Rapport de projet réseau

DELAR Emmanoe, RAKOTOARIJAONA Camille

26 avril 2017

Table des matières

1 Introduction

Lors de ce projet, réalisé en binômes, nous avions pour objectif de développer un jeu de bataille navale en réseau. Pour cela, nous avons utilisé le langage de programmation objet Python. On est parti du code source fourni par nos enseignants. Ce code nous permettait de jouer contre la machine uniquement. Dès lors, nous avons dû l'améliorer afin de pouvoir jouer à 1 contre 1 sur le réseau.

1.1 Présentation du jeu

La bataille navale est un jeu de société dans lequel deux joueurs doivent placer des navires sur une grille tenue secrte et tenter de toucher les navires adverses. Le gagnant est celui qui parvient à torpiller complètement les navires de l'adversaire avant que tous les siens ne le soient.

2 Fonctionnement du programme

2.1 Jouer contre la machine

Après avoir lancé le serveur, lorsqu'on lance le programme avec le numéro d'hôte de celui-ci en argument («main : :»), si on choisi de ne pas jouer en réseau alors le jeu commence directement. Quand le jeu commence, on choisi la colonne et la ligne à viser. Quand c'est au tour de l'ordinateur, une fonction fait de même en choisissant des coordonnées aléatoirement et joue son coup. Et ainsi de suite jusqu'à la fin de la partie.
On tient à préciser que l'address non spécifiée 0 :0 :0 :0 :0 :0 :0 :0 ou : : signifie que le port respectif n'est pas en mode «écoute » a une adresse précise mais toute.

2.2 Jouer en réseau local

Si on choisi de jouer en réseau, comme demandé dans le sujet, il faut dun côté, démarrer le serveur en lançant le programme «main.py» sans arguments, et du côté client, il faut lancer le programme avec ladresse ip du serveur en argument.
Lorsque ceux-ci sont lancés, le client aura la possibilité de choisir d'abord si il veut se connecter au serveur, ou si il veut jouer seul face au robot comme décrit au dessus. Si il décide de jouer en réseau, le programme attendra qu'une autre personne se connecte au serveur pour qu'ils puissent jouer ensemble.
Ensuite, chacun jouera à tour de rôle jusqu'a ce que la partie soit terminée, lorsque la partie est terminée chaque joueur verra le score, et le serveur se déconnectera et le jeu de chaque joueur aussi.
Il est à noter que les joueurs peuvent donc jouer tout seul peu importe le moment sans affecter le serveur lorsqu'ils jouent en local.

3 Structure

Contrairement à la partie contre l'ordinateur, en mode réseau, les coordonnées choisies par le joueur client doivent être envoyés au serveur pour qu'il puisse les retransmettre à l'adversaire. Nous avons alors créé un protocole de communication serveur/client TCP sur le port 7777. Ce port nous permettra de recevoir ou d'émettre des informations.

3.1 Serveur

La programmation du serveur à été un peu plus complexe car il a pour tâche de faire le lien entre les joueurs.
Au lancement du programme, nous nous retrouvons dans la fonction principale ≪main ≫ en attente de demande de connexion, tout cela se fait par la méthode ≪.select≫ vu en cours. Lorsque 2 joueurs se connectent, le serveur leur envoie à l'aide de la méthode ≪socket.send ≫, leur numéro, 0 ou 1 selon leur ordre de connexion. Ce numéro défini celui d'entre eux qui jouera le premier.
La deuxime étape, nous a demandé beaucoup plus de réflexion car nous avons choisi d'envoyer le jeu créé par le serveur, aux clients connectés.
En effet, créer un jeu aléatoire depuis le serveur puis l'envoyer aux clients nous offre beaucoup de possibililitées. Par exemple implémenter l'extension qui nous permet d'avoir des spectateurs devient plus simple car nous pouvons donc leur envoyer ce même jeu.
Pour parvenir à cela, nous avons dû modifier la fonction ≪randomConfiguration≫ qui génére aléatoirement une table de bateaux.
Dans cette fonction, tout se passe normalement, puis à chaque fois que l'on génére les coordonnées et la position d'un navire, on stocke ces informations dans un tableau. Une fois la génération terminée, on envoie le contenu du tableau, bit à bit (en utilisant la méthode ≪client.send ≫), aux joueurs/spectateur(s) connectés avec une boucle qui parcourt le tableau.
Cette partie nous a posée plusieurs problèmes, par exemple, au niveau de la taille des bateaux, les valeurs envoyés devaient être de type ≪bytes≫ . La taille de ces variables pouvait varier, de 1 à 2 bytes car les valeurs générées par la fonction ≪randomConfiguration≫ allaient de 1 à 10 et 10 est représenté sur 2 bytes. On a donc décidé de décaler de 1 chaque valeur pour avoir un interaval de 0 à 9 et ainsi avoir une taille au maximum de 1 byte pour chaque valeurs envoyées. La valeur reçue puis convertie en entier de l'autre côté par le(s) joueurs/spectateur(s) serait incrémentée de un.
Un autre des problèmes complexe que nous avons fixé à l'aide du débogage, provenait du fait que la fonction ≪isValidConfiguration≫ retournait de base plusieurs valeurs alatoirement sans avoir au préalable vérifié si c'était une configuration valide pour notre instance de jeu. Nous avons alors compris qu'il fallait modifier cette fonction afin qu'elle retourne les configurations valide et envoie le tout aux clients connectés.
Comme la fonction de base, la nouvelle fonction retourne un navire. On répète alors l'opération deux fois pour avoir notre table de jeu. Ensuite, c'est le début

