Ponginator

Palard Nicolas
Robert Etienne

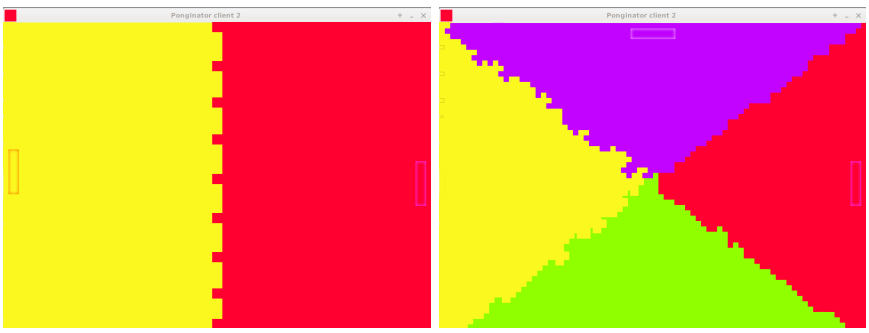
January 3, 2016

1 Introduction

Dans l'optique d'un Projet de Programmation Objet et Réseau, nous avons réalisé un **Pong**, un jeu créé en 1967 par Ralph Bear et ensuite amélioré et édité par Atari en 1972. Ce jeu est fortement inspiré du tennis de table et il constitue simplement un échange de balle entre deux (ou plus) raquettes réparties sur les extrémités du terrain.

2 Fonctionnalités

La partie réseau de ce projet a été réalisée de façon acentré, c'est à dire que chaque client dialogue avec tous les autres sans passer par un serveur. A chaque instant, on détermine un client nommé maître, chargé d'informer ses homologues de l'état du jeu. Le maître est choisi en fonction de la position de la balle, comme l'illustre l'image ci dessous :



La totalité de la partie réseau a été faite en utilisant des sockets configurés de façon non-bloquantes.

Cette conception nous a poussé à développer **une clock** interne pour ne pas momentanément couper le thread principal du programme, ce qui n'était pas en adéquation avec l'idée que l'on pouvait recevoir à tout moment n'importe quel

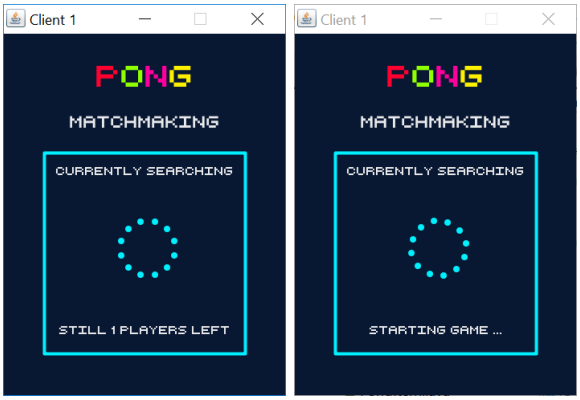
type d'information.

Nous avons décidé d'implémenter le mode **quatre joueurs** ainsi qu'un système anti-triche qui vérifie en permanence si **les données qui transitent entre les joueurs sont correctes** et si personne n'essaie de bidouiller son propre jeu pour gagner la partie. Pour ce faire, nous avons du utiliser un système qui traque les données et qui fait ensuite ses propres calculs pour les comparer avec les données qui sont reçues. De cette manière, si un joueur triche, les autres se déconnectent et la partie se termine.

Du fait du mode quatre joueurs, il fallait trouver un moyen d'**attribuer des points durant une partie** et pour cela nous avons imaginé un système de "tournoi" : A chaque fois qu'un joueur rate la balle, celle ci rebondit sur le "mur" en question, et tout les autres joueurs marquent 1 point. Ceci implique que plusieurs joueurs peuvent atteindre le nombre maximum de points en même temps, il a donc fallu les départager. Lors d'une telle égalité, le score de chaque joueur concerné est réinitialisé et la partie recommence sans les perdants. Une fois qu'il n'en reste plus que deux, une ultime partie se lance pour ainsi déterminer le grand vainqueur de la partie de Pong !

Pour qu'une partie ne se lance pas brusquement, nous avons aussi réaliser une petite fonctionnalité qui permet d'**attendre que tout le monde soit prêt** et ait appuyé sur la touche **ESPACE** de son clavier pour lancer la partie de Pong.

