Rapport projet final POA

-

Serveur Multi Threads pour Jeu Multijoueur

Halablian Serge - HALS17029807
Joulia Stephan - JOUS08099808
Zeni Florian - ZENF16129806

Hamid Mcheick - Programmation Objet Avancée - 8INF957 - Automne 2020

Sommaire

I Introduction

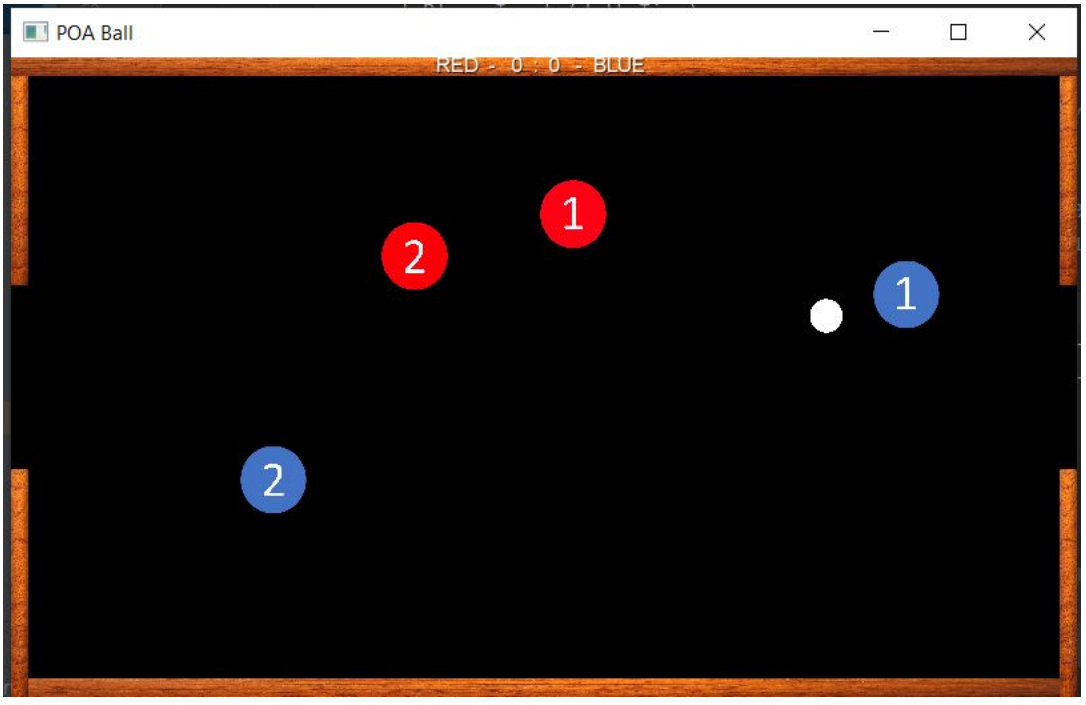
1) Thème / Choix

Après avoir présenté en début de cours, la technologie Peer to Peer lors de notre critique de technologie et avoir apprécié le premier TP sur le réseau, nous avons cherché un sujet de projet final qui continuerait à explorer cette voie de la programmation réseau en programmation orientée objet.

Les jeux multijoueurs en temps réel nous ont semblé être une bonne manière d'essayer d'appliquer ce que nous avions appris durant ce cours, que ce soit en réseau, ou en programmation objet. C'est pourquoi nous avons choisi comme sujet la réalisation d'un petit jeu multijoueur en temps réel, largement inspiré du jeu haxball. Il sera basé sur une architecture dite de serveur multithread.

2) Présentation

Notre jeu multijoueur oppose deux équipes, l'objectif est de marquer dans le camp adverse tout en défendant ses cages. Pour cela les joueurs disposent de plusieurs commandes. Ils peuvent se déplacer avec ou sans la balle et tirer pour marquer.



Cette application réseau amène plusieurs problématiques intéressantes à résoudre. Premièrement, l'architecture étant orientée multijoueur, on ne doit se retrouver dans une

architecture concurrentielle où des joueurs auraient un avantage certain sur d'autres puisque le jeu est en temps réel. Ainsi, l'architecture multithread du serveur à programmer permet d'éviter cet écueil en faisant en sorte que chaque joueur client se retrouve sur une entité commune, un thread.

