

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



PROJET DE SEMESTRE 2015

Chator - Rapport

Mélanie Huck
Jan Purro
Bastien Rouiller
Miguel Santamaria
Benoist Wolleb

Professeur René Rentsch

TABLE DES MATIÈRES

1	Introduction.....	4
1.1	Objectif.....	4
1.2	Technologies utilisées et conventions de codage	4
1.2.1	Technologies.....	4
1.2.2	Conventions de codage	4
1.3	Pairs	5
2	Conception et description technique	6
2.1	Le modèle Chator	7
2.1.1	Rôle du modèle.....	7
2.1.2	Interactions avec la vue et le contrôleur	8
2.2	Module d'inscription	9
2.2.1	Acteurs du module	9
2.2.2	Détails sur l'interface graphique	9
2.2.3	Validation des champs.....	10
2.2.4	Nom d'utilisateur.....	10
2.2.5	Image de profil	11
2.2.6	Avertissements à l'utilisateur	11
2.2.7	Cheminement pour la création de compte	11
2.2.8	Récupération des informations depuis le formulaire.....	12
2.2.9	Génération du sel pour le mot de passe et hachage de ce dernier	13
2.2.10	Génération du sel pour la paire de clé et chiffrement de la clé privée.....	13
2.2.11	Envoi au serveur	13
2.2.12	Réponse du serveur et connexion	13
2.2.13	Pour conclure	14
2.3	Connexion.....	14
2.3.1	Acteur du module.....	14
2.3.2	Détails sur l'interface graphique	14
2.3.3	Validation des champs.....	15
2.3.4	Cheminement pour la connexion	15
2.4	Module Chat	16

2.4.1	Acteurs du module	16
2.4.2	Détails sur l'interface graphique de la vue principale	17
2.4.3	Présentation	18
2.4.4	Détails techniques	20
2.4.5	Procédure post-connexion	22
2.4.6	Visualisation de compte	24
2.4.7	Raccourcis clavier	25
2.5	Implémentation.....	26
2.5.1	Liste des événements pouvant survenir.....	26
2.6	Salles de discussion	30
2.6.1	Module Room	32
2.6.2	Détails sur l'interface graphique	33
2.6.3	Procédure de création d'une salle.....	34
2.6.4	Procédure d'édition de la salle	36
2.6.5	Procédure d'adhésion à une salle	37
2.6.6	Processus de connexion au serveur	37
2.6.7	Fonctionnalités utilisateur.....	37
2.7	Sécurité.....	38
2.7.2	Mise en œuvre dans l'application	40
2.7.3	Résumé de la mise en œuvre de la sécurité.....	40
2.8	Serveur	44
2.8.1	Architecture.....	44
2.8.2	Interactions.....	44
2.8.3	Objets supplémentaires	45
2.8.4	Architecture.....	46
2.8.5	Protocole	49
2.8.6	Scénario de transmission d'un message.....	57
2.9	Base de données.....	59
2.9.1	Schéma de la base de données	59
2.9.2	Consultation des données	61
2.9.3	Interaction avec la base de données.....	62

3	Problèmes rencontrés	63
3.1	Problèmes organisationnels	63
3.2	Problèmes techniques	63
3.3	Points positifs	63
4	Conclusion	65
4.1	Fonctionnalités	65
4.2	Améliorations futures possibles	66
4.2.1	Côté serveur	66
4.2.2	Côté client.....	66
4.3	Conclusion finale	67

1 INTRODUCTION

1.1 OBJECTIF

Ce projet a pour but le développement d'une messagerie instantanée (chat) de type client-serveur. Celle-ci permettra de communiquer via des canaux de discussion (salles) privés ou publics, modérés par des personnes possédant des droits d'administration. Elle sera donc constituée d'un logiciel client, et d'un logiciel serveur.

L'application cliente fournira un module d'inscription et de connexion, un module de gestion de salles, ainsi que le module principal représentant les flux de discussion.

L'application serveur s'occupera de gérer l'échange de données (messages et autres), ainsi que les lectures / écritures dans une base de données, de manière sécurisée.

1.2 TECHNOLOGIES UTILISÉES ET CONVENTIONS DE CODAGE

1.2.1 Technologies

Les technologies utilisées sont les suivantes.

- Le C++ en tant que langage de production, norme 2011.
- SQLite en tant que moteur de base de données.
- Qt en tant que framework graphique et réseau.
- La bibliothèque OpenSSL pour gérer la sécurité de l'application (TLS, hachage, algorithme de chiffrement symétrique, algorithme de chiffrement asymétrique).

1.2.2 Conventions de codage

Les conventions de codage suivies sont celles présentées dans le document « C++ Programming Style Guidelines », Version 4.9, Janvier 2011, Geotechnical Software Services.

Nous avons décidé d'utiliser l'anglais pour tous les commentaires, la notation camelCase, le retour à la ligne lors d'une accolade.

Les noms de variables doivent être explicites, les noms de constantes doivent être écrits en majuscules.

Les codes envoyés sur le répertoire Git doivent être compilables.

Nous avons également nommé nos variables selon les conventions suivantes :

Labels : lbl_monLabel
Boutons : btn_mon
Combobox : cbo_monTruc
Checkbox : chk_maTruc
LineEdit : ldt_monTruc
Tree : tre_maListe
RadioButton : rbt_blab
ButtonGroup : bgp_balb
SpinBox : sbx_blab
StandardItemModel : sim_blab

Les entêtes de méthodes doivent respecter le format suivant :

```
/*  
  
    * Created by Miguel Santamaria, on 17.04.2015 21:20  
    *  
    * Get all the given user's rooms, and return them in a QList.  
    * Used in : - ControllerChat::loadRooms.  
    *  
    * Last edited by Miguel Santamaria, on 17.04.2015 22:05  
*/
```

1.3 PAIRS

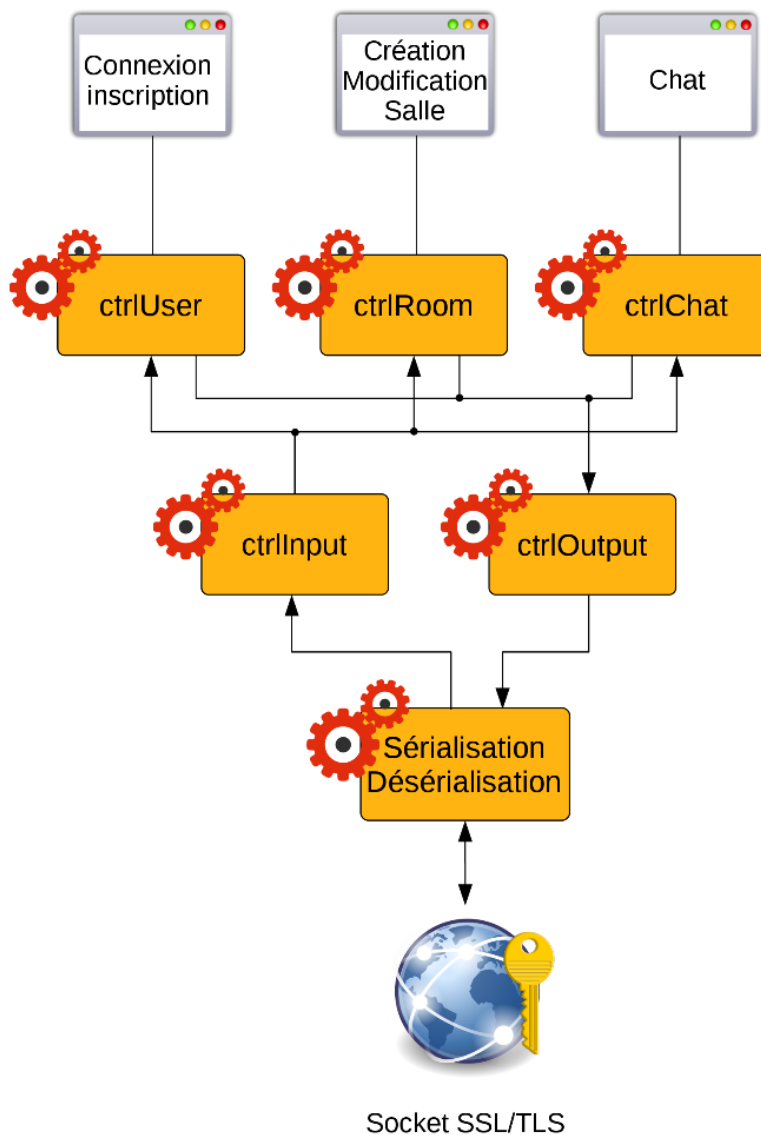
L'équipe de développement est composée des membres suivants :

- Miguel Santamaria (chef de groupe),
- Mélanie Huck,
- Jan Purro,
- Bastien Rouiller,
- Benoist Wolleb,
- René Rentsch (professeur et « client » du projet).

2 CONCEPTION ET DESCRIPTION TECHNIQUE

Les chapitres suivants décrivent de manière détaillée la conception et le fonctionnement de l'application.

Voici un schéma UML simplifié de notre application cliente. L'application est composée des vues d'inscription et de connexion, des vues de création et modification de salle, de la vue de chat. Les contrôleurs user, room, chat, sont liés au « modèle chator », qui n'est pas représenté sur ce schéma pour des raisons de lisibilité.



2.1 LE MODÈLE CHATOR

Afin qu'une application utilisateur puisse stocker toutes les informations relatives au chat, nous avons décidé d'utiliser une architecture inspirée du modèle MVC.

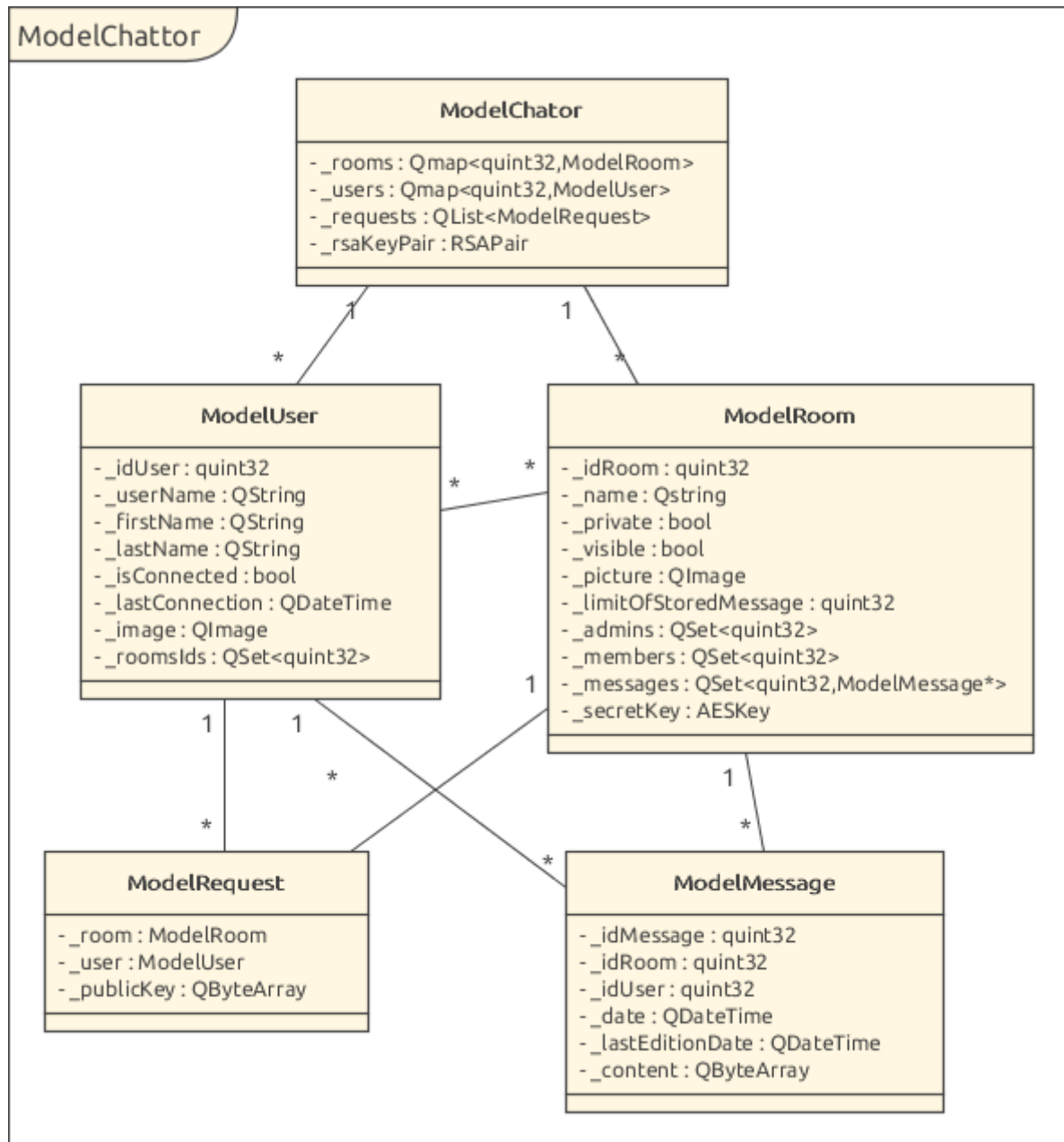


Figure 1: Schéma des modèles de l'application

2.1.1 Rôle du modèle

Le rôle de notre modèle et des modèles en général est de stocker des informations. Ces dernières sont stockées séparément du reste de l'application afin de permettre une certaine réutilisabilité et modularité. À noter que les informations contenues chez un utilisateur ne sont pas forcément les mêmes que chez un autre utilisateur, car ce dernier stockera seulement les informations utiles pour lui-même.

Nous utilisons différentes classes pour effectuer ce stockage (cf. figure 1) :

- ModelChator
- ModelRoom
- ModelMessage
- ModelUser
- ModelRequest

Chacune de ces classes propose des méthodes permettant le contrôle des données (lecture, ajout, modification et suppression) implémentées entre autres par des constructeurs, accesseurs et modificateurs.

ModelChator

Cette classe contient la liste des utilisateurs ayant des salles en commun avec l'utilisateur de l'application. Elle contient également les salles dont l'utilisateur fait partie ainsi que les salles visibles au public. C'est par cette classe que toute demande d'information doit transiter. ModelChator est en mesure de faire suivre les requêtes vers ModelRoom.

ModelRoom

Ce modèle décrit une salle de discussion, il contient la liste de tous ses membres ainsi que diverses informations sur la salle. Les messages sont aussi référencés dans ce modèle. Ainsi toutes les requêtes concernant un message (ajout, suppression, modification) arrivent au ModelRoom depuis le ModelChator afin d'être traitées. Le même cheminement est appliqué pour les opérations relatives à la modification des rôles d'administration.

ModelMessage

Ce modèle est le plus basique, il se situe tout en bas de la hiérarchie. Il stocke uniquement un message et en permet sa modification.

ModelUser

Au même titre que pour le modèle message, on a accès à différents modificateurs et accesseurs sur les champs privés consistant en plusieurs informations sur l'utilisateur, y compris s'il est connecté ou non.

ModelRequest

Permet de stocker les demandes d'adhésion à une salle privée.

2.1.2 Interactions avec la vue et le contrôleur

Il n'y a pas d'interaction à proprement parler entre le modèle et la vue. C'est le contrôleur qui est chargé de faire le lien entre les deux, c'est-à-dire d'afficher les informations du modèle dans la vue ou à l'inverse recevoir une interaction de l'utilisateur sur la vue et répercuter ces actions sur le modèle.

2.2 MODULE D'INSCRIPTION

Afin qu'un utilisateur puisse se créer un compte directement depuis l'application, il est essentiel d'avoir un module inscription intégré à l'interface.

2.2.1 Acteurs du module

ControllerUser

Ce contrôleur a pour but de faire l'intermédiaire entre *viewInscription* et le reste du programme (*modelChator* et serveur). Pour interagir avec le serveur, il se doit de respecter le protocole de sécurité décrit plus tard dans ce document.

ViewInscription

La vue de l'inscription permet de faire une première vérification des champs lors de l'inscription. Cette vérification est détaillée plus tard dans cette section. La vue est également chargée d'informer le *ControllerUser* lorsque l'utilisateur veut soumettre son formulaire d'inscription.

La vue contient également le GUI décrit ci-dessous.

2.2.2 Détails sur l'interface graphique

Pour atteindre la fenêtre d'inscription, l'utilisateur doit cliquer sur le bouton "Inscription" depuis la fenêtre de connexion.

L'interface pour l'inscription est basique. Elle permet à l'utilisateur de comprendre très rapidement la logique derrière l'application. Différents messages d'information lui seront affichés lors d'une saisie incorrecte. Ces messages sont explicités dans le chapitre suivant.

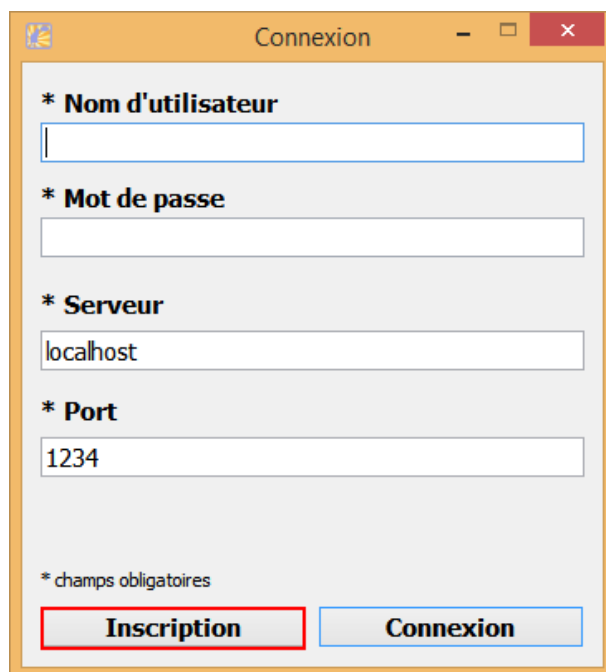


Figure 3: Accès à l'inscription depuis la fenêtre de connexion

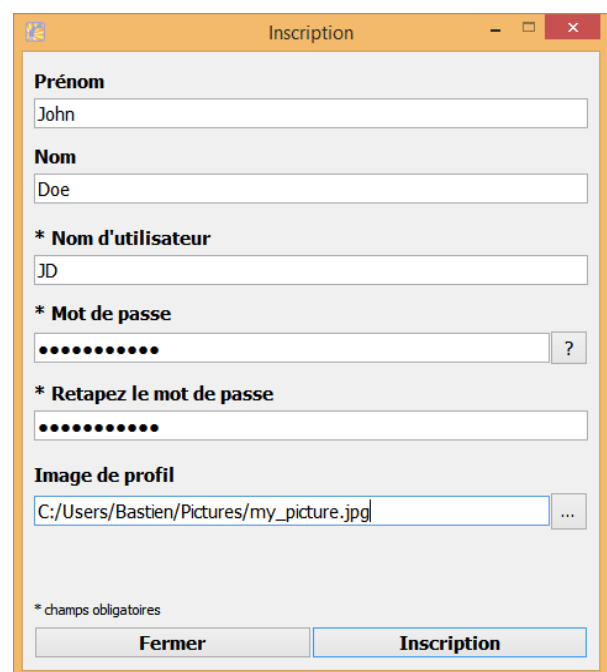


Figure 2: Exemple d'inscription

2.2.3 Validation des champs

Lors de la saisie d'une inscription, l'application vérifie les champs soumis par l'utilisateur pour des raisons de sécurité et de cohérence avec le serveur.

Champs requis

Nous demandons à l'utilisateur le strict minimum d'information lors de l'inscription : un nom d'utilisateur et un mot de passe.

Mot de passe

Dans l'optique d'une application sécurisée, nous imposons certaines contraintes concernant le mot de passe lors de l'inscription. Tout d'abord le mot de passe ne doit pas être identique au nom d'utilisateur. Ensuite, il doit contenir au moins 8 caractères et au moins :

- Une minuscule
- Une majuscule
- Un chiffre
- Un caractère spécial

Pour ce faire, nous utilisons une expression régulière à l'aide de l'objet QRegExp. Voici le pattern :

```
^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[^\a-zA-Z0-9]).{8,}$
```

"^" et "\$" indiquent le début et la fin du String respectivement.

Ensuite nous avons quatre blocs, ils requièrent l'utilisation d'au moins une minuscule, d'une majuscule, d'un chiffre et d'un caractère spécial respectivement. Le caractère spécial est simplement un caractère qui ne correspond à aucun des trois précédents blocs.

Pour finir, on indique avec {8,} le nombre minimum de caractères.

Cette même validation de mot de passe est bien sûr utilisée lors du changement de mot de passe d'un utilisateur.

2.2.4 Nom d'utilisateur

L'unique contrainte concernant le nom d'utilisateur est le type de caractère utilisé : il peut uniquement contenir des lettres, des chiffres ainsi que les caractères – et _ ainsi on évite d'avoir des noms d'utilisateur incompréhensibles.

Le nom d'utilisateur "Anonyme" (non sensible à la casse) est réservé pour des raisons d'implémentation. En effet ce nom est utilisé dans la fenêtre de chat, lorsqu'un utilisateur est inconnu. Ceci peut se produire qu'un utilisateur quitte une salle après avoir écrit des messages. Il devient "anonyme" pour les utilisateurs n'ayant aucune salle en commun avec lui

2.2.5 Image de profil

L'utilisateur a la possibilité de définir son image de profile. Cette image doit respecter un format standard d'image (jpg, jpeg, png, gif), pour ce faire on utilise une variable de type QImage qui se charge elle-même de faire les vérifications nécessaires. Cette image sera immédiatement redimensionnée pour des raisons de stockage sur le serveur ainsi que pour optimiser l'affichage dans l'application.