Pour un peu plus d'amusement, nous avons implémenté une fonctionnalité graphique qui permet de **changer la couleur de la balle en fonction du maître** actuel de celle ci. Comme autre fonctionnalité graphique nous avons réaliser un petit menu de "Matchmaking" qui **informe le joueur sur l'état de la partie** à laquelle il souhaite se connecter (en attente de joueurs ou la partie va se lancer).



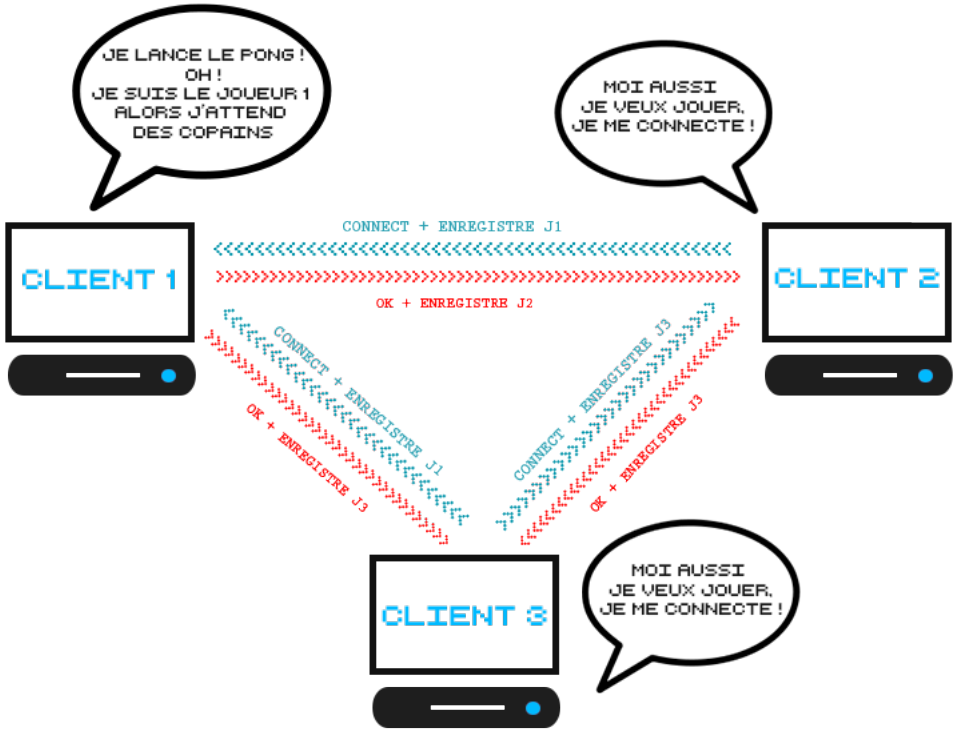
3 Valorisation du travail réalisé

3.1 Le non bloquant

Comme dit plus haut, nous avons choisi de réaliser la totalité du projet en utilisant des **sockets non-bloquantes** parce que nous voulions que le jeu ne soit jamais dans un état de pause et qu'ainsi l'échange perdure tout le long d'une partie. Comme on peut s'en douter, ce choix n'a pas été sans conséquences ...

3.1.1 La connexion

La connexion des clients entre eux a été un peu délicate. Elle se déroule de la façon suivante :



Bien sûr, ce schéma est une représentation très imagée de ce qu'il se passe en réalité.

Lorsqu'un client lance le Pong, il doit préciser les adresses IP des joueurs auxquels il souhaite se connecter. Le premier client ne précisera donc aucune adresse IP puisqu'il va juste attendre les connexions des autres joueurs. Le deuxième client, quant à lui, précisera l'adresse IP du premier client.

Le premier client va ainsi recevoir cette connexion, l'accepter si elle est conforme, et la stocker en la liant a un identifiant qui lui sera attribué pour le reste de la partie. **De cette façon on peut rapidement déterminer à quel joueur on envoie des informations quand on utilise telle ou telle connexion.** Ces données sont stockées dans un objet "Player" qui contient entre autres : l'identifiant du joueur et la connexion (sous forme de SocketChannel) établie. Cet objet contient d'autres données peu importantes dans cette partie.
Par analogie, cette opération se déroule de la même façon lorsque les joueurs 3 et 4 souhaitent se connecter, ils précisent seulement plus d'adresses IP.

