Rapport POO - application de clavardage

4IR - B2

Céléstin Moënne-Loccoz

Julien Rouzot

Sommaire

1.Conception du projet **3**

2.Description des choix d'implémentation **4**

2.1.Les communications 4

2.1.1.Le broadcast UDP 4

2.1.2.Les communications TCP 5

2.2.La base de données 6

2.2.1.Description des tables 6

2.2.2.Base de données locale 6

2.2.3.Base de données centralisée 7

2.3.Les interfaces utilisateurs 8

3.Tests de validation réalisés **10**

4.Manuel d'utilisation **13**

1.Conception du projet

Après la lecture du cahier des charges et l'étude de ses différents points d'exigences, l'application de clavardage livrée avec ce rapport a fait l'objet d'une modélisation mûrement réfléchie afin d'implémenter au mieux ses différentes fonctionnalités.

Tous nos diagrammes sont disponibles dans le dossier "Diagrammes" dans le projet GIT.

Nous avons choisi de nous baser sur une conception orientée objet dont les principaux diagrammes sont les suivants :

- Diagramme de cas d'utilisation.
    - But : Identifier les fonctionnalités principales demandées par le client

- Diagramme de séquences.
    - But : Identifier les interactions entre les différents acteurs

- Diagramme de classes.
    - But : identifier les différents composants (objets) de l'application et leurs interactions.

- Machines à états.
    - But : identifier les différents cycles de l'application pendant son exécution.

Nous avons choisi d'adopter le design pattern MVC (Modèle-Vue-Contrôleur), car il sied particulièrement bien à l'implémentation de l'application de clavardage. Ici, les principaux modèles sont les suivants :

- *User* : représente les utilisateurs, ce modèle nous permet de garder en mémoire, les informations de connexion (id, mot de passe), le pseudonyme, l'adresse IP de l'utilisateur.

- *Message* : représente les messages échangés pendant les sessions de clavardage. On veut pouvoir sauvegarder dans la base de données le contenu du message, les utilisateurs impliqués dans la conversation, l'horodatage du message.

Les vues sont les interfaces avec lesquelles les utilisateurs pourront interagir. Les principales vues sont les suivantes :

- *ConnectionWindow* : Point d'entrée de l'application, permet à l'utilisateur de s'identifier afin de garder en mémoire ses conversations dans la base de données et ainsi accéder à son historique.

- *PseudonymeWindow* : Permet de choisir un pseudonyme (étape obligatoire pour accéder à l'application) qui doit être unique. L'utilisateur est notifié en cas d'erreur et doit recommencer l'opération le cas échéant.

- *MainWindow* : Affiche la liste des utilisateurs connectés ainsi qu'une entrée texte pour modifier son pseudonyme en cours d'utilisation de l'application. Sélectionner un utilisateur permet de lancer une session de clavardage avec ce dernier.

- *ChatWindow* : Affiche une fenêtre de clavardage avec une entrée texte pour saisir les messages et une zone de texte déroulante pour afficher l'historique des messages.

Les contrôleurs permettent de gérer la logique des messages UDP (pour la recherche des pseudonymes des autres utilisateurs actifs) et TCP (pour les messages du clavardage). Ils permettent aussi la gestion des utilisateurs et l'écriture / lecture dans la base de données.

2.Description des choix d'implémentation

2.1.Les communications

2.1.1.Le broadcast UDP

Afin de simplifier l'utilisation des sockets UDP (DatagramSocket), nous avons créé une classe nommée UdpCommunication.

Tout d'abord, elle permet d'ouvrir un socket sur le port souhaité grâce à la méthode openSocket (à priori, un port aléatoire si on souhaite envoyer un message et un port prédéfini si on souhaite attendre certains types de messages.

Une fois ce socket ouvert, on pourra l'utiliser pour envoyer un message en broadcast sur le réseau local vers un certain port via la méthode broadcastMessage, ou envoyer un message en unicast vers une adresse IP sur un certain port grâce à la méthode unicastMessage.