De plus, puisque le jeu est temps réel, il faut donc s'assurer que chaque joueur voie la même chose au même moment et qu'aucun n'ait un avantage temporel sur les autres. En d'autres termes, il faut assurer la synchronisation des échanges. Pour cela on a aussi mis en place un système de prédiction-réconciliation pour chaque client et qui sera expliqué plus en profondeur dans la suite du rapport.

II Implémentation

1) Organisation

Concernant la conception du jeu, nous avons opté pour l'architecture suivante présentée sur l'UML. Nous disposons de deux fonctions Main différentes qui instancient la classe GameClientScreen ou GameServerScreen en fonction de qui est hôte ou client. Chacune de ces deux classes permet un fonctionnement totalement différent. En effet, notre serveur est maître de toute la logique du jeu, sa version est imposée à tous les clients. Côté client en revanche, il nous faut appliquer des traitements permettant de réduire l'impact des délais dû aux communications entre client et serveur. Pour cela, nous avons implémenté un système de prédiction client et de réconciliation serveur ainsi qu'une interpolation des positions dont nous reparlerons dans la suite.

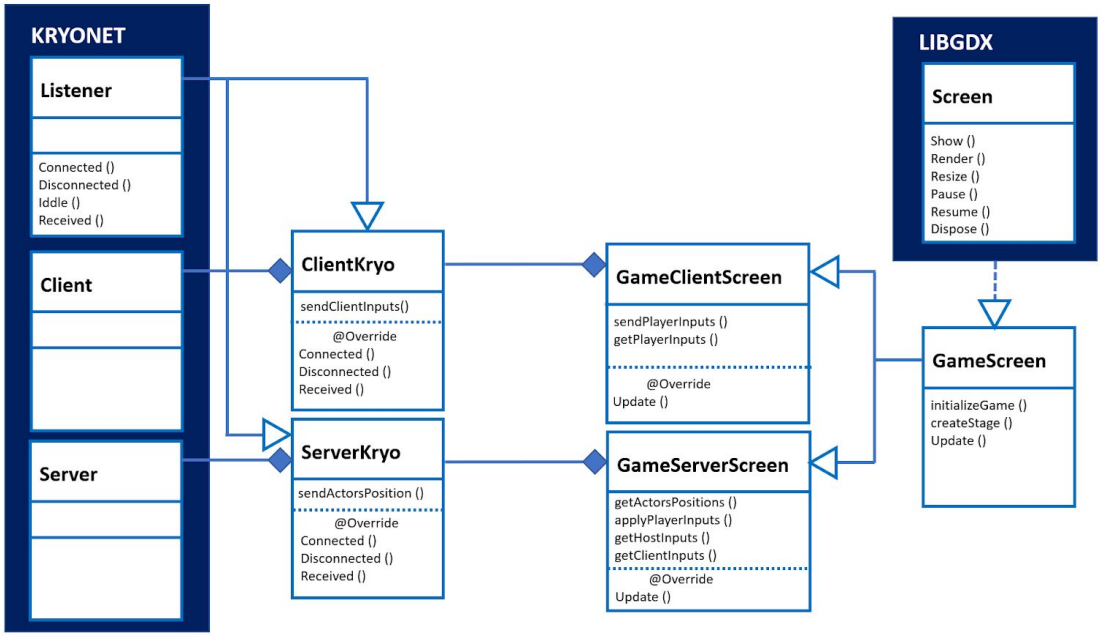


Diagramme UML Simplifié

La classe GameScreen contient toutes les fonctionnalités communes au client et au serveur afin d'éviter les répétitions de code comme par exemple la création de niveau, ou la gestion de l'affichage. Les classes ClientKryo et ServerKryo sont associées par composition aux classes principales respectives. Elles sont chargées de l'envoi et de la réception des messages transitant entre le serveur et les clients ainsi que de conserver les données reçues en attendant d'être lues et traitées.

2) Prédiction Client et Réconciliation Serveur

Si l'on attend que le serveur réponde aux requêtes du serveur envoyées à chaque changement au niveau des inputs du joueur, les clients observeront un léger décalage entre leurs actions et leurs conséquences, en plus de se voir se déplacer de manière saccadée. Un moyen d'éviter ces problèmes consiste à permettre au client de prédire les conséquences de ses actions. Le client ne sera alors corrigé que si la prédiction se révèle incorrecte.

