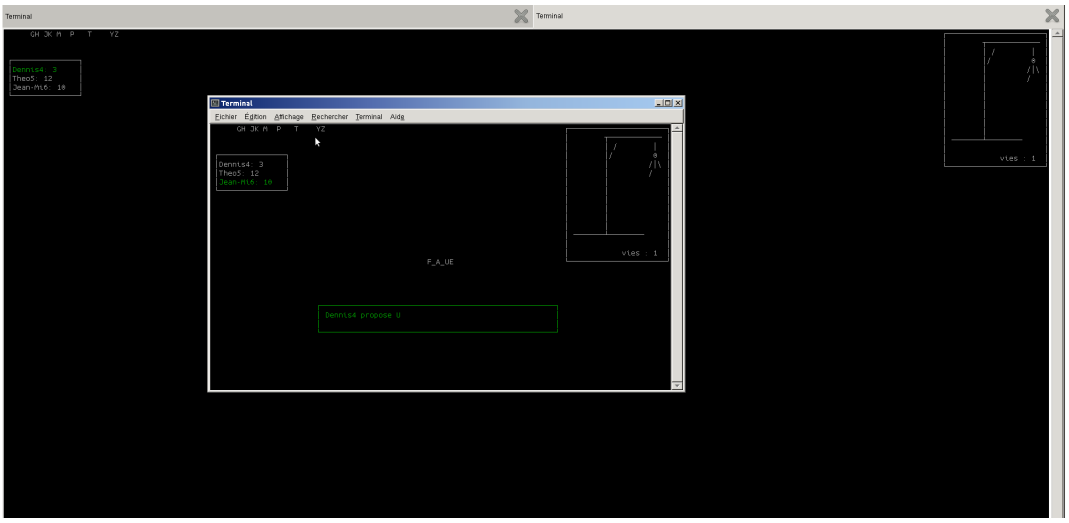
Rapport de projet : Pendu Réseau

Dennis Bordet, Théo Couraud

Mars 2016



Sommaire

1 Introduction

Dans le cadre du module de réseau, nous devions faire une application clients-serveur capable de gérer plusieurs clients à la fois. Pour cela nous avons choisi de faire une pendu en ligne avec un système de points. Afin de commencer le programme correctement nous avons commencé par lire le document fourni décrivant les sockets, leur utilisation et la description de la bibliothèque.

Par la suite, nous nous sommes familiarisé avec le client et le serveur fourni en exemple. Il a fallu lire le code, le comprendre, faire des essais avec puis le bidouiller en ajoutant des forks.

Notre projet nécessitants des threads, nous avons passé les premières semaines à se renseigner sur la bibliothèque <pthread> qui était toute nouvelle pour nous. Ce fût la première difficulté rencontrée.

2 Le pendu et ses règles

Notre pendu se déroule de cette façon : Le serveur génère en premier lieu un mot aléatoire et attend que les clients proposent des lettres pour mettre à jour l'avancement du mot. Lorsqu'un mot est trouvé ou que les joueurs sont pendus, le serveur demande à tous les joueurs si ils veulent continuer à jouer ou non. Lorsque nous avons toutes les réponses, nous recommençons le jeu avec un nouveau mot, et nous bouclons ainsi indéfiniment.

Pour le système de points nous avons fait ainsi :

- quand un joueur se connecte il part avec un capital de 10 points
- les joueurs perdent un point par proposition fausse, 3 si ils ont été pendu.
- ils ne peuvent pas avoir de points négatifs.
- plus un point par occurrence de lettre vraie dans le mot
- le joueur qui a trouvé la dernière lettre gagne 10 points.
- les points sont sauvegardés d'un match à l'autre.
- lorsqu'un joueur se déconnecte, tout ses points sont perdus, même s'il se reconnecte avec la même adresse et le même pseudo.

Pour les règles du pendu hors points :

- il n'y a pas de joueur minimum, i.e si tous les joueurs se déconnectent le serveur le relance pas de nouveau mot.
- Il ne peux y avoir que 10 joueurs sur la même partie, si un 11e essaye de se connecter il devra attendre une déconnexion pour pouvoir jouer.
- les joueurs peuvent arriver en cours de partie et jouer instantanément .
- on ne peux envoyer que des lettres sans accents et une par une.
- le groupe de joueurs a 10 vie qu'il perd dès qu'un joueur propose une mauvaise lettre.
- proposer une lettre déjà comptabilisée ne fait rien d'autre que la surligner.