On pourra également recevoir des messages grâce à ce socket. Nous pourrons choisir de recevoir un certain nombre de messages grâce à la méthode receiveMessages en précisant le temps maximal à attendre avant d'arrêter de recevoir. Nous pourrons également attendre de recevoir un seul message à l'infini ou en précisant un temps d'attente maximal.
Lorsque nous utilisons ces méthodes de réception, les messages reçus sont concaténés (à la fin des messages) avec l'adresse IP de l'émetteur et le port sur lequel il a envoyé ce message avant d'être retournés. Ainsi, toute classe utilisant UdpCommunication recevra

toutes les informations contenues dans les messages ainsi que toutes les informations nécessaires pour répondre à ces messages.

Cette classe sera utilisée dans deux cas :

- lorsqu'un utilisateur souhaite indiquer un changement de son état (connexion, déconnexion ou changement de pseudonyme) aux autres utilisateurs du réseau (via la méthode isNicknameAvailable)

- dans un thread analysant tous les messages de changement d'état reçus par d'autres utilisateurs afin de les afficher lorsqu'ils ont actifs, les enlever sinon, et changer leur pseudonyme au bon moment (via le thread NotificationListener)

2.1.2.Les communications TCP

Pour ce qui est de la communication TCP, nous avons 2 phases à traiter : la connexion à l'hôte distant ou la réception d'une demande de connexion, puis l'échange de messages avec cet hôte distant. Nous avons donc séparé la communication TCP en 2 classes :

- ChatManager qui permettra de lancer une demande de connexion TCP, et qui va tourner en tant que thread pour récupérer toutes les demandes de connexion et créer un socket pour chacune d'entre elles. Pour chaque connexion TCP, reçue, ou demandée, elle va lancer un thread ChatCommunication (décrit ci-dessous) en lui fournissant le socket correspondant.

- ChatCommunication qui permettra d'envoyer des messages sur la connexion établie, et qui va tourner en tant que thread pour récupérer tous les messages reçus sur la connexion TCP. Cette classe devra d'une certaine manière être liée à une interface utilisateur de clavardage pour lui fournir les messages reçus, et pour envoyer les messages écrits.

2.2.La base de données

2.2.1.Description des tables

Nous voulions laisser la liberté à chaque utilisateur de se connecter sur n'importe quelle machine du réseau et de tout de même pouvoir accéder à son historique de messages. De plus, nous ne voulions pas permettre à n'importe qui de pouvoir accéder à l'historique d'une autre personne et d'usurper son identité en utilisant son ordinateur à sa machine à son insu.

C'est pourquoi nous avons décidé de créer une table spécifiquement pour lister les différents comptes utilisateur. Chaque compte est identifié par un identifiant (entier) et un mot de passe (un entier). Nous avons donc pu mettre en place une phase de connexion, préalable au choix du pseudonyme lors du lancement de l'application. Nous pouvons ainsi identifier quelle personne est en train d'utiliser l'application, l'authentifier grâce au mot de passe, et récupérer son historique de messages sur n'importe quelle machine du réseau.

La seule autre table de notre base de données contiendra donc un ensemble de messages, constitués de leur contenu (le message envoyé), la date d'envoi, ainsi que l'identifiant de l'émetteur et celui du destinataire pour pouvoir associer ces messages aux bons utilisateurs.

2.2.2.Base de données locale

Nous avions commencé par implémenter une base de données locale SQLite. Cette base de données avait l'avantage d'être extrêmement rapide, mais impliquait évidemment certains inconvénients et contraintes.

Sachant que chaque machine possédait sa propre base de données, et ne pouvait pas accéder à celle des autres, pour que tous les messages d'une conversation soient stockés, il fallait stocker les messages à l'envoi ET à la réception. Ainsi, chaque machine possédait un exemplaire de chaque message. C'était tout à fait réalisable, et cela restait extrêmement rapide. Cependant, comme dit précédemment, des inconvénients non négligeables étaient générés :

- Pour accéder à son historique de messages, il fallait se connecter sur la même machine à chaque fois, or, c'était une contrainte que nous voulions éviter.
- Nous ne pouvions pas vérifier l'unicité des comptes utilisateur. Un même compte pouvait exister sur deux machines différentes avec un historique de messages différent.

Nous n'aurions même pas pu corriger ces problèmes en mettant en place des échanges entre les bases de données avant toute connexion, car il aurait fallu que toutes les autres

bases de données soient accessibles à tout moment. Or, nous ne pouvons pas demander aux utilisateurs de laisser tourner constamment leur application.

