Rapport de projet Middleware
Application distribuée de ventes aux enchères en ligne "Pay 2 Bid"

Arnaud Grall, Alexis Giraudet et Thomas Minier
Master 2 ALMA

3 Décembre 2016

Table des matières

Table des figures

Introduction

Le parallélisme en informatique est un procédé de calcul consistant à faire exécuter les tâches d'un programme informatique en parallèle les unes des autres. Ce paradigme est devenu de plus en plus dominant dans le monde de l'informatique avec l'accroissement de la puissance de calcul des ordinateurs et la naissance des processeurs multi-coeurs. Il présente des avantages indéniables : en réécrivant des programmes pour qu'ils s'exécutent en parallèle, on peut améliorer leur vitesse d'exécution de manière significative. Néanmoins, ce paradigme ne présente pas que des avantages et nécessite de repenser la manière de construire des applications lorsqu'il est utilisé.

Une application distribuée est une application parallèle dont l'exécution est répartie entre plusieurs machines. Pour modéliser une telle application, il faut la modéliser au travers d'un modèle de calcul qui permet de donner une vision simplifiée du système distribué. Ce modèle doit pouvoir englober le plus de situations possibles sans perdre de vue la réalité. On construit donc une abstraction d'un système de manière à en donner une représentation plus homogène.

Pour construire un tel modèle, il faut le penser le plus simplement possible, car un surplus de complexité est souvent superflu lors du développement d'une application répartie. Il faut également définir une synchronisation des différents acteurs (ou processus) de l'application, afin d'interdire les entrelacements pouvant entraîner des pertes de performances et des erreurs. Un mécanisme de synchronisation efficace dans ce genre de situation est un **moniteur de Hoare**, permettant de synchroniser une ou plusieurs tâches qui utilisent des ressources partagées.

Dans ce projet, il nous a été demandé de réaliser une application distribuée modélisant un système d'enchères en ligne. Cette application devra être réalisée sur le modèle client-serveur, où plusieurs clients communiquent entre eux par le biais d'un serveur. Elle sera réalisée en Java, avec le support du framework Remote Method Invocation, ou RMI, pour la gestion de la partie réseau. Dans ce rapport, nous analyserons d'abord les spécifications de l'application, puis nous évoquerons les problématiques liées à un modèle asynchrone ainsi que le modèle de calcul que nous utiliserons pour l'application. Enfin, nous détaillerons notre implémentation avant de conclure sur les potentielles améliorations.

1 Analyse des spécifications

1.1 Spécifications du problème

Nous souhaitons donc créer une application permettant d'effectuer des ventes aux enchères. Pour commencer, il nous faut définir les spécifications du problème.

Une vente aux enchères est un mécanisme de négociation au travers duquel un tiers met en vente un objet suivi d'une phase où plusieurs acheteurs sont mis en concurrence pour déterminer le prix, ou *bid*, auquel sera vendu l'objet, le tout dans un laps de temps limité. Afin de simplifier les choses, nous nous plaçons dans un contexte où les règles suivantes s'appliquent :

- Les clients ne peuvent pas s'inscrire si une vente est en cours. Ils sont mis en attente et, une fois la vente terminée, ils sont automatiquement inscrits pour la prochaine vente à venir.
- Une vente se déroule par round. À chaque round, chaque participant fait une enchère. Lorsque tous les participants ont enchéri ou que le temps imparti s'est écoulé, le round se termine. Un nouveau round se lance alors, avec comme nouveau prix la meilleur enchère du round précèdent. Une vente s'arrête lorsqu'aucun participant n'a surenchéri pendant un round, et le vainqueur est celui qui a effectué la plus haute enchère.
- Un participant ne peut pas quitter une vente lorsqu'elle est en cours.
- Un seul objet peut être vendu à la fois, il n'est pas possible d'avoir plusieurs ventes en simultané.

Dans le contexte de notre application, les participants d'une vente seront représentés par le client, et le coordinateur de l'enchère sera représenté par le serveur.

