Compte rendu Projet Réseau

Bataille Navale

par Zhengde QIU et Simon TRICOIRE

Sommaire

1-Description des protocoles

1.1-Côté client : Version normale (pas graphique)

Cette version du client est très simple, son seul rôle est de réceptionner des données provenant du serveur et d'afficher le contenu dans le terminal du client.
Il existe des cas spéciaux de réception de données, si le client reçoit un message où il y a écrit « yolo1234 »(la condition est volontairement aberrante pour ne pas survenir dans d'autres cas), alors cela appelle une fonction prompt qui affiche un « <You » et qui appelle aussi une fonction de base flush.

Un autre cas spécial se déclenche lorsque dans un message, il y'a écrit « [88] », là encore ce cas spécial nous permet de gérer l'affichage sur le terminal. En effet, de base si le serveur effectue deux send à la suite, notre client va afficher un « You » en trop, avec le [88] on évite ce cas-là.

Si le client ne reçoit rien du serveur, il est déconnecté de celui-ci, un message le prévient alors de la déconnection survenue.

Le client peut, bien entendu envoyer des données. Rien de particulier n'est mis en œuvre côté client pour l'envoi des données.

1.2-Côté client : Version graphique

Ici le client n'est pas tellement plus compliqué, la différence se pose seulement concernant la partie Tkinter. En effet, ici le client est implémenté de la même façon que précédemment, seul la partie graphique Tkinter change. De fait nous utilisons un thread s'exécutant en parallèle du code principale et nous permettant de gérer l'affichage graphique.

1.3-Côté serveur : Version normale (pas graphique)

Côté serveur, une boucle while nous permet de faire boucler notre programme, s'en suit une itération sur les socket présentent dans la liste. Si le socket détectée correspond à celle du serveur alors il s'agit d'une connexion d'un nouveau client, on lui envoi alors un message lui indiquant qu'il s'est connecté au serveur et on ajoute le socket correspondante à la liste des socket. Si le socket ne correspond pas à une nouvelle connexion, alors on va vérifier une à une plusieurs conditions.

Si le client s'est déconnecté, alors on gère sa déconnexion, en enlevant le socket correspondante de la liste des socket et en enlevant le client correspondant (voir description classe client en dessous) de la liste des clients.

Sinon on vérifie si le jeu est en pause, s'il est en pause, on doit vérifier si le client se reconnecte ou pas, s'il se reconnecte, le jeu reprend, autrement on gère normalement l'arrivée d'autres clients sur le serveur.

Sinon on vérifie si le client a bien un pseudo, s'il n'en a pas on lui demande d'en fournir un valide (cad pas déjà pris). Sinon on gère s'il y a « MSG » dans le message envoyé par le client, si oui, alors ce message est envoyé à tous les clients présents. Sinon on gère si le jeu est en cours et dans ce cas on affiche la grille de jeu pour le joueur qui joue et on lui demande d'entrer une colonne et une ligne correcte.

A l'autre joueur, on dit d'attendre les mouvements de l'autre joueur et aux observateurs on affiche la grille des deux joueurs.
Si jamais le jeu se finit, alors on revient à l'écran d'accueil où on peut à nouveau rejoindre des parties.

Sinon on gère les messages contenant « CMD » (affiche la liste des commandes), contenant « JOIN » (met le client dans la liste des playerReady), contenant « QUIT » (le client se déconnecte), contenant « MORE » (donne des précisions concernant les différentes commandes.

Sinon, finalement, on envoi « yolo1234 » au client pour lui signaler qu'entre deux send, on a pas reçu de data, ce qui permet de ne pas perturber l'interface graphique.

1.4-Côté serveur : Version graphique

Pour la version graphique du serveur, peu de choses changent, seulement nous ajoutons des mots clefs à la fin des messages envoyés pour gérer les différents cas d'affichage graphique Tkinter.

2-Choix d'implémentations et leur motivations

2.1-Class client

N’étant pas autant familier avec python (surtout sur la gestion en réseau) qu’avec le C, nous avons dès le début du projet entrepris de se renseigner sur Internet en fonction de ce que l’on cherchait à faire.
Très vite la nécessité et l’utilité d’implémenter une classe client nous est apparue. En effet, l’idée de pouvoir différencier chaque client et de lui associer des caractéristiques propres nous a convaincu.
Nous pouvions alors à partir d’un client, voir quel socket lui était lié, quel pseudo l’identifiait, quel était son statut ou encore quel était son nombre de victoire. Tout ceci a simplifié notre code et nous a permis un accès plus facile et plus rapide aux bonnes informations, autorisant une implémentation facilitée de modules tels que le module gérant les observateurs.

