Rapport de projet : Coincoin Christmas Edition

Dennis Bordet, Benjamin Moreau, Thibault Béziers-la-Fosse,
M2 ALMA

15 décembre 2016

Sommaire

1 Introduction

Dans le cadre du Master 2 ALMA, nous avons eu pour projet de développer une application en utilisant l'API de communication RMI. Cette application, Coincoin permet à plusieurs utilisateurs de participer à une enchère, afin d'acheter des canards vivants.

Dans ce compte-rendu vous trouverez un descriptif de l'architecture que nous avons mis en place, ainsi que les fonctionnalités de notre applications, mais aussi les technologies que nous avons utilisées, une explication sur les choix que nous avons fait, les problèmes rencontrés, et enfin les conclusions que nous avons tirées de ce projet.

2 Architecture du projet

Nous avons décidé de séparer le projet en trois modules : le client, le serveur et un dernier module qui sert à partager des informations pour les deux autres modules comme par exemple les interfaces comportant les méthodes utilisées via RMI.

2.1 Client

Le module du client est une application web, il est donc divisé en deux parties : le BackEnd et le FrontEnd.

- Le BackEnd représente la partie RMI du client qui communique avec le serveur.
- Le FrontEnd est l'interface exploité par l'utilisateur qui interroge le BackEnd régulièrement pour récupérer les informations du serveur et enchéri sur les produits.

2.2 Serveur

Le module serveur contient les méthodes d'interactions avec la base de donnée, la base de donnée elle-même (juste une liste de JSON) et bien évidemment, la classe serveur qui implémente les méthodes que le client connaît via le package commun. C'est lui qui choisi les produits à mettre en vente, et communique avec les clients par le biais de RMI.

2.3 Common

Ce module est partagé par les deux autres, il contient tout ce qui est nécessaire pour que la communication soit possible, notamment les interfaces. En passant par un module commun, on évite la redondance au niveau des interfaces communes au Client et au Serveur.

2.3.1 Les interfaces partagés

Il faut que le client connaisse les méthodes du serveur qu'il voudrait appeler et inversement.

Le client possède trois méthodes que le serveur peut utiliser :

- nouvelle_soumission : envoie le nouvel item en vente
- objet_vendu : prévient que l'objet courant est vendu (encore inutilisé puisque actuellement, les objets ne sont vendus uniquement à la fin du temps imparti)
- nouveau_prix : envoie le nouveau prix de l'objet au client, ainsi que la personne qui à émit cette proposition.

Le serveur lui, possède quatre méthode accesibles par le client :

- insc_acheteur : permet au client de s'inscrire sur le serveur de vente
- rencherir
- temps_ecoule : le minuteur de la vente coté client est terminé et en informe le serveur
- quitter : informe le serveur que le client quitte la vente.

2.3.2 DTO

Afin de passer des paramètres dans les méthodes partagées, et appelées par le biais de RMI, nous avons décidé d'utiliser des Strings uniquement, que nous sérialisons ou désérialisons en passant par des Data Transfert Object.

Effectivement, les Strings peuvent être très facilement passées en paramètre lorsque l'on appelle des méthodes avec RMI, et on évite ainsi les possibles problèmes lors de la désérialisons avec l'interface **Serializable**.

On trouve aussi des variables communes comme le port de connexion RMI.