Pour que cette réconciliation soit possible, il ne faut pas comparer les positions reçues provenant du serveur avec la position actuelle du joueur car le léger délai dû au transfert signifie que la position reçue est en réalité une position passée. Cela implique donc de conserver, côté client, les positions précédentes, et de comparer le message serveur, avec la position adéquate de l'historique. Pour stocker ces positions, nous avons créé une classe héritant d'une ArrayList avec comme argument supplémentaire une taille maximale N. Si l'on essaie d'ajouter un élément alors que la taille maximale est déjà atteinte, le premier élément de la liste est retiré. On obtient alors une liste des N dernières positions obtenues à intervalle régulier, ce qui nous permet, en fonction du temps de voyage moyen d'une requête, de comparer le message du serveur avec la position chronologiquement cohérente du client.

3) Interpolation

La méthode employée précédemment permet de réduire les délais entre les inputs du joueur et ses conséquences, mais n'impacte en rien le déplacement des autres acteurs, que ce soit les autres joueurs ou la balle. On observe donc toujours un déplacement saccadé au rythme des broadcast du serveur sur tous les autres acteurs.

La méthode couramment utilisée pour faire face à ce problème, consiste à utiliser l'interpolation entre les deux dernières positions des acteurs fournies par le joueur. Ce faisant, on calcule une série de positions intermédiaires modifiées à la vitesse réelle du jeu, ce qui donne une impression de fluidité comme on peut l'observer chez l'hôte. Cependant, cette méthode a un défaut, c'est celui d'afficher les positions des autres acteurs avec un retard d'une mise à jour serveur. Les acteurs affichés à l'écran du client sont donc "dans le passé", ce qui n'est malgré tout, pas réellement problématique, au vu de la vitesse de rafraîchissement du serveur.

Pour calculer ces positions intermédiaires, on utilise un coefficient d'interpolation calculé à partir du temps écoulé depuis la dernière mise à jour et de la fréquence de broadcast du serveur. Dans la capture ci-dessous, on utilise l'interpolation sur la position de la balle. Le procédé est strictement le même pour les autres joueurs.

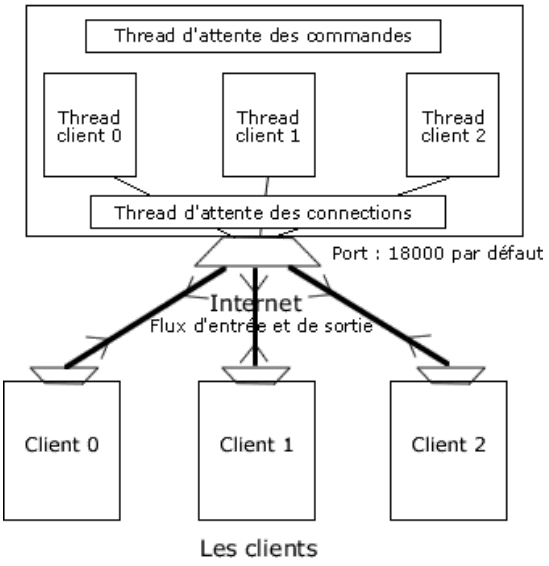
```
// Actors Positions are updated at the update loop frequency
interpolationCoef = Math.min(1, timeSinceLastServerUpdate / timeBetweenUpdates);

ball.setClientPosition( x: interpolationCoef * serverAnswer[0] + (1 - interpolationCoef) * previousServerAnswer[0],
                        y: interpolationCoef * serverAnswer[1] + (1 - interpolationCoef) * previousServerAnswer[1]);
```

III Réseau

1) Serveur Multithread

On peut aussi parler de serveur Java multiclient. Il redirige les flux arrivant d'un client connecté vers tous les autres et ceci indépendamment du type du client et a la possibilité d'accepter plusieurs clients simultanément.
Techniquement, le serveur reçoit un message du client, l'envoie alors à tous les autres connectés. Il y a aussi la possibilité de l'interroger au travers de commandes serveurs spécifiques. On a alors une véritable universalité du serveur.

