où les *x* correspondent à l'addresse IP visible sur le terminal du serveur.

Le client est alors exécuté et la fenêtre d'accueil se lance. Il s'agit maintenant de créer un compte puis de se connecter avec ces identifiants et finalement d'utiliser le logiciel.