3 Fonctionnement

3.1 Général

3.1.1 Inscription (Serveur)

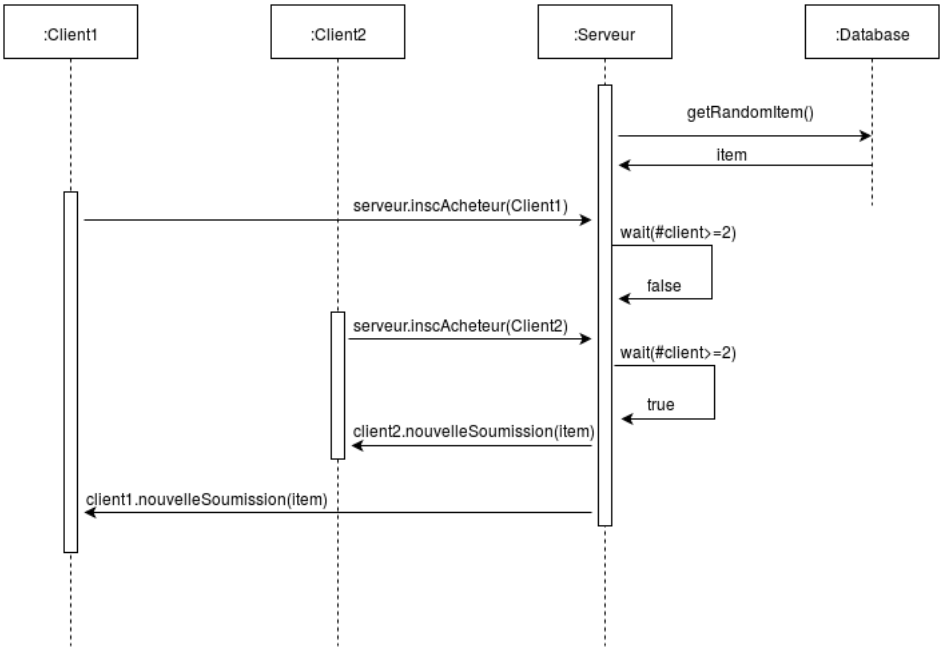

FIGURE 1 – Nouvelle Soumission

3.2 Méthodes Synchronized

3.2.1 Renchérir (coté client)

On imagine une situation où l'objet en vente vaut 180€.

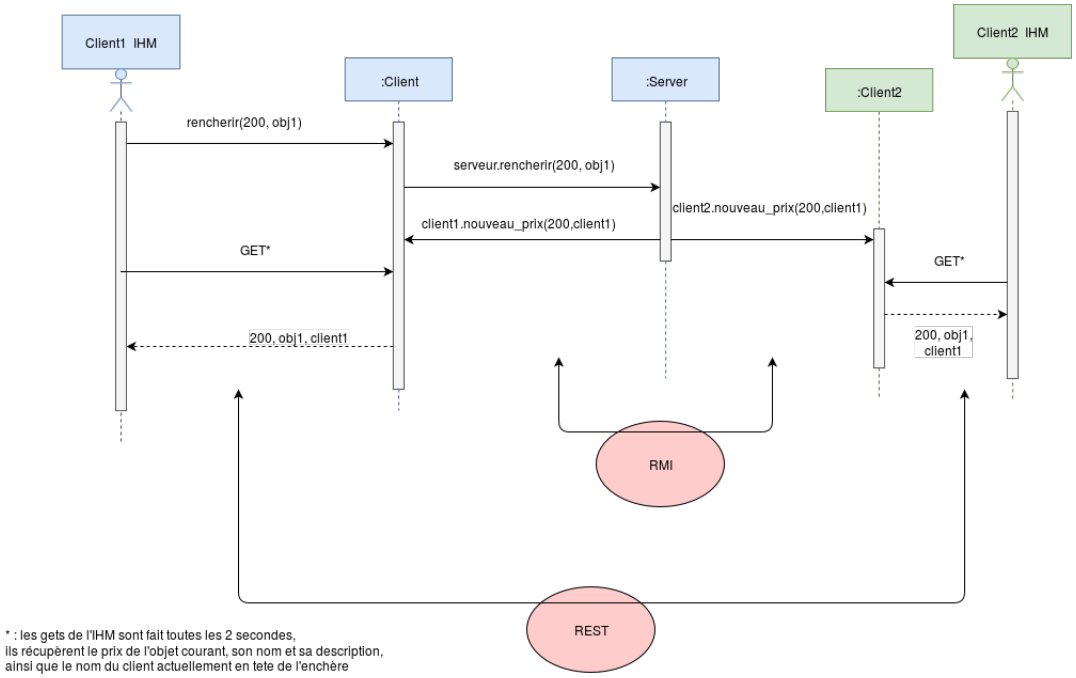

FIGURE 2 – Fonctionnement de Enchérir

Lors de la méthode de renchérissement, nous avons choisi d'envoyer le nouveau prix désiré, au lieu de la somme à enchérir. Par exemple, si l'objet en vente est au prix de 180€, on envoie **200** et non **"+20"**.