On a alors l'architecture suivante :



On observe dans notre cas le thread principal qui attend les connexions en écoutant le port ouvert pour le serveur. À chaque connexion il lance un nouveau thread de type client.

Les threads clients associés à chaque client connecté. Ils sont chargés d'attendre l'arrivée d'un message. Lorsqu'un de ces threads reçoit un message, il l'envoie à tous les clients connectés.

Le thread des commandes attend que l'administrateur du serveur tape des commandes dans la console pour faire une action.

2) Librairie Kryonet

Pour cette partie réseau, nous avons eu recours à la bibliothèque Kryonet. C'est une librairie Java fournissant une API simple et claire pour les applications réseau, et plus particulièrement pour les jeux. Son efficacité réside dans l'amélioration de la communication réseau pour les protocoles TCP et UDP.
Dans notre cas, on utilise les 2 protocoles. UDP trouve son intérêt pour la partie jeu quant à l'échange de commandes et la transmission des données du jeu. Il est moins fiable mais rapide et c'est ce dont nous avons besoin dans un jeu multijoueur en temps réel.

On utilise aussi le protocole TCP pour le tchat où la rapidité d'envoie n'est pas aussi primordiale. On peut alors se permettre d'améliorer la fiabilité de l'envoie des données via ce protocole mais nous y reviendrons plus en détail dans la suite du rapport.

Dans un serveur multithread, la bibliothèque KryoNet est une excellente solution car elle facilite la communication entre les processus, notamment grâce à la sous-librairie Kryo. Cette dernière permet une sérialisation automatique et optimale sur le réseau sans pour autant le ralentir. Elle orchestre le processus de sérialisation et mappe les classes sur des instances *Serializer* qui gèrent les détails de la conversion du graphe d'un objet en une représentation en octets. Une fois les octets prêts, ils sont écrits dans un flux en utilisant un objet *Output* . De cette façon, ils peuvent être stockés dans un fichier, une base de données ou transmis sur le réseau. Lorsque l'on souhaite désérialiser l'objet pour l'utiliser, une instance *Input* est utilisée pour lire ces octets et les décoder en objets Java.

3) Implémentation

La connexion de chaque client s'opère comme suit. Ils n'ont pas d'adresses IP propres et se contentent de rejoindre le serveur qui lui en possède une.

Ce serveur est d'ailleurs lancé sur un thread avec une adresse IP personnelle. Il attend les connexions entrantes, lit et écrit dans les sockets et interroge ses listeners personnels. Un listener vérifie chaque entrée qu'il reçoit. Si il s'agit d'une instance de client, il la traite sinon, il la rejette.

La première information envoyée par le client lui permet de s'initialiser, elle contient des informations sur l'index qui lui est attribué par le serveur et permet à la prédiction client de s'appliquer sur le bon joueur.

Les informations suivantes formeront le trafic routinier :

- Le client envoie les inputs sur les touches haut, bas, gauche, droite, espace
- Le serveur envoie les positions de tous les acteurs (joueurs et balle)