2.2.6 Avertissements à l'utilisateur

Si une de ces précédentes conditions n'est pas respectée concernant les champs, un de ces messages sera affiché à l'utilisateur dans un label dédié en bas de fenêtre :

Veuillez mentionnez tous les champs requis.

Le mot de passe ne correspond pas.

Votre mot de passe doit être différent de votre nom d'utilisateur.

**Votre mot de passe doit contenir au moins 8 caractère et:
une miniscule, une majuscule, un chiffre et un caractère spécial.**

Ce nom d'utilisateur est un nom réservé par le système.

Le nom d'utilisateur doit contenir des chiffres, des lettres ou - et _

2.2.7 Cheminement pour la création de compte

Afin de permettre l'utilisation du chiffrement et un certain niveau de sécurité au sein de l'application. Il est nécessaire d'effectuer des tâches bien précises lors de la création d'un compte. Ces tâches sont principalement effectuées dans `controllerUser` mais elles sont également partagées avec les différents acteurs de l'application pour permettre une communication avec le serveur.

Le module cryptor permet d'effectuer toute action relative à la sécurité. Il sera donc vivement sollicité lors de l'inscription.

Accès à la fenêtre d'inscription

Lors du clic sur le bouton "Inscription" depuis la fenêtre de connexion, une connexion va être établie (ouverture d'un socket TCP) avec le serveur afin de garantir le fait que le serveur est bien atteignable. Cette connexion va être conservée, de sorte que lorsqu'on s'inscrit, on n'a pas besoin de refaire cette connexion, on peut directement envoyer le paquet avec les informations de l'utilisateur au serveur.

Des messages d'erreurs concernant le certificat SSL vont s'afficher, car dans le cadre du projet nous n'avons pas fait l'acquisition de certificat SSL auprès d'un organisme certifié. Le certificat SSL permet de garantir l'identité d'un hôte, dans notre cas du serveur de chat.

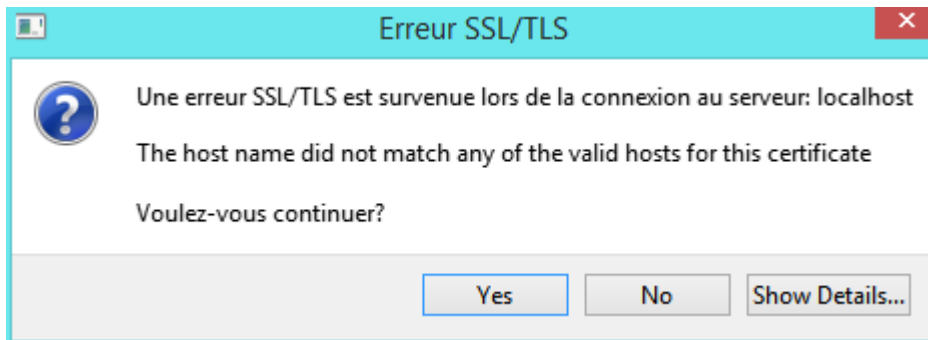


Figure 4: Premier message d'erreur à la connexion

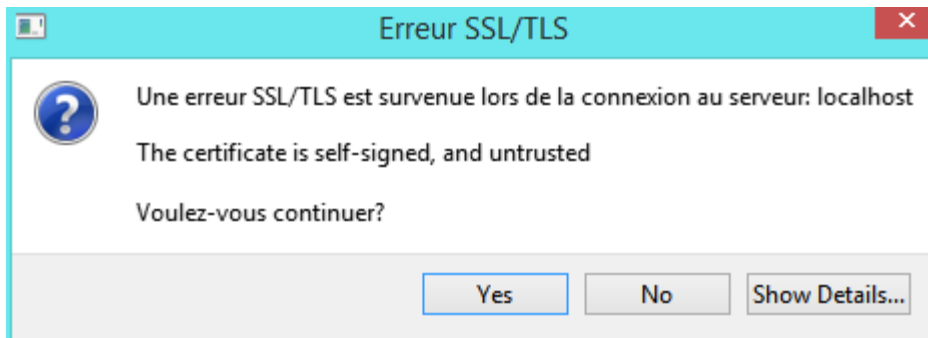


Figure 5: Second message d'erreur à la connexion

Timeout lors de la connexion

Dans le cas où le serveur est inaccessible, c'est-à-dire qu'il ne répond pas après un certain temps, une fenêtre d'erreur s'affichera et l'utilisateur sera invité à saisir à nouveau l'adresse et le numéro de port du serveur. Pour calculer ce timeout, un objet de type QTimer est utilisé. Ce chronomètre est enclenché au clic sur le bouton "Inscription" de la fenêtre de connexion et déclenché lors d'une réponse du serveur.

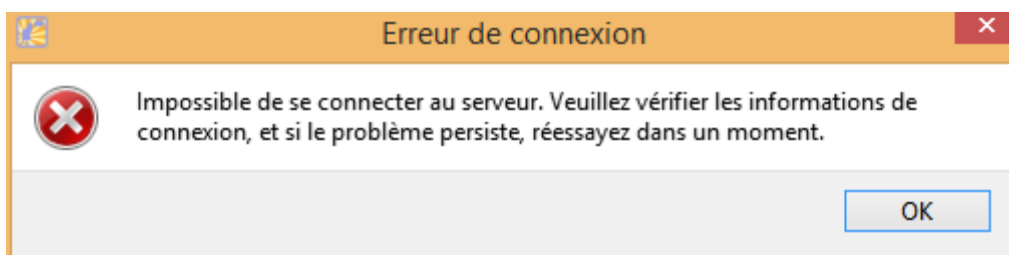


Figure 6: Message d'erreur lorsqu'une connexion ne peut pas être établie

2.2.8 Récupération des informations depuis le formulaire

Une fois que l'utilisateur a saisi les informations relatives à son nouveau compte (nom, prénom, nom d'utilisateur, mot de passe, image de profile). Le contrôleur se charge de récupérer les valeurs saisies après avoir été sollicité par la vue grâce au processus de signal.

2.2.9 Génération du sel pour le mot de passe et hachage de ce dernier

Etant donné qu'un mot de passe ne doit pas transiter sur le réseau en clair pour des raisons évidentes, il est nécessaire d'hacher ce dernier. De plus, le serveur lui-même ne doit pas connaître ce dernier. Il se contentera de le stocker sous forme de hash.

Grâce à la fonction "generateSalt" du Cryptor, on génère un sel. Ensuite ce sel est utilisé en complément du mot de passe afin d'obtenir un hash.

2.2.10 Génération du sel pour la paire de clé et chiffrement de la clé privée

Ensuite, on va générer une paire de clé, et ceci toujours en local, puis du sel. Ce sel contribuera au chiffrement AES de la clé privée. Par définition, la clé publique n'a aucun besoin d'être "cachée" aux autres.

De cette manière, on pourra stocker une paire de clé (dont la clé privée est chiffrée) directement sur le serveur. Lors d'une connexion, l'utilisateur pourra recevoir cette paire de clé et déchiffrer la partie privée.

Pour l'envoi d'un message chiffré à l'utilisateur dans le cadre d'une salle privée, on utilisera sa clé publique pour le chiffrement. Ainsi il pourra déchiffrer le message avec sa clé privée.

2.2.11 Envoi au serveur

Une fois toutes les opérations précédemment décrites effectuées, on peut maintenant transmettre toutes les informations au serveur.

2.2.12 Réponse du serveur et connexion

Pour finir, le serveur va indiquer si la création de compte s'est terminée avec succès. Si le nom d'utilisateur est déjà utilisé, un message d'erreur apparaîtra et l'utilisateur devra choisir un autre nom d'utilisateur.

Si aucune erreur ne se présente, on peut se connecter au chat grâce aux informations renvoyées par le serveur. La fenêtre de chat s'ouvrira et l'utilisateur pourra enfin communiquer avec d'autres utilisateurs à travers des salles. Le déroulement détaillé de la connexion est décrit dans le chapitre suivant "Module Connexion".

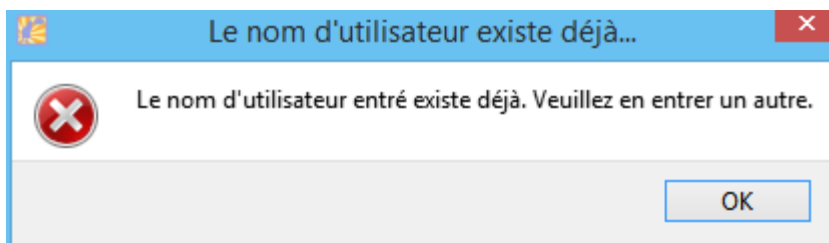


Figure 7: Message d'erreur lorsque le nom d'utilisateur est déjà pris

2.2.13 Pour conclure

Toutes ses manipulations permettent de rendre invisible à l'utilisateur les différentes couches de sécurité. A partir d'un seul mot de passe et nom d'utilisateur, il est capable d'accéder à une application sécurisée. Ceci marque l'importance d'un mot de passe avec une certaine complexité étant donné que toute la sécurité se base sur ce dernier.

Dans le cas où le serveur serait compromis, cela ne mettrait en aucun cas les informations de l'utilisateur en péril. Car comme expliqué précédemment aucune information critique (clé privée et mot de passe) n'est stockée en clair.

2.3 CONNEXION

Lorsque l'utilisateur ouvre le programme Chator, la première chose qu'il voit est la fenêtre de connexion. Après avoir s'être inscrit, il peut se connecter au serveur de chat à tout moment à l'aide de ces identifiants.

Dans ce document, nous allons traiter de l'implémentation de la connexion, dans le sens d'authentification auprès du serveur de chat. Ce module est étroitement lié au module d'inscription, il est donc nécessaire de se référer à ce dernier pour comprendre pleinement le fonctionnement de ce module.

2.3.1 Acteur du module

ControllerUser

Ce contrôleur est partagé avec le module d'inscription. Il permet en plus de son rôle vis-à-vis de l'inscription, de connecter la vue avec le reste du programme. Il permet la récupération des données saisies dans l'interface de connexion et l'interaction avec le serveur.

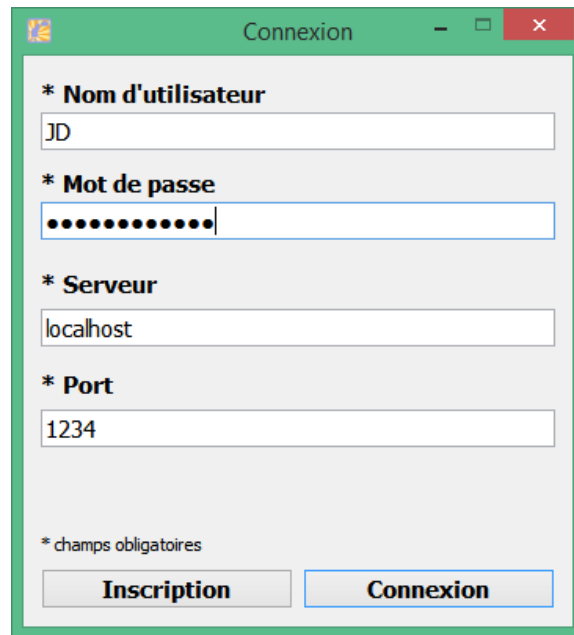
ViewUser

La vue utilisateur a un rôle mineur. Elle est chargée de vérifier que tous les champs sont remplis lorsque l'utilisateur veut se connecter, si c'est le cas elle envoie un signal au contrôleur.

Elle se charge également d'afficher les potentiels messages d'erreurs.

2.3.2 Détails sur l'interface graphique

L'interface utilisateur est très simple, à même titre que celle pour l'inscription.



The image shows a graphical user interface window titled "Connexion". It contains four input fields, each preceded by an asterisk indicating it is mandatory: "Nom d'utilisateur" (containing "JD"), "Mot de passe" (containing ten dots), "Serveur" (containing "localhost"), and "Port" (containing "1234"). Below these fields is a small text label "* champs obligatoires". At the bottom of the window are two buttons: "Inscription" and "Connexion".

Figure 8 Fenêtre de connexion

2.3.3 Validation des champs

La seule vérification faite au niveau de la vue est la présence de tous les champs. Si l'utilisateur n'a pas mentionné un des champs, le message suivant lui sera affiché :

Veuillez mentionnez tous les champs requis.

2.3.4 Cheminement pour la connexion

Récupération des informations depuis le formulaire

Dans un premier temps, au clic sur le bouton "Connexion", la vue va envoyer un signal au contrôleur afin de lui informer que les informations sont prêtes à être récupérées. Le contrôleur va ensuite prendre le relai.

Ouverture d'un socket avec le serveur

Tel que décrit dans la documentation sous "Module Inscription", une connexion est tout d'abord ouverte avant l'interaction avec le serveur. Cette connexion consiste en l'ouverture d'un canal de communication avec le serveur à l'aide d'un socket. Il faut la différencier de la connexion à proprement parler au serveur de chat qui englobe le tout (vérification des identifiants, autorisation d'accès au chat).

Si l'utilisateur a déjà un socket ouvert avec le serveur, par exemple s'il avait ouvert par erreur la fenêtre d'inscription, la connexion est conservée.

Demande du sel auprès du serveur

Une fois la connexion établie entre le serveur et le client. Une demande de sel est formulée envers le serveur. Une fois le sel reçu, l'utilisateur peut hacher son mot de passe.

Envoi des identifiants

On peut désormais prouver son identité auprès du serveur en envoyant son nom d'utilisateur et son mot de passe haché. Le serveur va comparer le hash avec celui contenu dans sa base de données. En effet, pour des raisons de sécurité évidente, le mot de passe n'est pas stocké en clair dans la base de données.

Récupération de la réponse du serveur

Si les informations fournies par l'utilisateur sont correctes, l'utilisateur reçoit les informations dont il a besoin : son `ModelUser` contenant ses détails du profil, sa paire de clé ainsi que le sel pour le chiffrement de sa clé privée.

En effet, les clés ont été générées à l'inscription par l'utilisateur puis stockée sur le serveur. Avant ce stockage, il a été nécessaire de chiffrer la clé privée de l'utilisateur à l'aide du sel et de son mot de passe. Il peut désormais déchiffrer sa clé privée qui lui permettra à son tour de déchiffrer les messages qui lui sont destinés dans l'application.

Dans le cas où l'utilisateur fournit des informations erronées (nom d'utilisateur ou mot de passe), le serveur lui enverra un message d'erreur.

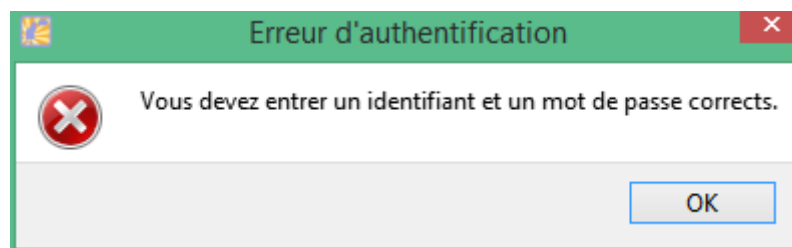


Figure 9 Erreur lorsque l'utilisateur fournit de mauvais identifiants.

Ouverture du chat

Maintenant que l'utilisateur a tout en main, il peut accéder à la fenêtre de chat et utiliser l'application à sa guise.

2.4 MODULE CHAT

2.4.1 Acteurs du module

Toutes les méthodes possèdent un entête précis et clair dans le code, détaillant leur fonctionnement. Pour cette raisons, nous ne nous étendrons pas (ou peu) sur celles-ci dans cette partie.

ControllerChat

Il s'agit du contrôleur du module. Il gère notamment les interactions entre le modèle (**ModeleChator**), la vue (**ViewChat**) et le serveur, et agit donc comme « noyau central » du module. Il est créé dans le `main.cpp` principal de l'application, en même temps que tous les autres contrôleurs, et devient actif directement après qu'une demande de connexion ait été validée par le serveur.

Il connaît, par attributs privés :

- Sa vue (**ViewChat**).
- Le modèle (**ModeleChator**).
- Un objet de type « pointeur sur **ModelUser** » représentant l'utilisateur courant (connecté) qui a été paramétré lors de la connexion de ce dernier.
- Les différents objets en rapport avec la communication avec le serveur.
- Le contrôleur du module Salle (**ControllerRoom**), qui lui sera utile pour ouvrir les différentes fenêtres de gestion de salles (ajout, édition, ...).
- L'objet **Cryptor** lui permettant d'implémenter la sécurité.

Tout ce qui doit transiter de la vue à une autre entité, ou d'une entité à la vue, passe par ce contrôleur ; il effectue les contrôles nécessaires et traite les données avant de les envoyer plus loin. C'est aussi lui qui lancera la vue au moment opportun.

ViewChat

Il s'agit de la vue principale du module. Elle contient notamment l'interface graphique (*viewChat.ui* – voir plus loin pour plus de détails) et est lancée par le contrôleur (**ControllerChat**) après la connexion d'un utilisateur.

Son but principal est de mettre en style et d'afficher les différentes données reçues par le modèle et le serveur, qui transitent via le contrôleur.

ViewMembershipRequests

Cette vue contient une interface graphique permettant de gérer les demandes d'adhésions aux salles privées auxquelles l'utilisateur connecté est administrateur. Elle est lancée par le contrôleur (**ControllerChat**) lorsque l'utilisateur appuie sur le menu « Notifications > Demandes d'adhésion... » (notifié par la vue **ViewChat**).

Cette vue est composée d'un arbre (**QTreeWidget** – plus de détails plus bas) contenant les demandes d'adhésion, ainsi que de deux boutons permettant à l'administrateur d'accepter ou de refuser une demande (pour plus de détails, se référer au chapitre « Liste des événements pouvant survenir » -> « Un utilisateur a fait une demande d'adhésion sur une salle privée dont je suis administrateur »).

ViewAbout

Cette vue affiche les détails d'à-propos de l'application. Elle est lancée par le contrôleur (**ControllerChat**) lorsque l'utilisateur appuie sur le menu « ? > À propos... » (notifié par la vue **ViewChat**).

2.4.2 Détails sur l'interface graphique de la vue principale

L'interface graphique de ce module a été réalisée à l'aide du générateur intégré à l'IDE Qt Creator.

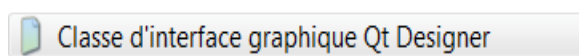


Figure 10 : Création de l'interface graphique avec Qt Creator.

Elle est lancée et stockée dans la vue du module, **ViewChat**.

2.4.3 Présentation

Elle se présente de la manière suivante :

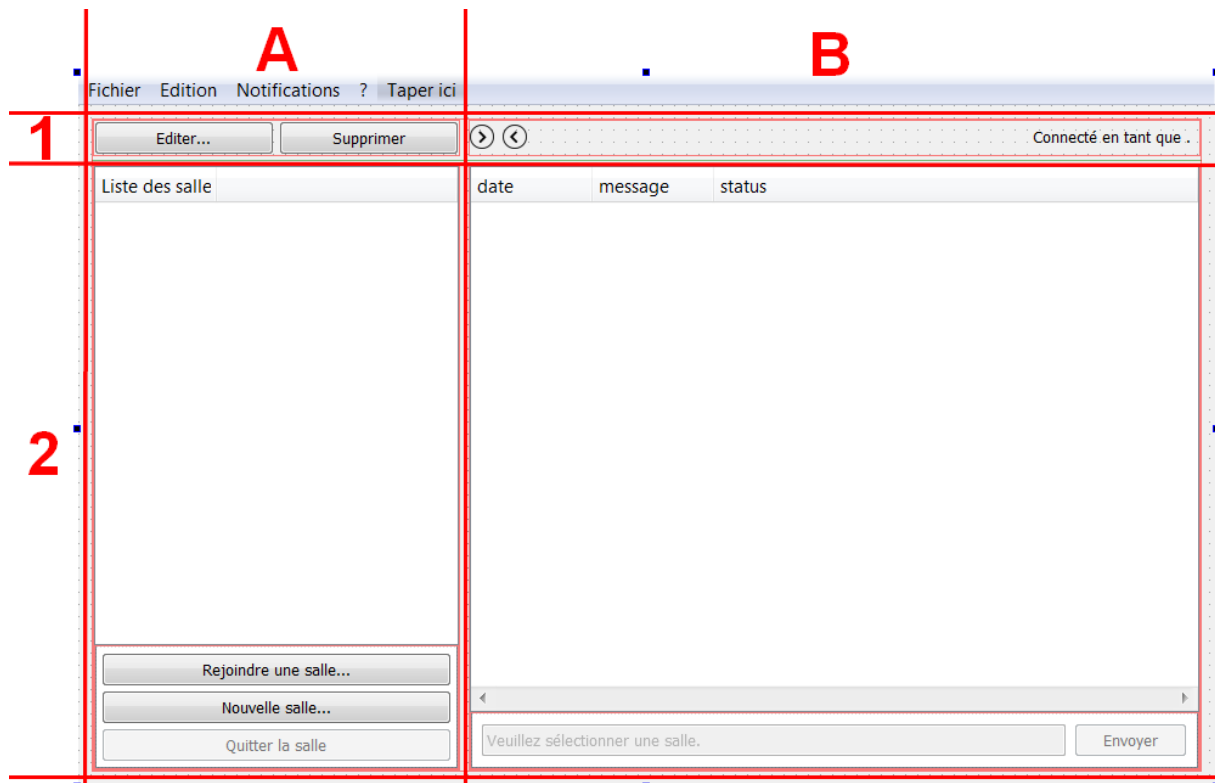


Figure 11 : Interface graphique du module Chat.

Les blocs sont organisés en « layouts » (mises en page), de la manière suivante (en essayant de rester bref) : la fenêtre contient un **QGridLayout** (mise en page dans une grille contenant des cellules) de deux lignes (1 et 2) et deux colonnes (A et B). La première ligne (1) est fine, et ne s'agrandit qu'en largeur (jamais en hauteur) ; elle contient des layouts de type **QHBoxLayout** (alignement horizontal des éléments). La deuxième ligne (2) s'agrandit par contre en largeur et en hauteur. Elle contient elle-même des layouts de type **QVBoxLayout** (alignement vertical des éléments) contenant deux éléments : celui du haut (les arbres de salles (A2) et de messages (B2)) qui peut s'agrandir verticalement, alors que celui du bas qui ne le peut pas.

