Compte Rendu Projet OS

Sherlock13

Soukarieh Ali
21113343

Introduction

Ce rapport détaille la construction et le fonctionnement du programme SH13, un jeu inspiré de l'univers de Sherlock Holmes, en mettant l'accent sur l'utilisation des différents aspects vus dans les TP du module (socket, processus, pipe, threads, mutex, etc.).

Structure du projet

Le projet est structuré autour de deux fichiers principaux :

1. **server.c** : Serveur qui gère la logique du jeu et les communications
2. **sh13.c** : Client avec interface graphique SDL permettant aux joueurs d'interagir

Construction du programme

Architecture générale

Le programme est basé sur une architecture client-serveur où le serveur central gère jusqu'à 4 clients simultanément. Chaque client représente un joueur qui doit deviner le personnage coupable parmi 13 cartes.

Complétion du code C

1. Dans le serveur (server.c)

J'ai complété le serveur en implémentant les fonctionnalités suivantes :

Gestion des joueurs et de leur état

```
int eliminatedPlayers[4] = {0, 0, 0, 0}; // Tableau pour suivre les joueurs éliminés
int remainingPlayers = 4; // Nombre de joueurs encore en jeu
int winByElimination = 0; // Indique si la victoire est due à l'élimination des autres joueurs
```

Système de révélation d'informations

```
int revealedValues[4][8]; // Suivi des valeurs spécifiquement révélées (1) vs marquées avec * (0)
```

Initialisation du jeu L'initialisation du jeu se fait dans la fonction `main()` où je mélange le deck, crée le tableau de jeu et distribue les cartes aux joueurs.

Gestion des actions des joueurs J'ai implémenté un système qui traite trois types d'actions possibles :

1. Accusation (G) - Le joueur accuse directement un personnage
2. Questionnement (O) - Le joueur demande combien d'objets spécifiques possèdent les autres joueurs
3. Suggestion (S) - Le joueur demande à un joueur spécifique combien d'un objet il possède

```
case 'G':
   // Gestion des accusations
   sscanf(buffer, "G %d %d", &playerId, &guiltIndex);
   // Vérification de l'accusation et élimination si incorrecte
   if (guiltIndex == deck[12]) {
      // Victoire
      sprintf(reply, "W %d", playerId);
      broadcastMessage(reply);
   } else {
      // Élimination
      eliminatedPlayers[playerId] = 1;
      remainingPlayers--;
      // Notification d'élimination
      sprintf(reply, "E %d", playerId);
      broadcastMessage(reply);
      // Vérification si un seul joueur reste
      if (remainingPlayers == 1) {
         // Déterminer le gagnant
       }
   }
   break;
```

Gestion du tour de jeu Pour gérer correctement les joueurs éliminés, j'ai implémenté un système qui saute leur tour :

```
do {
   joueurCourant = (joueurCourant + 1) % nbClients;
} while (eliminatedPlayers[joueurCourant]);
```

2. Dans le client (sh13.c)

Gestion des états du jeu J'ai ajouté des variables pour suivre l'état d'élimination du joueur :

```
int isEliminated = 0; // Indique si le joueur courant a été éliminé
```

```
int winByElimination = 0; // Indique si quelqu'un a gagné par élimination
```

Amélioration de l'affichage J'ai amélioré l'affichage pour distinguer les valeurs spécifiquement révélées (nombre exact) et les valeurs simplement marquées (symbole "*") :

```
char mess[10];
if (tableCartes[i][j] == 100) {
   sprintf(mess, "*");
} else if (tableCartes[i][j] != -1) {
   sprintf(mess, "%d", tableCartes[i][j]);
} else {
   mess[0] = '\0';
}
```

Traitement des messages du serveur J'ai complété la gestion des messages du serveur, notamment pour les messages d'élimination et de victoire :

```
case 'E': // Élimination d'un joueur
   {
      int eliminated;
      sscanf(gbuffer, "E %d", &eliminated);
      if (eliminated == gId) {
         isEliminated = 1;
         goEnabled = 0; // Désactiver le bouton go pour le joueur éliminé
         // Afficher un message indiquant que le joueur a été éliminé
         SDL_ShowSimpleMessageBox(
            SDL_MESSAGEBOX_WARNING,
            "Eliminated!",
            "You have been eliminated for guessing the wrong culprit!",
            window);
         }
   }
   break;
case 'W': // Victoire d'un joueur
   // Gestion des différents types de victoire avec messages appropriés
   break;
```

