Rapport Projet Réseaux STOMP

Enzo CISTERNINO - César TONDON - François XU

Table des matières

Introduction

Voici le projet final du cours intitulé “Réseaux informatiques” avec M. Bertrand.

Le but de ce projet est d’implémenter le protocole STOMP au sein d’un environnement “Client/Serveur” tout en ayant une solution créative et innovante afin de tester ce protocole, le but étant de se différencier d’un simple chat textuel.

Ainsi nous avons décidé de reprendre le concept de Reddit avec r/place, nous expliquerons cette idée dans la suite du document.

Dans ce document sera traité de :

- La répartition du travail dans l’équipe
- De l’architecture du projet
- Des choix techniques
- Des difficultés rencontrées et des solutions apportées
- D’une description détaillée de l’implémentation (ce qui a été implémenté et comment, ce qui n’a pas été implémenté)
- De tests du projet
- Et enfin d’une conclusion

Répartition du travail dans l’équipe

Dans notre fonctionnement du projet, nous avons divisés le développement en 3 parties :

- La programmation et l’implémentation du protocole STOMP
- La programmation de l’implémentation JavaFX du jeu type R/Place
- La documentation du projet et la création du rapport
- Les tests du projet

(Glarell = César TONDON) (EnzOuille = Enzo Cisternino) (ChineThug = Francois XU)

Répartition du travail :

- Enzo s’est majoritairement occupé du protocole STOMP, et César a aidé Enzo à résoudre les problèmes et finaliser l’implémentation de ce protocole.

- César s’est occupé entièrement de la modélisation et de la programmation du jeu type R/Place.

- Enzo s’est occupé de la création du rapport et César s’est occupé de la documentation du code

- Enzo et César se sont occupés de tester le bon fonctionnement du code à chaque nouvelle fonctionnalité ou grande étape du développement réalisée.

Architecture du projet

Le projet est découpé en trois parties :

- La partie JavaFX
- La partie network / STOMP
- La partie serveur

I - La partie JavaFX

Cette partie répertorie tout le jeu et sa conception. Elle regroupe le Controlleur FXML et les différents composants permettant de créer un client STOMP vers le serveur.

On a donc le fichier Button.Java qui représente le composant Button de JavaFX, cela nous permet via un constructeur de créer votre bouton et de l'associer directement à une position dans la grille de notre jeu et de lui attribuer une couleur par défaut.

Cela simplifie notamment le fonctionnement de notre application et surtout la réception des messages depuis le serveur qui permettra facilement de modifier notre grille en conséquence.

Le controller lui met en place les listeners sur la grille et crée notre client STOMP qui permet notamment de se connecter au serveur STOMP, de s'abonner, et de se désabonner et d'envoyer des messages. Pour chaque "MESSAGE" on va donc pouvoir parser le contenu des messages et récupérer les différentes cases de la grille qui doivent être modifiées.

II - La partie Network / STOMP

Cette partie permet d'implémenter le protocole STOMP, cette partie permet de gérer les différentes trames envoyées par le client et d'implémenter les files d'attentes / Queues ainsi que l'envoi des messages en fonction des abonnements aux différentes clients.

Avoir travaillé comme ça nous permet d'avoir toute l'implémentation de notre protocole STOMP au sein d'un même fichier et ainsi de pouvoir centraliser tout notre code.

C'est ce fichier qui doit être lancé afin d'avoir notre serveur STOMP avec qui communiquer.

III - La partie serveur

Nous avons décidé d'utiliser Tyrus afin de pouvoir avoir un serveur HTTP gérant les websockets de notre implémentation STOMP.

Cela nous permet d'avoir tout simplement une écoute sur un port en continu acceptant les connexions des différents websockets arrivant.

Les choix techniques

Nous avons dû faire plusieurs choix quant à notre implémentation, dans un premier cas nous avions décidés d'écrire le serveur et le client en React.Js avec Node.Js, hors nous avons mis énormément de temps sur la configuration et nous avons décidés de ne pas perdre de temp avec cela et plutôt de partir sur une implémentation en Java, où nous avons pris seulement une heure à configurer le projet.

I - Java WebSocket API

Nous avons utilisé l'Api Java Websocket pour notre projet, c'est une librairie qui nous permet d'avoir des websockets et plus particulièrement des callbacks comme (OnMessage, OnOpen, OnError, OnClose, etc.…) et ainsi pouvoir tout gérer.

Cela nous a aussi permis de pouvoir créer des EndPoint (des routes) afin d'amener nos différents utilisateurs aux bons endroits.

