Rapport Projet PSE

Quizz de culture générale automatisé

Thimoté Dupuch Théo Gachet

Programmation Système
Juin 2023

Table des matières

1 Exposé du sujet

Sur le plan technique, l'application est multi-threadée, par un thread principal serveur qui communique les questions aux clients et qui récupère leurs réponses, et leurs temps de réflexion respectifs. Au niveau des clients la réception des messages provenant du serveur s'effectue par un thread dédié.

Notre projet consiste à recréer un jeu de culture générale largement inspiré du jeu *Questions pour un Champion*, en particulier la première phase de celui-ci, le *9 points gagnants*.

2 Déroulement d'une partie

Le serveur ouvre une connexion et accepte 3 clients virtuels (des clients qui peuvent se connecter). Le nombre de 3 clients est choisi afin de correspondre au mode de fonctionnement du jeu *Questions pour un Champion* mais peut être librement modifié à l'aide de la ligne suivante dans `server.c` :

```
#define MAX_CLIENTS 3
```

Dans notre cas, nous nous contenterons donc de 3 clients. Si un nombre supérieur de clients souhaite se connecter malgré tout, alors une erreur sera levée. Chacun de nos clients possèdent une base de réponses prédéfinie. Nous avons choisi d'en constituer une qui est commune à tous les joueurs afin de leur donner la même chance de gagner la partie. Ces 100 connaissances sont contenues dans le fichier texte `reponses.txt` et peuvent être des noms de famille, des dates ou des lieux.

Lorsque le serveur pose une question aux clients, ceux-ci prennent leur temps pour répondre, de la même manière qu'un humain prendrait le temps de réfléchir avant de buzzer. Lorsqu'ils reçoivent la question du serveur, chacun des 3 client génère un nombre aléatoire de manière indépendante. Ce nombre constitue leur temps de rélexion propre. Ils vont donc patienter pendant cette durée avant de buzzer et de proposer une réponse.

Lorsque les 3 clients ont transmis leur proposition de réponse au serveur, celui-ci va s'intéresser à celle du client qui a répondu en premier.

Si la réponse est correcte, le client gagne 5 points, et le serveur passe à la question suivante. Dans ce cas, tous les buzzers sont libérés et le serveur envoie une nouvelle question, ce qui relance le processus précédent.

En revanche, si la réponse proposée est incorrecte, le serveur retire 1 point au client qui vient de se tromper, il bloque son buzzer (ce qui fait qu'il ne pourra plus participer pour la question en cours) et il va désormais s'intéresser aux clients encore en jeu.

Cela n'aurait aucun sens d'examiner leur proposition de réponse déjà envoyée, car un joueur réel peut tout à fait être influencé par la proposition de réponse du joueur qui vient de se tromper. Ainsi, les joueurs encore en jeu vont recommencer leur processus de réflexion et de buzz afin de proposer une nouvelle réponse, et le serveur va attendre leurs réponses de la même manière que précédemment, à l'exception près qu'il va désormais ignorer le joueur qui s'est trompé (puisque son buzzer est éteint pour cette question).

A chaque fois qu'un joueur propose une réponse, qu'elle soit bonne ou mauvaise, le serveur lui indique s'il a raison ou s'il se trompe, il met à jour son score et fait un point sur les scores en temps réel en affichant la modification du score et en donnant un classement à l'instant t.

Le nombre de questions est arbitraire et peut être modifié dans `server.c` :

```
#define NB_QUESTIONS 10
```

Une partie est un enchaînement de plusieurs questions qui peuvent chacune se terminer de deux manières différentes :

— un client a donné la bonne réponse
— tous les clients ont donné une mauvaise réponse

Vérification des réponses des clients :

```
int verif_reponse(char *reponse,char *reponse_client,Client *client)
{
    reponse_client[strcspn(reponse_client, "\n")] = '\0';
    reponse[strcspn(reponse, "\n")] = '\0';
    if (strncmp(reponse, reponse_client, strlen(reponse)) == 0)
    {
        printf("[BONNE REPONSE] +5 points");
        modif_score(client, 5);
        return 1;
    }
    else
    {
        printf("[MAUVAISE REPONSE] -1 point");
        modif_score(client, -1);
        bloquer_buzzer(client);
        return 0;
    }
}
```

3 Organisation des programmes

3.1 Découpage du code

En plus du `main.c`, nous avons choisi de bénéficier des avantages qu'offre la modularité [1] en segmentant notre code de manière suivante :

Fichiers sources :

- `server.c`
- `client.c`
- `functions.c`

Fichiers d'en-tête :

- `server.h`
- `client.h`
- `functions.h`

Nous disposons également de deux fichiers qui contiennent les questions ainsi que les bases de connaissances des clients : `questions.txt` et `reponses.txt`. Enfin, le fichier `Makefile` qui permet la compilation du code.

