Rapport Projet os_user

Système d'exploitation

Asdjad Bakary

21315742

EI4 FISA – 2024/2025

Encadré par : Thibault Hilaire

Introduction

Ce rapport présente la démarche suivie pour réaliser le projet final du module OS USER, consistant à créer une version réseau et graphique du jeu Sherlock 13. Ce projet a été mené à bien après plusieurs séances de TP qui nous ont progressivement initiés à la programmation système avec des concepts fondamentaux comme les processus, les threads, les pipes, les sockets, les mutex, etc.

1. Développement progressif au fil des TP

a. TP1 : Prise en main des appels système de base

Nous avons commencé par des manipulations simples en C avec *scanf*, *sscanf*, et la manipulation de chaînes. Ces fonctions nous ont permis de bien maîtriser les formats d'entrée et de traitement des chaînes, ce qui a été utile par la suite pour la lecture des messages reçus par socket.

Ensuite, nous avons appris à créer des processus avec *fork*, à comprendre leur comportement (différentiation parent/enfant), et à gérer leur exécution avec *sleep*, *exit*, etc. Ces connaissances nous ont permis de comprendre comment démarrer plusieurs clients ou serveurs simultanément, et ont été très utiles pour les tests ultérieurs.

b. TP suivants : Sockets, processus, pipe et thread

Nous avons étudié les sockets TCP pour créer un client et un serveur capables de communiquer via le réseau. Cela constitue la base de la communication dans notre jeu final : chaque interaction entre les joueurs passe par le serveur via ces sockets TCP.

Nous avons utilisé les **processus** pour faire tourner plusieurs clients/serveurs lors des tests. Cela nous a appris à gérer l'indépendance des entités et à simuler un environnement multijoueur.

Les **pipes** ont été abordés pour la communication entre processus. Bien qu'ils ne soient pas directement utilisés dans le jeu final, cette étape nous a aidés à comprendre les mécanismes de communication interprocessus.

Les **threads** sont utilisés dans le client pour écouter les messages du serveur sans bloquer la boucle principale de SDL (interface graphique). Cela permet à l'interface de rester fluide même lorsque des messages réseau arrivent.

Enfin, nous avons utilisé des **mutex** pour protéger l'accès concurrent au buffer partagé entre le thread TCP et la boucle graphique principale. Sans cela, des conflits pouvaient apparaître lors de la lecture/écriture du buffer.

2. Démarche de complétion et fonctionnement du programme

Avant de commencer la complétion du code, ma première démarche a été de tester s'il se compilait correctement afin de m'assurer que l'environnement de compilation était fonctionnel et que les bibliothèques nécessaires (SDL2, SDL_image, SDL_ttf, pthread) étaient bien installées. Le code compilait, mais à ce stade, il n'était pas jouable : la fenêtre graphique s'ouvrait, mais aucun joueur ne pouvait interagir et rien ne se passait côté serveur.

Ensuite j'ai pris le temps de lire en détail les fichiers **sh13.c** et **server.c** fournis**.** Ce qui m'a permis de comprendre la structure générale du programme et de repérer les parties à compléter.

Côté serveur

Le code du serveur avait pour rôle principal de gérer les connexions des clients, de leur attribuer des cartes, et de gérer la logique du jeu à travers les messages reçus et envoyés via TCP. La structure générale du serveur était bien présente, mais une grande partie de la logique restait à compléter.

Des blocs *// RAJOUTER DU CODE ICI* étaient présents pour nous indiquer où intervenir : lors de l'envoi des cartes à chaque joueur, lors des actions **G, S, O**, ou encore lors du changement de joueur courant.

J'ai compris que le tableau tcpClients stocke les informations réseau (IP, port, nom) de chaque client connecté. Le deck est mélangé, puis les cartes sont réparties entre les joueurs, et le dernier élément du deck correspond au coupable. La matrice tableCartes[4][8] indique combien de fois chaque symbole est présent dans les cartes de chaque joueur, ce qui est utile pour les enquêtes.

La machine à états fsmServer permet de gérer les différentes phases du jeu : l'attente des connexions (fsmServer == 0) puis le déroulement de la partie (fsmServer == 1). Le serveur reçoit les messages via read() sur socket, et envoie les réponses avec sendMessageToClient (pour un joueur) ou broadcastMessage (pour tous les joueurs). La gestion du tour est assurée par la variable joueurCourant, qu'il faut mettre à jour manuellement après chaque action.