3.1.2 La déconnexion

Lorsqu'un joueur se déconnecte, nous avons décidé que les autres joueurs continueraient à jouer sans lui. Pour ce faire, nous avons remplacé un joueur parti par **null dans l'implémentation** et à chaque fois que nous bouclons dans la liste des joueurs connectés, nous ignorons simplement ceux valant null.

3.1.3 Le protocole

Pour communiquer entre les clients, nous avons décidé d'opter pour un **protocole texte** assez simple afin de pouvoir **débugger nos communications** réseau facilement. Chaque message est divisé en champs séparés par un caractère spécial **SEPARATOR_CHAR = ':'**, et chaque message se termine par un caractère de fin **END_CHAR = '$'**. Ce caractère de fin permet, lorsque plusieurs messages sont reçus en un bloc, de les séparer facilement et de détecter un message non terminé. Le premier champ de chaque message est une chaîne de caractères définissant le type du message. Il peut être de trois types, **MESSAGE_MASTER = "M"**, **MESSAGE_RACKET = "R"** ou **MESSAGE_READY = "READY"**. Le nombre et le type des champs suivants dépendent du type de message.

Le premier protocole utilisait le type MESSAGE_BALL = "B" à la place de MESSAGE_MASTER

3.1.4 Des buffers ? Allons-y !

Nous n'avons, avec les bibliothèques utilisées, aucune garantie que les données reçues des autres clients soient envoyées toutes d'un bloc, aussi, pour être capables de reconstituer les messages reçus d'un tour de boucle à un autre, nous avons choisi d'implémenter pour chaque autre joueur un buffer dans lequel le message en cours de réception sera stocké à n'importe quel moment du programme sous la forme d'un StringBuffer.

3.2 Détails d'implémentation

3.2.1 Tic tac ... La clock

Afin de **réguler le réseau et éviter toute triche**, nous avons conçu un système de tours, comme dans un jeu en tour par tour. Le début d'un **nouveau tour est délimité par un déplacement de la balle**. De nombreuses actions du jeu sont basées sur les tours, appelés ticks dans l'implémentation. Notamment, chaque joueur ne peut déplacer sa raquette qu'une fois par tick.

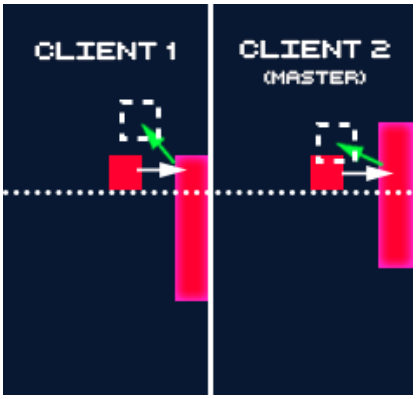
3.2.2 Être maître ou ne pas l'être

Comme il nous l'a été conseillé, nous avons décidé que l'un des clients, dit maître, est en charge d'**informer les autres joueurs des déplacements de la balle**. Le maître actuel est déterminé par la position de la balle. Le côté de l'écran le plus proche de la balle est celui du maître. Les autres clients sont appelés esclaves.

3.2.3 Synchroniser les raquettes

La synchronisation des raquettes nous a donné tant de fil a retordre qu'elle mérite sûrement une subsubsection. Pour que chaque joueur puisse pré-calculer les déplacements de la balle au tour suivant, il faut qu'ils aient tous les mêmes données.

Pour cela, nous avons tout d'abord fait en sorte que dès qu'un joueur souhaitait déplacer sa raquette (une fois par tick maximum), il changeait la dite position dans ses données et envoyait un message indiquant sa nouvelle position à tous les autres. Nous nous sommes vite rendus compte que cela allait engendrer des erreurs de synchronisation. En effet, si le message du déplacement de la raquette met longtemps à arriver et qu'un message de déplacement de la balle arrive au client entre temps, il calculera la nouvelle position de la balle au tick suivant à partir de la position de la raquette déplacée, alors que l'autre n'aura pas encore reçu la nouvelle position de la raquette. Les positions et vitesses calculées de la balle sont alors différentes d'un client à l'autre, les clients sont désynchronisés.



