Rapport Middleware

Juzdzewski Matthieu, Kowalski Vincent, Lebouis Clément, Santos Daniel.
Source code: `https://github.com/xdanielsb/RMIGame`

Novembre 2020

1 Introduction

L'objectif du projet est de mettre en place des situations de concurrence sur des variables partagées. Le support et le thème étant pour le reste libre, nous avons choisi de développer une copie modifiée du célèbre jeu Agar.io (celui-ci est disponible sur navigateur). Nous avons voulu rester au plus proche des notions enseignées dans cette UE, à savoir Java RMI et les moniteurs, ainsi que la programmation orientée Aspect enseignée dans une autre UE. C'est pourquoi nous avons basé notre approche multijoueur sur le RMI de Java et la gestion de la synchronisation sur les moniteurs de Hoare (disons les moniteurs de Hoare tels que permis par Java car leur implémentation est quelque peu différente selon le langage). Le projet est donc codé sur Eclipse en Java, avec un projet Server et un projet Client. Nous verrons plus loin les détails de l'implémentation. Quant aux mécaniques du jeu en lui-même, nous avons deux équipes : les rouges et les bleus. Un joueur contrôle un cercle de taille variable à l'aide de sa souris. Chaque équipe a un score qui peut être incrémenté de deux manières différentes : soit en mangeant de la nourriture qui est disséminée dans la zone de jeu, soit en mangeant un joueur de l'équipe adverse plus petit que soi. De plus, lorsque l'on mange, on grossit. Il existe un timer global qui, lorsqu'il arrive à 0, termine la partie et désigne l'équipe avec le score le plus haut comme étant gagnante. Par la suite, nous présenterons dans un premier temps le modèle UML de notre projet, puis nous détaillerons les axes principaux, particulièrement la synchronisation, le RMI et notre tentative d'intégrer la programmation orientée Aspect.

2 Modèle

3 Developpement

Tout d'abord, le projet ne fonctionne en l'état actuel qu'en localhost. Nous avons essayé de connecter le client sur l'adresse publique du routeur du serveur et de forward les ports correspondants mais cela ne semble pas fonctionner. Le

projet est donc divisé en deux parties : le serveur et le client. Les graphismes sont gérés à travers la librairie Java Processing.

3.1 RMI et multijoueur

La partie RMI est assez classique : un serveur qui inscrit dans l'annuaire l'objet distant et un client qui peut accéder aux méthodes partagées à travers l'interface IPlayerRemote. Le port choisi est le 1099 et comme dit précédemment l'adresse IP est localhost. Le client gère uniquement les parties graphique et input de son côté. Son cercle est toujours représenté au centre de sa fenêtre de jeu, ce qui implique un offset par rapport à sa position réelle sur le serveur. Cet offset est également répercuté sur l'ensemble des objets visibles par le client. Un effet zoom est également appliqué pour rendre compte de la différence de taille entre les objets (tous ces calculs sont des modifications vectorielles). Le client envoie son intention de déplacement au serveur qui va alors mettre à jour la position du client sur la zone de jeu. Parallèlement à la gestion des inputs de tous les clients, le serveur possède un "tick de régularisation" : toutes les 30ms, il va exécuter deux fonctions de collisions : les collisions joueur/nourriture et les collisions joueur/joueur. Le résultat de ces fonctions indiquera si une nourriture ou un joueur a été consommé pendant le tick, augmentant ainsi le score correspondant de l'équipe bleue ou rouge. La décision de choisir 30ms pour l'horloge serveur est arbitraire mais elle se base sur un compromis : donner suffisamment de temps au serveur pour effectuer ses calculs de collisions tout en paraissant fluide à l'oeil du joueur. Or l'oeil humain voit (en moyenne) une image comme étant fluide à partir de 28 images/s. En considérant des millisecondes, cela implique le calcul suivant : $1000/28 = 35.7$ (environ). Nous avons choisi 30ms pour arrondir, et nous avons de plus constaté que cette valeur est souvent utilisée dans les applications de ce type. Cet intervalle donne donc au serveur 30ms pour effectuer ses calculs tout en gardant une impression de fluidité visuelle du côté des clients.

3.2 Concurrence et synchronisation

Le projet comporte trois principales situations de concurrence que nous allons détailler ici : le score, l'ID attribuée au joueur à son arrivée et le nombre de joueur dans chaque équipe.