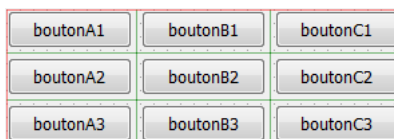


Figure 12 : Exemple de QGridLayout (3x3).

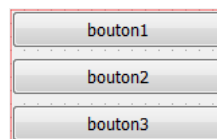


Figure 14 : Exemple de QVBoxLayout.

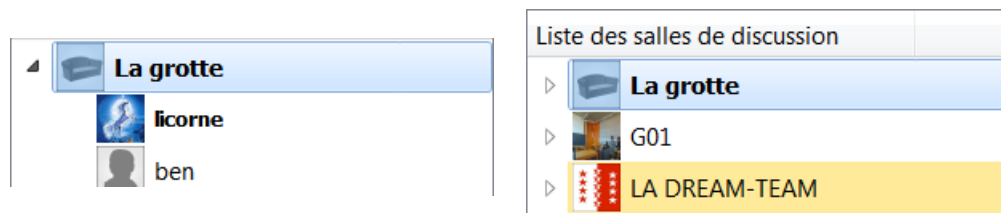


Figure 13 : Exemple de QHBoxLayout.

Trois blocs principaux sont présents dans cette interface :

1. La barre de menu, permettant d'accéder à diverses fonctionnalités :
 - a. Quitter l'application : Fichier > Quitter
 - b. Afficher les informations du compte de l'utilisateur courant (anciennement édition de compte) : Edition > Informations utilisateur...

- c. Gérer les demandes d'adhésion aux salles privées auxquelles nous sommes administrateur : Notifications > Demandes d'adhésion...
 - d. Afficher une fenêtre d'à-propos : « ? » > A propos.
2. Le bloc de gestion des salles (A, sur le schéma), il est composé (de haut en bas) de :
- a. (1) Deux boutons d'administration de salle, permettant de l'éditer, respectivement de la supprimer ; ils sont visibles et actifs uniquement si l'utilisateur courant est administrateur de la salle sélectionnée.
 - b. (2) La liste des salles, représentée par un arbre (fonctionnement détaillé plus bas) dont les nœuds de premier niveau correspondent au nom des salles, et ceux de deuxième niveau à ceux des utilisateurs présents dans la salle. Les utilisateurs connectés apparaissent en gras. Les salles privées apparaissent en doré.



- c. (2) Trois boutons permettant respectivement de rejoindre une salle (ouvre une nouvelle fenêtre), d'en créer une nouvelle (pareil) ou de quitter celle qui est sélectionnée dans l'arbre (après confirmation).
3. Le bloc de messagerie à proprement parler (B, sur le schéma), qui est composé de (de haut en bas, gauche à droite) :
- a. (1) Deux boutons permettant respectivement de dérouler tous les messages contenus dans les nœuds (a), ainsi que de les enrouler (b).

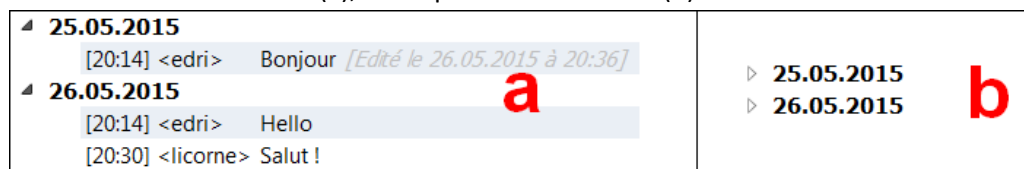


Figure 17 : Dérouler les messages (a), ou les enrouler (b).

- b. (1) Un libellé indiquant le nom de l'utilisateur courant.
- c. (2) Un arbre contenant la liste des messages de la salle, organisés par date ; les nœuds de premier niveau de cet arbre représentent les différentes dates, et ceux de deuxième niveau les messages de la salle. Un message est ordonné en trois colonnes :
 - i. La date et l'utilisateur.
 - ii. Le contenu du message.
 - iii. La date de dernière édition du message (facultative).

Les messages de l'utilisateur courant sont mis en valeur. Lorsque l'on double-clique sur l'un de ceux-ci (peu importe la colonne), il devient éditable ; il est alors possible de le modifier, ce qui enverra une requête au serveur.

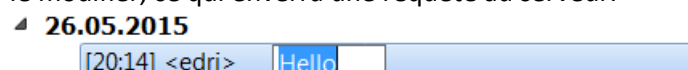


Figure 18 : Edition d'un message.

Il est aussi possible de réaliser un clic-droit sur l'un de nos messages, afin de faire apparaître un menu contextuel qui permet de l'éditer (comme ci-dessus) ou de le supprimer après confirmation.

26.05.2015



Figure 19 : Menu contextuel qui apparaît lors d'un clic-droit sur l'un de nos messages.

- d. (2) Un champ de texte permettant d'écrire un message, ainsi que le bouton permettant de l'envoyer au serveur.

2.4.4 Détails techniques

Les différentes actions réalisables sur les composants (clic sur un bouton, appui sur une touche, etc.) sont réalisées à l'aide de signaux et de slots, afin de respecter les conventions de Qt. En deux mots, l'action envoie un signal qui est récupéré dans une méthode (que l'on appellera un slot, dans ce contexte) qui agit en conséquence.

De nombreux composants C++ possèdent leur « homonyme » dans Qt, qui commencent tous par la lettre 'Q' par convention : `string` <=> `QString`; `unsigned int` <=> `quint32`, etc. Ils permettent notamment d'assurer une portabilité robuste entre les systèmes (un `quint32` fera toujours 32 bits, quel que soit le système utilisé, par exemple), et de simplifier les opérations (un `QString` agira plus comme un `String` en Java que comme un `string` en C++, par exemple). A noter que l'objet de type `QVariant` accepte n'importe quelle valeur, un peu à la manière d'un `Object` en Java.

Arbres

Les arbres contenus dans l'interface (liste des salles et liste des messages) sont des composants du type `QTreeWidget`, qui contiennent des nœuds de type `QTreeWidgetItem` (qui contient du texte et éventuellement des données « cachées » sur lesquelles nous pourrions travailler, le tout ordonné par colonnes). Ces composants possèdent de nombreux paramètres et de nombreuses fonctionnalités qui sont détaillés précisément dans la documentation officielle de Qt, raison pour laquelle ils ne seront pas tous expliqués ici.

Lorsque nous souhaitons ajouter un nœud enfant à l'arbre, il faut :

1. Créer un nouvel objet de type (pointeur sur) `QTreeWidgetItem`.
2. Lui ajouter du texte dans les colonnes, à l'aide de la méthode `setText(int colonne, const QString& texte)`.

Par exemple : « `item->setText(0, "Coucou");` » ajoutera le texte « Coucou » dans la première colonne du nœud.

3. Lui ajouter éventuellement des données « cachées », à l'aide de la méthode `setData(int colonne, in role, const QVariant& value)`. Le rôle à mettre par défaut est `Qt::UserRole` ; il devra être indiqué aussi lors de la récupération de la donnée.

Par exemple : « `item->setData(0, Qt::UserRole, 50);` » ajoutera la donnée « 50 » dans la première colonne du nœud.

Pour pouvoir récupérer ces données par la suite, il suffira d'appeler la méthode `data(int colonne, in role)`, avec éventuellement une fonction de conversion. Par exemple : « `int nb = item->data(0, Qt::UserRole).toInt();` » récupèrera la donnée ajoutée précédemment.

4. Eventuellement lui ajouter d'autres paramètres (police d'écriture, taille, couleur, etc.).
A noter que Qt implémente un système de fanions (flags) permettant de paramétrer le comportement de certains objets, donc les `QTreeWidgetItem` : il est possible d'en ajouter à l'aide de la méthode `setFlags(Qt::ItemFlags flags)` ; nous pouvons en placer plusieurs en série à l'aide de l'opérateur OU binaire (« | »). La documentation possède de nombreux détails quant à ce système de fanions ; voici cependant quelques exemples utilisés dans le code :
 - i. `Qt::NoItemFlags` => aucun fanion (le nœud est désactivé).
 - ii. `Qt::ItemIsEnabled` => le nœud est activé (n'apparaît plus en grisé).
 - iii. `Qt::ItemIsEditable` => le nœud est éditable (à nous de gérer le signal).
 - iv. `Qt::ItemIsSelectable` => le nœud est sélectionnable.
5. Insérer le nœud dans l'arbre à la position souhaitée, à l'aide de la méthode `insertTopLevelItem(int index, QTreeWidgetItem* item)`, si nous souhaitons l'ajouter en tant qu'enfant de premier niveau. Si nous souhaitons l'ajouter en tant qu'enfant de niveaux inférieurs, il faut y aller récursivement, en récupérant le nœud de premier niveau concerné, et en lui ajoutant l'objet en enfant : `topLevelItem(int index)->insertChild(int index, messageItem)`. Nous pouvons aller encore plus loin avec la fonction `child(int index)...`

Menus contextuels

Le menu contextuel est représenté par un objet de type `QMenu`, stocké dans notre cas comme attribut privé de la vue, afin d'éviter de le recréer à chaque fois (vu qu'il pourra être appelé souvent). Il faut indiquer à cet objet quel est son parent, à l'aide de son constructeur (dans notre cas, la vue, donc `this`), afin qu'il sache où et comment il doit se lancer.

Nous pouvons ensuite ajouter des éléments à ce menu, qui sont représentés par des objets de type `QAction`, pouvant posséder une icône (facultative) et un texte, à l'aide de la méthode `QAction* addAction([icône, texte, ...])`.
Par exemple : « `QAction* editAct = _menu->addAction(QIcon(":/icons/img/edit.png"), tr("Editer"));` » ajoute l'élément « Editer » au menu.

Finalement, il faut exécuter le menu à l'aide de la méthode `QAction* exec()`, puis récupérer les actions avec des conditions, par exemple :

```
// Nous plaçons le menu sur le curseur de la souris, dans l'arbre des messages.
QAction* act = _menu->exec(_ui->tre_messages->viewport()->mapToGlobal(pos));

if (act == editAct)
{
    // FAIRE QUELQUE CHOSE...
}
```

Messages de confirmation

Les différents messages de confirmation qui peuvent apparaître sont des objets de type `QMessageBox` (qui héritent de la classe `QDialog`), à partir desquels nous pouvons appeler des méthodes statiques, selon le type de message désiré :





	Question	For asking a question during normal operations.
	Information	For reporting information about normal operations.
	Warning	For reporting non-critical errors.
	Critical	For reporting critical errors.

Figure 20 : Les différents types de messages de `QMessageBox` disponibles, tiré de la documentation de Qt (<http://doc.qt.io/qt-4.8/qmessagebox.html>).

Par exemple, pour construire un message de type « Warning » :

```
int ret = QMessageBox::warning(this, tr("Attention"),
                               tr("Êtes-vous sûr de vouloir supprimer cette
                                   salle ?"),
                               tr("Oui"), tr("Non"));
```

Cela fera apparaître une fenêtre jaillissante (pop-up) avec le titre « Attention », et deux boutons « Oui » et « Non ». La valeur entière de retour permet de connaître le bouton qui a été cliqué (0 => premier bouton – Oui ; 1 => deuxième bouton – Non). Résultat :

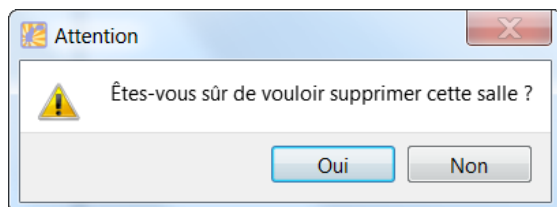


Figure 21 : Pop-up de suppression d'une salle.

2.4.5 Procédure post-connexion

Lorsque la connexion a été effectuée avec succès, le « module Chat » démarre : le **ControllerUser** passe l'objet représentant l'utilisateur connecté au **ControllerChat**, lui indique qu'il peut afficher sa vue (**ViewChat**), puis se ferme.

Dans un même temps, le serveur envoie tout d'abord la liste de toutes les salles auxquelles l'utilisateur connecté appartient (contenant aussi leurs messages respectifs), ainsi que la liste des utilisateurs qui appartiennent au moins à l'une de ces salles.

Côté client, la méthode « **join** » du **ClientControllerInput** insert en premier lieu chaque salle, chaque message et chaque utilisateur reçus dans le modèle – via la **ControllerChat** – puis indique à ce dernier qu'il peut charger la liste des salles dans la vue (à l'aide de la méthode « **loadUserRooms** »). A noter que l'ajout d'une salle privée au modèle s'occupe aussi de récupérer et de stocker sa clé secrète.

Cette méthode ajoute les salles une par une à la vue, ainsi que la liste des utilisateurs, pour chacune d'entre-elles.

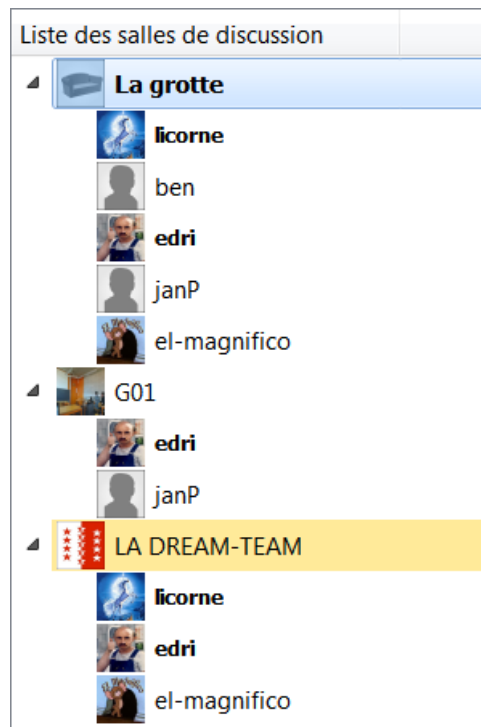


Figure 22 : Liste des salles de discussion et de leurs utilisateurs, dans la vue.

Chacun de ces objets contient des données cachées, utiles pour les traitements auxiliaires :

- Salles : l'ID de la salle dans la première colonne, ainsi que le nombre de nouveaux messages non-lus dans la deuxième (qui vaut 0, par défaut).
- Utilisateurs : l'ID de l'utilisateur dans la première colonne.

Une fois les salles chargées, la vue sélectionne la première de la liste, puis charge ses messages dans l'arbre de droite. Ceci se fait en envoyant le signal « **requestLoadRoomMessages** » au contrôleur, avec l'ID de la salle en paramètre ; celui-ci le rattrape dans le slot « **loadRoomMessages** » qui récupère la liste des messages depuis le modèle et les passe à la vue. A noter qu'il s'occupe aussi d'indiquer à cette dernière si les boutons d'administration (« Editer » et « Supprimer ») s'affichent ou non, selon le rôle de l'utilisateur courant.



Figure 23 : Boutons d'administration d'une salle, affichés uniquement si l'utilisateur est administrateur de la salle sélectionnée.

L'ajout des messages à l'arbre se déroule de la manière suivante dans la vue :

1. Pour chaque message de la liste de messages reçue :
 - a. Créer un nouvel objet du type **QTreeWidgetItem** (qui représente un enfant d'un arbre Qt) contenant la date du message dans la première colonne, son contenu dans

la deuxième, et éventuellement sa dernière date de modification dans la troisième. Chacun de ces objets contient des données cachées, utiles pour les traitements auxiliaires : la date du message, son ID, ainsi qu'une variable booléenne indiquant si le message appartient à l'utilisateur courant (*true*), ou non (*false*).

- b. Si le message appartient à l'utilisateur courant, il sera mis en évidence.

```
[20:14] <edri> Hello  
[22:30] <licorne> Salut !
```

Figure 24 : Mise en évidence des messages de l'utilisateur courant (ici, licorne).

- c. La vue regarde ensuite si l'arbre possède déjà un nœud correspondant à la date du message ; deux actions sont possibles à partir de là :
- Le nœud existe déjà => ajout du message à la fin de ce dernier.
 - Le nœud n'existe pas encore => il faut tout d'abord le créer, l'ajouter après le dernier nœud existant, puis finalement y ajouter le message.

```
▲ 25.05.2015  
[20:14] <edri> Bonjour [Edité le 26.05.2015 à 22:36]  
▲ 26.05.2015  
[20:14] <edri> Hello  
[20:30] <licorne> Salut !
```

Figure 25 : Des messages dans des nœuds représentant des dates.

A noter que le serveur envoie la liste des messages dans l'ordre croissant de leur date de création ; ceci nous permet d'ajouter le message courant ou le nouveau nœud simplement à la fin de la liste pour assurer l'ordre, sans avoir à y réfléchir plus.

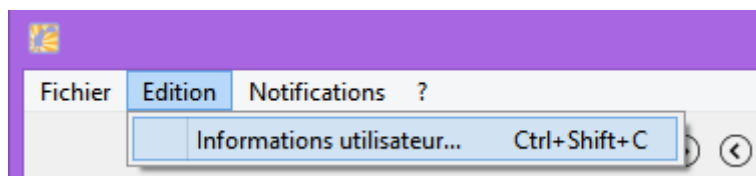
2. Redimensionner les colonnes de l'arbre, et s'y déplacer à la fin.

Une fois les messages chargés, le module est considéré comme lancé et chargé, et attend désormais des événements.

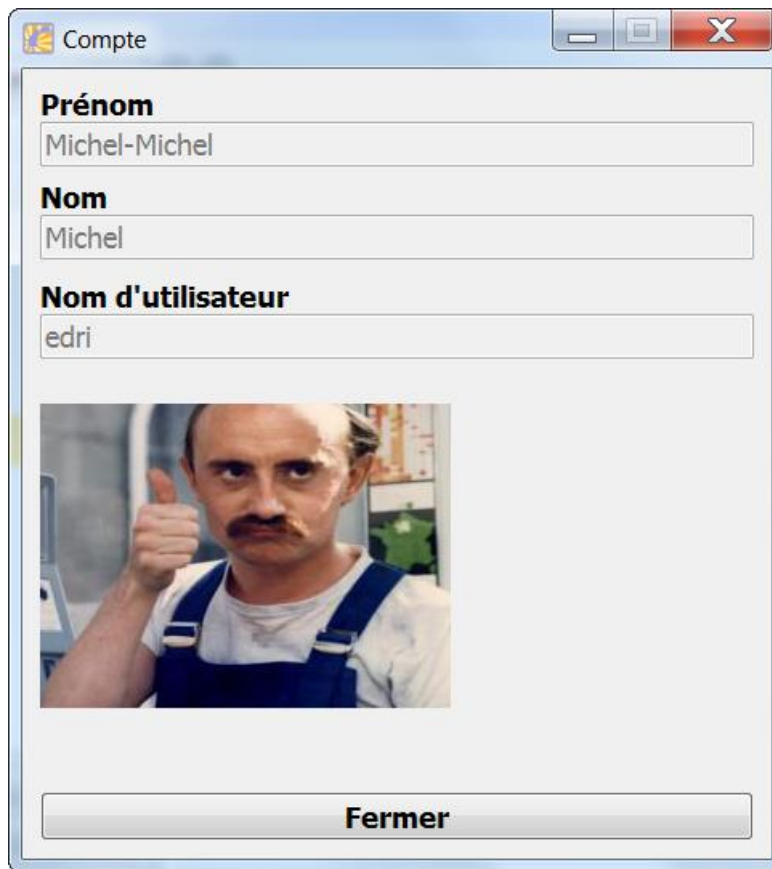
2.4.6 Visualisation de compte

L'utilisateur peut à tout moment consulter les informations relatives à son compte. Il était prévu à la base de pouvoir modifier ces informations mais nous avons dû faire des compromis afin de pouvoir une application complète fonctionnelle.

Afin d'atteindre cette fenêtre de visualisation de compte l'utilisateur doit être connecté sur l'application de chat. Puis se rendre dans le menu "Edition" -> "Informations utilisateur..." ou à l'aide du raccourci CTRL+Shift+C.



Voici à quoi ressemble le GUI :



2.4.7 Raccourcis clavier

L'application comporte les raccourcis clavier ci-dessous.

Lors de la connexion :

- enter : se connecter
- ctrl + i : ouvrir la fenêtre d'inscription
- enter : s'inscrire, une fois que les champs d'inscription sont remplis
- esc : fermer la fenêtre d'inscription

Une fois connecté :

- ctrl + shift + c : afficher les détails du compte
- F1 : ouvrir la fenêtre d'à propos
- enter : envoyer un message lorsque le champ de saisie du texte est sélectionné
- ctrl + e : ouvrir la fenêtre d'édition de salle pour éditer la salle courante
- ctrl + r : ouvrir la fenêtre pour rejoindre une salle
- ctrl + n : ouvrir la fenêtre pour créer une nouvelle salle
- ctrl + q : quitter la salle courante

2.5 IMPLÉMENTATION

Afin d'afficher les informations de l'utilisateur, nous avons utilisé une vue de type "viewInscription". En effet les mêmes types de données sont affichées, il est donc plus rapide reprendre cette vue. A la construction de cette dernière, nous avons modifiés à la volée l'aspect du formulaire en désactivant les champs et en supprimant certains. Nous avons récupéré les informations relatives à l'utilisateur connecté (nom, prénom et image de profil) afin de les afficher via le contrôleur utilisateur.

2.5.1 Liste des événements pouvant survenir

Dans les cas qui suivent, nous identifierons l'utilisateur actuellement connecté par « moi », ou « je », afin de simplifier les descriptions.