II - Tyrus Server

Tyrus est une librairie nous permettant de créer un serveur HTTP en Java en écoute sur un port et de pouvoir lui associer des services. Dans notre cas, nous avons créé notre Tyrus Serveur avec "MainServerEndpoint" qui nous permettait de gérer toute l'implémentation de notre protocole Stomp avec un websocket.

III - Maven

Maven est un gestionnaire de dépendances en Java, nous l'avons utilisé pour notre projet.
Cela nous a permis notamment de bien spécifier nos dépendances et cela nous permettra par la suite de pouvoir exporter notre projet en l'ayant bien formaté et en ayant généré des fichiers exécutables pour lancer notre projet.

IV - Javafx

Nous avons utilisé JavaFX pour représenter notre client de test et l'implémentation de notre jeu. Cela nous a permis d'avoir une interface ergonomique et rapide à développer plutôt que de devoir créer une application retournant des pages Web.

Je tiens néanmoins à préciser que même si nous avons utilisé JavaFX, la communication entre nos applications est entièrement faite en Websocket, nous n'avons pas fait de lien entre les objets de la partie Serveur et les objets de la partie Client.

Les difficultés rencontrées et leurs solutions

I - Installation et Configuration de React.js

Dans un premier temps, nous avions décidé d'effectuer le projet avec Node.Js et Reactjs, hors nous avons très rapidement rencontré des problèmes de configuration du projet et nous n'arrivons pas à arriver à une version satisfaisant nos besoins et permettant d'avoir un projet soigné.

Après une perte de motivation et de temps, nous avons décidé de partir sur le langage Java que nous connaissions très bien et cela nous a permis d'avancer beaucoup plus rapidement sur le projet.

II - Programmation concurrente

Nous avons très rapidement rencontré des problèmes de programmation concurrente entre les différents threads générés par notre application et notamment par l'écoute en continu de nos websockets et le programme en continu de lancement de JavaFx.

Aussi nous avons remarqué que lorsque nous créons plusieurs clients, ces clients modifiaient tous la même grille et ainsi posaient de gros problèmes de fonctionnement pour notre projet.

Pour cela nous avons réuni le Controller et le Client au sein d'une même classe et nous avons modifié l'instance passé à la connexion au serveur. Car au départ nous instancions la connexion avec un objet généré par l'api, alors que maintenant nous générons l'objet afin de toujours accéder à l'objet demandé et non pas à une autre référence non voulue.

III - Mise en place de l'idée créative

Comme expliqué, notre idée était de créer une grille que les utilisateurs pourraient colorier et où les modifications de la grille apparaîtraient pour tous les utilisateurs connectés et abonnés à la Queue correspondante.

Pour cela nous avons créé une grille avec JavaFX reliée à un contrôleur qui lui-même possède la session du websocket et ainsi le parsing des messages permettant de mettre à jour la grille.

Ainsi nous avons bien notre client qui se connecte au serveur STOMP, s'abonne à une queue, envoie ces modifications de la grille sur cette queue et reçoit les modifications à son tour.

Description détaillée de l’implémentation

I - Implémentation du serveur STOMP

Introduction

Afin d’implémenter le serveur et le protocole STOMP, nous avons utilisé comme précisé ci-dessus la librairie Java Web Socket.

Cela nous permet d’avoir un ServerEndpoint (“/main{username}”) qui fera office de serveur pour tout le projet.

Grâce à la librairie, nous pouvons programmer des CallBack qui permettent d’effectuer des actions en fonction d'événements, par exemple afficher un message quand on reçoit un message de clients.

Nous avons bien fait la différence entre une connexion de WebSocket et une connexion STOMP, en effet un client peut se connecter au serveur via un websocket mais ne pas être connecté avec le protocole STOMP, c’est pour cela que nous avons mis en place les différentes FRAMES qui sont en lien avec STOMP.

Structures de données

Avant d’expliquer le fonctionnement du protocole et comment nous l’avons implémentés, je tenais à expliquer quelles sont nos différentes structures de données nous permettant de stocker les différentes informations nécessaires au serveur.

Tout d’abord nous avons “serverEndpoints”, qui regroupe tous les endpoint côté serveur, car on pourrait également créer deux serveurs STOMP par exemple pour paralléliser la charge ou tout simplement pour faire des sauvegardes, dans notre application nous avons décidés de n’utiliser qu’un seul serveur.

