MINES Saint-Étienne
Une école de l'IMT

Rapport Projet PSE
Concours de vitesse de saisie au clavier

Introduction

Nous avons décidé de mettre en place un jeu client/serveur de vitesse de saisie au clavier.
Le concept est très simple, des joueurs se connectent à un serveur, lorsqu'ils sont tous prêts, des mots apparaissent. Il faut ensuite écrire le plus de mots possible dans le temps imparti sans fautes.
Le joueur ayant écrit le plus de mots est le gagnant.

Table de matières

Structure du programme

Nous avons utilisé la bibliothèque PSE `(c) P Lalevée, 2012`

```
.
├── appli
│   ├── client
│   ├── client.c
│   ├── liste_francais.txt
│   ├── Makefile
│   ├── serveur
│   └── serveur.c
├── assets
│   └── tree.png
├── include
│   ├── dataspec.h
│   ├── datathread.h
│   ├── erreur.h
│   ├── ligne.h
│   ├── pse.h
│   ├── resolv.h
│   └── words.h
├── lib
│   └── libpse.a
├── Makefile.inc
├── modules
│   ├── datathread.c
│   ├── erreur.c
│   ├── ligne.c
│   ├── Makefile
│   ├── resolv.c
│   └── words.c
└── README.md

6 directories, 23 files
```

- appli : contient **client.c**, **serveur.c** et liste_francais.txt (liste de mots)
- include : contient les headers des modules
- modules : contient les modules PSE utilisés plus words.c
- lib : contient libpse.a la bibliothèque statique

Nous avons créé en plus la librairie words. Cette librairie contient:

- `void generate_sentence(char sentence[TAILLE_PHRASE][TAILLE_MOT]);`

  Cette fonction permet de générer une phrase à partir de la liste de mots liste_francais.txt

- `char *get_word(int ligne);`

  Retourne le mot de la ligne du fichier liste_francais.txt

Coté serveur

Nous utilisons le modèle de serveur à workers dynamique du TP5, les threads pour chaque client sont gérés grâce à une liste chainée et la bibliothèque datathread.

Le programme serveur.c est un serveur multithread dynamique.

- Le programme commence par définir quelques constantes telles que le nom de la commande, la durée du jeu, et le nombre maximal de clients.
- Ensuite, il initialise un mutex statique utilisé pour la synchronisation des threads.
- La fonction `generateRanking` est définie pour calculer le classement des joueurs en fonction de leurs scores.
- La fonction `sessionClient` est la fonction exécutée par chaque thread client. Elle gère la communication avec un client spécifique.
- Dans la fonction `main`, le serveur crée une socket et se met en attente de connexions entrantes.
- Lorsqu'un client se connecte, le serveur vérifie si le nombre maximal de clients est atteint. Si c'est le cas, il ferme la connexion. Sinon, il crée un nouveau thread pour gérer ce client.
- Chaque thread client exécute la fonction `sessionClient`. Elle génère une phrase, attend que tous les clients soient prêts, puis lance un chrono.
- Chaque client doit répondre à la question "Êtes-vous prêt ?" en envoyant "o" pour oui. Les clients qui ne répondent pas ou qui envoient une réponse différente sont déconnectés.
- Une fois que tous les clients sont prêts, le serveur envoie "start" à chaque client pour commencer le jeu.
- Les clients doivent alors envoyer des mots correspondant à la phrase générée. Le serveur les reçoit et les compare à la phrase d'origine pour calculer les scores.
- Une fois que les clients ont envoyé tous leurs mots, le serveur envoie "stop" pour arrêter le jeu.
- Le serveur envoie ensuite les scores aux clients et annonce le classement.
- Après cela, le serveur vérifie s'il y a d'autres clients en attente de connexion et continue à écouter les nouvelles connexions.
- Lorsqu'un client se déconnecte, le serveur libère les ressources associées à ce client.

Ce programme est conçu pour gérer plusieurs clients en parallèle grâce à l'utilisation de threads. Chaque client communique avec le serveur via des sockets TCP.

Coté client

Le fichier "client.c" contient le code source du client.

1. Initialisation du client :
- Le client commence par créer une socket pour la communication réseau.
- Il résout l'adresse IP et le port du serveur auquel il doit se connecter.

2. Établissement de la connexion :
- Le client se connecte ensuite au serveur en utilisant la fonction "connect".
- Si la connexion échoue, une erreur est générée.