1.2 Modèle de calcul adopté

Notre modèle de calcul sera découpé de la manière suivante. Pour effectuer des actions, le client appellera des méthodes publiques du serveur et, inversement, le serveur coordonnera les clients en utilisant leurs méthodes publiques. Chaque acteur n'a donc conscience que du strict minium requis pour interagir avec le reste du système.

Spécification du Client/Acheteur Un acheteur est donc un des deux acteurs de notre système. Il expose les méthodes suivantes :

- **nouvelleSoumission** : le serveur appelle cette méthode pour indiquer au client qu'un nouvel objet vient d'être mis en vente et lui transmet l'objet en question. L'appel à cette méthode marque donc le début d'une nouvelle enchère coté client.

— **objetVendu** : le serveur appelle cette méthode pour indiquer au client qu'un objet a été vendu et lui transmet le prix de vente final ainsi que le vainqueur de l'enchère. L'appel à cette méthode marque donc la clôture d'une enchère.
— **nouveauPrix** : le serveur appelle cette méthode pour signaler au client que l'objet actuellement en vente vient de recevoir une enchère sur son prix. L'appel à cette méthode marque donc la fin du round précédent et le début d'un nouveau.

Spécification du moniteur Serveur Étant une ressource partagée, nous considérons le serveur comme un *moniteur de Hoare*. Les méthodes qu'il expose ne sont donc accessibles que par un processus à la fois, et le serveur peut gérer la mise en attente des processus via des files d'attente. Il possède aussi les variables partagées suivantes :

— **nombreParticipants** (type : entier, initialisé à 0) : le nombre de participants de la vente en cours.
— **vainqueur** (type : client, initialisé à *null*) : le vainqueur du round précédent.
— **venteEnCours** : une condition d'attente, liée à une file d'attente, qui indique si une vente est en cours.
— **encheresEnAttente** (type : file, initialisée à vide) : une structure FIFO qui contient les enchères placées par les clients et qui sont en attente.

Le serveur expose les méthodes suivantes :

— **inscrire** : un client appelle cette méthode pour s'inscrire à la prochaine vente aux enchères. Conformément aux spécifications du problème, si une vente est déjà en cours, alors le client est mis en attente. Il sera libéré (ainsi que tous les autres clients dans la file d'attente) lorsque la vente en cours sera terminée.
— **vendre** : un client appelle cette méthode pour mettre en vente un objet. S'il n'y a pas de vente en cours, une nouvelle est lancée en utilisant cet objet. Sinon, il est stocké dans une structure de type FIFO et placé en attente.
— **encherir** : un client appelle cette méthode pour signaler qu'il enchérit sur un objet dans le cadre d'un round. Son enchère sera alors stockée en mémoire.
— **tempsEcoule** : un client appelle cette méthode afin de signaler que, de son point de vue, le round est terminé. Cela peut être due au fait qu'il a enchérit sur l'objet ou que son temps imparti pour enchérir est écoulé. Lorsque le dernier client impliqué dans la vente appelle cette méthode, la plus grande enchère du round est calculée, elle devient le nouveau prix de départ de l'objet et un nouveau round de vente démarre pour les clients. Mais si aucune nouvelle enchère n'a été faite, l'appel à cette méthode clôt la vente, débloque les clients en attente et lance la prochaine enchère (s'il y en a une en attente).

Protocole de vente La figure 1 présente un exemple de mise en pratique de notre protocole de vente tel que décrit dans les spécifications.