Ensuite nous avons “users”, cette structure comprend pour chaque indice de session, l’id textuel que le client a fourni lors de la connexion.

“Users_sessions” : pour chaque id textuel, nous avons la session correspondante
“Users_connected” : pour chaque id textuel, nous avons la confirmation ou non que le client est bien connecté avec le protocole STOMP.
“Users_subscibes” : pour chaque id textuel, nous avons la liste des abonnements des clients (à savoir qu’un abonnement est représenté par une destination, un id et un curseur qui indique la position de l’abonnement dans la queue)
“Queues” : qui représente toutes les queues présentes au sein du serveur.

Fonctionnement

Maintenant que nous avons expliqués les différentes structures de données permettant le fonctionnement du serveur, passons aux méthodes et à la programmation du serveur.

Comme précisé précédemment, nous avons utilisé les callback de la librairie, et ainsi nous avons pu séparer les différentes actions.

Voici le comportement de chacune des actions :

- OnOpen : on ajoute à toutes les structures les informations du nouveau Client
- OnClose : on supprime de toutes les structures les informations du client qui se déconnecte
- OnMessage : on vérifie si l'utilisateur est connecté au niveau STOMP :
    - Si non : On vérifie que la trame qu'il a envoyé est bien une frame CONNECT valide, si c'est le cas on valide la connection et il peut envoyer d'autres frames, sinon on envoie une erreur et le client se déconnecter lui-même
    - Si oui : On connecte l'utilisateur au serveur STOMP et il peut donc effectuer les actions "SUBSCRIBE" "UNSUBSCRIBE" "SEND" via les trames
    - Dans le cas d'une trame "SUBSCRIBE" :
        - On vérifie que la trame est valide
        - Si la trame n'est pas valide alors on envoie une erreur
        - Si la trame est valide, on vérifie que la queue "destination" existe :
            - Si non, alors on crée la queue à partir de zéro et on ajoute l'abonnement
            - Si oui, alors on ajoute l'abonnement à la liste des abonnements
            - On place l'abonnement au premier élément de la queue
            - On envoie les messages de la queue au nouveau client afin qu'il se mette à jour
    - Dans le cas d'une trame "UNSUBSCRIBE":
        - On vérifie que la trame est valide
        - Si l'id de l'abonnement existe alors on le supprime
        - Sinon on envoie une erreur
    - Dans le cas d'une trame "SEND":
        - On vérifie que la trame est valide
        - On vérifie que la destination existe (Si elle n'existe pas, on la crée)
        - On ajoute le body de la trame à la fin de la queue
        - On appelle la méthode "UpdateQueues" qui s'assure que tous les clients sont à jour dans la queue (s'assurer que le curseur de l'abonnement est égal à la taille de la queue)

II - Implémentation du Jeu (en JavaFX)

Dans le cadre où nous avons mis en lien le controller et le client en même temps, les messages que reçoit le client à partir de son abonnement doivent être parsés pour récuperer les bonnes informations et notamment les informations de transitions d'une case de la grille à l'état noir ou blanc.

Le client est fait de manière à automatiquement vouloir se connecter en STOMP à la connexion du Websocket et à se subscribe une fois connecté, cela permet aussi de ne pas avoir d'erreurs car nos trames sont valides.

A chaque clic sur une case et si le client est bien abonné, alors il envoie un message au serveur qui lui transmet aussitôt la réponse afin de pouvoir modifier la grille (nous avons décidé que les utilisateurs pouvaient modifier la grille s'ils étaient abonnés).

Cela permet par exemple de pouvoir se désabonner, que des clients fassent des modifications et que l'on puisse se réabonner et tout de même avoir les informations que nous n'avions pas obtenues lors de notre désabonnement.

III - Implémenté ou pas ?

Nous avons implémenté l'ensemble des fonctionnalités demandées par défaut dans le sujet "Analyse des trames SUBSCRIBE, UNSUBSCRIBE, CONNECT, DISCONNECT, MESSAGE, ERROR" mais nous n'avons pas implémentés les FRAMES ACK et NACK ainsi que l'authentification ni le heartbeat, etc….

