Table des matières

Rapport TM1C : Projet Réseau

Auteurs : Alexis Perignon, Alexis Flazinska

25 avril 2018

1 Introduction

Dans le cadre de notre cours Réseau, nous avons eu comme mission de procéder à la transformation de la version d'un jeu solo (un seul joueur peut jouer) à sa version à multiples joueurs.

Le jeu en question est Bomberman, où le joueur incarne un poseur de bombes, le but étant de faire exploser les adversaires/ennemis pour gagner. Le jeu a connu un grand succès, surtout grâce à son mode multijoueur qui, suivant les machines, permet de jouer jusqu'à une dizaine de personnes en même temps. Celui-ci a été codé avec le langage de programmation Python 3 ainsi que la bibliothèque Pygame : l'adaptation de la SDL au service de Python, mais est aussi constitué de quelques ajouts et modifications de son auteur.

Notre principale mission est donc d'implémenter le jeu Bomberman donné à l'aide d'un serveur centralisé ne réalisant pas d'affichage graphique maintenant à jour l'état courant du jeu. Les utilisateurs seront en charge de l'interaction avec le serveur, chaque client dispose d'une copie du modèle du jeu en cours qu'ils devront maintenir à l'aide d'échanges réseaux avec le serveur. Ses objectifs principaux sont la récupération des diverses informations à travers le serveur et la gestion de celles-ci, ainsi que la gestion des erreurs relatives à ces échanges.

Dans un premier temps, nous aborderons la vue fonctionnelle du projet : c'est à dire ce qu'offre notre application, ce qui n'a pas pu être réalisé ou qui ne fonctionne pas correctement. Dans un second temps, nous décrirons l'architecture et l'implémentation de notre projet réseau : les moyens que nous avons utilisé, les technologies utilisées ainsi que les techniques opérées.

2 Vue d'ensemble

2.1 Ce qui fonctionne

Par rapport à la demande de base (les différents objectifs donnés sous Moodle), voici ce qui fonctionne :

- Plusieurs joueurs, plus de 2, peuvent jouer et envoyer des commandes au serveur simultanément sans problèmes.
- Lors de la connexion d'un client, il reçoit bien le jeu tel qu'il est (les bombes avec leur décompte actuel, les fruits, les personnages déjà présents et la carte).
- Chaque joueur peut se déplacer, manger des fruits et poser des bombes.
- Le modèle est correctement mit à jour à chaque modification de l'état du jeu validée par le serveur.
- Chaque client peut se déconnecter sans que cela perturbe le déroulement de la partie.
- Un client peut rejoindre le jeu en cours de partie et recevoir correctement toutes les données.
- Bonus non réseau – Les fruits ont des effets supplémentaires plus amusants.
- Bonus non réseau – Les joueurs sont bloqués par les bombes (nous y tenions trop, il s'agit de Bomberman).

2.2 Ce qui ne fonctionne pas

Par rapport à la demande de base, voici ce qui ne fonctionne pas :

- Si un client est déconnecté de manière «sauvage» -câble Ethernet débranché, ...-, cela peut provoquer un plantage du serveur.
- Pas d'autre erreur connue à ce jour.

2.3 Bonus

Nous allons tout de suite aborder les différents bonus que nous avons ajouté, avec la contrainte que ceux-ci soient orienté réseau :

- Des bombes apparaissent aléatoirement sur les joueurs, dont nous avons prit soin de limiter la fréquence tout de même pour que le jeu reste totalement jouable !
- La déconnexion du serveur ne fait pas planter les clients, ils reçoivent un message d'erreur .
- La connexion du client est faite de telle manière que plusieurs clients peuvent utiliser le même pseudonyme.

2.4 Ce qui pourrait être encore fait

Voici ce qu'on pourrait attendre du jeu, qui n'a pas été implémenté à ce jour :

- Que lorsque la partie se termine -qu'il ne reste plus qu'un seul joueur-, nous pourrions déclarer un vote pour la prochaine carte à se mettre en place, et que la partie recommence, sans aucuns problèmes et sans à avoir à relancer le serveur.
- Un chat pour pouvoir communiquer avec les autre joueurs avec une interface dans le jeu : Le client pourrait par exemple appuyer sur entrée, ce qui déclencherai l'apparition d'une petite fenêtre graphique où chaque joueur pourrait communiquer aisément et ce qui donnerai un rendu plus soigné qu'un chat dans le terminal.
- Une interface utilisateur, affichant par exemple des barres de vie, la liste de tous les joueurs connectés, les joueurs restants (i.e ceux qui sont encore en vie, le nom de la carte (et pourquoi pas ses dimensions)) et les scores actuels des joueurs...

- La possibilité de s'enregistrer sur le serveur et donc d'avoir un compte, pour avoir la possibilité de sauvegarder une progression.
- Qu'un serveur puisse lancer plusieurs parties sur différentes cartes, et que les joueurs puissent naviguer entre chacune d'elle, choisir une « room » (un canal où jouer), et la possibilité d' en changer lorsqu'ils ont fini leur partie...

Cette liste est bien sûr non exhaustive, selon l'imagination et les attentes de chacun, beaucoup de bonus ou de fonctionnalités non présentes ici pourraient voir le jour .