2.2-Vérification si le pseudo choisi existe déjà ou non

Alors que notre projet était déjà bien entamé et qu'il était possible de réaliser une partie sans encombre, dans l'optique d'implémenter le module de reconnexion par la suite et afin d'éviter des problèmes liés aux pseudos, nous avons décidé de vérifier si les pseudos proposés étaient déjà utilisés par des clients connectés et le cas échéant de demander à l'utilisateur d'en choisir un nouveau. Pour coder ce module, nous nous sommes appuyés sur un set (s_pseudoConnected) contenant le nom de tous les clients connectés, un pseudo étant rajouté à ce set une fois vérification faite. L'utilisation d'un set s'est faite de manière consciente puisqu'il permet la vérification de la présence en son sein d'un élément avec une complexité de 0(1). Ainsi même si notre set contient n pseudo, le temps de vérification n'en sera pas affecté. Il en aurait été autrement si nous avions choisi de l'implémenter par le biais d'une liste (qui a une complexité de 0(n) pour la recherche d'un élément)

3-Les extensions développées

3.1-Plusieurs parties / Counting Score

Il est possible de jouer plusieurs parties à la suite et à chaque victoire, la variable « win » contenue dans l'instance client du joueur en question s'incrémente. Si les scores sont comptés et stockées ils ne sont pas affichés en jeu. Ils nous seraient de fait facile d'y remédier.

3.2-Observateur

Nous avons développé l'extension gérant le troisième client comme observateur en nous appuyant sur notre classe client et plus particulièrement sur une variable la composant : le statut (« status » dans notre code).
En effet, si le statut est à « -1 » alors l'instance client correspondante sera détectée comme étant un observateur, de fait lors de la partie, quand on enverra des informations aux différents clients, les clients détectés comme étant des observateurs recevront la grille des deux joueurs.
Ceci est notamment géré par la fonction : fun_msgToOtherThanCurrentPlayer( ).

De plus, un client observateur, du fait de son statut, ne mettra pas le jeu en pause si jamais il se déconnecte en cours de partie.
Une fois la partie finie, un observateur peut bien entendu prétendre pouvoir jouer lui aussi.

3.3-Fin de partie, demande aux clients présents s'ils veulent jouer

De fait, comme indiqué juste au-dessus, nous gérons aussi cette implémentation.
En effet, comme le jeu est fini, tous les clients présents ont leur statut réinitialisé par défaut à celui d'observateur et le tableau contenant les joueurs (Player Ready) est lui aussi réinitialisé.

Ainsi les clients peuvent à leur guise choisir de rejoindre une partie (JOIN) ou de rester à nouveau observateur. Une fois que deux joueurs sont prêts, une partie est lancée.

3.4-Plusieurs Observateurs

Comme précisé au-dessus, nous gérons aussi cette extension, chacun des clients recevant la grille des joueurs. Cette extension s'est implémentée naturellement dès que l'extension gérant un observateur a été codé. En effet il nous a suffi d'ajouter les clients entrants à la liste des clients et de détecter s'ils étaient ou non des observateurs en fonction de leur statut.

3.5-Chat

Via une simple balise nous gérons aussi un module de chat. En effet si un client tape « MSG » suivi de son message, alors le message ira à tous les autres clients. Bien entendu, l'envoi d'un message alors qu'une partie est en cours ne pose pas de problème en théorie puisque même si la saisie du joueur n'est pas bonne on lui re-proposera de re-rentrer sa saisie.
Cependant, le spam de message n'est pas géré ce qui peut poser des problèmes de lisibilité.

3.6-Reconnexion Joueur

La reconnexion des joueurs est gérée de manière partielle, puisque nous ne gérons les reconnexions joueur seulement lorsqu'un client a été tué en cours de partie. En effet, si joueur est tué, alors la partie est mise en pause et si jamais le joueur en question se reconnecte avec le même pseudo que saisi précédemment, alors la partie sera relancée à son état antérieur. Il est possible qu'un autre client (ou même plusieurs) se connecte entre temps, il sera alors d'office observateur. Il est aussi possible pour le joueur restant, de choisir de ne pas attendre que l'autre joueur revienne, à ce moment-là, soit il se déconnecte du serveur et donc il n'est plus connecté, soit il choisit de revenir au lobby par le biais d'une saisie particulière («STOP»). D'ici, il peut choisir de lancer ou non une autre partie ou de rester

observateur de la prochaine partie en cours. Dans les deux cas, si le premier joueur s'étant déconnecté se reconnecte, il sera considéré comme un nouveau client avec des paramètres réinitialisés. Aucun timer n'est implémenté, de fait tant que le joueur ne se reconnecte pas ou que le joueur restant ne quitte pas la partie en cours, la partie reste en pause.
Enfin, la reconnexion des clients observateurs n'est pas gérée, leurs paramètres seront donc réinitialisés à chaque connexion, faisant d'eux de nouveaux clients.