Test du projet

Tutoriel

Afin de pouvoir utiliser notre projet, il faut avoir installer Java 11 sur votre machine.
Voici les commandes appropriées (il faut se situer dans le dossier contenant l'archive)

```
Pour lancer le serveur :"%JAVA-PATH%\java -jar stomp-1.0.0.jar server"
Pour lancer un client : "%JAVA-PATH%\java -jar stomp-1.0.0.jar client".
```

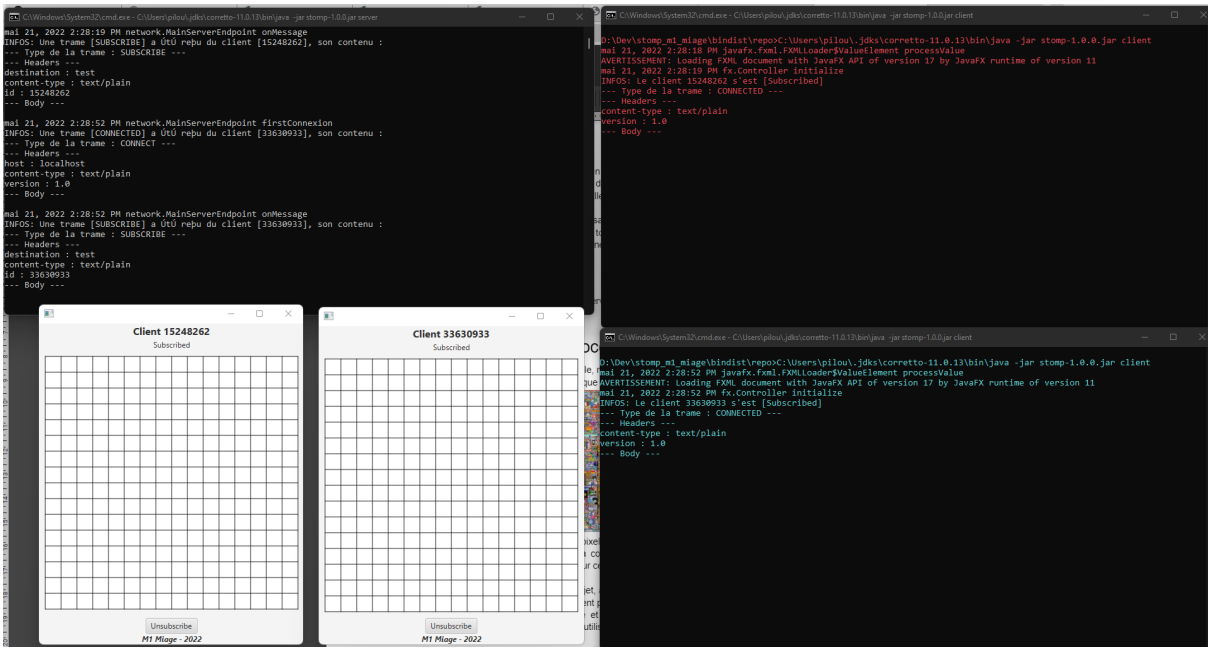
Ainsi vous aurez votre serveur dans un invite de commande, vos clients dans des invités de client mais également avec pour chaque une interface graphique représentant le jeu.

Test

Nous avons testé la création des trames client et server, à l'aide de tests Junit.

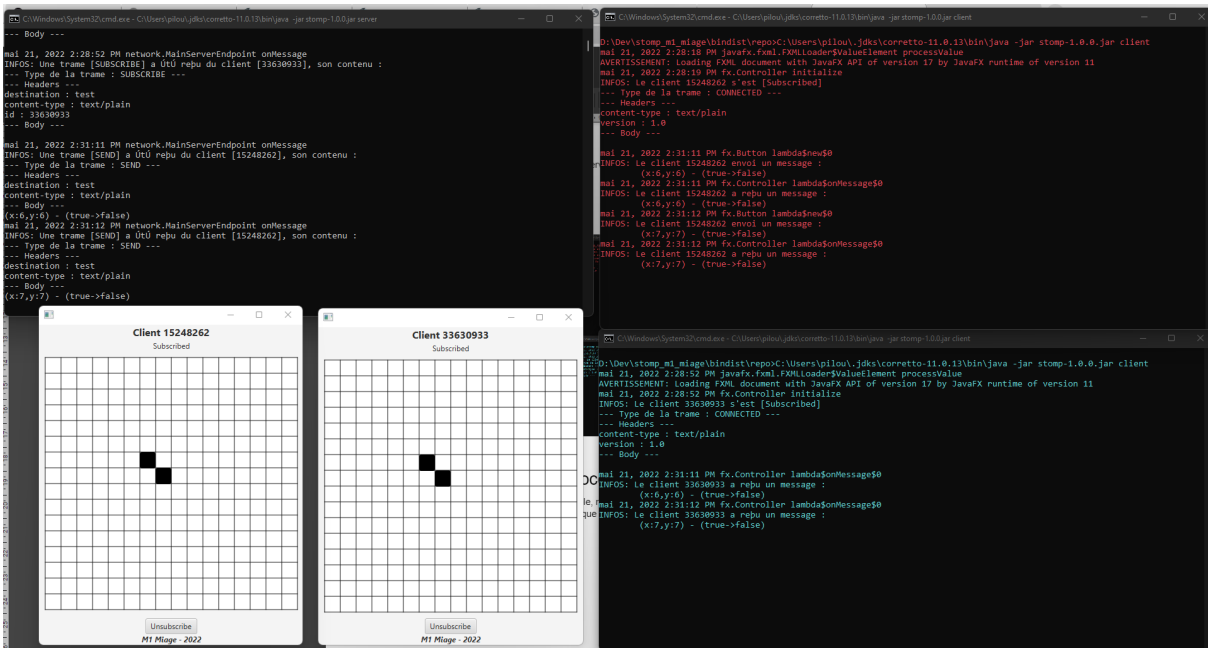
Initialisation :