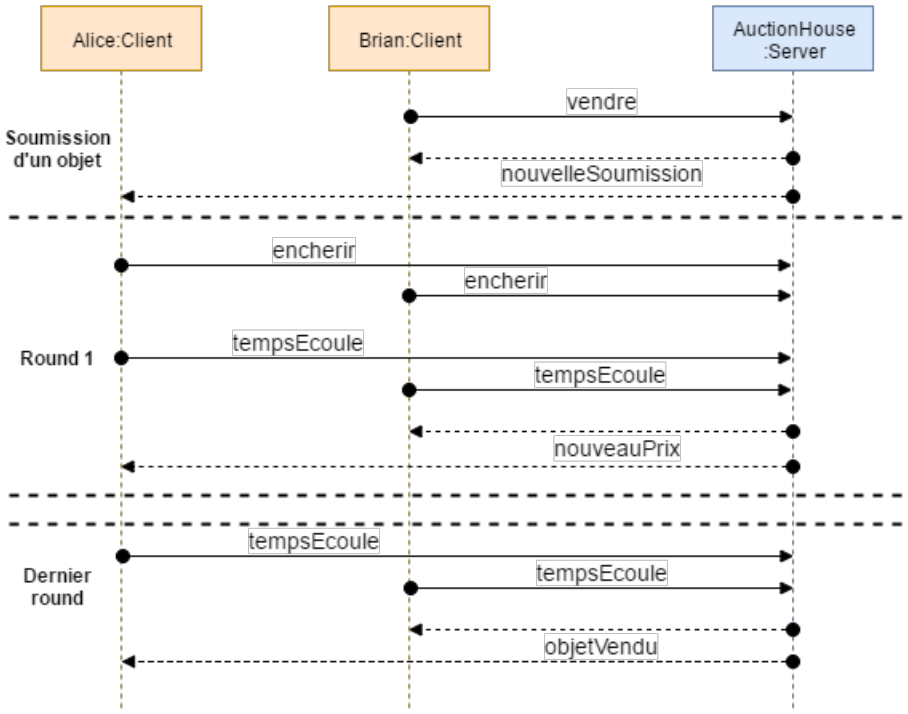

FIGURE 1 – Exemple de mise en pratique du protocole de vente

Gestion du temps imparti pour un round Pour gérer le temps de chaque round, nous utilisons un chronomètre qui est placé du côté du Client. S'il avait été placé du côté du serveur, cela n'aurait pas été juste pour les clients. En effet, si jamais l'un d'entre eux a une latence réseau plus importante que les autres, des actions émises avant la fin du temps imparti pourraient être reçues plus tard et donc être rejetées par le serveur.

En plaçant le chronomètre chez les clients, nous évitons ce type de problèmes, mais cela nécessite que chaque client prévienne le serveur que son temps est écoulé, ce qui génère plus de trafic réseau.

Gestion de la déconnexion Le cas le plus simple de déconnexion est celui où le client prévient explicitement le serveur qu'il se déconnecte et quitte le système. En revanche, le cas des déconnexions impromptues est plus complexe à aborder. Pour résoudre cela, nous proposons un système qui utilise un daemon qui tourne en tâche de fond et vérifie périodiquement si les clients sont toujours

connectés. S'il trouve une connexion morte, il la supprime et retire manuellement le client du serveur.

Cette solution présente néanmoins des inconvénients. Comme le daemon n'agit pas immédiatement après la déconnexion, il y a un laps de temps où le serveur voit toujours le client comme étant connecté, ce qui peut causer des erreurs s'il essaie de communiquer avec lui. L'idéal serait que le client puissent toujours notifier le serveur de sa déconnexion, volontaire ou non, au travers de la plateforme RMI.

1.3 Problématiques non abordées

Notre implémentation ne gère pas tous les cas d'utilisation que nous avons identifié lors de la phase de spécification. Par exemple, nous ne gérons pas le cas où un utilisateur se déconnecte alors qu'il est sur le point de remporter une enchère. De plus nous ne gérons pas l'inscription et l'authentification des utilisateurs sur le serveur (notions d'identifiant, mot de passe, etc.).

La sécurité des échanges entre le client et le serveur n'a pas été abordée dans le projet. En effet, les contraintes de sécurité sont très importantes dans les applications critiques telles qu'un système de vente aux enchères.