3 Architecture et implémentation

3.1 Choix techniques

A chaque tentative de connexion au serveur, le serveur en question récupère les informations relative au socket client et l'assigne dans un dictionnaire, utilisant comme ID l'IP du client, ou son pseudo une fois le pseudo défini (le pseudo n'est pas défini lors de la connexion avec le paramètre donné par l'utilisateur, car si un autre utilisateur utilise déjà ce pseudo, nous ajoutons un numéro généré aléatoirement entre 0 et 100 et le concaténons au pseudo voulu, pour lui laisser malgré tout le pseudonyme désiré mais nous laissant ainsi une identification unique dans notre base de données). Le serveur utilise ensuite ce dictionnaire pour lire les informations reçues et envoyées les modifications qu'il a traité.

Côté envoie des données, le client recevant pour la première fois les fruits et les bombes, celui-ci aura un "traitement spécial" des données (il s'agit de la phase d'initialisation des données client), car une fois la première lecture terminée, celui-ci ne traitera plus les informations concernant ces éléments, car la pose des bombes, leur décompte et explosions, ou le fait que les fruits soient mangés par les personnages sont gérés par le client, une fois que la commande correspondante, donnée et envoyé par un client , a été validée par le serveur.

Le choix technique principal s'avère peut-être peu recommandé car non optimisé, mais il est assez simple à mettre en œuvre et évite des cas d'erreurs bêtes : nous envoyons à chaque tick, côté serveur, sa version du modèle à chaque client, sans distinction de s'il y a eu modifications ou non (réduisant donc tout de même le temps de calcul), et tentons de recevoir et de traiter une

éventuelle commande d'un des clients. Tandis que, côté client, à chaque tick aussi, nous recevons et traitons l'information envoyée par le serveur. Chaque client a aussi la possibilité, à tout moment, d'envoyer une commande au serveur (ici à l'aide de son clavier).

Étant donné qu'on reçois et envoie des informations côté client, comme côté serveur, à chaque tick, mais aussi des informations plus irrégulières comme les commandes du client ou la déconnexion/ la mort d'un joueur ; nous utilisons des sockets non-bloquant, de manière à ne pas attendre qu'un joueur bouge pour afficher le décompte des bombes, ou pour lui afficher les mouvements des autres joueurs.

3.2 Le protocole réseau

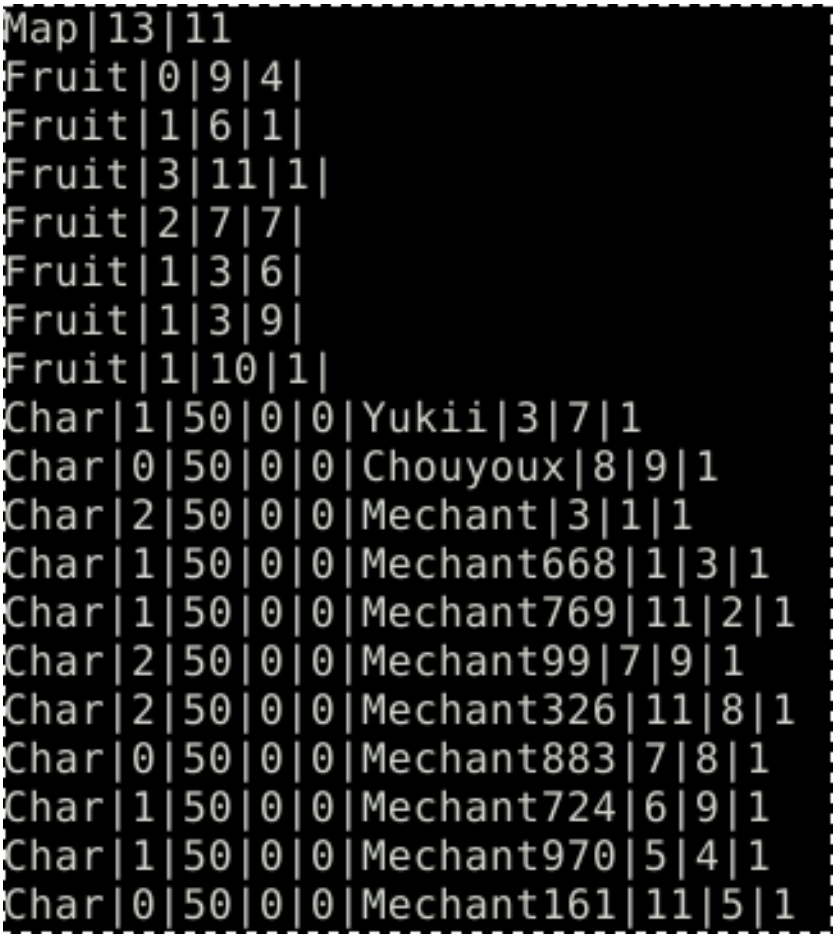
Le protocole de transport utilisé est le TCP, car nous avons besoin d'un flux continue d'informations, car certaines informations sont envoyées très régulièrement (nous ne sommes pas sur un jeu tour à tour), comme le décompte des bombes ou les déplacements des personnages. Cela permet aussi une meilleure gestion des erreurs et surtout la possibilité d'être en mode "connecté" contrairement au protocole UDP .