- Server (Blanc)
- Client 15248262 (Rouge)
- Client 33630933 (Bleu)



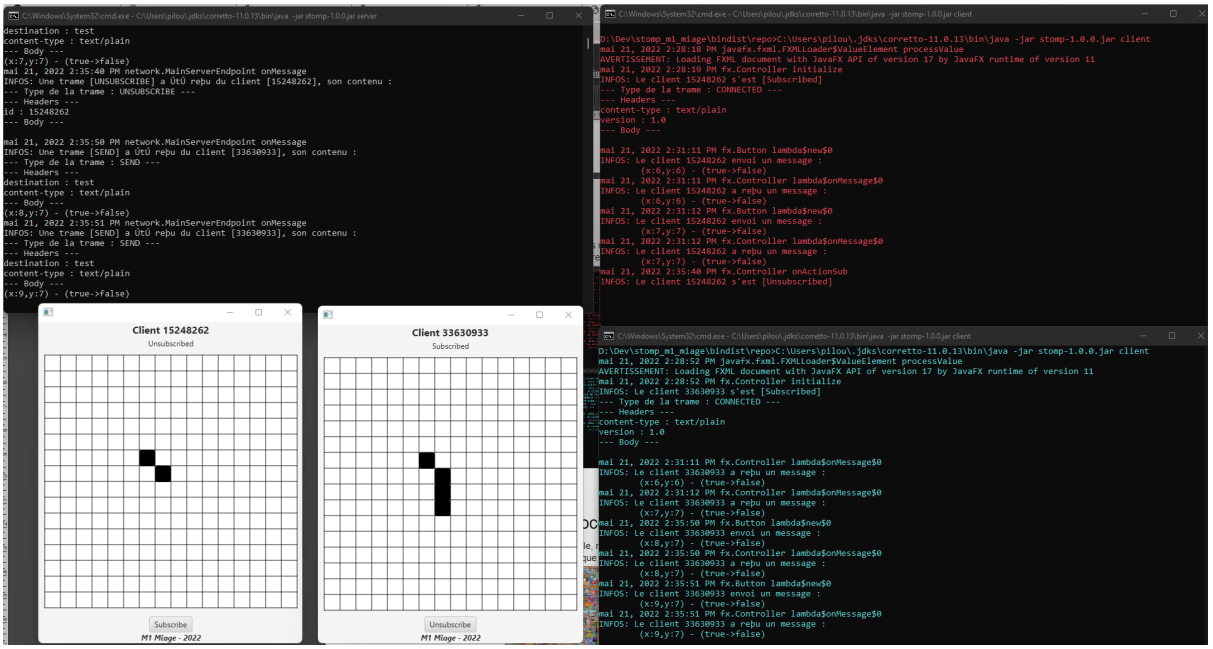
Test envoi message (Client est abonné) :

- Le client Rouge envoi deux messages dans une queue
- Le client Bleu est abonné à cette queue et reçoit les messages