3.6.1-Problèmes rencontrés

Cette partie à été particulièrement compliqué à implémenter, notamment du fait de notre implémentation particulière. En effet, la liste contenant nos sockets (l_socket) et la liste de nos instances client (l_client) partage des index communs en temps normal (lorsque le jeu n'est pas en pause) ce qui nous permet de gérer entre autres à qui on envoit les informations et ce que l'on envoit. Cependant, lorsqu'un des joueurs se déconnecte, le socket correspondant est nécessairement fermé et supprimé de la liste, modifiant ainsi les index de la liste de socket. Cette modification de la liste de socket nous a posé beaucoup de problèmes, provoquant des bugs lors de tentatives de reconnections. Une fois cette erreur repérée, nous avons décidé de nous baser plutôt sur les index de la liste de clients (nos instances clients contenant le nom de la Socket), de fait nous pouvions reconstruire notre liste de socket en se basant sur les mêmes index que notre liste de clients.
Pour gérer la reconnexion, comme dit plus haut, nous nous basons sur les pseudos, si le pseudo du nouveau client arrivant se trouve dans la liste des clients déconnectés (s_pseudoDisconnected), alors nous mettons le nouveau socket du client reconnecté à la place de l'ancien socket se trouvant dans l'instance client du joueur.
Tout ceci nous permet de relancer normalement le jeu.

Cette partie nous a pris un certain temps, parce que nous avons aussi essayé d'éviter au maximum les bugs provoqués par des manipulations particulières (ex: une partie est en cours, un des joueurs se déconnecte, un client se connecte sans

entrer de pseudo, un autre se connecte avec un pseudo différent et est donc reconnu comme un observateur. Finalement, le client qui n'avait pas entré de pseudo se connecte comme étant l'ancien joueur. Cette manipulation particulière posait problème puisqu'on se basait sur la valeur de certaines variables (b_newClient) pour tester si on avait bien à faire à une reconnexion de client, or le fait qu'un autre client se connecte entre temps réinitialisait ces variables et provoquait de fait des boucles sans fin).
Considérer ces manipulations particulières nous a permis d'implémenter de nouvelles choses au sein de notre code, notamment dans les statuts des clients (le statut «-2» qui correspond à un client n'ayant pour le moment pas entré de pseudo valide), ou encore la gestion de l'envoi de données par les clients encore connectés alors que le jeu est en pause, le tout évitant d'autres problèmes à d'autres endroits.

Après multiples tests, notre code semble assez robuste pour gérer les reconnections et les manipulations particulières (même si quelques bugs sont encore présents par moment), cependant par manque de temps nous n'avons pas pu implémenter un timer qui aurait déconnecté d'office le joueur parti trop longtemps. A la place nous avons décidé de proposer au joueur restant, le choix d'attendre ou non.

3.7-Implémentation graphique

A la fin du projet, comme Simon est parti à gérer la reconnexion, Zhengde de son côté est parti à implémenter un autre module : la partie graphique, ceci nous permettant d'éviter les conflits de versions.
La partie graphique a été implémenté via Tkinter.

4-Problèmes rencontrés non résolus

Au final le manque de temps ne nous aura pas permis d'implémenter la gestion du timer dans le cadre de la reconnexion.

Globalement, là encore avec plus de temps il nous aurait été possible de vraiment clarifier et optimiser notre code.
Ceci mis à part, nous n'avons pas vraiment rencontré de problèmes que nous n'avons pas su résoudre.

5-Répartition du travail

N'étant que deux dans le projet et afin de réellement disposer d'une base solide au début du projet, nous avions choisi de travailler tous les deux sur le même ordinateur. Le projet avançant à vitesse régulière et ne voyant pas trop comment scinder le code en deux, nous avons continué à travailler de la même façon. Finalement, à quelques jours du rendu projet, Simon s'est chargé de la partie reconnexion alors que Zhengde de son côté s'est chargé de la portabilité Tkinter. Les deux parties étant relativement éloignées, il nous a été relativement facile de travailler de manière scindée pour plus d'efficacité.

Concernant le dépôt git, comme nous n'avons pas pris le temps de le créer dès le début, son implémentation en cours de projet ne nous a pas semblé particulièrement pertinente.