3. Affichage et interaction avec l'utilisateur :
- Le client affiche un message d'accueil reçu du serveur.
- Il demande à l'utilisateur s'il souhaite jouer en saisissant "o" (oui) ou "n" (non).
- La réponse de l'utilisateur est envoyée au serveur.

4. Déroulement du jeu :
- Si l'utilisateur souhaite jouer, le client entre dans une boucle de jeu.
- Le client reçoit un message de démarrage du serveur et affiche un message indiquant que la partie va commencer.
- Le client reçoit des mots du serveur et les affiche à l'utilisateur.
- L'utilisateur doit saisir le mot correspondant dans un délai imparti.
- Le mot saisi par l'utilisateur est envoyé au serveur.
- Si le serveur envoie "stop", cela signifie que la partie est terminée, sinon le jeu continue.
- Une fois la partie terminée, le client reçoit son score calculé par le serveur et l'affiche.
- Le client reçoit le résultat final du serveur (gagné ou perdu) et l'affiche.

5. Terminaison de la partie :
- Si l'utilisateur ne souhaite pas jouer à nouveau, le client termine la boucle de jeu.
- Une fois la boucle terminée, le client affiche un message indiquant que la partie est terminée.
- La socket est fermée et le programme se termine.

Ce fichier "client.c" implémente la logique du client pour interagir avec le serveur et jouer au jeu en réseau.

Défis et solutions trouvées

Le serveur synchronise les actions des clients à l'aide d'un mutex et utilise des variables partagées pour suivre l'état du jeu et les scores. Nous utilisons des variables globales pour faire communiquer nos threads entre eux. Il s'agit généralement de flag déclencheur d'événements.

- ## Générer une même phrase pour tous les clients

Défi:
Le premier défi est de générer une phrase unique pour tous les clients. Si on la génère dans le même alors chaque client à la même mais la phrase se génère une seule fois si on relance la partie on a un problème. Si on la génère dans les threads clients chacuns aura une phrase différente, le jeu n'a plus de sens.

Solution:
Nous avons décidé d'utiliser une section critique et un flag pour générer la phrase.

```
/*section critique car un seul thread client va générer la phrase*/
   pthread_mutex_lock(&mutex);
   if (!phrase_flag)
   {
     generate_sentence(phrase);
     printf("%s: Phrase generated\n",CMD);
     phrase_flag = VRAI;
   }
   pthread_mutex_unlock(&mutex);
```

Si la phrase n'est pas encore générée, alors un thread la génère pour les autres car `phrase` est une variable globale. Sinon le thread ne fait rien

- ## La gestion du chronomètre

Défi:
Un défi que nous avons eu avec la gestion du temps est celui de la consommation de ressources. En effet, au début, nous avions créé un thread timer à partir du thread principal, et nous attendions le lancement de celui-ci dans une boucle while. Cependant cette solution était gourmande en ressource car un thread entier bouclait à répétition.

Solution :

```
/*On lance le thread timer*/
pthread_mutex_lock(&mutex);
if (!start_chrono){
  pthread_create(&timer_thread_id,NULL,timer,NULL);
  start_chrono = VRAI;
}
pthread_mutex_unlock(&mutex);
/*On lance le chrono*/
```

De manière similaire à précédemment, une section critique permet de lancer un chronomètre dans un thread. Si le chrono n'est pas lancé alors un thread le lance à l'aide d'une variable globale. Un thread suivant ne pourra pas le relancer jusqu'à la fin de la partie.

```
void *timer()
{
   time(&start_time);
   time(&elapsed_time);
 while(difftime(elapsed_time,start_time) < GAME_TIME)
 {
   sleep(0.5);
   time(&elapsed_time);
 }
 stop = VRAI;
 pthread_exit(NULL);
}
```

Le thread timer communique avec les autres la fin du timer grâce à la variable globale `stop`.

- **Relancer une partie**

Défi:
Une fois la partie terminée, il faut réinitialiser les variables globales. Cependant, une fois que un client a terminé sa partie, si on réinitialise les variables globales dans son thread (timer,score,classement, phrase etc..) alors si un autre client se termine après il n'aura plus accès aux résultats.

Solution:

```
if (clients_prets == 0)
{
   /*On réinitialise les variables*/
   /*On remet le chrono à 0*/
```

```
    start_chrono = FAUX;
    stop = FAUX;
        etc…
}
```

On a donc décidé de réinitialiser les variables de jeux quand le nombre de clients prêts est nul.