J'ai sélectionné une autre salle que celle qui était sélectionnée auparavant dans l'arbre

Lorsque l'utilisateur change la salle sélectionnée dans l'arbre des salles, il faut recharger les messages contenus dans l'arbre des messages. Pour cela, la vue notifie le contrôleur qui lui renvoie la liste des messages de la nouvelle salle sélectionnée, récupérée dans le modèle du client (pas de requête au serveur, car nous connaissons déjà tous les messages de toutes les salles). Cela assure un accès rapide aux données, empêchant ainsi que des problèmes de latence surviennent.

Envoi d'un message

Lorsque j'appuie sur le bouton « Envoyer » et que j'ai entré un message, son texte est converti en format binaire (après avoir été chiffré avec la clé de la salle, si je me trouve dans une salle privée) et est encapsulé dans un objet de type **ModelMessage**. Cet objet est envoyé au serveur qui va le stocker dans la base de données.

Côté vue, rien ne se passe lors de cet événement, si ce n'est l'effacement du contenu du champ de texte du message. On ne peut en effet pas ajouter le message à l'arbre sans être sûr qu'il ait été reçu par le serveur, pour des questions de cohérence.

Réception d'un nouveau message ou d'un message édité

Lorsqu'un utilisateur a envoyé/édité un message et que le serveur l'a reçu, ce dernier va le renvoyer en retour (un objet **ModelMessage** – avec un contenu binaire) à tous les utilisateurs présents dans la salle concernée (y compris l'utilisateur qui a envoyé le message, et qui a besoin d'une confirmation).

Le message est reçu par le contrôleur qui va tout d'abord le déchiffrer, si la salle du message est privée. Il va ensuite agir différemment selon le type du message :

- s'il s'agit d'un nouveau message, il va le stocker dans le modèle, dans la bonne salle.
- s'il s'agit d'un message qui a été édité, il va modifier le message déjà existant dans le modèle, afin qu'il soit à jour, puis va mettre à jour la dernière date de modification de ce dernier.

Le contrôleur va ensuite notifier la vue ; à partir de là, deux cas sont possibles :

- si la salle sélectionnée actuellement dans la vue correspond à la salle du message, la vue va afficher le contenu du message sous format texte dans l'arbre des messages s'il s'agit d'un nouveau message, ou qui va ajouter/mettre à jour la date d'édition de celui-ci dans la colonne la plus à droite de l'arbre.

▲ 25.05.2015

[20:14] <edri> Bonjour [Edité le 26.05.2015 à 20:36]

Figure 26 : Affichage d'un message édité.

- si la salle sélectionnée actuellement dans la vue ne correspond pas à la salle du message et qu'il s'agit d'un nouveau message, la vue va ajouter une notification à côté de la salle concernée, indiquant qu'un nouveau message non-lu est disponible. Le nombre de nouveaux messages non-lus est stocké à l'aide d'une donnée cachée liée à la salle, dans l'arbre.
- A noter que s'il s'agit d'une modification de message, rien ne va se passer dans ce cas.

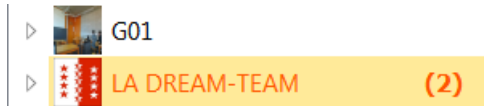


Figure 27 : Notification indiquant de nouveaux messages non-lus, dans une salle.

Suppression d'un message

Lorsque l'utilisateur supprime l'un de ces messages, une notification est envoyée au serveur (en indiquant l'ID de la salle, ainsi que l'ID du message) qui le supprimera dans la base de données, et renverra une confirmation à tous les utilisateurs de la salle (contenant elle-aussi les deux IDs).

Lorsque le contrôleur reçoit la confirmation du serveur, il va tout d'abord supprimer le message du modèle, puis indiquer à la vue qu'elle peut retirer le message de l'arbre, uniquement si la salle sélectionnée correspond à la salle du message supprimé (dans le cas contraire, elle n'a rien besoin de faire).

Un utilisateur s'est connecté/déconnecté

Lorsqu'un utilisateur se connecte/déconnecte, le serveur notifie tous les clients susceptibles d'être intéressés par cette action, à savoir tous les utilisateurs qui ont au moins une salle en commun avec lui.

Lorsque le contrôleur reçoit cette information (l'ID de l'utilisateur, avec une variable booléenne indiquant si l'utilisateur s'est connecté (*true*) ou s'il s'est déconnecté (*false*)), il la transmet à la vue qui s'occupera simplement d'afficher l'utilisateur en gras à chaque fois qu'il apparaît dans la liste des utilisateurs d'une salle s'il s'est connecté, ou de retirer l'affichage en gras s'il s'est déconnecté. L'utilisateur est recherché dans l'arbre à l'aide de son ID, qui est un champ caché (data) des éléments de cet arbre.

Un utilisateur a rejoint/quitté une salle dans laquelle je me situe

Lorsque qu'un utilisateur rejoint/quitté une salle, le serveur notifie tous les clients susceptibles d'être intéressés, à savoir les utilisateurs de cette salle.

Lorsque le contrôleur reçoit cette information (l'ID de la salle, ainsi que l'ID de l'utilisateur), il met à jour son modèle, puis appelle la vue qui retirera/ajoutera cet utilisateur à la liste des utilisateurs de la salle.

Je souhaite supprimer une salle dont je suis administrateur

La suppression se déroule en deux temps : on notifie premièrement le serveur, puis ce dernier envoie une confirmation à tous les utilisateurs de la salle.

Lorsque l'administrateur clique sur le bouton « Supprimer », puis sur celui de confirmation de suppression de la salle, la vue notifie le contrôleur qui envoie une requête (contenant l'ID de la salle) au serveur. Celui-ci supprime la salle, ses messages, ainsi que les liaisons des utilisateurs à la salle, puis leur envoie une confirmation.

Lorsque le contrôleur reçoit la confirmation, il supprime la salle du modèle, puis notifie la vue qui enlèvera de la liste des salles le nœud représentant la salle.

Je souhaite éditer les informations d'une salle dont je suis administrateur

Lorsque l'administrateur clique sur le bouton « Editer... », le module Chat ouvre la fenêtre d'édition de salle, et le module Salle prend le relai.

Lorsque le serveur envoie la notification d'édition à tous les utilisateurs de la salle (contenant un objet de type **ModelRoom**), le contrôleur notifie la vue qui va mettre à jour les informations de la salle, dans l'arbre des salles.

J'ai été ajouté à / ai rejoint une salle

Lorsqu'un utilisateur rejoint une salle, le serveur lui envoie l'objet **ModelRoom** correspondant, ainsi que les différents objets **ModelUser** correspondant à ses utilisateurs. Le contrôleur met à jour le modèle, en y ajoutant la salle et les utilisateurs encore non-connus (c'est-à-dire, les utilisateurs qui n'ont actuellement aucune salle en commune avec moi), puis indique à la vue qu'elle peut ajouter la salle et des utilisateurs à l'arbre des salles (de la même manière que pour la procédure post-connexion, plus haut).

Je souhaite quitter une salle

Lorsqu'un utilisateur appuie sur le bouton « Quitter la salle », la vue notifie le contrôleur qui envoie une requête au serveur, contenant l'ID de l'utilisateur ainsi que celui de la salle.

Une fois que le serveur a traité la requête, il en renvoie une à tous les utilisateurs de la salle concernée. Le contrôleur retire l'utilisateur de la salle dans le modèle, puis notifie la vue qui va retirer l'utilisateur de la salle, dans l'arbre (s'il s'agit d'un utilisateur autre que *moi*), ou alors va retirer la salle de l'arbre (s'il s'agit de *moi*).

A noter que si je suis le dernier utilisateur de la salle, celle-ci sera supprimée définitivement.

Je souhaite créer une nouvelle salle

Lorsque l'utilisateur clique sur le bouton « Nouvelle salle... », le module Chat ouvre la fenêtre de création de salle, et le module Salle reprend la main.

Une fois la salle créée, le serveur envoie une notification (contenant un objet de type **ModelRoom**) à ses utilisateurs. Le contrôleur notifie la vue qui va ajouter la salle ainsi que ses utilisateurs à l'arbre des salles.

Un utilisateur a fait une demande d'adhésion sur une salle privée dont je suis administrateur

Le serveur envoie au client une requête contenant l'ID de l'utilisateur qui fait la demande, sa clé publique, ainsi que l'ID de la salle privée. Lorsqu'il reçoit cette requête, le contrôleur l'ajoute à la liste des requêtes dans le modèle (**ModelRequest**), puis indique à la vue principale (**ViewChat**) qu'une nouvelle notification est disponible, et rafraîchit l'arbre de la vue de gestion des demandes (**ViewMembershipRequests**).

A noter qu'à la connexion d'un administrateur, le serveur lui envoie la liste de toutes les demandes d'adhésion.

La vue principale (**ViewChat**) possède un attribut interne `_nbNotifications` qui comptabilise le nombre de notifications disponibles et les affiche dans la barre de menu. A noter que le nombre de notifications ne peut pas être négatif, et que s'il vaut 0, la vue n'affiche simplement pas de nombre.

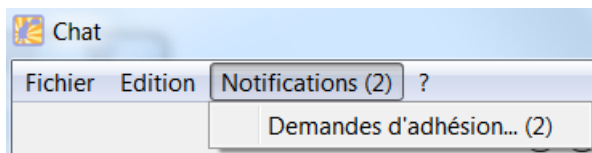


Figure 28 : Affichage des notifications (ici, 2) dans la vue principale du module Chat.

Lorsque l'administrateur appuie sur le menu « Notifications > Demandes d'adhésions... », la vue notifie le contrôleur qui va afficher la vue de gestion des demandes d'adhésion (**ViewMembershipRequests**), après l'avoir rafraîchie au préalable. Cette vue possède en effet une méthode lui permettant de mettre à jour son arbre de requêtes, à partir d'une `QMap` de `ModelRequest`. Si des demandes sont disponibles, les deux boutons « Accepter » et « Refuser » sont activés ; s'il n'y en a aucune, ils sont désactivés.

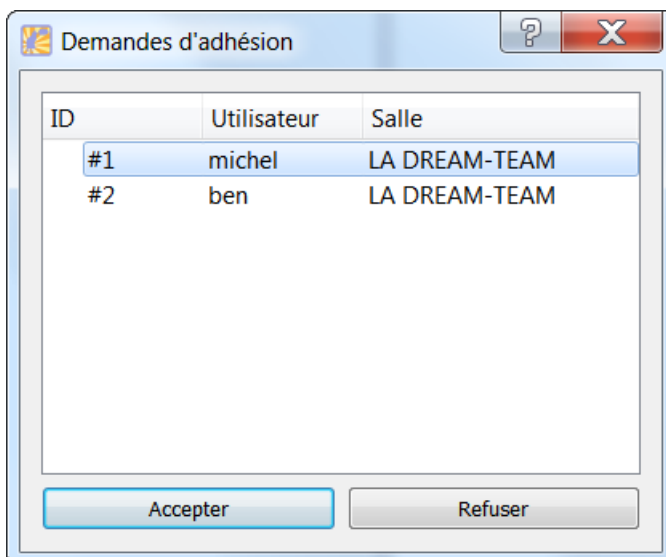


Figure 29 : fenêtre de gestion des demandes d'adhésion.

Les requêtes sont identifiées par des IDs, qui permettent au contrôleur de gérer plus facilement les requêtes. Pour chaque requête, l'administrateur possède deux choix : l'accepter ou la refuser.

Dans les deux cas, la vue se fige, puis envoie au contrôleur un signal qui contient une variable booléenne indiquant si la demande a été acceptée (*true*) ou non (*false*). Le contrôleur va ensuite envoyer au serveur le résultat, en lui indiquant l'ID de la salle privée, l'ID de l'utilisateur concerné, le statut de la requête (acceptée/refusée), ainsi qu'un tableau d'octets (`QByteArray`) qui possèdera une valeur différente en fonction du statut :

- Requête acceptée : le tableau possèdera la clé privée de la salle, chiffrée avec la clé publique de l'utilisateur qui a fait la demande (reçue lorsque le serveur avait envoyé la requête au client), puis convertie en tableau d'octets. Cette clé permet au serveur de gérer plus facilement l'adhésion du client, en la lui renvoyant.
- Requête refusée : le tableau est vide, car l'utilisateur n'a pas besoin de connaître la clé de la salle, dans ce cas.

Lorsque la requête a été traitée, le contrôleur indique à la vue (**ViewMembershipRequests**) qu'elle peut enlever l'objet sélectionné de son arbre, puis la réactive.

Si la requête a été acceptée, le serveur notifie l'utilisateur concerné qui met à jour sa vue.

Je souhaite accéder aux détails de mon compte (anciennement édition du compte)

Lorsque l'utilisateur appuie sur « Edition > Compte », ou sur la combinaison de touches Ctrl+Shift+C, le module Chat ouvre la fenêtre détails du compte, et le module User reprend la main.

Je souhaite afficher la fenêtre d'à-propos

En cliquant sur « ? > A propos », la fenêtre d'à-propos s'affiche.



Figure 30 : Fenêtre d'à-propos.

J'ai perdu la connexion avec le serveur

Lorsque le serveur se ferme, il envoie un signal « disconnected() » à tous ses clients. Le contrôleur du module Chat (**ControllerChat**), s'il est actif (c'est-à-dire, si la connexion utilisateur a déjà été effectuée), indique à la vue (**ViewChat**) qu'elle doit afficher un message d'erreur, puis lorsque celle-ci l'a fait, ferme l'application.

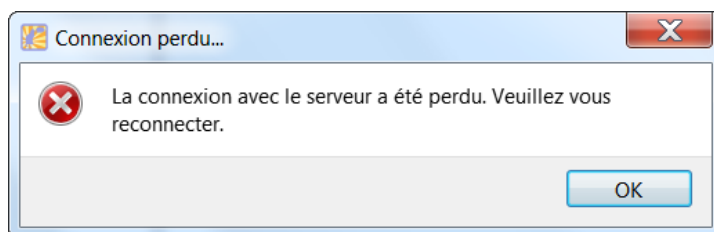


Figure 31 : Connexion perdue avec le serveur.

Je ferme la fenêtre

L'application se termine si la fenêtre est fermée (« Fichier > Quitter », Alt+F4, Ctrl+Q, croix rouge).

2.6 SALLES DE DISCUSSION

Le concept

L'application Chator étant une application de messagerie instantanée, elle permet à ses utilisateurs d'échanger des messages via des « salles de discussions ».

Une salle s'apparente à un groupe de discussion. Les utilisateurs membres d'une salle voient les messages échangés au sein de celle-ci et peuvent eux-mêmes y écrire des messages, qui s'afficheront pour tous les membres de la salle. Les messages écrits peuvent être édités ou supprimés par la suite.

Salles publiques et salles privées

Les salles peuvent être de deux types : publiques ou privées.

Tout utilisateur est libre de rejoindre une salle publique à n'importe quel moment. De plus, les messages échangés au sein d'une salle publique seront stockés « en clair » dans la base de données.

Les salles publiques se distinguent des salles publiques sur deux plans : l'adhésion des nouveaux membres, et le stockage des messages. Les salles privées ne peuvent être rejointes qu'avec le consentement de l'administrateur, et bénéficient de chiffrement des messages stockés sur la base de données. Seuls les membres de la salle peuvent déchiffrer ces messages.

Il y a deux types de salles privées : visibles ou non visibles. Dans le cas d'une salle visible, celle-ci est visible par tous les utilisateurs lorsque ceux-ci cherchent une nouvelle salle à rejoindre. L'utilisateur peut alors envoyer une demande d'adhésion à son administrateur (concept expliqué ci-après) qui pourra soit accepter, soit refuser la demande. Si la demande est acceptée, l'utilisateur devient membre de la salle. Il peut également être ajouté directement par un administrateur.

Dans le cas d'une salle non visible, celle-ci n'est pas visible lors d'une recherche effectuée par un utilisateur, et seul l'ajout par un administrateur est possible.

Quelle que soit la salle, un utilisateur est libre de la quitter à tout moment. Une salle qui n'a plus d'utilisateurs est supprimée.

Administrateurs

L'administrateur d'une salle est l'utilisateur qui a créé la salle. Toute salle possède au moins un administrateur et peut en posséder autant qu'il y a de membres dans la salle.

Un administrateur est un membre de la salle qui possède plus de droits que les autres. Lors de la création d'une salle, l'administrateur lui donne un nom, un nombre maximal de messages stockés (historique), spécifie si elle est privée (visible ou non) ou publique, et lui donne éventuellement une image. Il ajoute les utilisateurs qui deviendront membres de la salle.

Il peut à tout moment modifier ces paramètres ainsi qu'exclure un membre de la salle. Il peut également se faire épauler par un autre utilisateur en le nommant coadministrateur. Ce dernier possèdera alors les mêmes droits d'administration sur la salle.

Dans le cas d'une salle privée visible, c'est l'administrateur qui décide d'accepter ou non une demande d'adhésion de la part d'un utilisateur.

Il peut à tout moment décider de supprimer la salle qu'il administre.

Nombre de messages par salle

Pour des raisons d'espace de stockage et de confidentialité, il existe une limite de message stocké par salle. Ce nombre est déterminé par l'utilisateur au moment de la création de la salle.

Il y a une gestion à deux niveaux de cette limite de message. Au niveau de l'application, les messages en deçà de la limite sont chargés depuis la DB. Au niveau de la base de données, chaque jour les messages excédant la limite sont purgés. Cette opération ne s'effectue pas à chaque saisie de message car cela demanderait trop de ressources si nous gardons dans l'optique que cette application peut être réutilisée à grande échelle. Il est donc de notre devoir de développer des outils évolutifs. Cette gestion à deux niveaux est transparente pour l'utilisateur.

Attributs d'une salle

Toutes les salles sont caractérisées par ces mêmes attributs :

- leurs membres,
- leurs administrateurs,
- leur nom : c'est celui qui s'affiche dans l'interface principale ainsi que lors de la recherche d'une salle.
- leur logo : il s'agit d'une image qui sera affichée à côté de leur nom dans l'interface principale.
- le nombre de messages stockés : il s'agit de l'historique des messages conservés sur le serveur. Ces messages seront visibles par les utilisateurs lorsqu'ils se connectent, même s'ils n'étaient pas connectés au moment où les messages ont été écrits.

Ces attributs peuvent être modifiés par les administrateurs. La visibilité d'une salle ne peut pas être modifiée.

Mise en œuvre

Les salles sont stockées et manipulées par l'application de deux façons différentes :

1. En tant que lignes d'une table dans la base de données du côté serveur (cf. partie « base de données » pour plus d'informations).
2. En tant qu'objet *ModelRoom* au niveau du programme (cf. le fichier *modelChator.h* où la classe est définie pour plus d'informations sur les champs et méthodes de cette classe).

2.6.1 Module Room

Ce module gère les principales actions concernant les salles de discussion, c'est-à-dire la création de nouvelles salles, l'édition de salles déjà existantes, et l'adhésion d'un utilisateur à une salle. Toutes ces actions sont appelées depuis le module *Chat*, le module *Room* se chargeant de les implémenter. A noter que le départ d'un utilisateur d'une salle se fait directement depuis le module *Chat*, s'agissant d'une opération légère et concernant plus l'utilisateur que la salle.

Acteurs du module

Il s'agit d'une présentation succincte des fonctionnalités des principales classes de ce module. Pour plus de précisions sur l'implémentation, se référer aux commentaires et entêtes du code source.

2.6.1.1.1 ControllerRoom

Il s'agit du contrôleur du module. Il gère notamment les interactions entre le modèle (*ModeleChator*), les vues (*ViewRoom* et *ViewJoin*) et le serveur. Il est donc au centre de ce module.

Il est créé dans la fonction *main* de l'application, en même temps que les autres contrôleurs, et devient actif directement après qu'une demande de connexion ait été validée par le serveur.

Il connaît, par ses attributs privés :

- Ses vues : *ViewRoom* et *ViewJoin*.
- Le modèle : *ModeleChator*.
- Un objet de type « pointeur sur *ModelUser* », représentant l'utilisateur courant (connecté) qui a été paramétré lors de la connexion de ce dernier.
- Les différents objets en rapport avec la communication serveur.

- L'objet Cryptor, fournissant les fonctionnalités de sécurité (génération et chiffrement des clés).

2.6.1.1.2 ViewChat

Il s'agit de l'une des deux vues sur le module. Elle est lancée par le contrôleur ControllerRoom sur demande du module Chat. Elle fournit l'interface graphique pour la création et l'édition des salles. Son but principal est d'afficher les données composant une salle et de fournir une interface pour les modifier (édition).

2.6.1.1.3 ViewJoin

Il s'agit de la deuxième vue du module. Elle est lancée par le contrôleur ControllerRoom sur demande du module Chat. Elle fournit l'interface graphique pour l'affichage des salles existante et la demande d'adhésion à l'une de celles-ci.

2.6.2 Détails sur l'interface graphique

Contrairement aux autres modules, l'interface du module Room n'a pas été réalisée à l'aide de l'outil du générateur intégré à l'IDE Qt Creator. Toutefois, elle est également codée en Qt et, par conséquence, n'est pas différente des autres, si ce n'est que tout le code est contenu dans les fichiers .h et .cpp des vues et que le fichier .ui n'existe pas.

Les remarques techniques se trouvant dans la documentation du module Chat sont donc largement valides pour ce module et ne seront pas répétées ici.

Vues d'édition et de création de salle

Cette interface est globalement la même aussi bien lors de la création que de l'édition d'une salle. Seul le label de certains boutons et le titre de la fenêtre changent selon le cas.