C'est pourquoi nous avons décidé, malgré sa rapidité inégalable, d'abandonner la base de données locale pour préférer une base de données centralisée.

2.2.3.Base de données centralisée

Nous nous sommes tournés, pour la base de données, vers une approche SQL décentralisée. La robustesse du langage SQL contre une base de données noSQL nous a semblé préférable. De plus, l'approche décentralisée nous assure l'unicité de chaque utilisateur, où une approche centralisée n'aurait pas permis cette fonctionnalité et aurait pu mener à des erreurs.

La solution que nous avons adoptée pour ce projet est db4free.com qui offre une base de données SQL en ligne gratuite. La phase de connexion à cette base de données distante se fait au lancement de l'application, ce qui entraîne un délai de chargement. Cette base de donnée n'est pas la solution la plus performante, mais dans le cadre de ce projet c'est la solution que nous avons préféré mettre en place pour sa simplicité d'implémentation et sa gratuité.

Dans le cadre d'une base de données centralisée, l'avantage est que les messages n'ont pas besoin d'être stockés plusieurs fois, ils sont stockés à la réception uniquement. Pour chaque message, l'horodatage est effectué, les utilisateurs concernés sont identifiés, puis le message est sauvegardé dans la base de données. Quand une session de clavardage est lancée, une requête de tous les messages concernant les deux utilisateurs est envoyée pour charger l'historique des messages. La base de données n'a pas besoin d'être consultée à chaque nouveau message.

2.3.Les interfaces utilisateurs

La qualité des interfaces utilisateur est un point important pour maximiser la qualité de l'UX (user experience). Nous avons développé nos vues de la manière la plus simple et épurée possible pour que l'application soit intuitive et facile à utiliser.

Les interfaces utilisateurs que nous avons définies sont les suivantes :

- Phase de connexion :
  - LoadingView
  - ConnectionWindow
  - NicknameFrame
- Menu principal de l'application
  - MainWindow
- Fenêtre dédiée à la session de clavardage :
  - ChatWindow
- Fenêtre dédiée au partage de fichier :
  - SendFileWindow
  - GetFileWindow

LoadingView :

La phase de connexion à la base de donnée étant de 2 à 20 secondes en fonction de la vitesse de connexion Internet, une interface notifiant que l'application est en train de s'initialiser est impératif pour que l'utilisateur comprenne qu'il doive attendre et que l'application est en train de se lancer correctement.

ConnectionWindow :

La phase de chargement terminée, l'utilisateur doit s'identifier. Il peut aussi choisir de créer un nouveau compte. La logique de ces deux possibilités étant très similaire (CREATE ou GET dans la base de données), les regrouper dans la même Frame nous a semblé être la solution la plus simple et user-friendly. En cas d'erreur un message d'erreur s'affiche pour informer l'utilisateur.

NicknameFrame :

Si l'utilisateur s'est correctement identifié, une nouvelle vue se lance pour qu'il sélectionne un pseudonyme. Cette vue est composée d'une simple input, d'un bouton et d'un éventuel message d'erreur si le pseudonyme est déjà utilisé.

MainWindow :

Quand l'utilisateur est identifié et que son pseudonyme est unique parmi les utilisateurs connectés, la vue principale peut se lancer. Elle est composée de la liste scrollable des autres utilisateurs connectés (mise à jour dynamique) et d'une input permettant de changer de pseudonyme. Un fois un autre utilisateur sélectionné dans la liste, l'utilisateur peut choisir de démarrer une session de clavardage avec ce dernier (bouton clavarder), ou de lui envoyer un fichier (bouton partager fichier).

ChatWindow :

Quand un utilisateur lance une session de clavardage, une vue dédiée se lance pour les deux utilisateurs concernés. La vue est composée d'un TextArea listant les messages envoyés, d'une input pour saisir un message et enfin d'un bouton pour envoyer les messages. La textArea est scrollable est contient tous les messages de l'historique dans l'ordre chronologique.

SendFileWindow :

Quand un utilisateur choisit de partager un fichier, la fenêtre SendFileWindow se lance, avec un bouton Sélectionner pour choisir le fichier, et un bouton Envoyer pour envoyer le fichier.

GetFileWindow :

Si un utilisateur est le destinataire d'un fichier, une fenêtre s'ouvre pour lui permettre de télécharger le fichier en cliquant sur le bouton download, ou de l'ignorer en cliquant sur le bouton Fermer.