Afin de résoudre ce problème, notre première solution a été que chaque client ait la position de toute les raquettes enregistrées mais aussi **leurs positions au prochain tour**, valant null si elles doivent rester constante. Lorsqu'un client désire déplacer sa raquette, il enregistre alors la nouvelle position dans la variable correspondant à sa position suivante et informe les autres clients de là où il se trouvera au tour suivant. Ainsi, on esquive les problèmes de décalage de 1 tick. Cependant, toute latence supérieure à 1 tick impliquait toujours un décalage.

Nous avons alors totalement changé de perspective. En effet, nous avons choisi que lorsqu'un client souhaite déplacer sa raquette et qu'il n'est pas maître de la balle, il envoie une **requête de déplacement** au maître. Celui ci, quand il la reçoit, l'enregistre et, à son prochain envoi de balle, indique à tous les clients le déplacement effectif de la raquette. Ainsi, tout autre client, reçoit, **en même temps** (en effet les messages ne sont interprétés que s'ils sont complets), le début d'un nouveau tour (nouvelle position de balle) et son nouvel environnement au complet (nouvelles positions des raquettes).

Dans la théorie tout va bien, seulement concrètement, nous rencontrons toujours des problèmes de synchronisation avec cette méthode... Erreur d'implémentation ou méthode défaillante ? Nous sommes toujours en train de nous arracher les cheveux pour le déterminer.

3.3 Anti-triche

3.3.1 Peur des fantômes ? - La balle

Dans notre Pong, le maître annonce aux autres la nouvelle position de la balle à chaque tick. Lors de la réception de ce message, les autres clients comparent alors les valeurs reçues avec d'autres qu'ils avaient eux mêmes pré-calculés, afin d'éviter que ce client annonceur ne triche et déplace la balle où bon lui semble. Pour cela, à tout moment chaque client possède en mémoire la position actuelle de la balle ainsi que celle qu'elle aura au tour prochain, nommée dans l'implémentation "ghost", puisque qu'une balle ne se trouve pas réellement à cet endroit, c'est une balle "fantôme".

3.3.2 La raquette

Afin d'éviter toute triche concernant le déplacement des raquettes, les clients doivent, lors de la réception d'un déplacement de raquette, vérifier si cette position est atteignable depuis la précédente avec une vitesse de raquette normale. Par exemple, le joueur a gauche a une vitesse maximum de VITESSE_MAX en y et une vitesse maximum de 0 en x. Il faut alors vérifier, en admettant que la dernière position soit (old_x, old_y) et la nouvelle (new_x, new_y), si

$$(old_y + VITESSE\_MAX \leq new_y) \wedge (old_y - VITESSE\_MAX \geq new_y) \quad (1)$$

et réciproquement pour x.

Il faut aussi vérifier que plusieurs déplacements ne sont pas faits chaque tick par une même raquette. Cela se réalise assez facilement en ajoutant un booléen hasMoved à chaque joueur valant true si le joueur a déjà déplacé sa raquette depuis le dernier tick.

3.3.3 Le maître

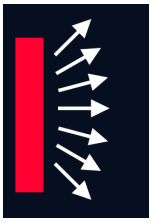
Il faut bien entendu tester si les messages que l'on reçoit sont cohérents par rapport à leur envoyeur, n'importe qui ne doit pas pouvoir se faire passer pour le maître actuel.

3.3.4 Aies confiance... Crois en moi...

Une des raisons pour laquelle il nous a été demandé de faire ce projet avec une approche acentrée est que pour permettre à chaque client de ne pas aveuglément faire confiance aux autres et de pouvoir tout vérifier soi même. Seulement l'approche que nous avons utilisée, où chaque client demande au client maître de bouger et ne bouge effectivement que lorsque celui ci le confirme, pose un problème de confiance majeur. En effet, s'il le souhaitait, le maître pourrait "tricher" et ne déplacer les gens que lorsque cela l'arrange.