On peut distinguer quatre parties différentes dans cette interface :

1. La partie « formulaire », où l'utilisateur peut entrer des informations (nom de la salle, nombres de messages conservés sur le serveur, logo) dans les champs appropriés. Lors de la création d'une salle tous ces champs sont vides. Dans le cas de l'édition, ces champs sont remplis par les informations actuelles de la salle, à l'exception du champ indiquant le chemin de l'image. En effet, celle-ci n'est peut-être pas située sur la machine de l'utilisateur modifiant la salle, ce champ n'est donc utilisé que si ce dernier souhaite modifier le logo.
2. La partie « membres » où l'administrateur peut ajouter les membres qu'il désire à la salle ou les enlever, ainsi que modifier les droits d'administration de la salle. Le champ de texte permet d'entrer le nom d'un utilisateur et le bouton « ajouter » permet de l'ajouter à la salle. La liste se situant en dessous permet d'afficher les membres actuels de la salle. Ceux ayant des droits d'administration sont affichés en gras. Les boutons « Enlever » et « Admin » se situant en dessous de la liste permettent d'effectuer des actions sur les membres préalablement sélectionnés dans la liste. Le bouton « Enlever » supprime un membre de la salle et le bouton « Admin » change les droits du membre (admin -> membre normal ou membre normal -> Admin). Lors de la création d'une salle l'utilisateur créant celle-ci est automatiquement ajouté en tant qu'administrateur de la salle. Lors de l'édition les membres sont visibles dans la liste avec leurs droits actuels.

3. Les cases à cocher permettent d'indiquer si une salle est publique ou si elle est privée. Dans le second cas, l'utilisateur peut choisir si la salle est visible par tous les utilisateurs ou non. Par défaut, une salle est publique, et privée visible si elle a été choisie privée. Cette partie est désactivée lors de l'édition d'une salle.
4. Le bouton « Annuler » en bas de la fenêtre permet d'annuler l'action en cours (la salle ne sera pas créée ou modifiée) et les boutons « Créer » ou « Modifier » permettent de valider la création ou la modification de la salle.

Vue d'adhésion

Cette interface est relativement simple. Elle est constituée d'un champ de texte permettant de rechercher une salle selon son nom, d'une liste de salles contenant toutes les salles dont l'utilisateur n'est pas membre qui sont soit publiques soit privées et visibles (celles-ci s'affichent en gras) et de deux boutons permettant soit d'annuler l'action, soit d'envoyer la demande d'adhésion à une salle préalablement sélectionnée dans la liste.

2.6.3 Procédure de création d'une salle

Le ControllerChat du module Chat indique au ControllerRoom du module Room lorsqu'un utilisateur souhaite créer une nouvelle salle en appelant la méthode `showViewRoom()` de ce dernier. Le ControllerRoom initialise alors une nouvelle vue (ViewRoom) l'affiche et ajoute l'utilisateur comme administrateur par défaut de la salle.

Celui-ci peut alors préciser les différents attributs de la salle.

Nom de la salle

Le premier champ de texte permet d'indiquer le nom de la salle (maximum de 32 caractères). L'objet `QLineEdit` permet de fixer la taille maximum du texte saisi.

Nombre maximum de messages

Le second champ permet d'indiquer le nombre de messages stockés sur le serveur. Il peut être compris entre 0 et 10 000. L'utilisateur peut directement entrer le nombre souhaité ou utiliser les flèches sur le côté pour incrémenter ou décrémenter ce nombre par intervalles de 50.

L'objet utilisé pour cette valeur est une `QSpinBox`, elle permet de s'assurer que le champ ne contienne que des chiffres et que le nombre ne dépasse pas les bornes fixées.

Ajout d'un membre dans la salle

Le champ de texte permet d'entrer le pseudonyme d'un utilisateur (maximum de 16 caractères).

Lors de l'appui du bouton d'ajout, une requête est envoyée au serveur pour vérifier si le pseudonyme est déjà utilisé. Si c'est le cas, une fenêtre jaillissante en informe l'utilisateur. Sinon l'utilisateur est ajouté à la liste située en dessous.

Par défaut les utilisateurs ajoutés ne disposent pas des droits d'administrateur sur la salle. Un utilisateur ne sera ajouté qu'une seule fois à la salle. En effet, la vue garde les utilisateurs de la salle dans une *QMap* (ce n'est pas le modèle qui stocke cette liste car elle n'est que temporaire. La salle n'existant pas encore, il serait trop lourd de communiquer ces données à travers le serveur, surtout si la salle n'est finalement pas créée) avec comme clé leur identifiant et comme valeur leurs clés. Une seule valeur par clé étant possible il ne peut y avoir plusieurs fois le même utilisateur.

Logo de la salle

Le champ de texte permet d'indiquer le chemin de l'image désirée comme logo de la salle (maximum de 256 caractères).

Changement des droits d'un utilisateur

Lors de l'appui du bouton "Admin" de la vue, si un utilisateur a été sélectionné dans la liste (un seul utilisateur peut être sélectionné à la fois), ses droits sont modifiés. S'il était administrateur de la salle il devient un membre normal, et vice-versa.

Un administrateur est affiché en gras et les autres membres normalement. En interne, la vue dispose de la liste d'utilisateurs de la salle ainsi que celle des administrateurs.

Suppression d'un utilisateur de la salle

Il ne s'agit bien évidemment pas d'une suppression physique de la personne mais de la suppression de l'utilisateur de la salle en cours de création.

Lors de l'appui du bouton "Enlever", l'utilisateur sélectionné (s'il y en a un) est enlevé de la liste des utilisateurs. Il ne sera donc pas ajouté à la salle lors de sa création.

Choix de la visibilité de la salle

La case à cocher "Salle privée" permet à l'utilisateur d'indiquer qu'il souhaite créer une salle privée. Par défaut la case est décochée.

Les deux boutons radio sont alors activés et permettent à l'utilisateur de spécifier s'il désire que la salle privée soit visible par tous ou seulement joignable sur invitation. Les deux boutons sont mutuellement exclusifs, un seul peut être sélectionné. Par défaut une salle privée sera visible.

Annulation de la création de la salle

L'utilisateur peut à tout moment choisir d'annuler la création en cours en appuyant sur le bouton "Annuler" de la vue, en appuyant sur le bouton en forme de croix de la fenêtre, ou en utilisant le raccourci "Alt- + F4". La vue sera alors fermée et détruite. La salle ne sera bien entendu pas créée.

Création de la salle

Le bouton "Créer" permet à l'utilisateur de créer la salle. Le contrôleur récupère alors les valeurs de la vue et vérifie que la salle soit valide. Il vérifie en particulier :

- que la salle ait un nom,
- que si un chemin a été précisé pour le logo il mène à une image valide,
- que la salle contienne au moins un utilisateur et au moins un administrateur.

Si une de ces conditions n'est pas respectée, le contrôleur en informe l'utilisateur grâce à un message d'information à l'intérieur d'une fenêtre jaillissante.

Si le champ du logo de la salle est laissé vide par l'utilisateur, une image vide est créée et le serveur lui attribuera le logo par défaut.

Si toutes les informations sont valides, le contrôleur crée un objet de type *ModelRoom*, avec un identifiant valant '0' et une clé vide.

Si la salle est privée, le contrôleur génère une clé AES pour le chiffrement des messages spécifiques à la salle. Il transmet ensuite une demande au serveur pour obtenir les clés publiques de tous les utilisateurs de la salle, en lui indiquant leurs identifiants. Le serveur transmet alors les clés publiques au contrôleur, qui se charge de chiffrer la clé AES à l'aide des différentes clés publiques (une fois pour chaque utilisateur).

Le *ModelRoom* est ensuite transmis au serveur ainsi qu'une *QMap* contenant les identifiants et la clé chiffrée correspondante de tous les utilisateurs, ainsi qu'un booléen indiquant qu'il s'agit d'une nouvelle salle.

Le serveur traite alors le paquet. Il stocke la nouvelle salle dans la base de données et ajoute la clé de la salle, chiffrée, dans les trousseaux respectifs des utilisateurs. Il transmet alors aux utilisateurs connectés la nouvelle salle. Le module Chat se charge de la réception de la nouvelle salle.

2.6.4 Procédure d'édition de la salle

Le ControllerChat du module Chat indique au ControllerRoom du module Room lorsqu'un utilisateur souhaite éditer salle en appelant la méthode *showViewRoom(roomId)* de ce dernier. Le ControllerRoom initialise alors une nouvelle vue (ViewRoom), l'affiche et remplit les différents champs de la vue avec les valeurs de la salle. Il conserve également l'identifiant de salle en mémoire. L'utilisateur peut alors modifier les différents attributs de la salle.

Globalement, la procédure reste la même que pour la création de la salle. Ne seront donc précisées ici que les différences par rapport à cette dernière procédure.

Choix de la visibilité de la salle

Il n'est pas possible de modifier la visibilité d'une salle une fois celle-ci créée. Les boutons concernant ces actions sont donc désactivés dans la vue lors de l'édition de salle.

Annulation de la création de la salle

L'utilisateur peut à tout moment choisir d'annuler l'édition en cours en appuyant sur le bouton "Annuler" de la vue, en appuyant sur le bouton en forme de croix de la fenêtre, ou en utilisant le raccourci "Alt- + F4". La vue sera alors fermée et détruite. La salle ne sera donc pas modifiée.

Edition de la salle

Le bouton "Editer" permet à l'utilisateur de valider les modifications faites à la salle. Le contrôleur procède aux mêmes vérifications que lors de la création.

Un nouvel objet *ModelRoom* est alors créé. Toutefois, l'identifiant de la salle reste le même que celui de l'originale. Si la salle est privée l'ancienne clé est également conservée.

Si le champ du logo est laissé vide, le logo de la salle ne sera pas modifié.

Le contrôleur envoie ensuite un paquet contenant le *ModelRoom* et un booléen indiquant qu'il s'agit d'une salle modifiée. Le serveur se charge ensuite d'enregistrer les modifications dans la base de données et de notifier les utilisateurs connectés des changements de la salle.

2.6.5 Procédure d'adhésion à une salle

Le ControllerChat du module Chat indique au ControllerRoom du module Room lorsqu'un utilisateur souhaite adhérer à une salle en appelant la méthode *showViewJoin()* de ce dernier. Le ControllerRoom initialise alors une nouvelle vue (ViewJoin).

Avant d'afficher cette vue, le contrôleur demande au serveur de lui envoyer la liste des salles publiques et privées visibles (leur identifiant et leur nom) dans deux listes séparées. Une fois la réponse du serveur reçue, la vue est affichée avec la liste de toutes les salles.

Recherche d'une salle

Il est possible d'utiliser le champ de texte de la vue ViewJoin pour rechercher une salle spécifique par son nom. Lorsque l'utilisateur modifie ce champ la vue filtre les salles en n'affichant que celles contenant la chaîne entrée par l'utilisateur (la casse est ignorée).

Annuler

L'utilisateur peut à tout moment décider de ne pas rejoindre de salle en appuyant sur le bouton "Annuler" de la vue, en appuyant sur le bouton en forme de croix de la fenêtre, ou en utilisant le raccourci "Alt- + F4". La vue est alors fermée et détruite et aucune demande d'adhésion n'est envoyée au serveur.

Adhésion à une salle

Une fois que l'utilisateur a trouvé la salle qu'il désire, il peut la sélectionner dans la liste et appuyer sur le bouton "Rejoindre" de la vue. Le contrôleur envoie alors un paquet précisant l'identifiant de la salle au serveur. Celui-ci se charge alors d'ajouter l'utilisateur à la salle si elle est publique ou d'envoyer une notification aux administrateurs de la salle si elle est privée. Le module Chat se chargera ensuite de gérer les demandes d'adhésion.

2.6.6 Processus de connexion au serveur

Lorsque l'utilisateur lance l'application, il doit saisir manuellement l'IP du serveur ainsi qu'un numéro de port. Il peut ensuite entrer ses identifiants (nom d'utilisateur et mot de passe) ou se créer un compte s'il n'en possède pas, puis se connecter.

2.6.7 Fonctionnalités utilisateur

Une fois membre d'une salle de discussion, un utilisateur peut y envoyer un message. Lorsqu'un message a été envoyé, il est toujours possible de le modifier ou de le supprimer. Un utilisateur peut être membre de plusieurs salles. Il peut voir les autres membres des salles auxquelles il appartient. Il est avisé des nouveaux messages arrivant dans une salle autre que celle qu'il consulte sur le moment. Il peut décider d'afficher ou non l'heure d'arrivée des messages. Il a la possibilité de modifier les informations relatives à son compte.

Lors de la connexion d'un utilisateur, l'application cliente récupère les messages envoyés durant son absence dans les différentes salles dont il est membre.

2.7 SÉCURITÉ

Il existe plusieurs aspects sécuritaires dans l'application Chator, mais ils peuvent se résumer sur trois points principaux, desquels découle le reste :

- la connexion client-serveur,
- l'identification des utilisateurs,
- le chiffrement des messages dans les salles privées.

Connexion client-serveur

La connexion entre un client et un serveur doit se faire de manière sécurisée. Il est important que le serveur auquel le client se connecte soit bien identifié et que les messages échangés entre le client et le serveur ne soient pas lisibles pour une tierce partie indésirable.

Afin d'assurer cette sécurité, le protocole TLS (Transport Layer Security) est utilisé. Le serveur doit notamment s'identifier à l'aide d'un certificat. Un certificat a été généré dans le cadre du projet, mais il n'est pas certifié par une organisation et ne sera donc pas reconnu.

Identification des utilisateurs

Une fois un utilisateur inscrit, il est important qu'il puisse se connecter à nouveau depuis n'importe quelle autre machine où Chator est installé. Il est également important que l'application dispose d'un moyen de l'identifier afin d'éviter un cas d'usurpation d'identité au sein de l'application (une personne se connectant en tant qu'une autre et ayant ainsi potentiellement accès aux messages des salles privées dont elle n'est pas réellement membre).

Afin de pouvoir identifier l'utilisateur, un système de mot de passe avec hachage et sel est utilisé.

Les mots de passe

Lors de l'inscription, l'utilisateur doit obligatoirement entrer un mot de passe. Ce dernier lui sera demandé lors de la connexion afin de vérifier son identité. Comme le mot de passe est un élément sensible de la sécurité de l'application, il doit répondre à certains critères de robustesse (cf. documentation sur l'inscription).

Le hachage

Même si la connexion entre les clients et le serveur utilise TLS et que les paquets échangés ne sont donc théoriquement pas lisibles par une tierce personne, il est préférable de ne pas échanger directement le mot de passe en clair entre les deux. En effet, il est toujours possible d'intercepter ces paquets et une personne ayant accès à la base de données aurait également accès aux mots de passe. Afin d'éviter ces problèmes, le mot de passe est haché avant d'être envoyé au serveur.

Afin d'éviter des attaques par tables, le hachage utilise un sel généré aléatoirement. Ce sel est également transmis au serveur lors de l'inscription, et le serveur le retransmet à l'utilisateur lors de la connexion.

Plutôt que de simplement utiliser une fonction de hachage classique, une fonction de dérivation de clé est utilisée, spécifiquement PBKDF2 qui est l'une des méthodes recommandée par l'OWASP pour le stockage des mots de passe. L'algorithme de hachage utilisé est SHA-512.

Chiffrement des messages

L'application offre la possibilité de chiffrer les messages échangés au travers des salles privées. Les messages doivent donc être chiffrés durant leur transit sur le réseau, mais également lorsqu'ils sont manipulés et stockés par le serveur. Une personne ne faisant pas partie d'une salle privée ne doit pas pouvoir lire les messages à l'intérieur de celle-ci, pas même l'administrateur du serveur qui aurait accès à la base de données.

AES a été choisi comme moyen de chiffrer les messages. Il s'agit en effet de l'un des standards les plus utilisés et les dernières générations de processeurs disposent d'instructions spécifiques à AES assurant un chiffrement et déchiffrement rapide des données. Afin de pouvoir effectuer ce chiffrement, plusieurs étapes sont nécessaires.

2.7.1.1.1 Génération des clés AES

Lors de la création d'une salle privée, une clé AES est générée aléatoirement par le client. C'est cette clé qui sera utilisée pour chiffrer et déchiffrer les messages (clé symétrique) de la salle.

2.7.1.1.2 Partage des clés AES

Afin que les différents membres puissent lire les messages d'une salle privée, il faut pouvoir leur partager sa clé. Il est toutefois préférable que la clé ne transite pas sur le réseau et par le serveur telle quelle. Un mécanisme doit donc permettre de partager ces clés de manière sûre : il s'agira de chiffrer la clé elle-même avant de la partager.

Un mécanisme utilisant des clés asymétriques a donc été choisi pour cette tâche, dans ce cas RSA. Chaque membre dispose d'une paire de clé RSA (une clé privée et une clé publique).

Lorsqu'un utilisateur souhaite partager la clé d'une salle avec un autre utilisateur, il utilise la clé publique de ce dernier pour chiffrer la clé de la salle et la lui transmettre à travers le serveur. L'utilisateur est ensuite à même de pouvoir déchiffrer la clé de la salle à l'aide de sa clé privée.

2.7.1.1.3 Génération des clés RSA

Il est nécessaire que les utilisateurs possèdent une paire de clés RSA afin de pouvoir se partager les clés des salles privées. Lorsqu'un nouvel utilisateur s'inscrit, cette paire de clés est automatiquement générée de manière aléatoire.

2.7.1.1.4 Stockage des clés

Il est nécessaire que les utilisateurs puissent se connecter et lire les messages des salles privées dont ils font parties à tout moment. Pour cela, ils doivent pouvoir récupérer les clés de ces salles. Deux alternatives existent : soit stocker les clés chez le client, soit stocker les clés sur le serveur. La seconde solution a été préférée car elle est plus flexible pour le client. En effet, il ne risque ainsi pas de perdre les clés et il peut se connecter depuis n'importe quelle machine où Chator est installé.

Il serait toutefois dommageable, dans la logique de notre approche sécuritaire, de stocker les clés en clair sur le serveur. La solution choisie a été de stocker, pour chaque utilisateur, un "trousseau" de clés. Ces clés sont les clés des salles privées dont il est membre. Elles sont stockées sous la même forme chiffrée que lors du partage de ces clés, c'est-à-dire que les clés sont chiffrées à l'aide de la clé publique de l'utilisateur.

La dernière problématique concerne la clé privée de l'utilisateur. Si sa clé publique doit être stockée en clair, afin d'être transmise à n'importe qui, sa clé privée doit, elle, rester secrète. Il est donc souhaitable de la conserver, elle aussi, sous forme chiffrée.

Pour ce faire la solution suivante a été choisie. Pour chiffrer la clé privée, il est nécessaire d'utiliser une clé. Une clé symétrique semble idéale. Il faut toutefois que cette clé reste également secrète. Il a donc été décidé de générer une clé AES à l'aide du mot de passe de l'utilisateur.

En effet, le hash du mot de passe peut être utilisé comme une clé AES (il ne s'agit après tout que d'une suite de bits). On ne peut toutefois pas utiliser celui qui est stocké sur le serveur, mais l'on peut générer un hash différent à partir du mot de passe en utilisant un sel différent.

Ce sel est généré aléatoirement à l'inscription de l'utilisateur et transmis au serveur. Lorsqu'un client se connecte il reçoit sa paire de clés RSA, dont la clé privée chiffrée ainsi que le sel. Il est ensuite à même de générer le hash à partir de son mot de passe et du sel et d'utiliser le hash comme clé pour déchiffrer la clé privée.

2.7.2 Mise en œuvre dans l'application

Au niveau de l'application, les aspects sécuritaires sont gérés de deux façon différentes :

- La connexion client-serveur (TLS) est gérée à l'aide d'objets Qt. (*QSslConfiguration* et *QWebSockets*).
- La génération des clés, leur chiffrement, celui des messages, ainsi que leurs déchiffrements respectifs, et la génération des sels et hashes est assuré par un objet *Cryptor* défini dans le fichier *cryptor.h*. Cet objet utilise la librairie *OpenSSL*.

2.7.3 Résumé de la mise en œuvre de la sécurité

Inscription

Le nouvel utilisateur doit fournir au serveur toutes les données nécessaires. En plus des informations utilisateurs (obligatoirement un pseudonyme), il doit fournir le hash de son mot de passe, le sel qui a servi à générer ce hash, une paire de clés RSA (la clé privée étant chiffrée) et le sel ayant servi à générer la clé utilisée pour chiffrer la clé privée.

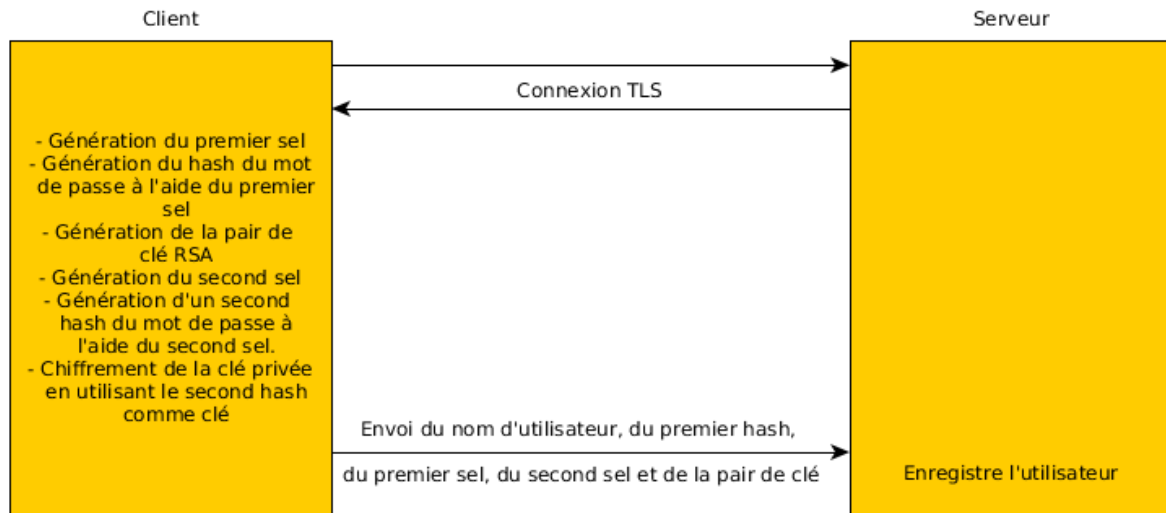


Figure 32: Inscription

Connexion

Une fois la connexion avec le serveur établie, le client transmet le nom (pseudonyme) de l'utilisateur au serveur. Celui-ci retourne le sel servant à générer le hash utilisé pour l'authentification. Le client génère le hash et le transmet au serveur. Celui-ci le vérifie et transmet les données utilisateurs s'il est correct. Le client déchiffre ensuite sa clé privée en générant la clé avec le second sel.