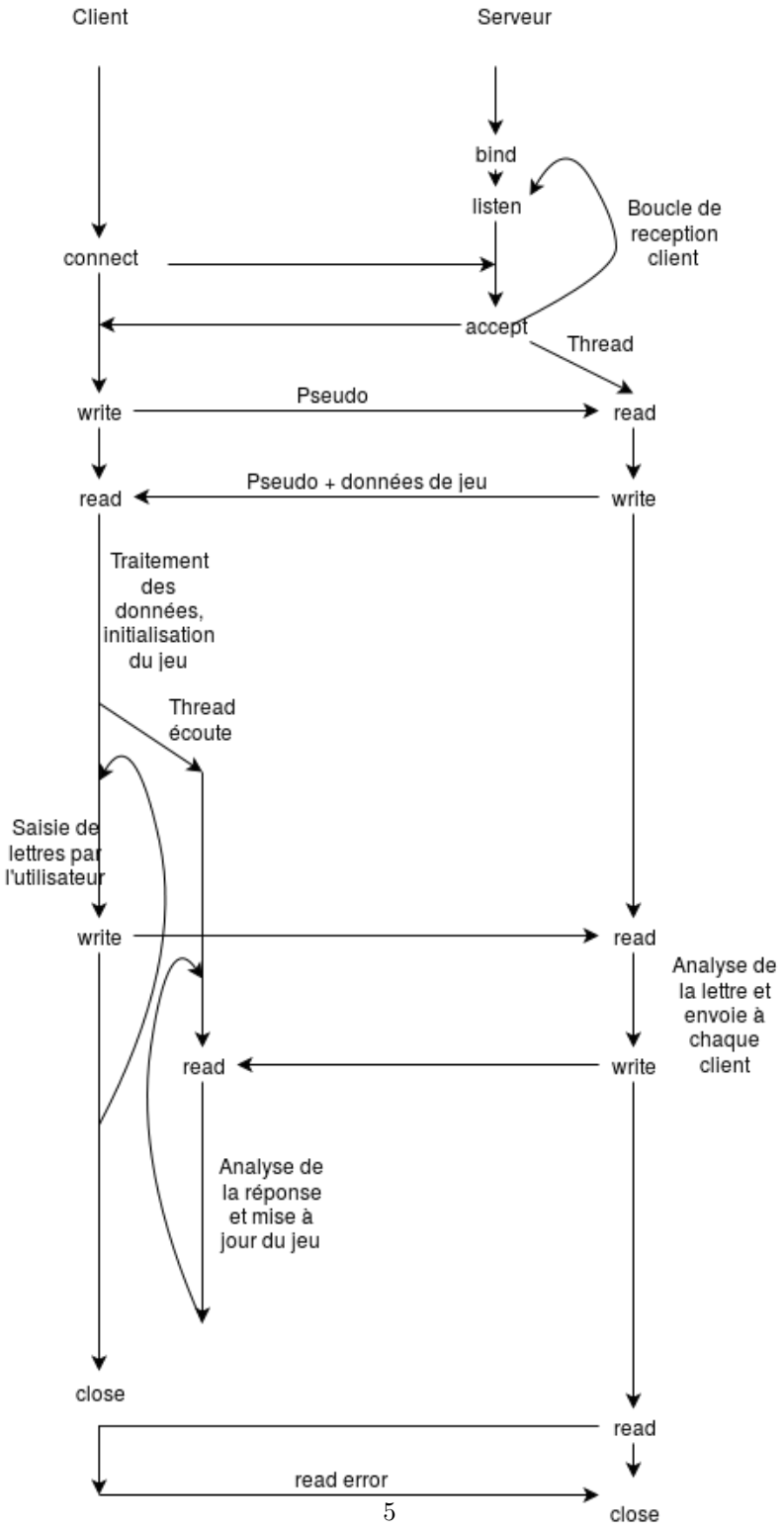
3 Choix d'implémentation

3.1 Présentation des sockets

Afin de faire communiquer les clients et le serveurs, nous nous sommes servis des sockets.