Le serveur ne contenait encore aucune logique métier concernant les actions du jeu : il n'y avait pas de traitement pour l'accusation (G), l'enquête globale (O) ou l'enquête ciblée (S). Toute cette partie était à implémenter en comprenant et en respectant le protocole de communication prévu.

Côté client

Le code du client s'appuyait sur la SDL2 pour l'interface graphique, et contenait des éléments réseau pour se connecter à un serveur distant, mais aucune logique métier n'était implémentée (aucune interaction utilisateur ne faisait quoi que ce soit).

Des blocs de commentaires *// RAJOUTER DU CODE ICI* étaient présents pour nous guider.

Côté client, une structure de données globale est utilisée pour stocker l'état du jeu : les informations sur les joueurs, les objets, les cartes reçues, etc. Le client crée un thread secondaire (fn_serveur_tcp) qui écoute en permanence les messages réseau, ce qui permet à l'interface graphique SDL de rester fluide pendant les

échanges avec le serveur. La fonction sendMessageToServer est utilisée pour envoyer des messages au serveur via socket TCP. L'affichage des cartes, des objets et des sélections est réalisé avec SDL_RenderCopy, en fonction de l'état courant du jeu. Au départ, la logique du jeu (enquêtes, accusations, levée de main...) n'était pas encore implémentée dans le code client, et devait être ajoutée manuellement à partir des structures existantes.

a. Etape de complétion

- **Connexion client /serveur**

Ce qu'il fallait faire :
Permettre à plusieurs clients de se connecter au serveur et d'attendre le début de la partie une fois tous les joueurs connectés.

Interventions dans le code :

server.c : boucle principale du serveur, gestion du fsmServer == 0, ajout dans tcpClients.

sh13.c : fonction de connexion au serveur, gestion de la transition vers l'écran d'attente.

J'ai complété la boucle de connexion dans le serveur pour accepter jusqu'à quatre joueurs.
Chaque nouvelle connexion est stockée dans le tableau tcpClients.
Quand le nombre de joueurs requis est atteint, le serveur envoie un message à tous les clients pour démarrer la partie.
Côté client, j'ai affiché un écran d'attente avant le début du jeu.
Le client utilise un thread secondaire pour écouter les messages du serveur tout en maintenant l'affichage SDL actif.

Concepts mobilisés :

- Sockets TCP (accept, read, write)
- Boucle de gestion des clients
- Threads (le client écoute le serveur en parallèle)

- **Distribution des cartes et définition du coupable**

Ce qu'il fallait faire :
Répartir les cartes de manière aléatoire entre les joueurs et désigner un coupable.

Interventions dans le code :

server.c : fonction de début de partie (fsmServer == 1), initialisation du deck et de la matrice tableCartes.

J'ai ajouté le mélange du deck côté serveur.
Les cartes sont ensuite distribuées équitablement aux joueurs.
La carte restante est définie comme le coupable.
Pour chaque joueur, j'ai calculé les symboles présents dans ses cartes et rempli la matrice tableCartes.
Chaque joueur reçoit ses cartes via un message personnalisé.
Le client affiche ensuite les cartes qu'il a reçues.

Concepts mobilisés :

- Manipulation de tableaux
- Génération pseudo-aléatoire
- Envoi de données structurées via TCP

- **Tour de jeu et gestion du joueur courant**

Ce qu'il fallait faire :
Définir un système de tour pour que chaque joueur puisse jouer à son tour.

Interventions dans le code :

server.c : mise à jour du joueurCourant après chaque action réussie.

sh13.c : affichage d'un message ou changement de bouton actif selon si c'est le tour du joueur.

Le serveur gère le joueur courant grâce à la variable joueurCourant.
Après chaque action valide, il passe au joueur suivant.
Un message est envoyé à tous les clients pour indiquer de qui c'est le tour.
Le client vérifie s'il peut jouer ou non et désactive les boutons si ce n'est pas son tour.
Cela évite que plusieurs joueurs agissent en même temps.

Concepts mobilisés :

- Partage d'état global
- Thread + mutex pour synchronisation côté client

- **Enquête ciblée (action 'S')**

Ce qu'il fallait faire :
Permettre à un joueur de demander à un autre s'il possède un certain symbole.

