Rapport de projet

OS USER

SHERLOCK 13

DAGHER Jana
21314951
TP A

Sommaire

I. Bases du projet et construction du programme

Il s'agit d'un projet qui vise à exploiter nos acquis en gestion de communications réseau et d'implémentation d'appels de fonctions fondamentales du système d'exploitation POSIX afin de mettre en marche le célèbre jeu d'enquête Sherlock 13 dont le but est de trouver la carte coupable à partir d'informations récoltées sur les autres joueurs au fur et à mesure que la partie progresse. Le dossier du projet a été fourni avec deux fichiers de base (server.c et sh13.c), dont certaines parties étaient à compléter.
La logique serveur (server.c) fait appel à la :

- Gestion des connexions des joueurs,
- Distribution des cartes,
- Gestion des différentes requêtes de type G (accusation), O (question générale), S (question spécifique).

L'implémentation côté client (sh13.c) des différentes commandes reçues depuis le serveur nécessite de :

- gérer l'affichage des cartes,
- répondre aux requêtes,
- passer le tour, etc.

II. Fonctionnement général

Le jeu se déroule en tour par tour avec 4 joueurs.
Au lancement :

- Le serveur est lancé avec un port
- Chaque client se connecte avec une adresse IP, un port propre à lui, et un nom
- Le serveur distribue à chaque joueur ses cartes (3 personnages et leur implication (traduite par des symboles) dans l'enquête)

C'est le joueur 0 qui commence la partie, et le numéro du joueur à qui devient le tour au fur et à mesure de l'avancée de la partie est broadcasté en début de chaque round.

Communication :
Chaque message suit un format simple : une lettre en majuscule qui préfixe chaque commande et indique l'opération en cours, et des arguments à sa suite selon le besoin :

- C : connexion
- I : message d'identifiant du joueur
- L : liste des joueurs
- D : distribution des cartes
- V : mise à jour du compte d'un objet (tableCartes)
- M : tour du joueur
- G : accusation de culpabilité
- O : question ouverte à tous
- S : question spécifique à un joueur et sur un objet en particulier

Le joueur courant peut donc poser une question (O, S), ou accuser (G) un personnage d'être le coupable. Dans ce dernier cas, selon son accusation, il peut vaincre la partie s'il a raison, ou être disqualifié, perdant ainsi sa capacité à entreprendre et poser des questions, mais maintenant tout de même son devoir de répondre à celles des autres joueurs encore actifs.

À la fin de son action, le tour passe au joueur suivant via le message M.

La complétion des fonctions impliquait globalement une communication avec le serveur pour lui envoyer, en recevoir, ou y programmer l'envoi via buffer, que ce soit vers un client particulier (donc adresse IP et numéro de port dédiés dans 'sendMessageToClient()') ou en broadcast vers tous les joueurs, de messages encodés de la manière introduite ci-dessus, avec l'initiale d'une commande et les arguments qui la suivent. Ensuite venait le parsing des commandes coté client, en lisant du buffer les valeurs envoyées à la bonne position comme convenu, en les associant à leurs variables et en les affichant ensuite sur le GUI SDL.

III. Utilisation des éléments des TP

Des sockets TCP sont bien utilisés tout le long de la communication client-serveur, et cela à l'aide d'appels de fonctions fondamentales en réseau : accept, connect, read, write, bind, etc.
Chaque client écoute aussi sur son propre port pour recevoir des messages directs du serveur.

Grâce au make run du Makefile, 5 process distincts sont lancés (1 serveur et 4 clients) pour simuler les 4 fenêtres de joueurs distinctes avec un serveur central à l'écoute de tous.
Il s'agit là d'un point important du Makefile, car mises à part les commandes basiques de compilation du code, de liaison des librairies, et de création d'un fichier exécutable, le make run permet de lancer immédiatement une instance du serveur, puis quatre clients dans quatre terminaux différents, avec des identifiants différents (a, b, c, d) et des ports distincts grâce au gnome-terminal.

Mais gnome-terminal, c'est quoi ?
Il s'agit du terminal par défaut de l'environnement GNOME, dont la commande permet d'ouvrir un nouveau terminal graphique, et avec le – bash, un nouveau shell Bash y est lancé, prenant en argument les commandes que l'on souhaite y exécuter.
[!] "exec bash" nécessaire en fin de commande pour que le terminal reste ouvert, ce qui permet de voir les messages, erreurs, ou outputs éventuels (sinon, le terminal se fermer automatiquement après la fin de l'exécution).
NB : Le Makefile ne fait pas de fork() lui-même, mais pour chaque commande qui y est lancée, le shell crée un process, donc le système fait bien un fork() under the hood à ces moments
Et dans au cours du lancement d'un "gnome-terminal -- bash -c "./sh13 ...""", l'OS ouvre un terminal, qui lance un shell Bash, qui exécute une commande → plusieurs fork() en interne pour réaliser cette chaîne de terminal + shell + exec (gnome-terminal est un child process du Makefile, et bash est un fork de gnome-terminal).

En ce qui concerne les threads, le client utilise un thread d'écoute pour recevoir les messages asynchrones du serveur tout en continuant à permettre des actions utilisateur via SDL.

Pas de mutex utilisés mais ceci aurait pu être une implémentation utile pour rendre le code actuel plus robuste (plusieurs structures partagées modifiées à la fois par le thread SDL et le thread réseau, mutex servirait à éviter les race conditions et collisions) (plus de détails dans les axes d'amélioration).

Pas de pipes utilisés dans ce projet car non nécessaires ici.

IV. Améliorations possibles

Quoique globalement fonctionnel, le code ajouté pourrait bénéficier de plusieurs amélioration pour renforcer la robustesse du jeu, notamment :

Les mutex :
Le client utilise un thread secondaire pour écouter les messages réseau (via socket) pendant que le thread principal gère l'interface graphique SDL. Ces deux threads partagent des données (comme les cartes du joueur, l'état du tour, etc.) sans synchronisation explicite, ce qui présente des risques de race conditions potentielles. La solution serait donc l'implémentation des mutex (pthread_mutex_t) pour éviter ces complications, surtout si le jeu devenait plus complexe/multithreadé côté serveur aussi.

Gestion des erreurs :
Actuellement, la plupart des erreurs (connexion, lecture socket, etc.) sont soit ignorées, soit traitées par des exit(1), ce qui coupe brutalement l'exécution. Une manière de lisser cette exécution lors d'erreurs serait l'introduction d'une vraie gestion d'erreurs avec messages explicites, tentatives de reconnexion, etc. (ou au moins un exit propre côté client avec nettoyage et libération des ressources SDL). Ajouter des SDL_Quit() serait aussi utile pour fermer proprement les fenêtres ouvertes lors de l'interruption d'une partie ou de la fin du jeu.