INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

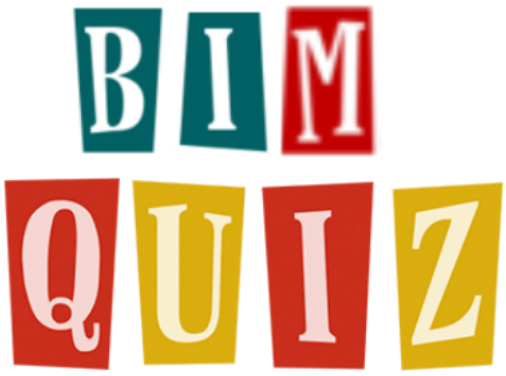
Programmation réseau

4ème année Biosciences 2018 - 2019

Rapport sur la création d'un Quiz en réseau

Marie Codet, Alice Genestier, Elise Jacquemet, Cloé Mendoza

Enseignant : Christophe Pera



29 mars 2019

1 Introduction

Dans le cadre du module de programmation réseaux, nous avons décidé que notre projet porterai sur un Quiz en réseau.
Ce programme permet à plusieurs utilisateurs de répondre à des questions de type vrai/faux, en multi-joueurs, sur un thème de leur choix (parmi trois propositions) et d'obtenir leur score et leur classement à la fin de la partie.

2 Architecture logiciel

Pour ce projet, nous avons utilisé le langage Python 3 pour créer un système client/serveur TCP, avec le module **multiprocessing**. Ce dernier permet en effet de gérer les connexions client-serveur de manière simultanée.
Nous avons créé deux fichiers : un correspondant au code serveur nommé `Serveur.py` et un autre correspondant au code client nommé `Client.py`. Le serveur doit être lancé par la commande `python3 Serveur.py` avant que le fichier client soit lancé, ce dernier pouvant être lancé autant de fois que l'on veut de joueurs. Les clients peuvent être lancés sur n'importe quelle machine appartenant au réseau INSA. Plusieurs possibilités sont offertes à l'utilisateur pour lancer le client :

1. `python3 Client.py <pseudo> <adresse serveur>` est à utiliser lorsque le serveur n'est pas en local : la commande lancera le client avec le pseudo donné en premier argument, et l'adresse IP de la machine sur laquelle le serveur a été lancé. Les tests ont été réalisés en 4BIM avec des adresses de type `134.214.XXX.XXX`
2. `python3 Client.py <pseudo>` est utilisé pour un seveur local, l'adresse par défaut du serveur étant `127.0.0.1`. Si le joueur ne rentre pas son pseudo dans la commande il sera demandé de le faire à l'exécution du programme.

À chaque connexion client, le serveur lance un nouveau processus et stocke dans un dictionnaire la socket et le pseudo du nouveau client (attribut `connexions`). Nous avons choisi d'utiliser un objet **Manager().dict** pour stocker ces informations, afin qu'elles soient partagées et synchronisées entre les différents processus. Une fois connectés, les joueurs doivent choisir entre trois options :

- Attendre que d'autres joueurs se connectent (`wait`) : le client passe alors en mode lecture.
- Lancer une partie (`start`) : débute la partie au niveau du serveur.
- Quitter le serveur (`quit`) : termine la connexion du client. la saisie de ctrl+C produira le même effet tout au long du programme.

La saisie d'une chaîne invalide ou l'absence de réponse du client pendant plus de 30 secondes produira le même effet que `wait`.
Le code `Client.py` gère donc tout ce qui est propre à un joueur et la communication avec le serveur, via le processus `handle_conn()` qui traite les données envoyées par le Client. Toutes les lectures de socket (`recv()`) côté serveur sont situées dans cette fonction. Les communications sont ensuite traitées différemment selon les cas :
Pour un client déjà en jeu, les données envoyées sont partagées entre les processus via un objet **Manager().Queue()** (attribut `data`). Pour un nouveau client, elles sont simplement lues par le serveur.
Le lancement de la partie fait l'objet d'un processus à part nommé `partie()`, qui sera lancé par le processus du client ayant saisi la commande `start`. Ce processus accède aux données partagées pour interagir avec les différents clients. Au lancement d'une partie, les joueurs présents sont ajoutés au jeu qui se déroule comme suit :

1. création d'un dictionnaire `joueurs` contenant le pseudo (clé) et le score (valeur) de chaque client en jeu.
2. choix du thème et du nombre de questions envoyées au lanceur
3. importation des questions depuis le fichier `questions_quizz.csv` de stockage et sélection selon le thème et le nombre de questions choisies (thème aléatoire et 3 questions par défaut)
4. envoi d'une question à tous les joueurs
5. collecte des réponses simultanément (possible grâce à la queue `data`) et stockage dans une liste temporaire. Cette étape est chronométrée.
6. parcours de la liste des réponses et traitement en conséquence : mise à jour des scores et envoi du résultat aux joueurs.
7. les étapes 4 à 6 sont répétées jusqu'à ce que toutes les questions aient été posées, puis le classement est calculé et envoyé aux joueurs.
8. on termine la partie par la fonction `fin` qui permet aux clients de revenir à l'état d'écriture de départ (choix start/wait/stop)

Le chornométrage de l'étape 5, ainsi que de la saisie de départ est possible grâce à la fonction `select()` du module **select** qui se termine soit lorsqu'un changement d'état d'une socket est détecté, soit lorsque le timeout est terminé. Nous avons donc utilisé cette fonction pour détecter une entrée clavier dans `Client.py`, de sorte qu'à chaque fois que le serveur demande une saisie, notamment pour une question, le client dispose de 30 secondes pour répondre, sans quoi il retourne un mot clé signifiant qu'il n'a pas répondu à temps et repasse en mode lecture.