Côté envoie des données, nous envoyons des chaînes de caractères en claire aux clients et au serveur. Les informations sont tirées du modèle représentant l'état actuel du jeu pour le client et pour le serveur (variables concernant les joueurs, la carte, les bombes et les fruits). Nous séparons les informations par avec un séparateur pour les traiter correctement une fois reçues par le client/serveur, et utilisons un préfixe pour distinguer ce que concerne l'information envoyée. Par exemple, une information sur un personnage pourrait être "Char|"+"Info1"+"|"+"Info2"+"|"+"Info3" etc.

Le client, comme le serveur, savent que Info1 correspond, par exemple, à la position du personnage, tandis que Info2 correspond à son pseudo, et chaque information est découpée par le caractère "|".

De manière plus générale, le protocole réseau sur les informations concernant le modèle est délimité ainsi :

- Le premier champ correspond à la nature de l'information, en somme les personnages (Char), les bombes (Bomb), la carte (Map) et les fruits (Fruit).
- Il existe ensuite des champs relatifs à ces informations qui seront délimités chacun par un pipe (" | "), on notera notamment les informations



qui reviennent souvent : les positions, le décompte des bombes et les événements claviers.

Lors de l'envoie du modèle, on regroupe toutes les informations concernant les bombes, les personnages, les fruits et la carte en une seule chaîne de caractère, séparées par des "\n". Une fois reçu, l'information est découpée par "\n" pour obtenir des "lignes de commandes" comme expliqué ci-avant. (ex : "Char|...\n Char|...\n Fruit|...").

Certains tag spéciaux sont utilisés pour les informations qui sont envoyées seules, comme la déconnexion ou la mort (KILL), le pseudo du client (qu'il envoie au départ, Pseudo), le placement des bombes en cours de partie (BOMB), et un mouvement commandé par un client (Move).

3.3 Pseudo-code Protocole réseau

//Initialisation
Coté Client :

- Création du socket
- Connexion du socket
- Envoi du pseudo

Coté Serveur :

- Création socket
- Vérification / Modif pseudo
- Assignation du socket dans le dictionnaire des connexions clients

// Tick
Coté Client :

- Reception du modèle et application des mises à jour .
- Si première reception traitement fruits et bombes déjà présentes sinon traitement déplacements / morts ...

Coté Serveur :

- Envoie du modèle (Bomb , Fruits ...)
- Reception du modèle serveur

//Evenements Commandes :
Coté Client :

- Envoi de la commande (Move |, BOMB |, KILL|)

Coté Serveur :

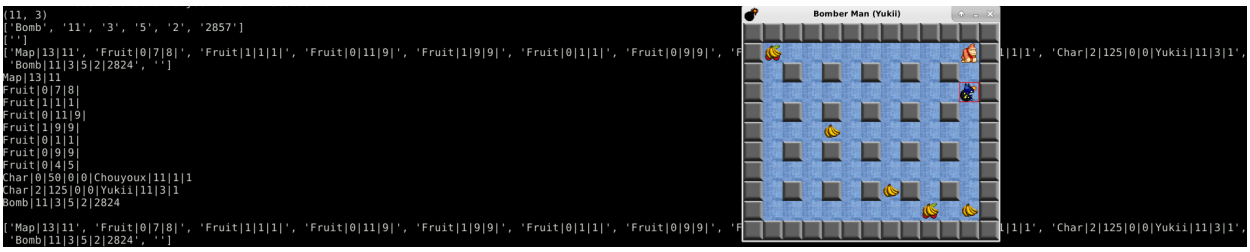
- Rien : la commande sera traitée lors du prochain tick

// Disconnect :
Coté Client :

- Déconnexion
- Fermeture Socket /Programme

Coté Serveur :

- Tue le joueur sur son modèle pour le prochain envoie
- Fermeture socket et suppression dans le dictionnaire



4 Conclusion

Dans l'optique de notre cours de réseau, ce projet nous a permit de mettre en application les différentes notions que nous avions appris au fil du semestre. Ainsi, et de par la liberté que nous avions au sein de ce projet, nous avons du effectuer des choix entre différentes technologies, peser le pour et le contre des méthodes que nous voulions utiliser, organiser nos données, réfléchir à un protocole réseau qui conviendrait à la situation et nous avons dû mettre en œuvre différents tests pour essayer de répondre le plus possibles aux attentes sur les différentes erreurs possibles.

De plus, le fait de transformer un jeu à un joueur vers un jeu multiple-joueurs était ludique, intéressant et nous a permis de consolider certaines connaissances qui pouvaient être encore fébriles, et nous avons pu également découvrir le principe des classes en Python, bien qu'en dehors du cadre du réseau, que nous pourrons réutiliser plus tard dans de futurs projets.

Ainsi nous pensons aujourd'hui pouvoir vous rendre un projet fonctionnel du jeu Bomberman en version multijoueur et nous espèrons que nous aurons su vous détailler tout au long de ce rapport.