- **Le score :** chaque équipe possède un score initialisé à 0 au début de la partie. Un joueur va pouvoir demander une incrémentation du score de son équipe en consommant un objet. Or, il est possible que plusieurs requêtes d'incrémentation arrivent au même moment par exemple si deux joueurs de la même équipe consomment simultanément un object A et un object B. La variable score est donc partagée à l'ensemble d'une même équipe. Nous avons alors créé un moniteur qui empêche l'accès simultané en écriture à la variable représentant le score de l'équipe. Un joueur de l'équipe A va verrouiller l'objet teamOne et un joueur de l'équipe B va

verrouiller l'objet teamTwo (ces deux Objects sont donc les variables de synchronisation). Notons que les deux équipes ne peuvent pas interagir avec le score de l'équipe adverse. L'accès en lecture aux variables de score n'est pas synchronisé car nous avons pensé que c'était inutile dans le cadre de ce jeu. Nous avons donc une cohérence faible au niveau de l'affichage des scores : ils finiront par être corrects pour tous les acteurs mais il est possible que la valeur affichée sur l'écran d'un client soit momentanément non cohérente.

- **L'ID du joueur:** lors de la connexion au serveur, un client reçoit une ID qui doit être unique car elle permettra au serveur d'identifier précisément l'origine de l'appel de méthodes distantes. Cette unicité du numéro d'identification est rendue possible à l'aide d'un Object de synchronisation (que nous avons appelé mutex) dans la fonction de connexion du joueur à la partie. Ainsi, si plusieurs joueurs se connectent en même temps, ils devront passer chacun leur tour dans la section critique de la méthode RegisterPlayer qui est celle qui gère l'arrivée d'un joueur. De manière plus détaillée, le serveur stocke un entier qui s'incrémente à chaque nouvelle arrivée. Notre approche comporte cependant un risque (assez peu probable) de dépassement de capacité : par soucis de simplicité, nous ne décrémentons pas la variable lorsqu'un joueur quitte la partie, donc si le serveur reste actif jusqu'à ce que 2147483647 joueurs se soient connectés, nous aurons effectivement un dépassement.

- **Le nombre de joueurs par équipe:** l'attribution d'un joueur à une équipe ou l'autre est entièrement gérée par le serveur. Or, il a pour objectif de garder le nombre de joueurs dans chaque équipe équilibré. Donc si l'équipe A possède 3 joueurs et l'équipe B en possède 2, le prochain qui se connectera sera attribué à l'équipe B. Le serveur doit donc compter le nombre de joueurs dans chaque équipe avant de prendre sa décision. Or, cette étape est réalisée au même moment que l'attribution de l'ID, à savoir dans le mutex. On pourrait donc penser qu'il n'y a pas de problème de synchronisation puisque l'incrémentation se fera de manière synchronisée. Et c'est vrai pour l'incrémentation, mais cette fois-ci nous décrémentons également lorsqu'un joueur quitte la partie (sinon l'équilibrage serait compliqué). Or un joueur peut quitter à n'importe quel moment -et donc décrémenter la variable de manière non contrôlée- et donc le mutex ne suffit pas. Il faut donc à nouveau faire appel à notre moniteur. Nous avons ajouté un Object de synchronisation addTeamate : lorsqu'un joueur verrouille le mutex à son arrivée, il va aussi verrouiller cet Object et incrémenter le compte de l'équipe (qui est également stocké localement sur le moniteur). Dans le même temps, lorsqu'un joueur décide de quitter la partie, il va aussi demander le verrou sur ce même Object addTeammate. Ceci permet donc une cohérence forte sur le nombre de joueurs dans une équipe. Cependant, le système ne permet pas en l'état de garantir un équilibre entre les arrivées et les sorties. Imaginons la situation suivante : équipe A possède 6 joueurs et équipe B en possède 5. Au même moment, 3 joueurs

de l'équipe A quittent et 1 nouveau joueur tente de se connecter. Si c'est le nouvel arrivant qui récupère le verrou, alors il va être placé logiquement dans l'équipe B. À la suite de quoi l'équipe A va alors perdre 3 joueurs. On aurait donc à présent 3 joueurs dans l'équipe A pour 6 joueurs dans l'équipe B. Nous avons pensé à deux solutions à ce problème : soit favoriser en premier les départs de joueurs (donc on aurait dans cet exemple 4 joueurs dans A et 5 dans B), soit garder le système tel qu'il est mais ajouter un algorithme d'équilibrage permanent qui changerait les joueurs d'équipe selon les besoins.