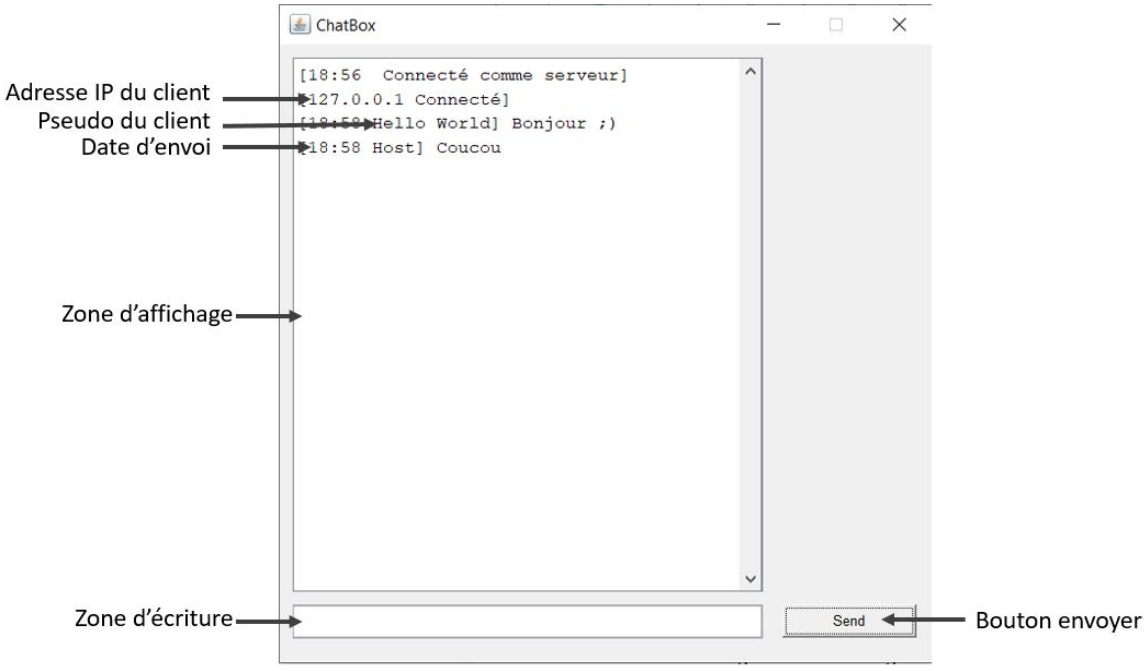
Chacun de ces messages disposera d'un index assigné chronologiquement qui permettra de ne prendre en compte uniquement les messages plus récents que le dernier message reçu. Cela permet de ne pas prendre en compte des données périmées.

A partir de ces inputs, le serveur pourra simuler sa version du jeu qui fera loi et sera appliquée à tous les clients.

Côté client, les traitements expliqués plus haut seront réalisés afin de réduire les désavantages ressentis par le joueur client par rapport à l'hôte qui lui voit ses inputs traités plus rapidement et les déplacements des acteurs de manière fluide.

4) Tchat

Afin de rendre le jeu plus amusant à jouer entre amis nous avons décidé d'ajouter une fenêtre de messagerie instantanée. Nous nous sommes contentés de faire un tchat avec de simples fonctionnalités, ainsi notre fenêtre contient : une zone d'affichage, une zone d'écriture et un bouton d'envoi (voir ci-dessous).



Fenêtre du tchat du serveur

Ainsi chaque joueur peut écrire dans la zone d'écriture puis appuyer sur “send” afin d’envoyer le message. Tous les joueurs de la partie qui sont connectés au tchat pourront voir les messages. Les messages sont envoyés grâce au protocole TCP (*Transmission Control Protocol*) qui est un protocole de transport fiable.

Implémentation du tchat

Pour créer le tchat, nous avons séparé le serveur des clients car ils ont deux rôles bien distincts. Le serveur doit recevoir les messages des clients et les renvoyer à tous les clients sauf au client qui est à l’initiative du message. Les clients, quant à eux, envoient juste leurs messages au serveur. Pour cela, on a réécrit les méthodes *send* et *received* dans la classe *ClientChat* et *ServeurChat* qui proviennent de *Listener* dans la librairie Kryonet, en utilisant des méthodes telles que *sentToAllExceptTCP( )* qui permet d’envoyer des objets en TCP en omettant un client.