Nous nous sommes servis uniquement des sockets en mode connecté qui utilisent le protocole TCP sous IP. Ce protocole nous garantit donc une liaison sécurisé et nous sommes sûr que les messages parviennent à leurs destinataires, tout ceci dans l'ordre. C'était très important pour pouvoir initialiser les données des clients qui se connectent mais aussi pour garantir l'ordre des réponses des clients et donc une gestion plus juste des points.

Vous pouvez voir ci-dessous les échanges principaux entre un client et le serveur.


Client
Serveur
bind
listen
Boucle de reception client
connect
accept
Thread
write
Pseudo
read
read
Pseudo + données de jeu
write
Traitement des données, initialisation du jeu
Thread écoute
Saisie de lettres par l'utilisateur
write
read
Analyse de la lettre et envoie à chaque client
read
write
Analyse de la réponse et mise à jour du jeu
close
read
read error
close

3.2 Serveur

Pour le serveur nous avons repris la base du code fourni, avec les sockets. Nous sommes partis sur une programmation en multi-threading, avec un thread par socket plus le thread principale d'écoute qui attend de nouvelles connexions clients. Le but des ces threads est de pouvoir gérer les informations du pendu de manière synchronisée avec tous les clients.

La fonction d'un thread est globalement la suivante :

— initialisation,
— boucle du jeu,
— fermeture de la socket puis du thread.

Pour gérer les informations des clients nous avons créé un tableau qui contient pour chaque client : son pseudo, ses points, son numéro de socket et son thread.

Nous avons aussi créé une structure qui gère le mot du pendu en entier, les lettres trouvées et le mot "haché" qui est sous cette forme : RE_EAU, chaque _ étant une lettre non découverte. Nous initialisons cette structure à la création du serveur, en piochant un mot aléatoirement dans un dictionnaire, et la réinitialisons quand le mot à été trouvé ou les joueurs pendus.

Les deux structures citées ci-dessus sont partagées et modifiées par les threads s'occupant des clients.

Afin de voir les déconnexions "sauvages" des clients, nous écoutons à tout instant les sockets, et si jamais cela nous renvoi une erreur nous fermons le socket et terminons le thread après avoir avertis les autres clients de cette déconnexion.

3.3 Client

En ce qui concerne le client, pour le confort visuel, nous avons choisi de développer une interface Ncurses. Le client stocke toutes les données de jeu qu'il peut, tout comme le serveur, la seule donnée qu'il n'a pas est évidemment le mot à trouver, pour des éviter des triches éventuelles.

Il fonctionne avec deux threads :

— Un thread qui écoute en permanence les réponses du serveur, nous détaillerons les réponses plus bas.
— L'autre qui lit les entrées du client et les traite.

3.3.1 Thread d'écoute

Les messages reçus par le client viennent uniquement du serveur. Il sont concis afin d'accroître la vitesse de transmission des messages. Il peuvent concerner :

— La déconnexion d'un client : "d :nomClient."
— La connexion d'un client : "c :nomClient."
— La proposition d'une bonne lettre par un client et le mot actualisé en conséquence : "v :Lettre,Client,NouveauMot."
— La proposition d'une mauvaise lettre par un client : "f :Lettre,Client."

De cette façon, le client reçoit le minimum de données nécessaires pour pouvoir actualiser le jeu. A chaque réception de message, l'affichage est rafraîchit.

Le nom du client présent dans les message est potentiellement celui qui le reçoit, sauf en cas de déconnexion.

3.3.2 Thread saisie client

Ce thread analyse donc les saisies du client qui peuvent être :

- Un appui sur la touche "échap" -> envoie un message de déconnexion au serveur puis quitte l'application.
- Une lettre déjà proposée -> pas d'envoi, la lettre est surligné sur l'écran du client concerné.
- Une lettre qui n'a pas encore été proposée -> envoi de la lettre au serveur qui la renverra en broadcast après traitement.
- Autre chose (lettre avec accent, virgule, dollar, chiffre ...) -> pas d'envoi, attente de nouvelle saisie.