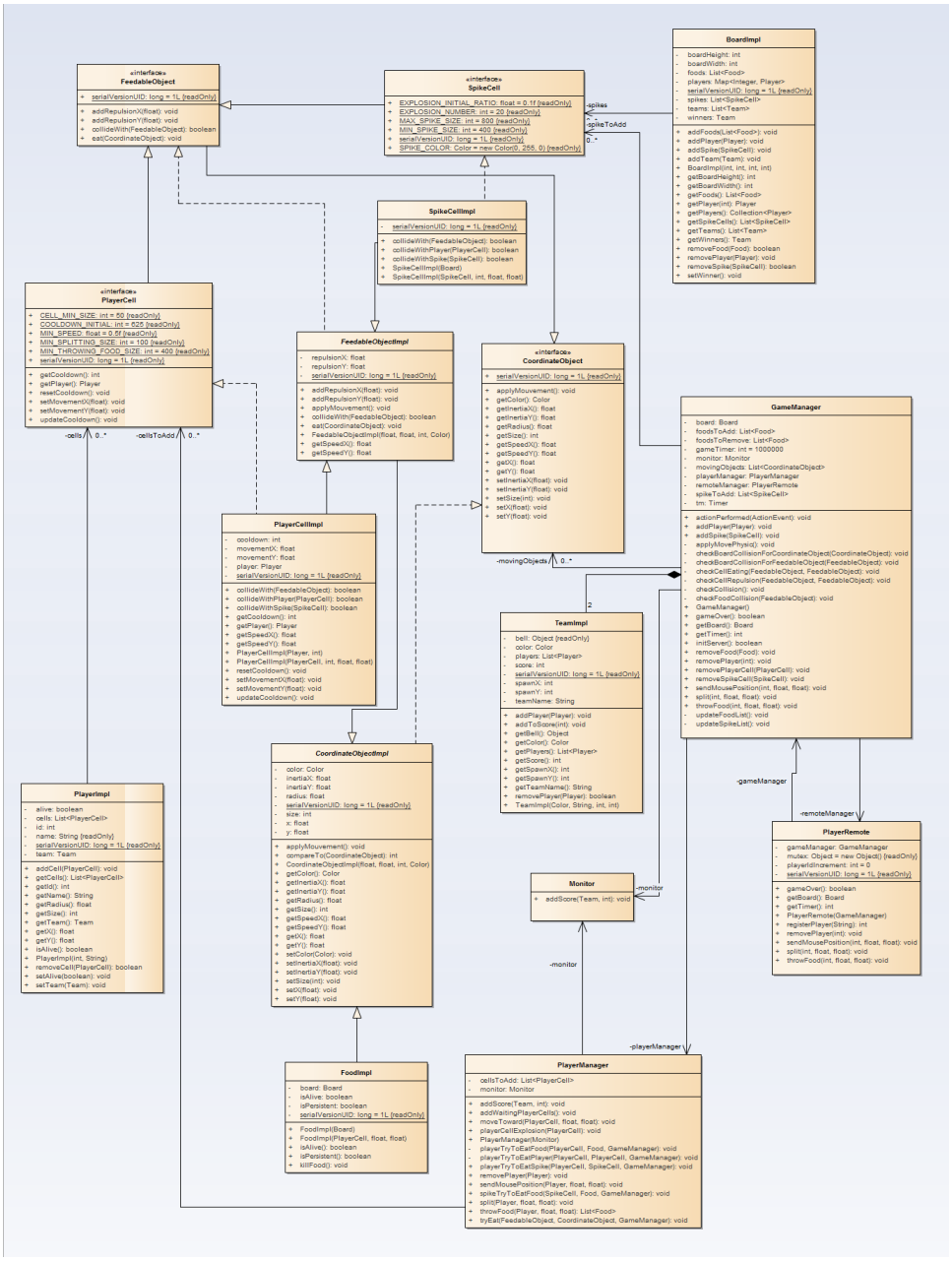

Figure 1: Class Diagram of the Server.