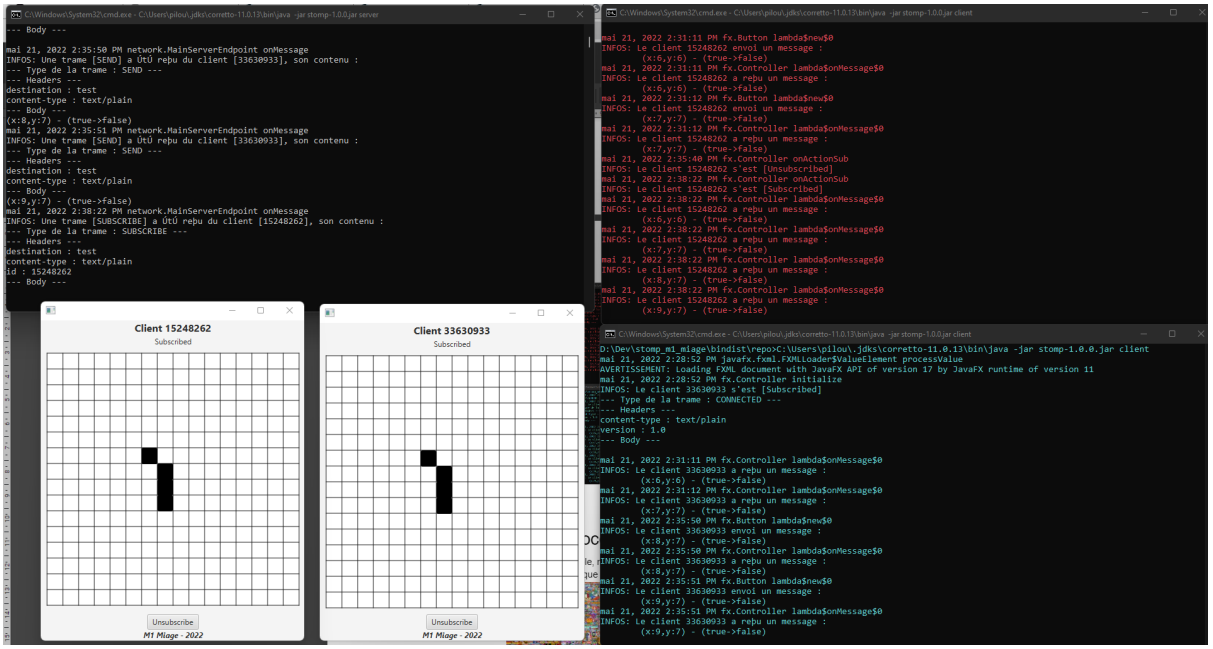
Test envoi message (Client n'est pas abonné) :

- Le client Rouge s'est désabonné, il ne peut plus émettre de messages
- Le client Bleu peut envoyer et recevoir des messages
- Le client Rouge ne reçoit plus les messages envoyés



Test abonnement :

- Client Rouge s'est abonné à nouveau
- Il reçoit tous les messages envoi pendant son absence
- Il peut émettre et recevoir à nouveau des messages



Utilisation créative du protocole

Afin d'utiliser de manière créative notre protocole, nous avons décidé de reprendre l'idée de Reddit avec le r/place, si vous ne savez pas ce que c'est, voici une brève explication :

Reddit a mis en place un carré de 8000*8000 pixels à la portée des collaborateurs, chaque utilisateur pouvait ainsi colorier un carré de la couleur de son choix toutes les 5 min, et voyait les modifications des autres utilisateurs sur cette toile.

On peut donc transposer ce petit jeu à notre projet, avec un carré de plus petite taille (16*16) et moins de couleurs. Cela permettrait notamment pour chaque client de pouvoir ajouter à la queue (publish) leur modifications de la grille et pour le serveur de pusher à tous les subscribers les modifications de tous les autres utilisateurs.

Ainsi on aura une utilisation du protocole STOMP implémentée avec une solution créative.

Conclusion

Etant donné que Enzo avait déjà travaillé avec Kafka et RabbitMQ, il a pu comprendre et expliquer le principe du protocole STOMP à César, qui lui aussi en avait déjà entendu parler et a donc rapidement pu comprendre le concept.

Mis à part le petit désagrément avec Reactjs nous avons su très rapidement partir sur de bonnes bases avec Java et ainsi réussir le projet. Mettre en place le protocole nous a permis de comprendre les échanges entre un serveur et un client et notamment toutes les étapes de discussions mais également tous les cas extrêmes qui peuvent mettre à mal un serveur.

C’est surtout très satisfaisant de pouvoir faire une application qui ne communique pas via des objets mais par le réseau, qui même en Master, reste quelque chose de mystique et dangereux pour nous.

Pouvoir reproduire à une échelle beaucoup plus petite un concept que nous avons aimé nous a motivés à réussir le projet et à nous donner à fond.