Fonctionnement du programme

Initialisation et connexion

1. Le serveur est lancé et attend la connexion de 4 joueurs
2. Chaque joueur lance le client qui se connecte au serveur
3. Le serveur attribue un identifiant unique à chaque joueur
4. Une fois les 4 joueurs connectés, le serveur distribue les cartes et commence le jeu

Déroulement d'une partie

1. Le serveur détermine le premier joueur et lui donne la main
2. Le joueur actif peut effectuer trois types d'actions :
3. Accuser un personnage d'être coupable (action à risque)
4. Demander des informations sur un objet aux autres joueurs
5. Demander des informations précises sur un objet à un joueur spécifique
6. Après chaque action, le tour passe au joueur suivant
7. Un joueur qui fait une accusation incorrecte est éliminé
8. La partie se termine par la victoire du joueur qui découvre le coupable ou du dernier joueur non éliminé

Utilisation des aspects vus en TP

1. Sockets et communication réseau

Le projet utilise les sockets TCP pour établir la communication entre le serveur et les clients :

Côté serveur :

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
listen(sockfd, 5);
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
```

Côté client :

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
server = gethostbyname(ipAddress);
connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
```

La communication est bidirectionnelle :

1. Le serveur envoie des messages aux clients via `sendMessageToClient()`
2. Les clients envoient des messages au serveur via `sendMessageToServer()`

J'ai utilisé une approche de messages textuels pour simplifier le protocole de communication entre le serveur et les clients.

2. Threads et synchronisation

Le client utilise un thread dédié pour écouter les messages provenant du serveur, ce qui permet à l'interface graphique de rester réactive :

```
pthread_t thread_serveur_tcp_id;
ret = pthread_create(&thread_serveur_tcp_id, NULL, fn_serveur_tcp, NULL);
```

La synchronisation entre ce thread d'écoute et le thread principal est assurée par une variable volatile `synchro` et un mutex :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
volatile int synchro;
```

Quand un message arrive du serveur :

1. Le thread d'écoute stocke le message dans le buffer partagé `gbuffer`
2. Il active le flag `synchro = 1` pour signaler au thread principal qu'un message est disponible
3. Le thread principal traite le message et réinitialise `synchro = 0` quand il a fini

Cette approche évite les problèmes de concurrence tout en permettant une interface utilisateur fluide.

3. Communication par messages

J'ai implémenté un protocole de communication par messages textuels entre le serveur et les clients :

1. `C [ip] [port] [nom]` - Connexion d'un client
2. `I [id]` - Attribution d'un identifiant
3. `L [nom1] [nom2] [nom3] [nom4]` - Liste des joueurs connectés
4. `D [carte1] [carte2] [carte3]` - Distribution des cartes
5. `M [joueur]` - Définition du joueur courant
6. `V [ligne] [colonne] [valeur]` - Mise à jour du tableau de jeu
7. `E [joueur]` - Notification d'élimination
8. `W [gagnant]` - Notification de victoire
9. `G [joueur] [carte]` - Action d'accusation
10. `O [joueur] [objet]` - Action de questionnement
11. `S [joueur] [cible] [objet]` - Action de suggestion

Ce système de messages permet une communication structurée entre les composants du système.

4. Gestion des ressources

Le projet utilise plusieurs techniques de gestion des ressources :

1. **Libération des ressources SDL** : Les textures et surfaces sont correctement libérées à la fin du programme
2. **Fermeture des sockets** : Les sockets sont fermées après utilisation pour éviter les fuites de ressources
3. **Gestion des descripteurs de fichier** : Les descripteurs de socket sont correctement suivis et fermés

Améliorations implémentées

1. **Système d'élimination des joueurs** - Un joueur qui fait une accusation incorrecte est éliminé de la partie
2. **Suivi des états révélés** - Distinction entre les valeurs spécifiquement révélées et celles marquées
3. **Messages de fin de partie** - Affichage de messages différents selon le type de victoire/défaite
4. **Gestion des tours adaptative** - Le système saute automatiquement les joueurs éliminés

Conclusion

La construction de ce programme a nécessité l'intégration de plusieurs concepts de programmation système :

1. Communication réseau par sockets TCP
2. Programmation concurrente avec threads
3. Synchronisation entre threads avec mutex
4. Communication par messages
5. Interface graphique avec SDL

Le résultat est un jeu multijoueur fonctionnel où les joueurs peuvent interagir en temps réel, échanger des informations et tenter de résoudre le mystère. L'architecture client-serveur utilisée est modulaire et pourrait être étendue pour supporter d'autres fonctionnalités comme un chat entre joueurs ou des variantes du jeu.