Projet 2 : Programmation système

[Cours : INFO-F-201]

Innocent Antoine
BA2 Informatique

Décembre 2016

1 Introduction

Dans le cadre de notre cours de système d'exploitation, nous devons implémenter une application client/serveur permettant de gérer plusieurs clients en parallèles. Plus précisément, notre application doit illustrer le jeux : "Cadavre exquis".

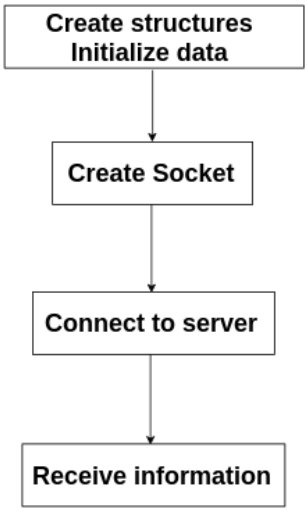
Afin de facilité notre travail nous allons considérer que :

— Le maître du jeu utilise le programme serveur.
— Le nombre de participants est limité.
— Le nombre de parties de phrase est limité.
— Un joueur ne peut pas quitter en pleine partie.

Ainsi, pour ma part, chaque joueur (après s'être connecté au serveur) sera demandé d'entrer un mot d'une certaine longueur prédéfinie. Une fois la totalité des mots reçus par le serveur, celui-ci renvoi l'ensemble des mots entrés à tous les joueurs.
Nous allons séparer ce rapport en deux parties : une partie client et un partie serveur.

2 Le client

Commencer par décrire la démarche utilisée par rapport a l'implémentation du programme client me parait le plus adéquat. En effet, celle-ci est plutôt basique. En me basant, sur le procédé vu en cours j ai structuré mon code ainsi :

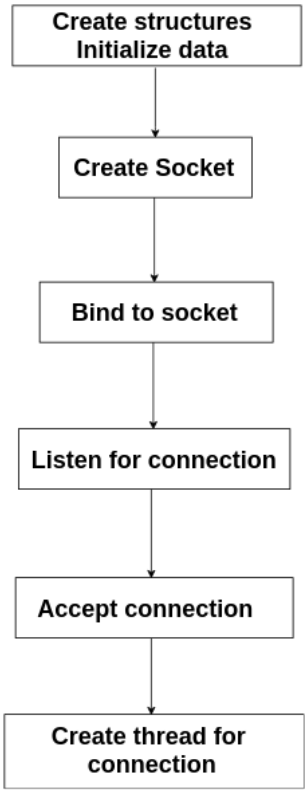


Le cœur de l'implémentation du code client réside dans une boucle "*while*". Cette dernière a pour condition principale l'appel système *recv*() qui va ensuite traiter différemment les informations reçu.

Dans le cas ou nous recevons le message : "*ready*" du serveur, le programme client est chargé de demander au joueur d'entrer un mot et de le renvoyer au serveur.
Finalement , une fois la partie terminée notre code interrompt la boucle et termine le programme.

3 Le serveur

Rentrons dans le vif du sujet. Le serveur était de loin la plus grande difficulté de ce projet. Même si le début du code suit le schéma du cours (expliqué ci-dessous), l'implémentation du maître du jeu n'était pas des plus intuitifs.



Après de longues réflexions sur le meilleur procédé à utilisé pour arriver a mes fins ($select(), fork, thread$), j'ai décidé d'opter pour les threads. Je me suis dit que c'était la solution la plus économe en termes processeur et ressources.

Ainsi, après de longues périodes d'essai/erreur je suis parvenu à avoir un code serveur qui communiquait correctement avec le code client et qui arrivait à émettre la solution finale. Effectivement, dans une boucle "*while*" infini, j'utilise l'appel système "*accept*" pour établir la connexion avec le client.

Une fois cette connexion établie, elle est attribuée a un thread (un thread par client) qui sera chargé d'envoyer et recevoir des informations du joueur.
La partie la plus compliquée n'était pas celle d'obtenir les données envoyées des clients. Pour cela, j'ai utilisé un buffer et manipulé son indice. Le grand défi était la gestion des threads. N'ayant jamais fait ça avec les appels systèmes de $I|O$, j'ai dû utiliser le concept de "Variable de condition".
Dans une boucle "*while*", chaque thread (à l'exception du dernier) ayant reçu l'information du client, doit attendre que tous les utilisateurs aient entré leurs données (manipulation du buffer) afin de renvoyer le résultat final.
Ici j'ai été confronté à un problème de "$I|O$" qui ne permettait pas à l'utilisateur d'obtenir le résultat correct. Après de longues recherches, j'ai finalement utilisé la fonction "*sleep*" qui m'a permis de régler le problème.

4 Conclusion

Le programme final n'est pas le plus optimal mais fonctionne. La fonction "*sleep*" induit un temps d'attente pour l'utilisateur qui peut être problématique dans le cas d'un jeu à beaucoup de joueurs. Et le serveur doit être quitté par l'intermédiaire d'un "$KeyboardInterrupt$".
Toutefois, ce projet m'a permis de solliciter la plupart des concepts vus en cours (gestion des threads a l'aide des mutexs, syntaxe serveur/client) et de les appliquer à un problème concret.