3.2 Bibliothèques utilisées et normes

Afin de ne pas avoir des dépendances trop importantes, nous avons décidé de ne pas include le fichier d'entête `"pse.h"`, qui contiendrait éventuellement des bibliothèques de code non essentielles à notre projet. Nous avons décidé d'inclure, sur l'ensemble de fichier ou uniquement sur une partie d'entre eux, les bibliothèques suivantes :

- `stdio.h`
- `stdlib.h`
- `string.h`
- `unistd.h`
- `sys/socket.h`
- `arpa/inet.h`
- `pthread.h`
- `fcntl.h`
- `semaphore.h`
- `time.h`

Remarque : cela réduit de moitié les dépendances par rapport aux séances de TD.

Notre code respecte les deux normes ANSI C (C89 ou C90) et POSIX.1b-1993 (POSIX). Nos flags de compilation sont les suivants :

```
CFLAGS= -D_POSIX_C_SOURCE=199309L -W -Wall -ansi -pedantic
```

Les drapeaux `-W` et `-Wall` permettent de détecter un large éventail d'éventuels problèmes de programmation. `Wall` signifie *all warnings* (tous les avertissements).

Le drapeau `-pedantic` active des avertissements supplémentaires pour se conformer aux normes du langage C. Il permet de détecter les constructions non conformes et encourage ainsi une programmation plus rigoureuse.

1. cf. la présentation de Théo Gachet sur le `Makefile` disponible sur Campus

4 Difficultés rencontrées

4.1 Communication entre le serveur et les clients

Établir une communication bidirectionnelle entre le serveur et les clients était un défi. Il fallait lire les messages envoyés par les clients et envoyer d'autres messages à tous les clients connectés. La synchronisation par mutex et sémaphores, ainsi que la gestion des threads pour chaque client, furent donc des étapes cruciales et particulièrement délicates.

Julien Lepers utilise `send_to_all_clients()` pour communiquer :

```c
void send_to_all_clients(char *message)
{
    int i;
    pthread_mutex_lock(&clients_mutex);
    for (i = 0; i < MAX_CLIENTS; i++)
    {
        if (client_socks[i] != 0)
        {
            int client_sock = client_socks[i];
            int write_size = write(client_sock, message, strlen(message));
            if (write_size == -1) { perror("Erreur d'envoi au client"); }
        }
    }
    pthread_mutex_unlock(&clients_mutex);
}
```

4.2 Synchronisation des actions des clients

Lorsque plusieurs clients étaient connectés, il était essentiel de synchroniser leurs actions. Par exemple, attendre que tous les clients soient connectés avant de commencer la partie, permettre à un seul client de buzzer à la fois pendant une question, etc. La gestion des sémaphores ou des mutex fut nécessaire pour assurer une exécution cohérente et ordonnée. L'utilisation d'une structure pour représenter un client a grandement facilité nos opérations :

Chaque client est une instance de `Client*` :

```c
typedef struct {
    int fd;      /* File descriptor pour le socket du client */
    int score;   /* Score du client */
    int buzzer;  /* Etat du buzzer du client */
    int etat;    /* Envoyeur, Receveur, En attente */
} Client;
```

4.3 Génération pseudo-aléatoire

Une partie importante de notre quizz se base sur la notion de génération pseudo-aléatoire. En effet, les ordinateurs sont des machines déterministes et il n'est donc pas possible en programmation de générer de vrais nombres aléatoires. À la place, on utilise des générateurs de nombres pseudo-aléatoires (PRNG - Pseudo-Random Number Generators) pour simuler des séquences de nombres qui semblent être aléatoires.

Dans notre code, la génération pseudo-aléatoire intervient dans la sélection des réponses par les clients. Il n'est pas possible de simplement utiliser une fonction comme `rand()`, car elle générerait pour chaque client la même séquence. Pour différencier les clients, on utilise leur *Process ID* avec un appel de la fonction `getpid()`.

Génération de nombres pseudo-aléatoires :

```c
long int reflexion(int seconds)
{
    struct timespec ts;
    long int temps_reflexion;

    ts.tv_sec = abs((rand() + seconds) % 8 + 1);
    ts.tv_nsec = (int)abs((rand() + seconds) % 1000000000);
    temps_reflexion = abs((rand() + seconds) % 100000 + 1);

    nanosleep(&ts, NULL);
    return (temps_reflexion * 10000);
}
```

4.4 Gestion de la séquence d'action à effectuer

Au niveau du déroulement du quizz, il y a un enchaînement d'instructions qui doivent se produire dans le bon ordre (en plus d'une bonne synchronisation comme évoqué plus haut) pour que le dialogue client serveur soit le bon. Tout d'abord, le serveur doit être prêt à recevoir les demandes des clients et à leur envoyer les questions. Les clients, quant à eux, doivent se connecter au serveur et attendre de recevoir les instructions. Une fois les questions envoyées aux clients, ces derniers doivent proposer une réponse et transmettre leurs différents temps de réflexion au serveur. Le serveur doit alors collecter les réponses de tous les clients et les traiter pour calculer les scores. Enfin, le serveur doit envoyer les résultats aux clients afin qu'ils puissent les afficher.