3.3.3 Difficultés

Concernant les difficultés que nous avons eu, la plus importante à été la gestion des char* et char[ ]. En effet que ce soit pour avoir la taille avec strlen +1 et sizeof, ou les conversions entre un char[ ] et char* qui sont notre seule source de communication entre les clients et le serveur. Pour la fonction write, on envoi le tableau de caractères en passant en dernier paramètre sa longueur (strlen)+1. Et pour la fonction read reçoit le message dans un autre tableau de caractères et le dernier paramètre est la taille de ce tableau (sizeof).

Parfois nous recevions un peu plus de caractères que prévu, des caractères aléatoires. Nous avons pu régler ce soucis en terminant nos message par un point à l'envoi. Ainsi, pour la réception, la partie du message située après le point (si elle existe) est tronquée.

Nous avons aussi eu beaucoup de bugs à gérer lors de l'inter-phase où les joueurs décident s'ils rejouent, surtout quand un nouveau joueur se connectait à se moment là. En effet cette phase est assez critique car les clients ne doivent rien recevoir qui pourrait les perturber, comme des messages de connexion ou déconnexion d'autres joueurs. Ce problème doit se régler côté serveur. En effet, il doit vérifier que les joueurs ne sont pas dans cette phase un peu spéciale avant de leur envoyer des données.

Quand nous avons essayé d'introduire le fait qu'il faille deux joueurs minimum pour jouer avec le serveur, nous nous sommes rendu compte qu'il nous faillais beaucoup plus de temps pour le faire correctement, nous l'avons donc enlevé du projet.

3.3.4 Améliorations

Pour les possibilités d'amélioration, nous avions pensé lors de la conception à un système de bannissement si jamais un joueur rentrait trop de lettres à la suite (flood).

Il y a aussi le fait que nous ne pouvons faire jouer simultanément que 10 joueurs, nous avons pensé à faire un système de rooms pour pouvoir avoir plus de joueurs par serveur.

Lors de notre présentation, il a aussi été dit que notre gestion des points est très permissive, et très inégale. Revoir le système de points serait assez essentiel pour rendre le jeu grand public.

Concernant les points, avoir un système qui les sauvegarde côté serveur grâce au pseudo permettrait de voir une vrai progression des joueurs à travers le temps. Bien sûr il faudrait limiter le nombre de parties par jour afin d'être le plus équitable possible.

Il devrait aussi être possible de rendre les règles consultables à tout moment coté clients, afin qu'il ne soit pas perdu lors des différentes phases.

4 Jeux d'essai

Nous avons fait énormément de tests. En réalité, nous avons d'abord implémenté la base des clients et du serveur puis nous avons procédé à une implémentation par tests. Nous testions notre programme, dans des conditions simples pour commencer, puis nous augmentions progressivement les risques de bug par divers moyens :

- Connexion de nombreux clients en simultanés.
- Connexion d'un client un peu n'importe quand.
- Flood d'un client.
- Flood de plusieurs clients.
- Déconnexion de clients un peu n'importe quand (avec échap ou en fermant le programme)
- Déconnexion du serveur.
- ...

5 Conclusion

Pour conclure, ce projet nous a appris à créer des applications qui utilisent le réseau, enfin ! Grâce aux sockets, nous pouvons désormais développer des programmes qui permettent d'échanger des données entres plusieurs utilisateurs. Mais pas seulement, on peut aussi utiliser les sockets pour faire communiquer deux programmes différents sur le même PC (sur l'adresse localhost).

Par ailleurs, au début du projet, le langage C ne nous était pas très familier mais grâce à ce projet, nous le maîtrisons bien mieux.

L'idée de faire un pendu était un peu plus ambitieuse qu'un simple chat de discussion mais nous avons réussi, même si il reste des points à améliorer.

6 Annexe

Ligne de commande pour compiler le serveur :

```
gcc serverPendu.c -lpthread -w -o server
```

Ligne de commande pour compiler le client :

```
gcc clientPendu.c -lpthread -lncurses -w -o client
```

Ligne de commande pour exécuter le client :

```
./client <adresse-serveur>
```