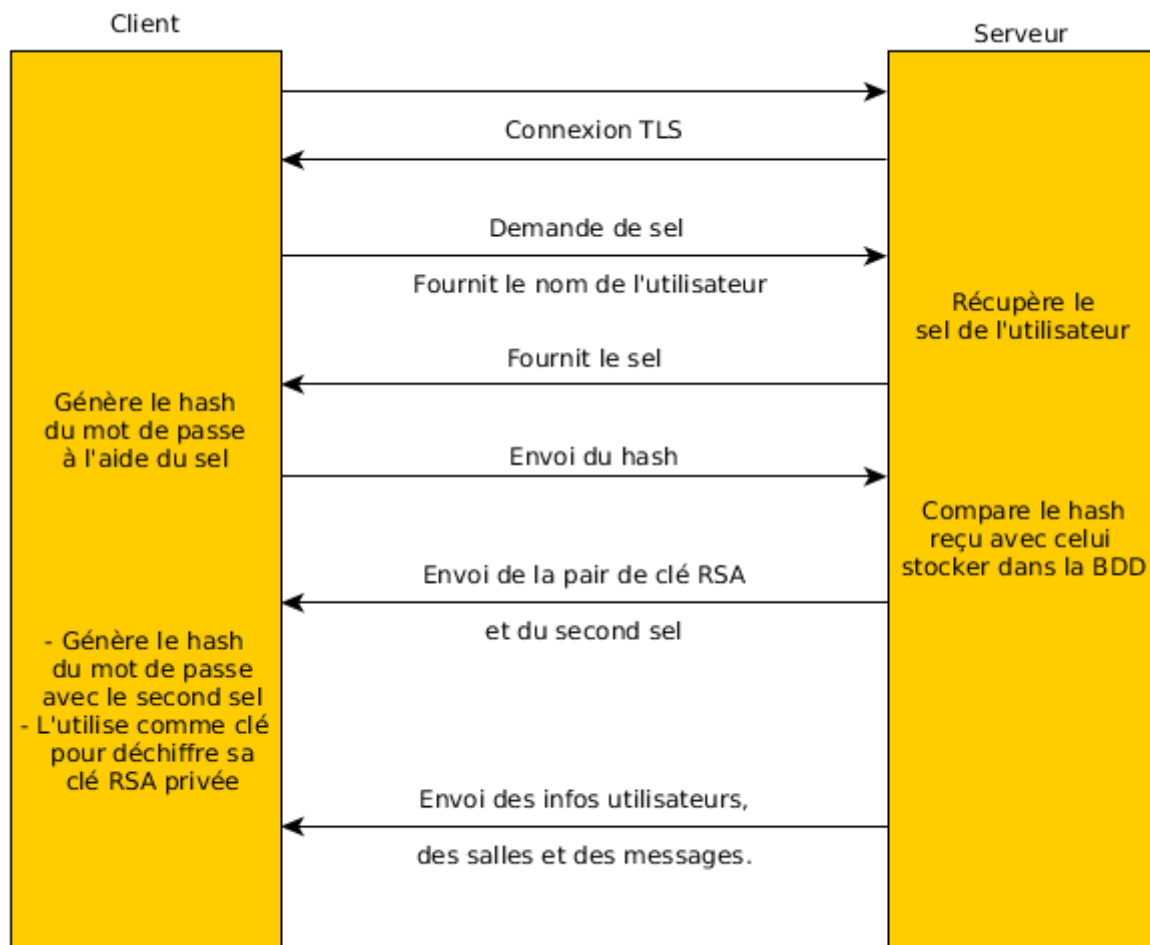


Figure 33: Connexion

Partage de clés

Un utilisateur souhaitant rejoindre une salle privée envoie une demande au serveur. Celui-ci transmet la demande à un administrateur de la salle. Si celui-ci accepte la demande, il chiffre la clé de la salle à l'aide de la clé publique du demandeur (qui lui a été fournie avec la requête). Il transmet ensuite la clé au serveur qui la stocke dans le trousseau du demandeur. La clé est ensuite transmise avec le reste des données de la salle au demandeur, immédiatement s'il est encore connecté et lors des connexions ultérieures.

Dans le cas où une personne créant ou éditant une salle privée souhaite directement ajouter un utilisateur (c'est d'ailleurs le seul moyen d'ajouter un utilisateur à une salle privée non-visible), la procédure reste semblable. Toutefois comme il n'y a pas de demande c'est au client de demander la clé publique de l'utilisateur au serveur. Cette demande est faite globalement pour tous les membres de la salle au moment de la création de celle-ci.

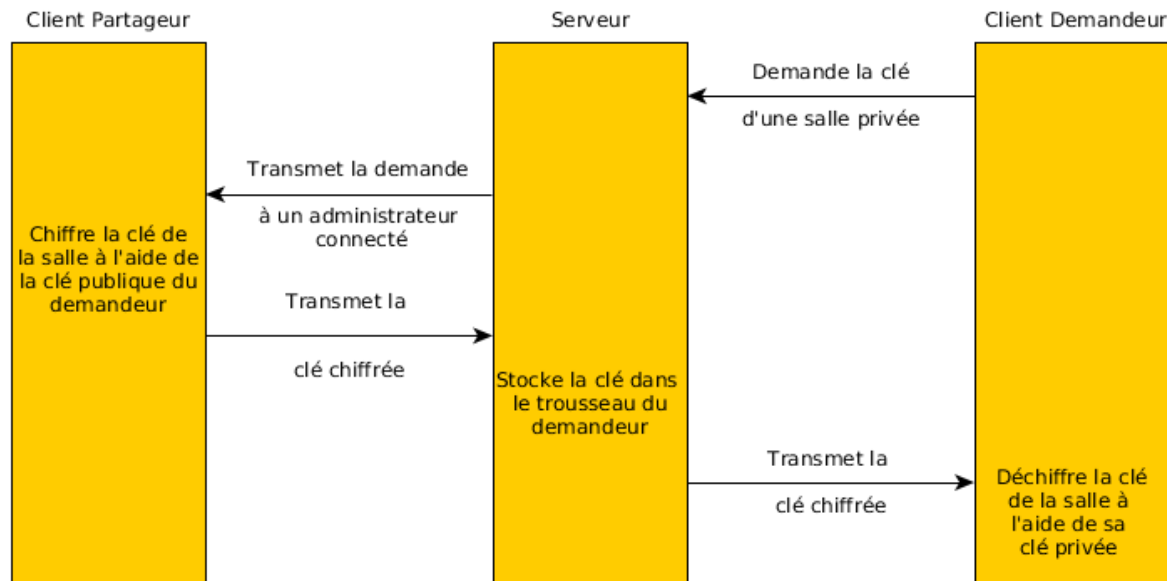


Figure 34: Partage des clés

Envoi de message

Lorsqu'un utilisateur envoie un message dans une salle privée dont il est membre, le client utilise la clé de la salle pour chiffrer le contenu du message et transmet ensuite le message au serveur. Celui-ci le stocke dans la base de données et le transmet à tous les utilisateurs connectés, y compris à celui qui a envoyé le message.

À la réception du message, les clients déchiffrent le contenu du message et le stockent dans le modèle de l'application. S'il y a lieu de le faire, il sera affiché dans la vue du chat.

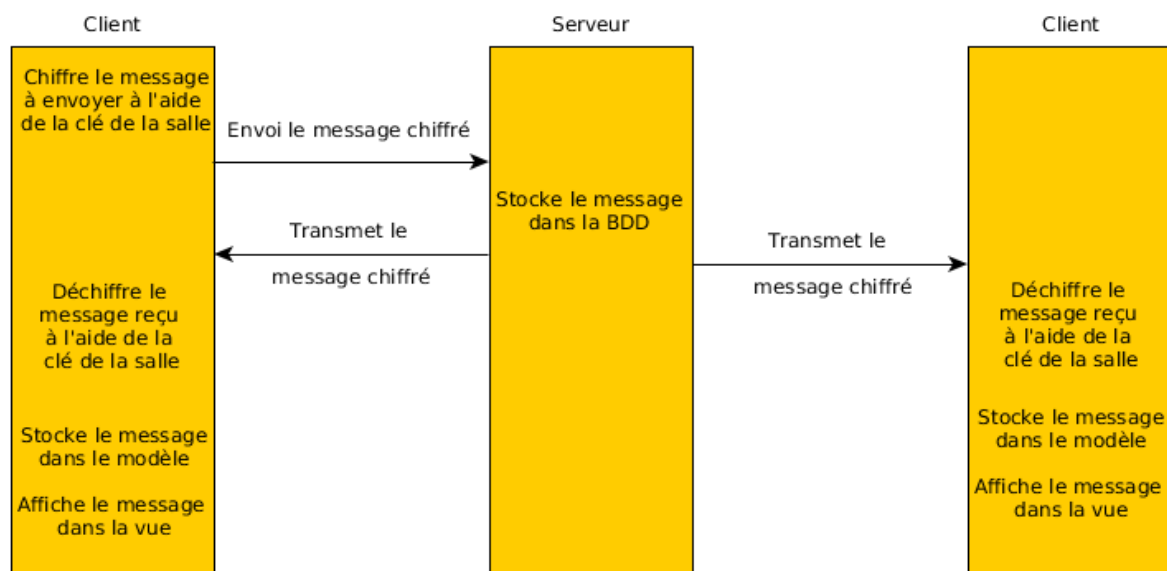


Figure 35: Envoi d'un message

2.8 SERVEUR

Aucune interface graphique n'est réalisée pour le serveur. En effet, une simple console d'administration suffit amplement pour les besoins de l'application. Lors du lancement du serveur, la personne responsable devra éventuellement indiquer certains arguments, comme le numéro de port, ou le nombre de connexions maximum par exemple.

Certains logs sont affichés en temps réel (inscription d'un utilisateur, création d'une salle, ...), afin de pouvoir debugger facilement l'application, et d'avoir un aperçu de l'activité à chaque instant.

2.8.1 Architecture

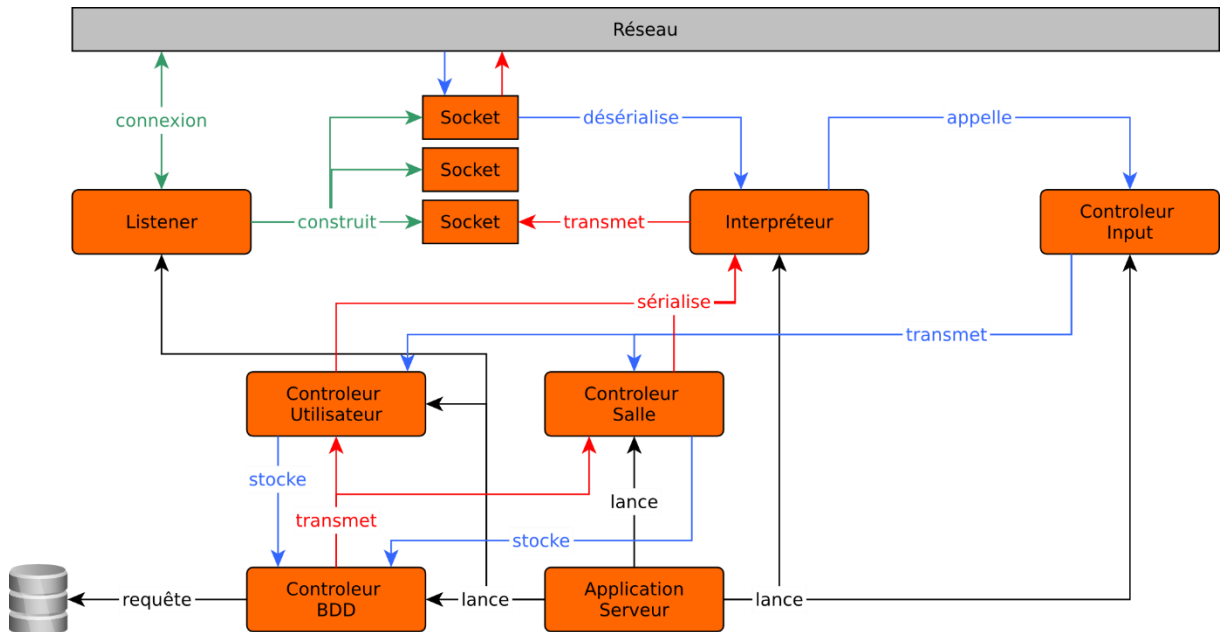
Notre projet étant une application réseau, nous avons deux choix d'infrastructure possible :

1. Architecture décentralisée : aussi appelé pair-à-pair (P2P) ce mode implique que tous les clients soient connectés les uns aux autres. Le système ainsi obtenu a l'avantage d'être très robuste et tolérant aux pannes (si une ou plusieurs machines se déconnectent, cela n'impacte en rien la fiabilité de transmission dans le reste du réseau) mais a le désavantage d'être assez complexe et lourd à mettre en place car il nécessite des mécanismes de découverte automatique des pairs et d'ouverture des ports sur NAT.
2. Architecture centralisée : il y a deux programmes différents, soit un serveur et un client. Les clients se connectent tous à un serveur et ce dernier fera office de relai entre tous les clients (qui ne sont pas connectés les uns aux autres). Cette solution a l'avantage d'être très simple à mettre en place car seul le serveur nécessite d'être accessible (donc lui-seul nécessite une configuration spéciale des ports à ouvrir sur les NAT des routeurs) mais a le désavantage d'être sensible aux pannes car le serveur est un nœud principal et s'il tombe, le système ne fonctionne plus. De plus, le serveur doit être capable de prendre en charge les requêtes de tous les clients, ce qui représente une charge considérable tant au niveau calcul qu'au niveau du réseau et doit donc être lancé sur une machine puissante dans une infrastructure réseau performante.

Malgré les désavantages de la seconde solution, c'est celle-ci que nous avons décidé d'utiliser car la découverte automatique de tous les pairs et la mise en place automatique des règles de routage est une tâche que nous savons délicate et préférons partir du principe que seul le serveur est critique et nécessite une configuration réseau spécifique.

2.8.2 Interactions

Le serveur de notre projet comporte un ensemble de classes et de sous-systèmes détaillés ici :



Flèches noires: représente l'ordre de construction des objets au lancement de l'application

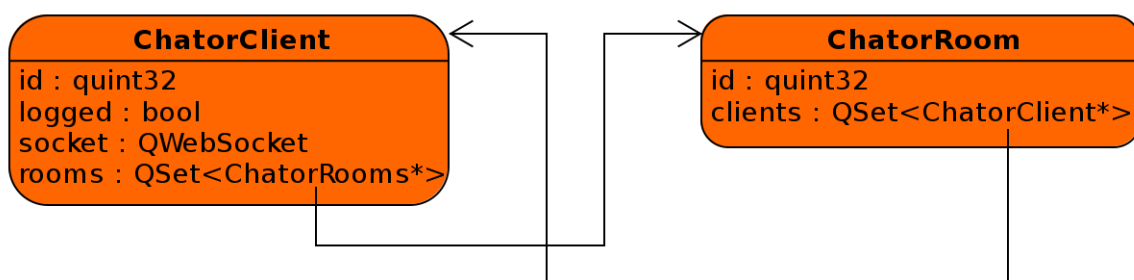
Flèches vertes: représente une construction dynamique d'objets pendant l'exécution

Flèches bleues : représente le cheminement d'une donnée qui arrive du réseau

Flèches rouges: représente le cheminement d'une donnée provenant de la base de données ou résultant d'une opération sur une requête reçue

2.8.3 Objets supplémentaires

Deux classes supplémentaires sont utilisées uniquement côté serveur et servent à gérer les utilisateurs et les salles qui sont en ligne : ChatorClient et ChatorRoom



ChatorClient

Cette structure permet d'encapsuler le socket permettant de joindre un utilisateur et de stocker des informations à propos du client utilisant ce socket :

Son numéro d'identifiant dans la base de données est stocké afin de rapidement pouvoir effectuer des requêtes dans la base de données le concernant

Un booléen indiquant s'il a correctement effectué sa connexion. En effet, il arrive des situations où un client est connecté mais n'a pas effectué le login, par exemple à l'ouverture de l'application cliente.

Une liste de pointeurs vers les salles auxquelles cet utilisateur est associé.

ChatorRoom

Cette structure représente conceptuellement une salle en ligne. Dès qu'un utilisateur se connecte, toutes les salles auxquelles il est associé contiennent au moins un utilisateur et sont donc "en ligne", cette classe est alors instanciée pour chacune de ces salles si elles n'existaient pas avant.

Cette structure contient l'identifiant de la salle qu'elle représente ainsi qu'une liste de pointeurs vers les clients qui y sont inscrits et qui sont en ligne. Dès lors, envoyer un message à chacun de ces clients est très aisé car il suffit de parcourir cette liste et d'envoyer le paquet désiré dans le socket de chacun des clients.

Nous avons choisi de concevoir ChatorClient et ChatorRoom en tant que struct car nous n'avons pas besoin d'encapsuler les données internes de manière protégée, le serveur est seul responsable des données qui s'y trouvent et cela simplifie leur utilisation.

2.8.4 Architecture

Listener : Cette classe hérite de la classe QWebSocketServer. Cette classe permet de créer un serveur TCP qui écoute le réseau, comme le ferait la classe QtcpServer, mais gère de plus nativemeent SSL/TLS pour ses connexions, ce qui nous simplifie grandement la vie.

Pour sécuriser les échanges du côté serveur, notre application nécessite un certificat SSL/TLS au format standard X.509. Il est possible d'acheter un tel certificat auprès d'un organisme de certification, ce qui permet de ne pas provoquer d'erreurs de signature. Cependant, pour notre projet, nous utilisons un certificat que nous avons généré nous-même en utilisant openssl. Notre certificat a été généré avec la commande suivante.