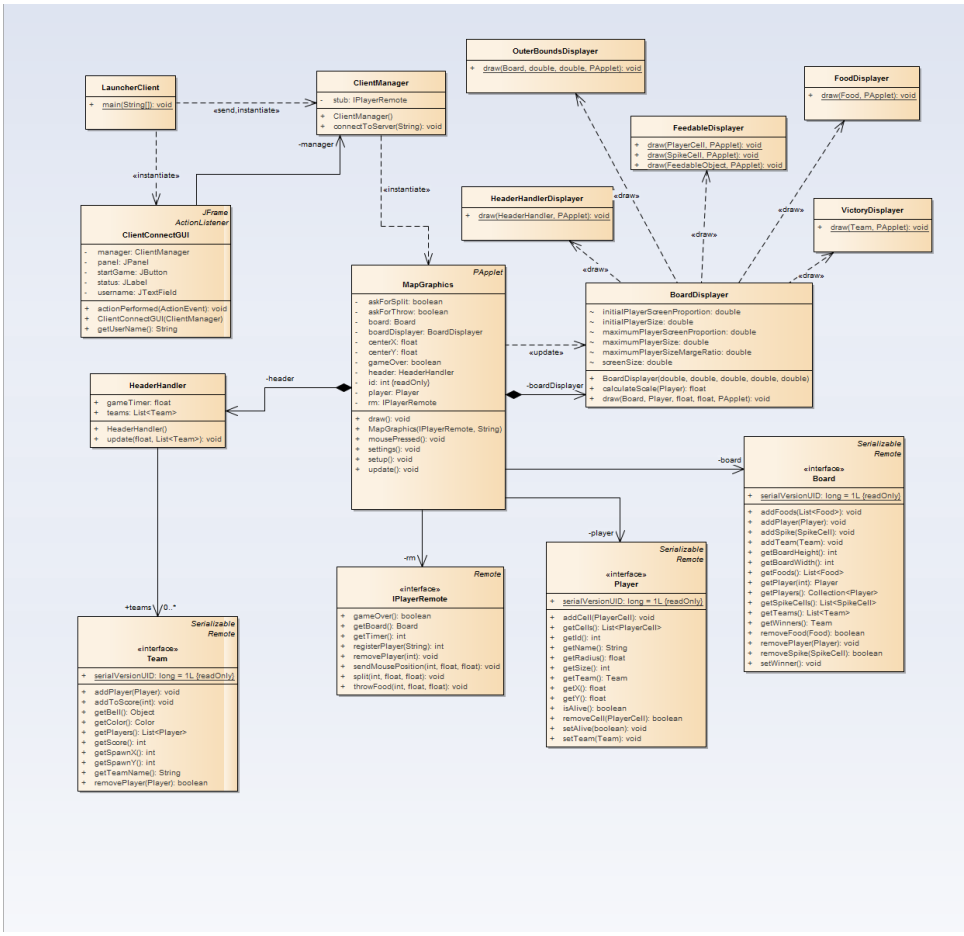

Figure 2: Class Diagram of the Client.

3.3 AspectJ

Dans une autre UE, nous avons été initiés à la programmation orientée Aspect et nous avons pensé qu'il pourrait être intéressant d'implémenter un aspect simple dans ce projet. Avant de parler de cet aspect, définissons le contexte dans lequel il va s'appliquer : comme vu précédemment, un joueur est un cerle qui se déplace sur une zone de jeu avec des limites prédéfinies. Ces limites définissent donc la zone "explorable" de la zone de jeu totale. Au lieu de coder les limites au niveau de la fonction de mouvement du serveur, nous avons décidé de déléguer cette tache à un aspect. Un pointcut est défini sur la méthode Move de la classe PlayerManager. Un advice around permet ensuite d'effectuer les vérifications de positions nécessaires, de modifier les coordonnées s'il s'avérait qu'elles soient

hors jeu, puis de proceed() avec les nouvelles valeurs.

4 Conclusion

On peut donc dire que notre "clone" du jeu Agar.io fonctionne comme prévu. Les différents tests effectués ne montrent pas de problèmes non prévus. La synchronisation sur les différentes variables évoquées semble effective, en tout cas dans le cadre d'une utilisation pratique du jeu. Il reste néanmoins des axes d'amélioration, notamment sur la partie équilibrage des équipes. En conclusion, le projet peut être considéré comme un prototype : le gameplay est fonctionnel, la cohérence -qu'elle soit faible ou forte- est assurée par le moniteur et l'utilisation du mutex qui permettent une flexibilité dans la manière d'appréhender le code. Notre seul regret serait de n'avoir pas réussi à faire fonctionner le projet sur des réseaux distants (WAN).