3.Tests de validation réalisés

Ce projet comportait plusieurs étapes bien distinctes, toutes aussi importantes pour valider l'application générale. Certaines ont pu être réalisées en parallèle pour optimiser l'avancement du projet. Mais, pour nous assurer qu'elles fonctionnent bien par la suite lors de l'intégration au projet final, nous avons dû fixer des attentes minimales pour chacune de ces tâches.

Nous avons pu identifier au total 4 tâches principales. Pour chacune d'entre elles, nous allons définir les attentes que nous avons fixées pour valider leur fonctionnement :

- **Broadcast UDP :**

Nous avons mis en place un thread écoutant les notifications des autres utilisateurs sur un port fixe. Cependant, il est impossible d'ouvrir deux sockets en UDP sur le même port sur une même machine. Or, nous avons dû tester notre application sur une seule et même machine à chaque fois (étant donné que nous ne possédions tous deux qu'un seul ordinateur). Nous pouvions donc ouvrir plusieurs instances de l'application, mais peu importe ce que nous faisions, une seule d'entre elles pouvait récupérer les notifications de connexion, de changement de pseudonyme…). Nous avons donc tout de même vérifié que la première instance lancée de l'application actualisait correctement sa liste d'utilisateurs connectés, même si nous démarrions un grand nombre de fois l'application. Par chance, cette fonctionnalité a également fonctionné par la suite quand nous avons pu nous voir pour tester réellement l'application avec différentes machines.

Affichage console de la première instance de l'application suite à de multiples autres connexions

Le serveur d'écoute a bien reçu tous les broadcasts de demande de connexion et les a tous ajoutés à sa liste d'utilisateurs actifs

```
Listening for messages on port 5000
NotificationListener - message reçu : login_request:jean:2:46938:48683:192.168.1.2:15810
Notification listener : adding user jean to the active user list
NotificationListener - message reçu : login_request:gerard:3:45008:45989:192.168.1.2:48068
Notification listener : adding user gerard to the active user list
NotificationListener - message reçu : login_request:gertrude:4:42596:11286:192.168.1.2:14596
Notification listener : adding user gertrude to the active user list
```

Il nous fallait également vérifier qu'en fonction des messages reçus, la réponse était adaptée. Par exemple, si l'utilisateur reçoit une demande de connexion avec un pseudonyme qu'il utilise déjà, il doit lui répondre que le pseudonyme est déjà pris.

```
isNicknameAvailable - message envoyé : login_request:celestin:3:14986:50408
Broadcast depuis le port 10545
Timeout reached. We suppose that every other user has answered.
isNicknameAvailable - message recu : login_response:0:celestin:1:61772:55619:192.168.1.2:5000
```

Affichage console suite à la connexion avec un pseudonyme en cours d'utilisation

Le "0" dans le message "login_response" signifie que le pseudonyme est déjà utilisé par la personne ayant envoyé ce message

Une fois ce broadcast fait, les autres types de notifications comme le changement de pseudonyme ou la déconnexion étaient relativement simples à implémenter.

- **Communication TCP :**

Pour cette partie, nous n'avions pas autant de contraintes pour réaliser nos tests qu'avec l'UDP. En effet, durant le broadcast UDP qui est fait après le choix de pseudonyme, nous envoyons un port aléatoire sur lequel l'utilisateur sera en écoute pour les demandes de connexion TCP. Ainsi, chaque utilisateur ouvrira (sauf si on a peu de chance) un socket d'écoute sur un port différent. Nous avons donc pu essayer de lancer de nombreuses connexions TCP. Nous avons donc vérifié qu'un thread (ChatCommunication) était bien créé et lancé pour chaque demande de connexion TCP (à l'émission ET à la réception d'une demande).

```
Chat connection received on port 35428
/192.168.1.2 has received a message on port 35428 :
jean

Chat connection received on port 35428
/192.168.1.2 has received a message on port 35428 :
gerard

Starting a chat connection between port 35428 and port 12675(gertrude)
```

Affichage console après le lancement de "ChatManager" pour gérer les connexions TCP pour le clavardage

On peut voir que ce thread a reçu 2 demandes de connexion de jean et gerard et a effectué une demande de connexion avec gertrude