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -nodes -days 365
```

Afin de ne pas être obligés de lancé le serveur en tant qu'administrateur, il nous faut utiliser un port supérieur à 1023. Nous avons opté pour le port 1234, facile à retenir.

Cette classe est donc instanciée au début de l'exécution du serveur et tente d'ouvrir le certificat et la clé à utiliser pour le chiffrement (dont les emplacements sont définis dans le fichier chatorConstants.h). Si un de ces fichiers n'est pas trouvé ou est illisible, c'est considéré comme une erreur critique et le serveur se termine avec une erreur.

Lors de l'exécution du serveur, le seul travail de cette classe est d'écouter le port et de recevoir les connexions de nouveaux clients. Lorsqu'un nouveau client se connecte, il est immédiatement encapsulé dans une structure ChatorClient et stocké dans une liste. D'autre part, les événements du socket sont traités comme suit :

Chaque paquet qui arrive dans un socket est automatiquement transféré à l'interpréteur

Lorsque le socket se déconnecte, sa déconnexion est gérée par le contrôleur d'utilisateurs et il est enfin détruit

Interpréteur

Appelée *Interpreter* dans notre application, cette classe est très particulière et centrale à notre application car c'est elle qui fait le lien et établit la compatibilité entre le client et le serveur :

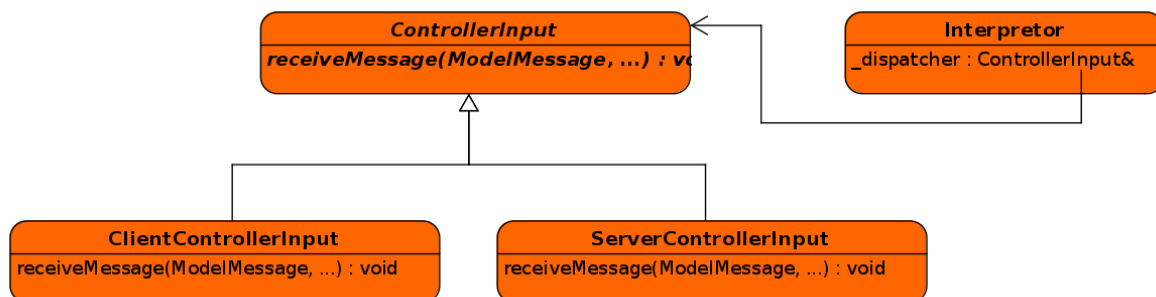
À l'envoi d'un paquet dans le socket, c'est cette classe qui va placer un identifiant permettant, à la désérialisation, de connaître le type du packet, et de sérialiser les données utiles dans tableau d'octets (un `QByteArray`). Ce tableau d'octets représente le contenu du paquet qui peut directement être envoyé dans un socket car dans la bibliothèque Qt, `QWebSocket` possède la méthode *sendBinaryMessage* qui prend directement en paramètre un `QByteArray`.

À la réception d'un paquet, on sait que la donnée reçue sous forme de `QByteArray` a été créée par un autre *Interpreter* qui a placé l'identifiant du paquet au début. Cet identifiant, qui est un simple *enum* nous permet de faire un grand *switch* qui, pour chaque type de paquet, désérialise les bons éléments et les transmet au contrôleur des données entrantes qui va se charger de les passer plus loin

Interpreter est la classe charnière de notre application et nous avons décidé de la partager entre le client et le serveur. En effet, c'est exactement la même classe qui est instanciée des deux côtés de la communication. Cependant, la structure et les actions qui sont à effectuer à l'arrivée de paquets sont singulièrement différentes lorsqu'il s'agit du client ou du serveur. C'est donc le travail du contrôleur des données entrantes de faire "l'aiguillage" afin d'envoyer les données au bon endroit.

Contrôleur des données entrantes

Il ne s'agit pas d'une classe, mais d'une structure de trois classes permettant d'aiguiller les données reçues dans les bons contrôleurs qui sont différents du côté client et du côté serveur.



Ces classes jouent le rôle d'adaptateurs, là réside la subtilité du processus : en effet, l'interpréteur ne sait pas s'il est instancié du côté client ou du côté serveur, il va donc transmettre les données reçues à ce contrôleur qu'il connaît en tant que la classe abstraite *ControllerInput*. Cette classe abstraite définit une interface que l'interpréteur peut utiliser. Les deux sous-classes *ClientControllerInput* et *ServerControllerInput* sont des classes concrètes qui héritent de *ControllerInput* et implémentent son interface.

Dès lors, lorsqu'un message est reçu par exemple, l'interpréteur passe simplement à la méthode *receiveMessage* définie dans l'interface *ControllerInput* qui, du côté serveur, respectivement du côté client, est en fait un *ServerControllerInput* ou un *ClientControllerInput*.

Le polymorphisme et la liaison dynamique faisant alors leur office, les actions qui sont effectuées dans leur propre implémentation de la méthode *receiveMessage* sont bien différentes.

Contrôleur des utilisateurs

Avec le contrôleur des salles, le contrôleur des utilisateur, appelé `ControllerUser` s'occupe de toute la logique métier de l'application serveur. Possédant une référence sur le contrôleur de base de données, ce contrôleur va s'occuper des tâches spécifiques aux utilisateurs comme :

- Les opérations relatives au login comme la récupération du sel et le login à proprement parler
- Les opérations relatives à l'inscription d'un nouvel utilisateur
- La récupération des informations concernant des utilisateurs (leur clé publique par exemple)
- Les opérations de déconnexion

Contrôleur des salles

Appelé `controllerRoom` dans l'application, ce contrôleur gère tout ce qui est relatif aux salles :

- Création de salles
- Modification de salles
- Suppression de salles
- Traitement des requêtes d'abonnement à une salle
- Réception des messages
- Modification des messages

Contrôleur de base de données

Appelé `controllerDB`, ce contrôleur gère les requêtes à la base de données. Ce contrôleur n'est finalement qu'une abstraction des données contenues dans la base de données SQL et ne contient que très peu d'intelligence fonctionnelle.

À l'instar du `Listener`, au démarrage du programme, le contrôleur de base de données va essayer d'ouvrir le fichier de base de données SQLite ou d'en créer un nouveau s'il n'existe pas. Si cette opération échoue, le serveur se termine en affichant une erreur. En effet, la base de données est un élément critique de notre conception et il n'est pas possible de lancer un serveur si les données ne peuvent être stockées.

Le fait d'avoir créé une classe spécialement pour abstraire le stockage et la récupération de données dans une base de données relationnelle est assez extensible. En effet, actuellement, seules les bases de données de type SQLite sont prises en charge dans notre programme, mais Qt gérant nativement d'autres types de bases de données comme IBM DB2, Borland InterBase, MySQL, Oracle, ODBC, PostgreSQL ou encore Sybase il serait tout à fait facile d'adapter cette classe pour supporter ces autres variantes.

Application principale

Ce bloc n'est en fait que la représentation du *main* du logiciel serveur. Son seul travail est d'initialiser la bibliothèque Qt et de créer les différents contrôleurs dans le bon ordre en passant les bons paramètres.

2.8.5 Protocole

Pour que le client et le serveur puissent se communiquer et se comprendre, nous avons dû établir un protocole. Bien sûr, nous aurions pu utiliser une bibliothèque qui implémente un protocole de messagerie déjà existant comme Jabber ou IRC par exemple, cependant nous avons préféré partir de zéro et créer nous-même un protocole. C'est effectivement un très bon exercice de devoir réfléchir en terme de paquets et d'état de la connexion.

Pour la sérialisation des différents éléments, nous avons utilisé le système mis en place par Qt. En effet, la plupart des classes de base dont nous nous servons peuvent directement sérialisées en utilisant l'opérateur d'écriture sur le flux <<. Cet opérateur est défini pour les flux spécifiques à Qt, soit `QDataStream`. Pour les types que nous avons nous-même définis (soit `ModelMessage`, `ModelRoom`, `ModelUser`, les types relatifs au module de chiffrement, etc ...) nous avons nous-aussi défini l'opérateur d'écriture dans un `QDataStream`. Ainsi, nous parvenons à sérialiser presque tous nos éléments de manière transparente.

Afin de ne pas surcharger notre protocole qui est déjà relativement complexe, certains paquets possèdent des paramètres qui peuvent être vides dans un sens ou dans l'autre (client -> serveur ou serveur -> client).

Par exemple, pour demander si un utilisateur existe, le client envoie la requête en spécifiant le pseudonyme concerné et le serveur répond par un booléen et un entier qui représente l'identifiant de l'utilisateur, si existant. Lorsque le client fait sa requête, le booléen n'est évidemment pas utilisé côté serveur, encore moins l'entier car il s'agit des champs de réponse.

Les paquets qui existent dans notre protocole sont les suivants :

Nom du paquet	EDIT_ACCOUNT
Paramètres	<code>ModelUser user</code> , <code>QByteArray password</code> , <code>QByteArray privateKey</code>
Client -> Serveur	Ce paquet est envoyé au serveur lorsque l'utilisateur souhaite modifier son profil. Le paramètre <i>user</i> spécifie les nouvelles informations à mettre à jour dans la base de données. Le champ <i>password</i> contient le hash du nouveau mot de passe de l'utilisateur. Si l'utilisateur change son mot de passe, sa clé privée doit être chiffrée en utilisant ce nouveau mot de passe, le dernier champ permet de stocker la nouvelle version de cette clé RSA privée.
Serveur -> Client	Le serveur signale à tous les autres utilisateurs qui connaissent l'utilisateur concerné qu'il a modifié son profil et leur transmet ses nouvelles informations. Les champs <i>password</i> et <i>privateKey</i> ne sont évidemment pas utilisés.

Nom du paquet	MESSAGE
Paramètres	ModelMessage message, bool edited
Client -> Serveur	Le client envoie un message dans une salle de discussion spécifiée dans le modèle du champ <i>message</i> au moyen de ce paquet. Ce paquet permet aussi d'éditer un message existant et, le cas échéant, met le booléen <i>edited</i> à la valeur vrai.
Serveur -> Client	Les deux paramètres sont transmis afin de transmettre le nouveau message à tous les utilisateurs de la salle concernée ou de les aviser de la modification de son contenu par son auteur d'origine. Le serveur transmet aussi le message à l'auteur lui-même en guise de confirmation.

Nom du paquet	DELETE_MESSAGE
Paramètres	quint32 roomId, quint32 messageld
Client -> Serveur	Le client souhaite supprimer un message dont il est l'auteur. Il lui suffit de spécifier l'identifiant du message concerné ainsi que l'identifiant de la salle dans laquelle ce message apparaît (pas nécessaire, le serveur pourrait tout aussi bien déduire l'identifiant de la salle en effectuant une requête dans la base de données, mais cela évite une requête rendant le processus plus léger).
Serveur -> Client	Une fois le message supprimé dans la base de données, le serveur transmet la suppression à tous les usagers de la salle concernée en spécifiant les deux champs. Le serveur transmet aussi le message à l'auteur lui-même en guise de confirmation.

Nom du paquet	CONNECTED
Paramètres	quint32 userId
Client -> Serveur	Paquet non utilisé dans ce sens.
Serveur -> Client	Le serveur signale à tous les utilisateurs en relation avec l'utilisateur dont l'identifiant est passé en paramètre qu'il s'est connecté avec succès, ceci dans le but de marquer dans l'interface que ce client est en ligne (pseudonyme en gras). En effet, les autres clients en relation (partageant au moins une salle de discussion avec ce client) connaissent déjà son ModelUser qu'ils auront reçu à leur propre connexion.

Nom du paquet	DISCONNECT
Paramètres	quint32 userId
Client -> Serveur	Non utilisé dans ce sens, la déconnexion est gérée lorsque le socket est fermé.
Serveur -> Client	Très similaire au paquet CONNECTED, ce paquet permet de notifier tous les utilisateurs en relation avec l'utilisateur spécifié qu'il s'est déconnecté. Cela permet de mettre à jour l'interface en marquant cet utilisateur comme non connecté (pseudonyme en non gras)

.

Nom du paquet	LOGIN
Paramètres	QString pseudo, QByteArray hashedPWD
Client -> Serveur	Le client tente d'effectuer un login en utilisant le nom d'utilisateur <i>pseudo</i> et son mot de passe hâché dans le champ <i>hashedPWD</i>
Serveur -> Client	Ce paquet n'est pas utilisé dans ce sens. Lorsqu'un client se connecte, le serveur lui envoie le ModelUser qui le représente ainsi que son trousseau de clés dans le paquet INFO_USER.

Nom du paquet	SALT
Paramètres	QString pseudo, QByteArray salt
Client -> Serveur	Avant d'effectuer une tentative de login, le client doit récupérer le sel qu'il doit utiliser pour hâcher le mot de passe. Ce dernier se trouvant dans la base de données, le client doit faire une requête au moyen de ce paquet en spécifiant un nom d'utilisateur dans le champ <i>pseudo</i> . Le champ <i>salt</i> est le champ de réponse et n'est pas utilisé.
Serveur -> Client	Le serveur renvoie le sel de l'utilisateur spécifié dans le champ <i>salt</i> ou renvoie une erreur de type AUTH_ERROR si l'utilisateur n'existe pas.

Nom du paquet	INFO_USER
Paramètres	ModelUser user, QByteArray keySalt, QByteArray publicKey, QByteArray privateKey
Client -> Serveur	Non utilisé dans ce sens.
Serveur -> Client	Lorsque le client réussit la connexion (l'utilisateur et le hash correspondent) le serveur lui renvoie le ModelUser qui représente son profil dans le champ <i>user</i> ainsi que son trousseau de clés RSA via les champs <i>publicKey</i> et <i>privateKey</i> . Comme la clé privée est protégée par un hash différent selon le mot de passe de l'utilisateur et un second sel, ce dernier est transmis via le champ <i>keySalt</i> .

Nom du paquet	PUBLIC_KEY
Paramètres	List<QPair<quint32, QByteArray>> usersIdAndKey
Client -> Serveur	Lorsqu'un utilisateur crée ou modifie une salle de discussion dont il est administrateur, il doit transmettre la clé AES de la salle à chacun des utilisateurs qu'il y ajoute. Afin de transmettre cette clé de manière sécurisée, elle est chiffrée de manière différente pour chacun des nouveaux utilisateurs avec leur clé RSA publique, il faut donc les récupérer. Le client envoie donc ce paquet avec une liste d'identifiants dont il souhaite connaître les clés. Le deuxième champ de la paire est laissé vide car il s'agit du champ de réponse.
Serveur -> Client	Le client renvoie les clés demandées associées aux identifiants spécifiés dans le premier champ de chaque paire et les stocke dans le second champ.

Nom du paquet	JOIN
Paramètres	QMap<quint32, ModelRoom> rooms, QMap<quint32, ModelUser> users
Client -> Serveur	Non utilisé dans ce sens.
Serveur -> Client	<p>Ce paquet peut être envoyé par le serveur à plusieurs occasions. Premièrement, lorsqu'un utilisateur se connecte, le serveur lui envoie la liste des salles dans lesquelles il est inscrit dans le champ <i>rooms</i> ainsi que les ModelUsers représentant tous les autres clients avec qui il est en relation au travers de ces salles dans le champ <i>users</i>.</p> <p>Deuxièmement, à la création d'une nouvelle salle, le serveur envoie ce paquet à tous les utilisateurs qui y sont inscrits afin qu'ils aient directement toutes les informations nécessaires pour que leur interface affiche cette nouvelle salle. Si la salle est privée, et la clé privée de la salle étant chiffrée différemment pour chacun, le paquet est personnalisé pour chacun des clients et nécessite d'être modifié entre chaque envoi.</p> <p>Troisièmement, lorsqu'un nouvel utilisateur rejoint une salle publique, ou qu'un administrateur l'a accepté ou ajouté manuellement, ce paquet est aussi envoyé à tous les utilisateurs de la salle, y compris le nouveau venu, afin de notifier ce changement.</p>

Nom du paquet	ROOM
Paramètres	ModelRoom room, bool edited, QMap<quint32, QByteArray> usersAndKeys
Client -> Serveur	Lorsqu'un utilisateur crée une salle ou qu'un administrateur d'une salle édite ses propriétés, ce paquet est envoyé contenant les données de la salle dans le champ <i>room</i> . Le booléen <i>edited</i> indique s'il s'agit d'une création ou d'une modification. Dans les deux cas, si la salle est privée, le champ <i>usersAndKeys</i> contient la clé AES chiffrée avec la clé RSA publique de chacun des utilisateurs ajoutés.
Serveur -> Client	Le serveur envoie ce paquet à tous les membres d'une salle lorsqu'un administrateur en a modifié les propriétés (nom, nombre de messages stockés, image, administrateurs, etc ...).

Nom du paquet	LEAVE
Paramètres	quint32 userId, quint32 roomId
Client -> Serveur	Lorsqu'un utilisateur souhaite se désinscrire d'une salle, il envoie ce paquet en spécifiant la salle qu'il souhaite quitter dans le champ <i>roomId</i> . Placer son identifiant dans le champ <i>userId</i> n'est pas essentiel car, du côté serveur, le socket est directement associé à l'identifiant utilisateur dans la structure <i>ChatorClient</i> .
Serveur -> Client	Une fois l'utilisateur retiré de salle dans la base de donnée, le serveur notifie tous les membres de la salle qu'un utilisateur la quitte afin que leur interface l'enlève de la liste. Le paquet est aussi transmis à l'utilisateur concerné en guise de confirmation.

Nom du paquet	JOIN_ROOM
Paramètres	quint32 roomId
Client -> Serveur	Lorsqu'un client souhaite rejoindre une salle à laquelle il n'est pas inscrit, qu'elle soit publique ou privée-visible, il utilise ce paquet pour en faire la demande en transmettant simplement l'identifiant de la salle concernée dans le champ <i>roomId</i> . Si la salle est publique, il sera ajouté de suite, sinon un administrateur doit valider sa demande.
Serveur -> Client	Non utilisé dans ce sens.

Nom du paquet	USER_ID
Paramètres	Qstring userName, bool exists, quint32 userId
Client -> Serveur	<p>Récupérer l'identifiant d'un utilisateur est utile dans deux cas :</p> <p>Premièrement, lors de l'inscription, il faut s'assurer que le pseudonyme désiré n'est pas déjà utilisé.</p> <p>Deuxièmement, lorsqu'un utilisateur crée une salle ou qu'un administrateur modifie la salle et y ajoute des membres, il doit associer les pseudonymes des utilisateurs ajoutés à leur identifiant.</p> <p>Dans les deux cas, l'utilisateur va placer le nom recherché dans le champ <i>userName</i> et attendre la réponse du serveur. Les champs <i>exists</i> et <i>userId</i> ne sont pas utilisés.</p>
Serveur -> Client	Le serveur effectue une recherche dans la base de données et retourne le résultat. Le booléen <i>exists</i> indique si l'utilisateur existe et, le cas échéant, place son identifiant dans le champ <i>userId</i> .

Nom du paquet	DELETE_ROOM
Paramètres	quint32 roomId
Client -> Serveur	Lorsqu'un administrateur souhaite supprimer une salle, il utilise ce paquet en spécifiant simplement l'identifiant de la salle concernée dans le champ <i>roomId</i> .
Serveur -> Client	Dès que le serveur a contrôlé que la requête provient bel et bien d'un administrateur de cette salle, elle est supprimée et tous les membres qui la composent, administrateurs compris, sont notifiés de sa suppression afin de mettre à jour leur interface.

Nom du paquet	LIST_ROOMS
Paramètres	QList<QPair<quint32, QString>> publicRooms, QList<QPair<quint32, QString>> privateVisibleRooms
Client -> Serveur	Lorsqu'un utilisateur souhaite rejoindre une nouvelle salle, il doit pouvoir visualiser lesquelles sont disponibles. Il envoie alors la requête au serveur au moyen de ce paquet en ne spécifiant aucun paramètre.
Serveur -> Client	Le serveur répond à la requête de liste en plaçant les salles publiques et leur identifiant dans <i>publicRooms</i> et les salles privées-visibles dans le champ <i>privateVisibleRooms</i> .

Nom du paquet	REQUEST
Paramètres	quint32 roomId, ModelUser user, QByteArray publicKey, bool accepted
Client -> Serveur	Lorsqu'un utilisateur souhaite rejoindre une salle privés-visible, sa demande doit être acceptée par un des administrateurs de la salle. Ce paquet permet à un administrateur de donner son approbation ou son refus. Il spécifie l'identifiant de la salle concernée dans le champ <i>roomId</i> et le ModelUser de l'utilisateur ayant fait la demande dans le champ <i>user</i> . Son refus ou son acceptation est représenté via le booléen <i>accepted</i> et, en cas de réponse positive, la clé AES de la salle sécurisée par la clé RSA publique de l'utilisateur est passée dans le champ <i>publicKey</i> qui est vide sinon.
Serveur -> Client	Dès qu'un utilisateur fait une demande d'adhésion à une salle privée-visible, ou dès qu'un administrateur se connecte, ce paquet est transmis pour notifier les responsables d'une salle qu'il y a des requêtes en attente à traiter.

Nom du paquet	SERVER_ERROR
Paramètres	ModelError error
Client -> Serveur	Non utilisé dans ce sens.
Serveur -> Client	Notifie le client qu'une erreur est survenue, spécifiée par l'objet <i>error</i> dont il est possible d'extraire un code d'erreur et un message textuel à afficher.

2.8.6 Scénario de transmission d'un message

Afin de mieux illustrer comment fonctionne notre protocole, voici le déroulement complet de l'envoi d'un message du client au serveur :

1. L'utilisateur saisit du texte dans le champ d'édition de texte d'une salle dont il est membre et active le bouton d'envoi ou presse sur la touche Enter.
2. La vue viewChat signale qu'il faut envoyer un message
3. Le contrôleur controllerChat récupère le texte et, si la salle est privée, le chiffre avec la clé AES de la salle qu'il récupère dans ModelChator.
4. Le contenu du message est encapsulé dans un nouvel objet de type ModelMessage dans lequel est aussi spécifié l'identifiant de l'utilisateur courant ainsi que l'identifiant de la salle dans laquelle le message doit être envoyé.

5. Le message est transmis au contrôleur de données sortantes `controllerOutput` via sa méthode `sendMessage(const ModelMessage& message, const bool edited)` en lui passant le message et la valeur `faux` car il s'agit d'un nouveau message et non d'une édition de message existant.
6. Le contrôleur de données sortantes récupère le résultat de la sérialisation de ces données dans un paquet de type `MESSAGE` en appelant la méthode `sendMessage(const ModelMessage& message, const bool edited)` de l'objet `Interpreter`.
7. L'`Interpreter` place en premier un entier, valeur effective du type énuméré `MessageType` avec la valeur `MESSAGE` puis place les autres champs.
8. Le paquet et ses données est stocké sous la forme d'un tableau d'octet défini par Qt via le type `QByteArray` et, étant prêt pour l'envoi, est transmis au `ClientConnector`, responsable de l'envoyer via sa méthode `send(const QByteArray& data)`.
9. Le `ClientConnector` envoie les données dans le socket via la méthode `sendBinaryMessage(const QByteArray& data)` de la classe `QWebSocket`.
10. Le paquet transit sur le réseau jusqu'au serveur
11. Le socket connecté au client signale qu'une donnée est prête à être traitée.
12. L'`Interpreter` du serveur récupère le paquet complet en tant que simple tableau d'octets et désérialise le premier champ de type `MessageType` qui représente le type du paquet. De cette valeur dépend le traitement des données contenues dans le paquet.
13. L'`Interpreter` passe dans une structure de type `switch` selon le type du paquet et, dans le case correspondant au type `MESSAGE`, effectue la désérialisation des autres paramètres du paquet, soit le `ModelMessage` et le booléen.
14. L'`Interpreter` passe les paramètres à la méthode `receiveMessage(ModelMessage& message, const bool edited, QObject* sender)` du contrôleur de données entrantes qu'il connaît en tant que `ControllerInput` mais qui est en fait un `ServerControllerInput`. Le champ `sender` est en fait le pointeur vers la structure de type `ChatorClient` qui contient les informations du client dont provient la requête. Ce champ est nécessaire du côté serveur mais n'est pas utilisé côté client.
15. La liaison dynamique est faite sur le `ServerControllerInput` et le corps de sa méthode `receiveMessage` appelle la méthode `processMessage(ModelMessage& message, const bool edited, ChatorClient* client)` du contrôleur de salles de discussion (`controllerRoom`).
16. Le contrôleur de salles de discussion contrôle que l'utilisateur est bien en ligne (a effectué un login correctement), que la salle est en ligne et que l'utilisateur en fait bien partie.
17. Comme le message est nouveau, il est stocké dans la base de données qui en appelant la méthode `storeMessage(const ModelMessage& message)` du contrôleur de base de données. Celui-ci va en renvoyer l'identifiant une fois inséré.
18. L'identifiant nouvellement attribué est placé dans le `ModelMessage` et sa date de création est figée au temps présent.
19. Le contrôleur de salles de discussion crée un paquet de type `MESSAGE` en appelant la méthode `sendMessage(const ModelMessage& message, const bool edited)` de l'objet `Interpreter`.
20. L'`Interpreter` place en premier un entier, valeur effective du type énuméré `MessageType` avec la valeur `MESSAGE` puis place les autres champs.
21. Le contrôleur de salles de discussion parcourt tous les utilisateurs en ligne de la salle concernée et envoie à chacun le paquet en appelant directement la méthode `sendBinaryMessage(const QByteArray& data)` de la classe `QWebSocket`.
22. Le paquet circule sur le réseau vers chacun des utilisateurs de la salle.
23. Chez les clients, l'objet `ClientConnector` signale qu'une donnée est prête à être traitée.
24. L'`Interpreter` du client récupère le paquet complet en tant que simple tableau d'octets et désérialise le premier champ de type `MessageType` qui représente le type du paquet. De cette valeur dépend le traitement des données contenues dans le paquet.

25. L'Interpretor passe dans une structure de type switch selon le type du paquet et, dans le case correspondant au type MESSAGE, effectue la désérialisation des autres paramètres du paquet, soit le ModelMessage et le booléen.
26. L'Interpretor passe les paramètres à la méthode `receiveMessage(ModelMessage& message, const bool edited, QObject* sender)` du contrôleur de données entrantes qu'il connaît en tant que `ControllerInput` mais qui est en fait un `ClientControllerInput`. Le champ *sender* est bien passé mais n'est pas utilisé côté client.
27. La liaison dynamique est faite sur le `ClientControllerInput` et le corps de sa méthode `receiveMessage` appelle la méthode `receiveMessage(ModelMessage& message, const bool edited)` du contrôleur de chat.
28. Le contrôleur de chat récupère la salle dans laquelle doit aller le message et, si la salle est privée, déchiffre le contenu du message avec la clé se trouvant dans le `ModelRoom`.
29. Le contrôleur de chat examine la valeur du booléen indiquant l'édition. Comme celui-ci est faux pour notre exemple, le message est nouveau et il faut le stocker, le message est donc passé à la méthode `addMessage(const ModelMessage& message)` du `ModelRoom`.
30. Le message est ajouté dans la table de hachage qui stocke les messages dans le `ModelRoom`.
31. Le contrôleur de chat appelle la méthode `loadRoomMessage(ModelMessage& message, const bool edited)` de la vue du chat qui va se rafraîchir et enfin afficher le message à l'utilisateur.

2.9 BASE DE DONNÉES

Afin de stocker toutes les informations persistantes relatives à notre serveur de chat, nous avons utilisé une base de données de type SQL.

Nous avons décidé d'utiliser une base de données de type SQLite. Ceci permet d'avoir une solution légère et portable. En effet, la base SQLite est directement liée au programme et permet d'être exportée facilement car elle se compose uniquement de deux fichiers :

- `init.sql` : ce fichier contient le schéma de la base de données, c'est-à-dire la structure de tables de la base de données. Il permet d'initialiser un fichier `db.sqlite`.
- `db.sqlite` : ce fichier contient à nouveau la structure de la DB ainsi que les données à proprement parler, stockées en partie sous forme binaire.

2.9.1 Schéma de la base de données

Les données sont stockées dans un fichier indépendant sur le serveur. La base de données est constituée des éléments décrits dans le schéma suivant.

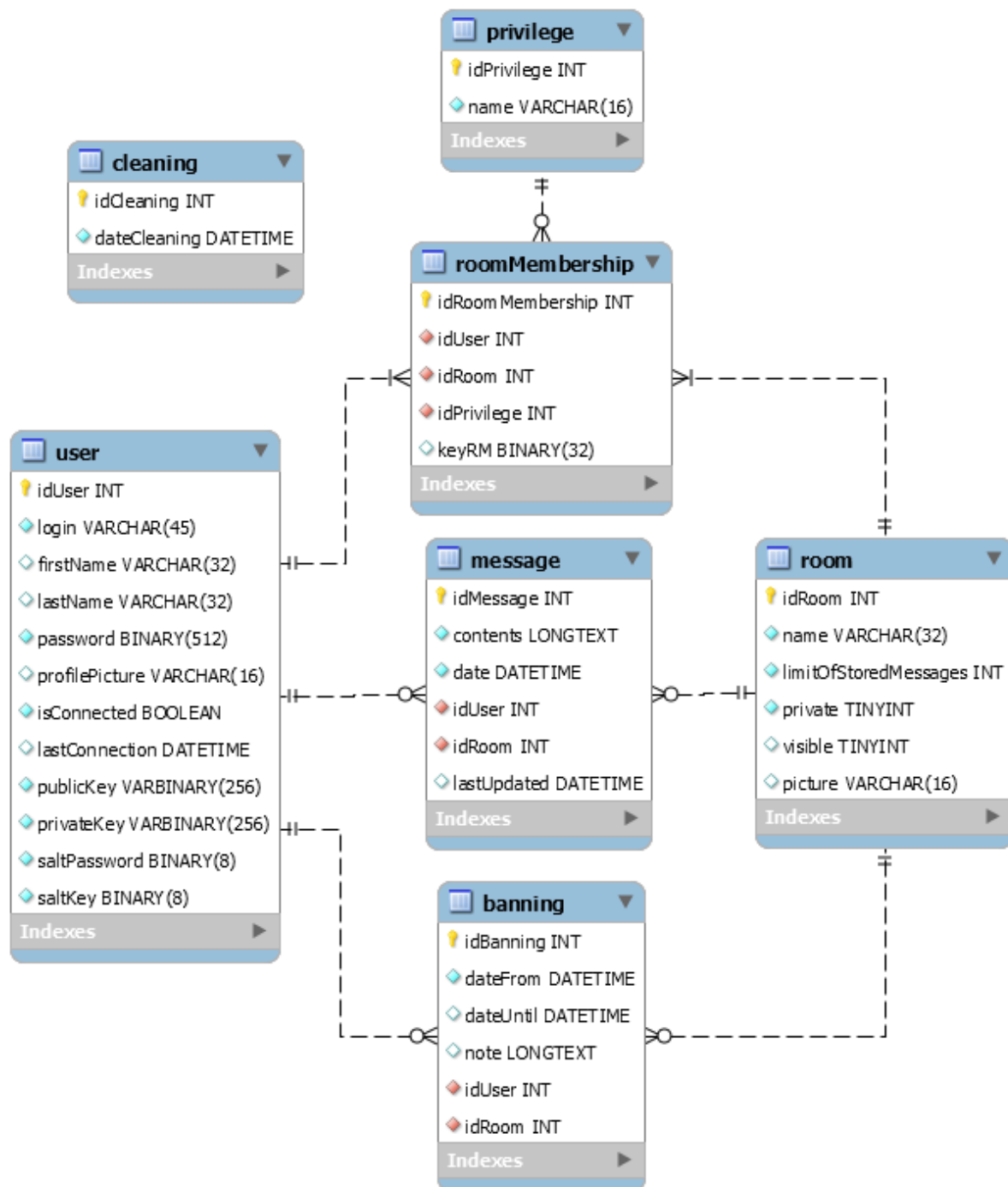


Figure 36: Schéma de la base de données

Il s'agit d'un schéma plutôt simple doté de six tables. Il n'y a rien de particulier à noter à l'exception des différents champs propres à la sécurité de l'application tels que les sels et les paires de clés. Ces éléments permettent de chiffrer des messages afin de les envoyer de manière sûre et également de ne pas stocker des informations en clair dans notre base de données.

Table user

Cette table contient toutes les informations relatives à un utilisateur. Le stockage des clés est primordial car il permet à l'utilisateur, une fois ces dernières récupérées, de déchiffrer les messages des salles dont il est membre.

Table message

Cette table permet de stocker les informations concernant un message : son contenu, son auteur, la salle dans laquelle il a été envoyé, ainsi que la date de sa dernière édition dans le cas où il aurait été modifié ultérieurement.

Table room

Cette table contient les informations décrivant les salles, telles que le type de salle (privée ou publique), sa visibilité ou encore son logo.

Table membership

Lorsqu'un utilisateur a rejoint une salle, une nouvelle entrée est ajoutée dans cette table. L'entrée contient également une référence sur le privilège de l'utilisateur, ainsi qu'une clé propre à chaque adhésion à une salle.

Table privilege

Il y a trois privilèges possibles : membre, administrateur ou « en demande d'adhésion ».

Le privilège membre est le plus basique et permet uniquement d'envoyer des messages.

Le privilège administrateur permet de supprimer un membre d'une salle, inviter un membre dans une salle et accepter un membre dans le cas où il s'agit d'une salle privée.

Le privilège « en demande d'adhésion » correspond à un utilisateur désireux de rejoindre une salle privée visible. Une fois accepté, son privilège changera en membre.

Dans un souci d'évolutivité, une table à part entière est utilisée afin de gérer les différents privilèges, bien que pour l'instant il n'en existe que trois différents.

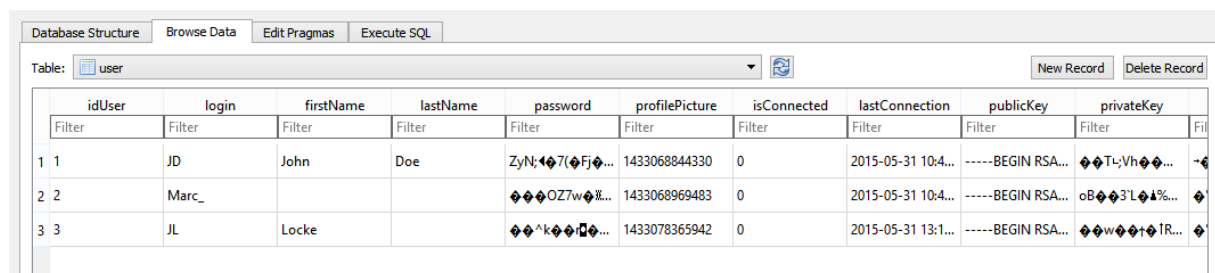
Table banning

Nous ne sommes pas parvenu à développer une fonction de bannissement. Néanmoins, dans une extension future, cette table aurait permis de gérer ces bannissements.

Par exemple, un administrateur peut décider, pour diverses raisons, d'exclure un membre d'une salle pour une durée déterminée ou illimitées.

2.9.2 Consultation des données

Pour des raisons de développement, nous avons été amenés à vouloir consulter les informations stockées dans la base SQLite. Pour ce faire nous avons utilisé des programmes tels que SQLiteBrowser.



The screenshot shows the SQLiteBrowser interface with the 'user' table selected. The table has 11 columns: idUser, login, firstName, lastName, password, profilePicture, isConnected, lastConnection, publicKey, and privateKey. There are three rows of data displayed.

	idUser	login	firstName	lastName	password	profilePicture	isConnected	lastConnection	publicKey	privateKey
1	1	JD	John	Doe	ZyN;7(Fj...	1433068844330	0	2015-05-31 10:4...	-----BEGIN RSA...	-----BEGIN RSA...
2	2	Marc_			000OZ7w0%...	1433068969483	0	2015-05-31 10:4...	-----BEGIN RSA...	oB003'L4%...
3	3	JL	Locke		00^k0000...	1433078365942	0	2015-05-31 13:1...	-----BEGIN RSA...	0w0001R...

Figure 37: Aperçu de la base de données depuis SQLiteBrowser

On s'aperçoit que certaines informations, telles que les mots de passe, ne sont pas stockées en clair.

2.9.3 Interaction avec la base de données

Les requêtes formulées pour interroger la base de données sont des requêtes SQL standards. Une fois le programme lié au fichier SQLite, on peut exécuter des requêtes à notre guise. Ces requêtes sont envoyées depuis le fichier controllerDB.h du côté du serveur.

Exemple :