3.4 Les collisions

Pour rendre le jeu intéressant, et pour inciter les joueurs à prendre des risques nous avons implémenté des collisions faites maisons. Lors d'une collision d'une balle avec une raquette, on lui assigne comme position le projeté orthogonal de sa position sur la surface de la raquette rencontrée et on va calculer sa nouvelle vitesse. Celle ci dépend uniquement du point d'impact :



On calcule tout d'abord l'angle de projection en déterminant la proportion de raquette en "haut" de la balle (sur le schéma, à adapter pour chaque bord) et à l'aide d'un produit en croix on lui assigne un angle en fonction d'un angle minimum atteint à l'extrémité haute de la raquette et d'un angle maximum atteint à l'extrémité basse. On trouve alors les coordonnées du nouveau vecteur vitesse de la balle en multipliant respectivement le cosinus et le sinus de cet angle par la norme du vecteur vitesse qui est une constante du code.

3.5 Qui sera le grand vainqueur ?

Pour dynamiser notre jeu, nous avons fais en sorte que notre Pong possède des "rebords" sur lesquelles la balle rebondit, pour qu'elle soit en mouvement permanent. La partie ne subit donc aucune interruption et l'attention du joueur ne doit donc se relâcher à aucun moment. Cette dynamisation du jeu inclut aussi le fait que lorsqu'un joueur perd il est "déconnecté" de la partie. En réalité, les autres clients ferment leur(s) connexion(s) avec le(s) perdant(s) et l'écran de fin de partie apparaît sur l'écran de celui/ceux ci.

3.6 Une autre implémentation ?

Lorsque nous avons commencé ce projet, nous en avons réalisé une première version en utilisant le principe de client/serveur : Un serveur de jeu tourne en permanence, attendant les connexions de chaque client. Une fois que tous les clients nécessaires au bon déroulement d'une partie sont présents, la partie se lance. Tout au long de celle ci le serveur sert d'intermédiaire entre les clients. Chaque client envoie ses informations (positions) au serveur, qui "met à jour la partie" et envoie aux clients la dite partie mise a jour.

Le serveur gère tout ce qui se passe dans le Pong : déplacement de la balle, collisions etc... Les clients n'ont alors besoin que d'afficher et d'envoyer les actions qu'ils désirent effectuer au serveur. Pour communiquer, chaque client possède deux threads : Un thread d'émission qui permet d'envoyer les messages au serveur, et un thread de réception qui permet de recevoir les messages du serveur. Le serveur quant à lui, fonctionne exactement de la même façon et possède lui aussi ces deux threads.

Cette implémentation nous a coûté un temps assez considérable sur l'avancée du projet.

3.7 Conclusion

Tout au long du développement de ce projet, nous avons rencontré diverses difficultés qui nous ont permit de découvrir les avantages et les inconvénients des différentes "organisations réseau", notamment ceux des réseaux acentrés et ceux des réseaux centralisés. Nous avons en effet pu comparer la complexité d'une implémentation avec clients multiples avec une implémentation de type clients/serveur. Cependant, à travers le développement du système d'anti-triche, nous nous sommes rendu compte qu'une implémentation acentrée permet un bien meilleur contrôle sur la partie mais au prix d'une complexité assez déroutante.

Dans ce projet, nous aurions voulu développer plus de fonctionnalités telles :

- **Les bonus :** Ceux ci étaient extrêmement intéressants à développer et nous y avions déjà réfléchi cependant il nous était impossible de les implémenter alors que notre système anti-triche n'était pas finalisé.

- **La recherche de groupe :** Un système qui permettrait, non pas de se connecter a des clients connus à l'avance mais de détecter les joueurs disponibles sur le réseau afin de créer une partie.

- **Améliorer les collisions :** Nous voulions, pour rendre le jeu plus intéressant, que la prise de risque influence de manière significative une partie et que ainsi une collision dite dangereuse (sur les bords de la raquettes) augmente la vitesse de la balle. De plus, nous aurions aimé améliorer le réalisme des rebonds en adaptant ceux ci en fonction de la vitesse des raquettes et de la balle au moment du choc. Nous n'avons pas pu développer cette fonctionnalité non plus encore une fois a cause du système d'anti-triche non finalisé.

Durant toute la période de développement du projet nous n'avons eu de cesse d'acquérir de nouvelles connaissances que ce soit sur la partie réseau que sur la programmation objet en elle même, mais surtout sur la gestion du travail, et l'importance du travail en groupe.

3.8 Bibliographie

Nous avons principalement utilisé la documentation JAVA:
`https://docs.oracle.com/javase/7/docs/api/`

Nous avons aussi utilisé le forum d'entraide :
`https://www.stackoverflow.com`