- Suivre l'état du jeu et les scores

Défi:
Il faut que les clients et le serveur puissent partager les données du jeu sans conflits.
Solution:
Pour éviter les conflits, pour écrire dans une variable globale nous utilisons des mutex.

- Connaître en permanence le nombre de clients connectés

Défi:
Les clients peuvent terminer leur processus à n'importe quel instant du programme en utilisant 'Ctrl+C'. Cela est un réel problème lorsque le serveur attend une réponse de la part du client, et lorsque le client n'existait plus, le serveur se terminait. Il a donc fallu trouver une solution pour pallier ce problème.

Solution:

```c
int verif_client_state(DataSpec *dataTh, int in_game)
{
 ssize_t bytes_read;
 char buffer[1024];
 bytes_read = recv(dataTh->canal, buffer, sizeof(buffer), MSG_DONTWAIT);
 if (bytes_read == 0) {
   close_client(dataTh, in_game);
   return 0;
 }
 return 1;
}
```

Nous avons trouvé une solution qui consiste à tenter de recevoir une réponse de la part du client en utilisant la fonction `recv()` avec l'argument `MSG_DONTWAIT` qui nous permet de rendre la fonction non bloquante.
Ainsi, si le client est encore connecté, on retourne 1, sinon on déconnecte le client du côté serveur et on retourne 0.

- **Gestion des numéros de clients**

Défi:
Un autre défi a été l'attribution des numéros de clients. En effet, nous utilisions un numéro client qui permettait d'identifier les clients dans le tableau score et classement. Celui-ci est différent de l'ID client car celui-ci était un long int et n'était pas utilisable tel quel pour utiliser notre tableau.

Une première solution qu'on a trouvé a été une table de hachage de correspondance entre l'ID du client (=id du thread du client) et le numéro du client. Mais nous avons plutôt opté pour la solution décrite ci-après :

Solution:

```
n_client = trouver_premier_libre(score_tab);
printf("Je viens de trouver un numéro de client pour le thread, n_client =
%d\n",n_client);
dataThread->spec.n_client = n_client; /*On spécifie le numéro du joueur*/
score_tab[n_client] = 0;
n_client++;
nb_clients_connectes++;
pthread_mutex_unlock(&mutex);
```

Ainsi une fonction `trouver_premier_libre()` permet de trouver l'indice libre dans le tableau de score ce qui correspond à un client disponible. En effet, le tableau score est initialement à -1 l'indice est disponible pour un client.

Nous avons aussi modifié la structure DataSpec de la librairie PSE pour ajouter le numéro client qui correspond à chaque thread.

```
/* module datathread : donnees specifiques */
typedef struct DataSpec_t {
 pthread_t id;                /* identifiant du thread */
 int libre;                   /* indicateur de terminaison */
/* ajouter donnees specifiques après cette ligne */
 int tid;                     /* identifiant logique */
 int canal;                   /* canal de communication */
 sem_t sem;                   /* semaphore de reveil */
 int n_client;                /*numero du client connecte*/
} DataSpec;
```

Nous avons rajouté le numéro du client.

Conclusion

L'objectif était de permettre à plusieurs joueurs de se connecter simultanément au serveur, de participer à un jeu vitesse d’écriture de mots et d'afficher un classement à la fin.

Le serveur, implémenté dans le fichier "serveur.c", a été conçu pour gérer plusieurs connexions clientes en parallèle grâce à l'utilisation de threads. Chaque client se connecte au serveur via une socket TCP et interagit avec le serveur pour jouer au jeu de mots. Le serveur assure la coordination entre les clients en utilisant des mécanismes de verrouillage tels qu'un mutex pour éviter les conflits d'accès aux variables partagées.

Le client, quant à lui, a été implémenté dans le fichier "client.c". Il permet aux utilisateurs de se connecter au serveur, de participer au jeu en envoyant des mots correspondant à la phrase générée, et de recevoir leur score ainsi que le classement à la fin du jeu.

Ce projet a permis d'illustrer plusieurs concepts importants en programmation système tels que les sockets, les connexions TCP, la communication client-serveur et la synchronisation de plusieurs threads.

Les résultats obtenus sont satisfaisants, car l'application est capable de gérer plusieurs connexions simultanées, d'organiser le déroulement du jeu de mots et d'afficher un classement final.

En conclusion, ce projet a permis d'approfondir notre compréhension des concepts fondamentaux de la programmation système et de mettre en pratique nos compétences en développement. L'application de jeu de mots multijoueur développée constitue une base solide pour des extensions futures et représente une expérience enrichissante dans le domaine de la programmation réseau.