3 Travail en groupe et difficultés rencontrées

Notre groupe était constitué de quatre étudiantes. Au départ, nous nous sommes partagées les tâches (serveur, client, questions), puis nous avons tout regroupé. Une fois les différents codes mis en commun, nous avons essayé de nous retrouver régulièrement (une fois par semaine) pour résoudre les problèmes rencontrés. Nous allons ici évoquer brièvement les principales difficultés rencontrées et les choix technologiques associés pour y répondre :

Si le module multiprocessing a été essentiel pour gérer les connexions multiples en simultané, il a néanmoins posé de nombreux soucis de synchronisation. Premièrement, au niveau du stockage des connexions clients arrivant sur le serveur : au départ, nous avons utilisé un dictionnaire classique passé en attribut de la classe serveur, pensant qu'il serait mis à jour par chaque processus. Or, les processus fonctionnent avec des espaces mémoire séparés, il était donc impératif d'utiliser un objet spécifique du module multiprocessing. Nous avons donc commencé par utilisé une Queue, mais son utilisation n'était pas adéquate : les données sont stockées "en vrac" et la lecture d'une valeur via `queue.get()` retire la valeur lue de la queue. Des problèmes de synchronisation étaient également observés. Notre choix final s'est donc porté sur un dictionnaire partagé via la classe `Manager()` qui synchronise automatiquement les connexions, et permet d'avoir un contrôle de l'unicité des connexions. La recherche d'un client particulier est également possible, là où nous étions obligées de parcourir et re-remplir la queue à chaque fois.

Ce choix technologique nous a poussé à réfléchir à la clé qui serait utilisée pour stocker les connexions. Nous avons commencé par utiliser les sockets des clients, en plaçant le pseudo en valeur. Cependant lors des tests réalisés nous avons observé des dysfonctionnements, notamment liés à des `Key error` : les sockets des clients ne sont en effet pas complètement fixées, l'attribut `fd` pouvant varier au cours de l'exécution. Nous avons donc inversé le dictionnaire en

plaçant le pseudo comme clé, et avons donc ajouté une sécurité afin d'assurer son unicité : un nombre aléatoire est rajouté au pseudo si celui-ci est déjà utilisé.

Nous souhaitions également que le serveur puisse détecter qu'une partie était déjà en cours et qu'il informe les nouveaux clients : nous avons ici aussi dû utiliser une variable partagée grâce à la classe `Valeur` qui fourni également un verrou, afin de s'assurer qu'un seul processus ne modifie la variable partagée.

Le deuxième problème de synchronisation a été la gestion des communications au sein de la partie. En effet, nous avions au départ utilisé `recv()` directement dans le processus `partie()` pour lire chaque socket. Or, cette fonction bloquait régulièrement le programme, sans qu'il y ait eu de déconnexion de la part des clients. Nous avons donc choisi d'utiliser `select()` afin de détecter les sockets qui envoyaient des réponse au serveur, puis de les regrouper dans une liste avant de les lire. Nous nous sommes alors rendues compte que les entrées clavier n'étaient pas détectées correctement, car les clients ne communiquaient pas directement avec le processus `partie()` mais avec le processus initié à leur connexion `handle_conn()`. Nous avons donc choisi de n'utiliser `select()` et `recv()` que dans ce dernier processus, et d'utiliser une `Manager().Queue` dans laquelle les données à envoyer à la partie sont stockées. La queue est ensuite lue (et vidée) par le processus `partie()` aux moments nécessaires.

Le programme a fonctionné assez rapidement avec un seul client. Nous avons utilisé cette version simplifiée pour gérer les déconnexions imprévues, puis nous avons testé avec plusieurs clients. Le fonctionnement est correct, lorsqu'un client est déconnecté inopinément les autres peuvent continuer à jouer. Un seul cas particulier pose cependant toujours problème : si le lanceur de la partie se déconnecte alors qu'il est connecté avec d'autres joueurs. En effet, le processus `partie()` est lancé par le processus `handle_conn()` initié par le client qui lance la partie. Si ce dernier se déconnecte, le processus est terminé et par conséquent ses processus enfant ne fonctionnent plus. Nous avons essayé de prévoir ce problème en utilisant des exceptions, mais le comportement du programme reste aléatoire : la déconnexion est parfois bien détectée, parfois non, une exception n'est pas toujours renvoyée. Le serveur n'est en revanche pas interrompu et les clients encore présents peuvent relancer une partie.

Enfin avons eu certains problèmes sur l'importation des questions, notamment à cause de caractères non reconnus par le serveur, des séparateurs différents... Au final, un traitement des questions et une restructuration du fichier ont été effectués et nous avons choisi d'utiliser un fichier .csv pour importer les questions simplement.

4 Améliorations possibles

Notre projet est un quiz en réseau assez simple mais on peut réaliser plusieurs améliorations :

- Faire choisir au joueur en début de partie le temps pour répondre aux questions
- A l'instar de quiz type Quizup, on pourrais faire en sorte que le joueur qui répond en premier a plus de points (ou nombre de points attribué au joueur inversement proportionnel à l'écoulement du temps)
- On pourrais faire choisir au joueur un thème à l'entrée dans le serveur et tous les joueurs connectés sur ce thème lancent une partie. Il pourrait donc y avoir plusieurs parties lancées en simultané.
- Pour aller plus loin on pourrait aussi faire comme dans "Tout le monde veut prendre sa place" et proposer un format Duo, Carré ou Cash ou le joueur choisi le format de question/réponse désiré.