2 Implémentation de l'application

Nous avons développé notre application en Java 1.6 avec la technologie RMI pour les échanges entre les clients et le serveur et Swing pour notre interface graphique. Nous avons choisi cette version de Java pour permettre le support d'un plus grand nombre de plateformes.

2.1 Structure de l'application

Nous avons implémenté notre application en suivant le modèle de calcul décrit précédemment. Elle se compose de deux modules : le modèle Client-Serveur en lui-même, qui gère le système d'enchères, et l'interface graphique. La figure 2 présente un diagramme de classes simple de notre application.

Pour connecter l'application à son interface graphique sans créer un fort couplage entre les deux, nous avons décidé d'utiliser un système d'évènements basé sur le patron de conception Observateur. Ainsi, dès qu'un module a besoin d'envoyer des informations à l'autre, il passe par un observable qui notifie ses observateurs. Grâce à cela, nous pouvons connecter n'importe quelle interface graphique au modèle client-serveur et inversement.

2.2 Intégration de Java Remote Method Invocation

Comme dit précédemment, nous avons utilisé la technologie RMI dans notre projet pour gérer les communications sur le réseau entre les différents acteurs. Il a trois types d'objets qui doivent être échangés entre les acteurs de l'application :

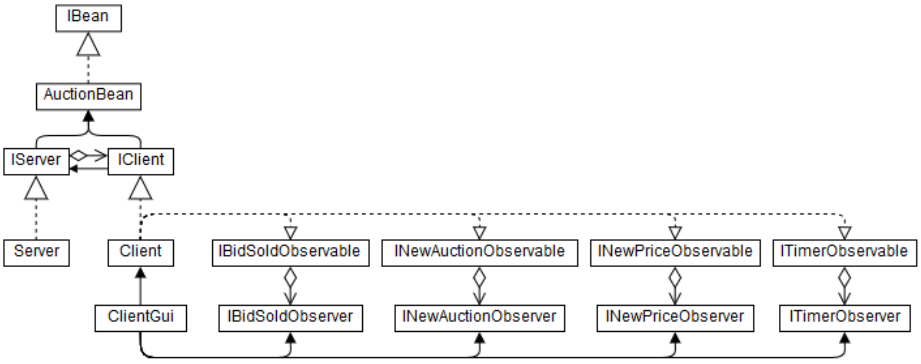

FIGURE 2 – Diagramme de classe de l'application PAY 2 BID

— Les classes **Client** et **Server** sont des sous-classes de **UnicastRemoteObject**, car nous avons besoin d'avoir accès aux références de ses objets et non à des copies.
— L'interface **IClient** implémente l'interface **Serializable**, car c'est un prérequis de la norme RMI pour que qu'elle soit utilisable comme paramètre d'une méthode appelée sur une machine distante.
— L'interface **IBean**, qui implémente l'interface **Serializable**, car toutes ses implémentations étant de simples conteneurs de données, elles peuvent être accédées comme des copies.

2.3 Gestion de plusieurs salles d'enchères

La fonctionnalité permettant de gérer plusieurs salles de vente aux enchère n'a pas été implémentée.

Néanmoins nous avons estimé le travail à réaliser pour implémenter cette fonctionnalité :

— Modification du serveur : ajout d'une liste de salles de vente en cours et en attente avec les méthodes correspondantes.
— Modification du client en conséquent (récupération de la liste des salles de vente, ...)
— Ajout d'un écran listant les salles de vente dans l'interface graphique.

Conclusion

En conclusion, ce projet nous permis de visualiser les problématiques liées au développement d'une application distribuée. Nous avons constaté que la phase de spécification et de conception est cruciale et détermine la complexité du modèle de calcul. De plus, nous avons aussi réalisé qu'il est complexe, et même parfois irréaliste, de vouloir créer une application distribuée qui a les mêmes fonctionnalités qu'une application non distribuée. Il faut donc savoir faire des choix et gérer les compromis, quitte à mettre de côté des aspects de l'application.