de la partie, la boucle qui fait jouer le serveur est la même que celle du robot et des joueurs. Cependant, celle-ci se place dabord dans la partie du premier joueur et attendra qu'il envoie son coup. Il verra le coup s'afficher sur sa table de jeu et l'enverra à son adversaire, tout cela avec la méthode ≪recv≫ et ≪.send≫ . Le tour passera ensuite au deuxième joueur puis la boucle fera de même et ainsi de suite jusqu ce que le jeu soit fini. Une fois le jeu terminé, le programme sort de la boucle et libère les sockets puis ferme le serveur.

3.2 Connexion client

Pour que le client se connecte au serveur, il nous faut le nom de l'hôte et le numéro de port.
Ensuite, on va créer une socket (client) de type ipv6 qui va nous permettre d'ouvrir une connexion avec une machine locale et d'échanger des informations (à l'aide de la méthode ≪.bind≫).
Pour récuperer les premieres informations, notamment le numéro du joueur et les navires, on intercepte les premiers bytes envoyés par la socket serveur (à l'aide de la méthode ≪client.recv≫).
Avec ces informations, le client peut afficher la partie générée par le serveur grâce à la fonction Game.
Une fois la partie crée puis lancée, si c'est au joueur actuel de jouer, il entre les coordonnées de la grille adverse à viser. Ensuite, on va envoyer ces coordonnées (x,y) sous forme de paquets de bytes au serveur. Le serveur va retransmettre ces paquets de byte à l'autre joueur (à l'aide de la méthode ≪client.send≫).
Par contre si c'est au tour de l'autre joueur, la socket client se met en mode réception et attend les coordonnées choisies par l'adversaire (à l'aide de la méthode ≪client.recv≫). Ces données sont également envoyées par le serveur.
Les données reçu sont converties en coordonnée de type entier puis passer en arguments de la fonction ≪addShot ≫ . Cette fonction tente de viser la flotte ennemie à ces coordonnées. Si le tire est juste, l'adversaire voit la case correspondante barrée, sinon la case est entourée pour représenter un échec.
Ce protocole est répété jusqu'à la fin de la partie.

3.3 Protocole de communication

Une fois la connexion établie, la communication serveur/client est très simple. Chaque information nécessaire à l'initialisation d'une partie est envoyé par le serveur bit à bit et ainsi récupéré du côté client. Il faut noter la nécessité à convertir les bytes recus en entier (int).

4 Extensions

4.1 Joueur vs robot

Pour implémenter l'extension ≪jouer vs robot ≫ , il nous a juste fallu mettre une condition et si elle est respectée, on lance le programme initiale qui simule déja une partie joueur/robot.

4.2 Observateurs

L'extension ≪Observateurs ≫ nous a demandée une fonction en plus. Cette fonction définie l'observateur comme un joueur en plus et initialise sa nouvelle table de jeu. Ce n'est qu'au moment d'afficher la table de jeu que ça change. Notre fonction ≪observator ≫va demander au serveur d'afficher deux tables à la fois ≪displayGame(game,0) ≫et ≪displayGame(game,1) ≫respectivement la table du joeur 0 puis celle du joueur 1 . À la fin, cette fonction teste également si le jeu est terminé et lequel des joueurs l'a remporté puis l'affiche. Et ainsi, nous avons un spectateur qui voit les coups joués par les deux joueurs.

5 Conclusion

Sur une période d'un semestre, les différentes notions sur l'organisation générale des protocoles réseaux vues en cours nous ont permis de réaliser ce projet. À l'aide de la bibliothèque standard ≪socket ≫ du langage de programmation objet Python. Cette interface de programmation bas-niveau permet aux applications de communiquer en passant directement par la couche transport du modèle OSI. Nous avons alors créé un serveur de jeu bataille navale en réseau.
Tout au long du projet, nous sommes tomber sur de nombreux problèmes par rapport au mode de communication de ce système. Par exemple pour que deux sockets communiquent, il faut échanger des paquets de bytes et non pas des entiers. Nous avons aussi vu que ...
En prenant du recul, nous pouvons dire que le projet nous a apporté un bagage supplmentaire. Nous avons consolidé nos connaissances générales sur la couche transport du systeme OSI, la programmation réseau bas niveau et avons aussi appris à coder à un niveau supérieur.