Nous avons également pu vérifier que chacun de ces threads nous permettait d'envoyer de manière asynchrone un message textuel, et réceptionne également de manière asynchrone tous les messages envoyés depuis l'autre côté de la connexion.

```
Sending the message COUCOU to gertrude
Received the message HEY from gertrude
Received the message Hey from jean
Sending the message Ca va ? to jean
```

Affichage console des messages reçus et envoyés de Robert avec Jean et Gertrude

On peut voir que Robert a pu recevoir ou envoyer des messages à Jean ou à Gertrude de manière asynchrone

Une fois ces deux modes de communication terminés, le fond de notre application était relativement fonctionnel ! Mais il nous restait toute la forme à mettre en place, sans laquelle notre application ne serait absolument pas déployable au sein de l'entreprise.

- **Création de toutes les interfaces utilisateur requises :**

Les tests ici étaient relativement simples, il nous suffisait de créer toutes les fenêtres nécessaires (connexion/inscritpion, choix de pseudonyme, fenêtre principale, et fenêtre de clavardage) et de permettre le passage de l'une à l'autre avec certaines fonctions précises fournissant les informations nécessaires. Par exemple, pour passer de la fenêtre de choix de pseudonyme à la fenêtre principale, il fallait lui fournir l'utilisateur courant. Ou alors, pour ouvrir une fenêtre de clavardage depuis la fenêtre principale, il fallait lui fournir l'autre utilisateur afin d'afficher son nom avant ses messages. Le plus difficile avec les interfaces utilisateurs était l'intégration. En effet, il fallait bien penser à effectuer toutes les vérifications, appeler les bonnes fonctions au bon moment pour éviter les incohérences, et aussi penser à fermer tous les sockets et arrêter tous les threads lors de la fermeture de certaines fenêtres afin d'éviter des bugs.

- **Configuration d’une base de données fonctionnelle :**

L’utilisation de la base de données étant simple (création/lecture d’user et de messages), les tests ont été simples également. Les tests pour la table User ont été les suivants :

- Création de 4 User avec les ID 1, 2, 3, 4 et mot de passe 1234.
- Tentative de création d'utilisateurs déjà existants dans la base de données.
- Tentative de création d'utilisateurs avec ID et/ou mot de passe invalides.
- Connexion aux comptes des utilisateurs enregistrés.

Les tests pour la table Message ont été les suivants :

- Création de Messages.
- Créations de Messages vides.
- Lecture de Messages.
- Chargement d’un historique (vérification de l’ordre temporel des messages et de la cohérence des sender/receiver).

4.Manuel d’utilisation

Thalès clavardage est une application de clavardage, utilisable sur un réseau local.

Installation

Télécharger le fat JAR chatApp.jar sur la page github dédiée au projet :

https://github.com/JulienRZ-Dev/chat_project.git

Usage

Ouvrir le jar pour lancer l'application. Les étapes de connexion sont les suivantes :

- Créer un compte (id, mot de passe). Si l'id est déjà utilisé, un message d'erreur sera affiché. l’id et le mot de passe doivent être des nombres entiers (int). A savoir que les ids 1, 2, 3 et 4 ont été utilisés pour nos tests avec le mot de passe 1234. Si vous voulez accéder rapidement à un long historique de messages, vous pouvez utiliser les ids 1 et 2.
- Choisir un pseudonyme pour la session active. Le pseudonyme peut changer entre les sessions, mais il doit être unique sur le réseau local. Si le pseudonyme choisi est déjà utilisé, un message d'erreur sera affiché.
- Sélectionner un autre utilisateur et cliquer sur Clavarder pour démarrer une session de clavardage. L'historique des messages est automatiquement chargé, si un des utilisateurs quitte la fenêtre de clavardage ou l'application elle-même, l'autre utilisateur est notifié.
- Sélectionner un autre utilisateur et cliquer sur Partager fichier pour lui envoyer une image, schéma,... Le bouton Sélectionner permet de choisir un fichier dans l’explorateur de fichier et le bouton envoyer permet d’envoyer le fichier sélectionné. Si un autre utilisateur vous envoie un fichier, un pop-up apparaît, sélectionnez Télécharger pour télécharger le fichier et Ignorer pour ignorer le fichier.

License