Effectivement, une première raison est que ça se rapproche plus d'une vraie enchère, où les participants donnent leur nouveau prix, et pas la somme qu'ils ajoutent.

La seconde raison est que ça évite de nombreux problèmes. Par exemple :

Un objet est en vente à 180€. Je veux ajouter 20€, mais entre temps quelqu'un a ajouté 50€. Si bien qu'au lieu de mettre un prix à 200€, je me retrouve avec une enchère à 250€.

Ainsi avec notre système actuel, les participants envoient le prix qu'ils désirent payer pour l'objet. Si un prix supérieur est arrivé au serveur entre-temps, le serveur ne prend pas en compte le prix inférieur. Comme ça, aucune mauvaise surprise pour les acheteurs.

3.2.2 temps_écoulé (coté client)

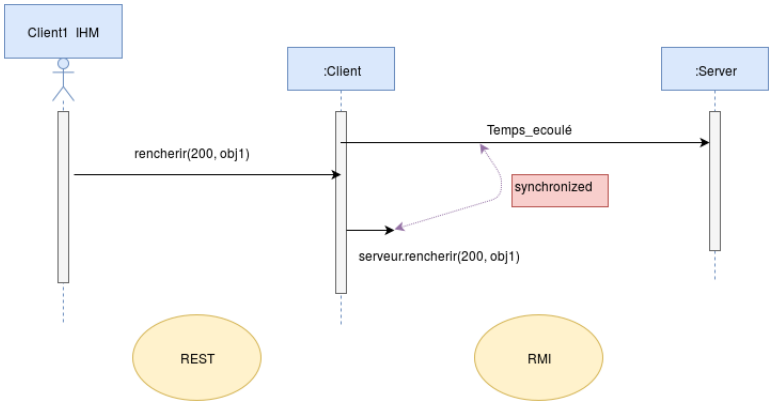

FIGURE 3 – Fonctionnement de TempsEcoulé

3.2.3 insc_acheteur

Les inscriptions commencent sur l'IHM, l'utilisateur donne son pseudo, son mail et choisit son serveur de vente. A cet instant, l'utilisateur est inscrit dans le registre de RMI et le serveur est stocké dans un attribut du client.

L'inscription à l'enchère est l'étape qui suit, le client envoie son nom et une référence sur lui même pour que le serveur puisse le stocker dans ses attributs. La fonction insc_acheteur se découpe en trois parties, selon le nombre d'utilisateurs dans l'enchère car nous avons définit un nombre minimal de clients pour lancer les enchères (2 pour le schéma ci-dessous).