```
query.prepare("UPDATE message SET contents = :content, lastUpdated =  
datetime('NOW') WHERE idMessage = :idMessage");  
  
query.bindValue(":content", message.getContent());  
query.bindValue(":idMessage", message.getIdMessage());  
query.exec();
```

3 PROBLÈMES RENCONTRÉS

Si aucun problème n'est rencontré au cours d'un projet, c'est que les choses vont mal. Nous avons eu notre lot de petits soucis et complications en tout genre.

3.1 PROBLÈMES ORGANISATIONNELS

- Expliquer son travail aux autres membres du groupe afin qu'ils puissent le comprendre et l'utiliser (notamment au niveau de la cryptographie et du serveur). Il est souvent difficile d'expliquer aux autres ce que nous avons implémenté, même après l'avoir bien compris.
- Il s'agit ici plutôt d'un regret, mais il aurait été plaisant d'appliquer les choses vues en cours de GEN (Génie Logiciel) concernant les projets.
- Nous aurions dû déterminer plus précisément et ensemble la manière dont les fichiers partagés par le client et le serveur étaient gérés, en particulier le modèle ModelChator. En effet, les types qui y sont définis sont utilisés dans l'ensemble de l'application ; le moindre changement sur les méthodes des classes qui y sont déclarées implique donc des changements à tous les endroits où elles sont utilisées. Cela nous a fait perdre un certain temps.
- En parlant du temps, justement, celui-ci s'est avéré plutôt difficile à gérer, d'autant plus que nous avons une dose considérable de travail dans les autres branches, certaines fois.
- Nous avons dû faire face à beaucoup de nouveautés techniques qui, concoctées avec le temps qui se faisait avare, nous ont poussé à faire des formations qui nous ont pris encore plus de temps.
- Suivi des conventions de codage établies.

3.2 PROBLÈMES TECHNIQUES

- L'établissement de la communication entre le serveur et les clients nous a pris un temps relativement long.
- L'installation des bibliothèques tierces (OpenSSL) sur Windows s'est avérée très pompeuse.
- Les dépendances circulaires : le contrôleur User connaît le contrôleur Room, qui connaît lui-même le contrôleur User, etc.
- Le déverminage est long et fastidieux, surtout lorsque des fonctionnalités problématiques dépendent en partie du code de quelqu'un d'autre, code que l'on maîtrise forcément moins bien que le nôtre.

3.3 POINTS POSITIFS

- Une bonne organisation, on savait ce qu'on avait à faire, on était généralement bien coordonnés, la mise en commun des différentes parties, bien que pas une partie de plaisir c'est relativement bien passée (sauf la cryptographie, qui a été un peu plus complexe).
- Une bonne ambiance, tout le monde y a mis du sien.
- Le thème du projet s'est avéré très intéressant et plein de défi.

- Acquisition de nouvelles connaissances (Qt, C++, etc.).
- Le découpage des tâches était optimisé.

4 CONCLUSION

4.1 FONCTIONNALITÉS

De par le manque de temps et la complexité du travail que représente le développement d'une application telle que Chator, il se peut que quelques bugs subsistent encore dans notre application. Le document « Chator : Tests » recense les tests effectués et leurs résultats. A noter que nous avons laissé tomber quatre parties non vitales au projet, qui auraient rapidement pu s'avérer pompeuse, à savoir :

1. La fenêtre permettant de générer une nouvelle paire de clés pour l'utilisateur connecté ; pour rappel, celle-ci devait s'afficher lorsque l'utilisateur appuyait sur le menu « Edition > Sécurité... », et se présentait ainsi :

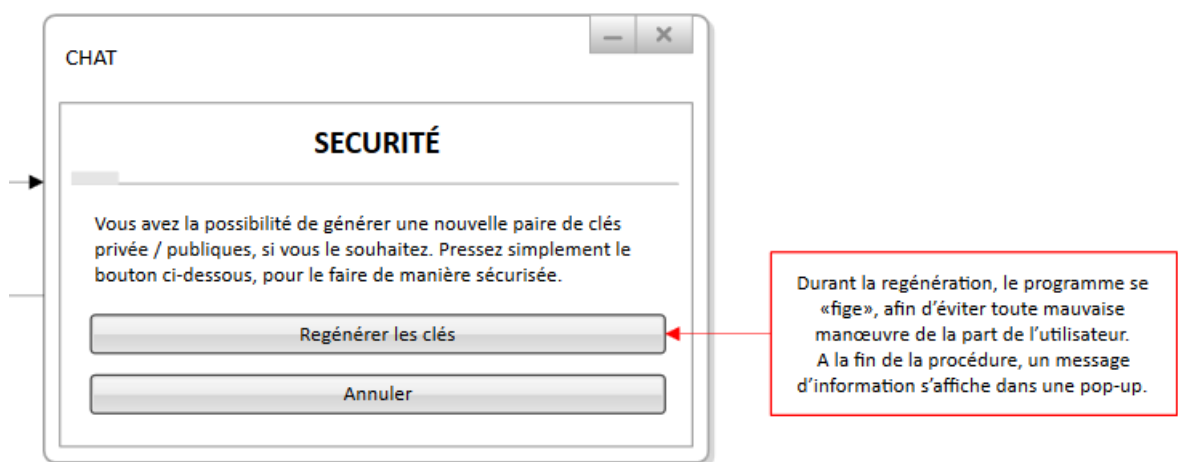


Figure 38 : Fenêtre initiale permettant de générer une nouvelle paire de clés.

L'abandon de cette fonctionnalité – que nous jugions peu importante pour l'application – se justifie par le fait qu'elle nous aurait pris un certain temps, que nous préférons mettre à disposition dans des fonctionnalités plus importantes, comme l'adhésion à une salle ou l'inscription, qui étaient plus critiques. En effet, le fait de régénérer les clés de l'utilisateur nous aurait obligé à devoir récupérer toutes les clés AES de celui-ci, les déchiffrer avec l'ancienne clé privée RSA, et les rechiffrer avec la nouvelle clé publique RSA.

2. Il était prévu à la base de pouvoir changer le type d'une salle lors de son édition, soit permettre de passer une salle publique en salle privée, ou inversement. Nous nous avons aussi décidé d'abandonner cette fonctionnalité mineure en cours de projet. En effet, celle-ci nous aurait aussi pris du temps, car elle demanderait un traitement assez lourd, se traduisant par la nécessité de chiffrer, respectivement de déchiffrer tous les messages de la salle, opération que seul un administrateur peut effectuer.
3. L'édition de compte s'est transformée en simple fenêtre d'affichage des détails du compte, car nous n'avons simplement pas eu le temps de l'implémenter complètement. Par soucis de rigueur, nous avons tout de même décidé de faire quelque chose avec ce qui avait déjà été implémenté (à savoir, la récupération et l'affichage des informations de l'utilisateur).
4. Finalement, nous avons laissé tomber le bannissement d'un utilisateur, aussi par manque de temps, et parce que nous jugions cette fonctionnalité comme étant peu importante.

A noter aussi que même si nous avons testé et débogué notre programme, la gestion des erreurs est encore loin d'être complète. Il s'agit pour la grande majorité de fonctionnalités qui ne sont pas censées planter, comme la génération de nombre aléatoires.

Malgré présents points, nous pensons avoir fourni un travail de qualité, et, dans les grandes lignes, sommes parvenus à nos fins en respectant la plupart de ce qui avait été planifié. Le projet livré est propre, documenté et opérationnel.

4.2 AMÉLIORATIONS FUTURES POSSIBLES

De nombreuses améliorations sont envisageables pour rendre notre application Chator éclatante. Il faudrait tout d'abord implémenter les fonctionnalités qui n'ont pas pu l'être (voir ci-dessus), la gestion d'erreurs y compris.

Outre cela, de nombreuses autres fonctionnalités pourraient être implémentées ; vous pouvez retrouver ci-dessus une petite liste non-exhaustive.

4.2.1 Côté serveur

- Refactorisation du code : plusieurs parties du code du serveur sont un peu redondantes ou inutilement compliquées et il serait de bon augure d'effectuer un nettoyage et une factorisation.
- Optimiser le login / garder les ModelUser et ModelRoom en cache : Actuellement, le serveur est assez lent lors du login ou de toute opération nécessitant de transférer un grand nombre de ModelUser ou ModelRoom. Cela s'explique par le fait que pour la construction de chacun de ces éléments, une requête est effectuée dans la base de données pour y récupérer les champs et un accès disque est fait pour y lire et décoder l'image de profil. Il serait très utile de garder les ModelUser et les ModelRoom en cache afin de ne pas avoir à les recharger à chaque utilisation, mais uniquement lors de changements qui, par nature, sont beaucoup plus rares.
- Utiliser plusieurs serveurs simultanément, pour pouvoir optimiser le transfert des données.

4.2.2 Côté client

- La première amélioration envisageable serait d'améliorer l'ergonomie de l'application – bien que celle-ci ne soit pas handicapante actuellement – en ajoutant des boutons personnalisés, ainsi qu'un peu plus de couleurs, par exemple.
- Une fonctionnalité intéressante dans ce type d'application est la déconnexion d'un utilisateur, permettant à un autre de se connecter sans avoir à quitter l'application. A noter qu'il existe déjà un paquet prévu du côté serveur, ainsi que certains en-têtes de méthodes.
- Lors du démarrage de l'application, il serait très intéressant de scanner le réseau pour rechercher d'éventuels serveurs disponibles, et les proposer dans une liste.
- Permettre l'envoi de messages privés entre utilisateurs.
- Gérer l'envoi des fichiers.
- La possibilité de changer la clé secrète d'une salle privée, pour des questions de sécurité.
- Dans une optique de renforcement des aspects sécuritaires et cryptographiques de l'application, nous pourrions obliger les utilisateurs à changer de mot de passe au bout d'un

certain délai, par exemple tous les 3 mois. Il s'agit en effet du talon d'Achille de nombreuses applications sécurisées.

- Recherche de messages spécifiques dans une discussion.
- Recherche d'une salle dans la liste de salles dont l'utilisateur est membre.
- Tri des salles et des membres appartenant à ces salles par ordre alphabétique.

4.3 CONCLUSION FINALE

Ce projet de semestre s'est avéré être un réelle « plus » dans notre formation, nous apportant une expérience réelle et solide, de par ses aspects techniques, ainsi que par les aspects liés à la gestion de projet. Nous avons pu acquérir de nouvelles connaissances techniques, tant au niveau du fonctionnement global d'un serveur qu'au niveau de la sécurité ; celle-ci s'est avérée être un véritable défi, que nous avons relevé avec intérêt et enthousiasme.

Nous nous sommes rapidement rendus compte que la gestion d'un tel projet – se composant d'un certain nombre de personnes – posait rapidement des problèmes au niveau de l'organisation et de la communication. Cependant, grâce à l'expérience déjà acquise au fil de notre formation, ainsi que grâce à la bonne foi et la motivation générales qui régnait au sein du groupe en tout temps, nous avons toujours pu nous en sortir sans réel problème.

Nous évaluons notre travail général comme étant plutôt bon, bien que pas parfait.