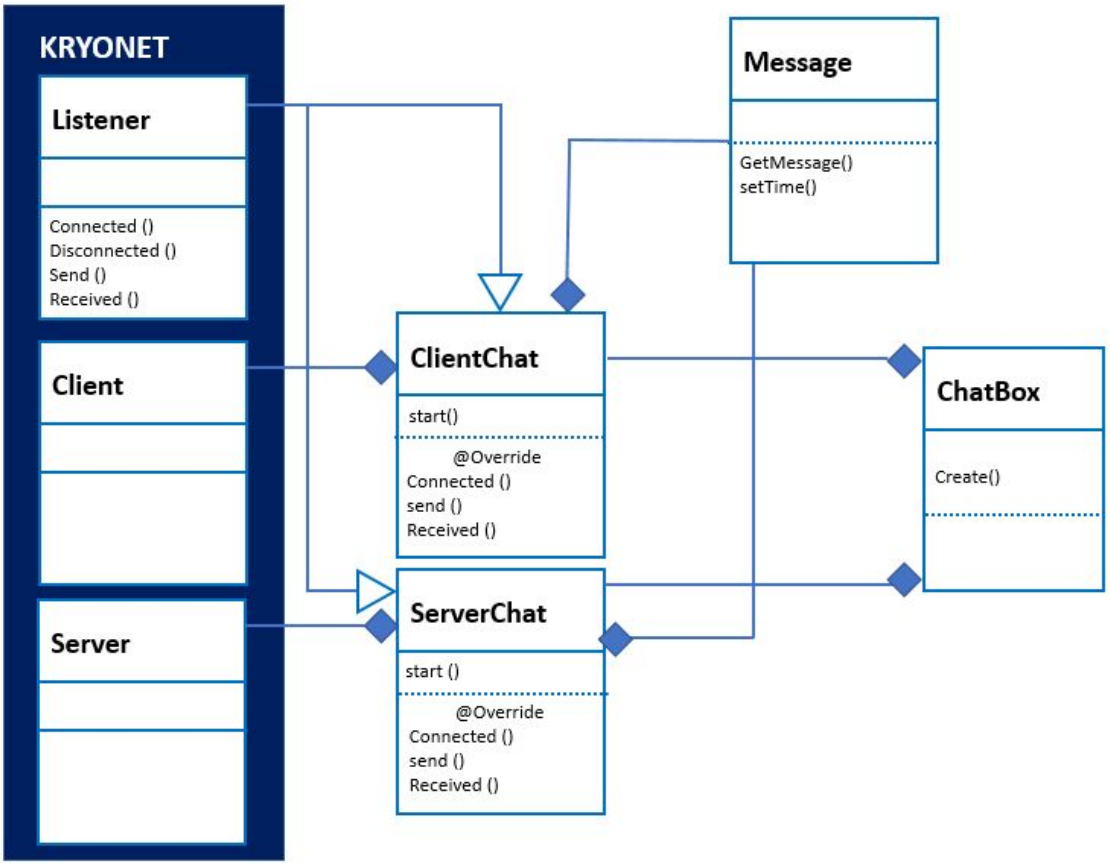


Diagramme UML simplifié du tchat

IV Utilisation

Vous trouverez dans le fichier zip de rendu, notre projet libgdx complet ainsi que deux .jar, un pour le serveur, l'autre pour le client paramétrés pour fonctionner en local. Le projet fonctionne avec Gradle. Pour tester les effets des différentes technologies de prédiction et d'implémentation, le fichier ClientGameScreen contient deux booléens paramétrables pour les activer et désactiver au besoin. L'ip du serveur, est définie dans ClientKryo et est actuellement paramétrée sur localhost.

Notre projet est également disponible sur github au lien suivant :
https://github.com/FlorianZeni/POABall

Pour pouvoir essayer le jeu dans des conditions réelles, avec des connexions plutôt lentes, nous avons utilisé le logiciel LogMeIn Hamachi qui permet de simuler un réseau local et ainsi outrepasser les vérifications des pare-feux ordinateurs et routeurs. Nous avons alors pu essayer le jeu à grande distance, avec un ping non négligeable.

V Conclusion

Ce projet a été pour nous l'occasion d'approfondir nos connaissances à propos des préoccupations réseau, et de réaliser une application complète. Nous en sommes satisfait, mais si nous devions trouver des axes d'amélioration, on pourrait citer les suivants :

- Intégration du tchat sur le jeu et le client Kryonet principal
- Essayer et comparer d'autres structures réseau (P2P décentralisé, serveur dédié, ...)
- Essayer d'autres protocoles réseau sans passer par Kryonet
- Régler la problématique de sécurité des pare-feux (sans Hamachi)

Malgré cela, nous avons un jeu terminé et fonctionnel qui fonctionne en multijoueur, est hébergé par un joueur, et applique les mécaniques de prédiction client, réconciliation serveur et interpolation comme nous avions prévu au début du trimestre.

VI Bibliographie

Kryonet : https://github.com/EsotericSoftware/kryonet

Kryo : https://github.com/EsotericSoftware/kryo

LibGDX : https://libgdx.badlogicgames.com/documentation/

Box2D : https://libgdx.info/box2d-basic/

Prediction Client :

https://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html

Tutoriel LibGDX : https://github.com/julienvillegas/libgdx.info-Box2D-basic