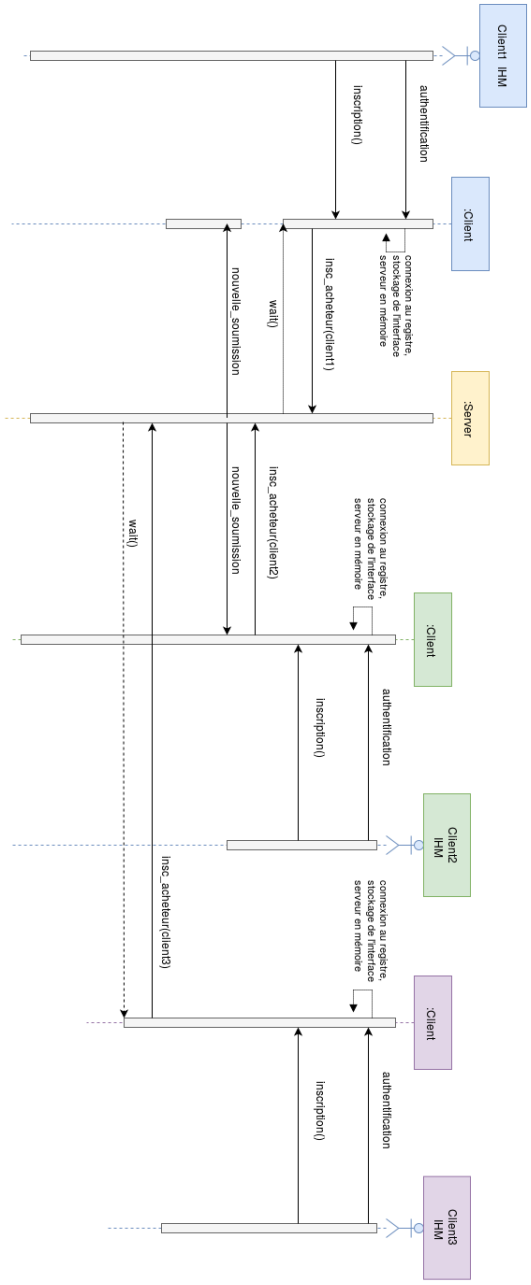

FIGURE 4 – Fonctionnement de insc_acheteur

Nombre d'utilisateurs insuffisant

Dans ce cas, le client qui se connecte est mis en attente.

Nombre d'utilisateurs tout juste bon

C'est donc le dernier client nécessaire à la vente qui vient de se connecter, il va donc réveiller tous ceux qui attendaient ainsi que la fonction initiate_sell.

Nombre d'utilisateurs supérieur à la limite

Cela veut dire que la vente est déjà commencée. Chaque client arrivé après les n premiers requis est mis en attente. En effet, il sera réveillé à la fin de la fonction de vente, il pourra ainsi participer à la vente de l'objet suivant.

3.2.4 initiate_sell

C'est la fonction qui est lancée à chaque vente d'objet. Le serveur envoie le nouvel objet à tous les utilisateurs inscrits et réveille les utilisateurs qui sont arrivés en cours de vente.

4 Limites du projet

Au cours du développement de ce projet, nous avons trouvé plusieurs fonctionnalités qui auraient pu être intéressantes, mais que nous n'avons pas fait. Ces fonctions sont :

- Un système permettant de retrouver sa session en cas de départ.
- De multiples salons d'enchères.
- Un historique d'achat.
- Un connexion fonctionnant autrement qu'en LocalHost, chose que nous n'avons pas fait à cause du Proxy de l'université (Bien que logiquement, ça fonctionne par IP).

5 Conclusion

RMI est un moyen assez efficace pour communiquer en réseau, bien plus simple que gérer des sockets à la main. Il est orienté objet, facile à mettre en place.

Toutefois, aujourd'hui la communication via API-REST est beaucoup plus utilisée. Cependant, d'un point de vu pédagogique, RMI reste une alternative intéressante pour mettre en place des algorithmes concurrentiels. La concurrence étant surtout gérées par des Frameworks comme Spring ou JaxRS dans le cadre des API-REST.

6 Annexe

6.1 Comment utiliser Coincoin ?

Pour utiliser notre application, vous devez d'abord lancer le **main** présent dans la classe Application du module Serveur.

Ensuite, pour chaque instance de Client désirée, lancer un serveur d'application (Tomcat, Jetty ...), sur un port différent à chaque fois.

Ouvrez ensuite les clients dans votre navigateur, à l'url **localhost :8080**. Remplissez les champs pour vous authentifier, avec **localhost** comme URL du serveur.

Les enchères débuteront dès qu'il y aura suffisamment d'utilisateurs de connectés. Le temps d'enchère peut-être modifié dans la classe CommonVariables du module Common, il est par défaut fixé à 15 secondes.