Interventions dans le code :

server.c : gestion du message S reçu, réponse conditionnelle.

sh13.c : envoi de la demande au serveur.

Quand un joueur utilise l'action 'S', un message est envoyé au serveur. Le serveur vérifie si le joueur ciblé possède le symbole demandé. Si oui, il envoie une réponse au joueur demandeur. Sinon, il lui indique que rien n'a été trouvé.
Le client affiche le résultat de l'enquête dans une zone dédiée. L'action ne modifie pas le tour, donc le joueur peut continuer à jouer.

Concepts mobilisés :

- Protocole de communication via messages TCP
- Logique conditionnelle métier
- Mutex pour accès aux données partagées

- **Enquête globale (action 'O')**

Ce qu'il fallait faire :
Permettre à un joueur de poser une question à tous les autres concernant un symbole, et compter les occurrences.

Interventions dans le code :

server.c : ajout de la boucle de comptage.

sh13.c : affichage du résultat (case que j'ai rajouté pour pouvoir mieux tester le fonctionnement du '0' dans le server).

Le joueur peut interroger tous les autres joueurs sur un symbole. Le serveur compte combien de fois le symbole apparaît parmi les autres joueurs. Le résultat est envoyé uniquement au joueur demandeur.
Le client affiche ce nombre sur son interface ce qui lui permet d'éliminer certaines possibilités.

Concepts mobilisés :

- Boucle réseau + condition
- Envoi structuré de réponse

- **Accusation (action 'G')**

Ce qu'il fallait faire :
Permettre à un joueur de désigner un coupable.

Interventions dans le code :

server.c : gestion du message G, comparaison avec le vrai coupable.

sh13.c : réception du résultat et affichage.

Le joueur peut désigner un suspect avec l'action 'G'. Le serveur compare cette accusation avec le coupable réel. Si l'accusation est correcte, un message de victoire est envoyé à tous les clients.
Sinon, le joueur est éliminé et ne peut plus jouer. Le jeu se termine quand un joueur trouve le bon coupable.

Concepts mobilisés :

- Logique de fin de jeu
- Envoi de message à tous les clients
- SDL : affichage d'un message et bouton quitter

- **Interface graphique SDL**

Ce qu'il fallait faire :
Rendre le jeu interactif et agréable visuellement.

Interventions dans le code :

sh13.c : gestion des événements SDL, rendu des cartes, boutons, texte.

Ajouts réalisés :

- Boucle sur nbnoms[] + affichage avec SDL_RenderText_Solid() et les icônes via SDL_RenderCopy().
- Gestion de goEnabled après réception du message M; affichage du bouton si goEnabled == 1.
- Mise à jour de leveMainClients[] après réception du message O, affichage du carré avec SDL_RenderFillRect() si levée de main.
- Traitement du message F, affichage du texte final avec TTF_RenderText_Solid(), ajout d'un bouton rouge avec "Quitter".
- Mutex autour de gbuffer et synchro pour éviter les accès simultanés entre le thread réseau (fn_serveur_tcp) et la boucle principale.

Concepts mobilisés :

- SDL2 (SDL_RenderCopy, SDL_Event, SDL_Rect, etc.) : représente visuellement les réponses des autres joueurs.
- Communication réseau (message TCP) + logique conditionnelle pour empêcher les joueurs de jouer hors de leur tour.
- pthread_mutex_t (vue en TP Threads) pour empêcher les conflits d'accès mémoire entre threads.

b. Fonctionnement du programme

Le programme repose sur une architecture **client-serveur** utilisant des **sockets TCP**. Chaque joueur lance un client SDL et se connecte au serveur, qui gère l'ensemble de la logique du jeu. Une fois quatre joueurs connectés, le serveur commence la partie en distribuant les cartes, en désignant un coupable, et en lançant le premier tour.

• Communication réseau

La communication se fait par messages texte, transmis via les sockets.
Chaque message suit un format simple : une lettre indiquant le type de message (C, D, G, O, S, etc.), suivie de paramètres.
Le client envoie les actions du joueur, et le serveur renvoie les réponses ou met à jour tous les joueurs avec broadcastMessage.

• Tour de jeu

Le serveur maintient un identifiant joueurCourant.
Seul ce joueur a le droit d'agir, ce qui est signalé par le message M.
Le client active alors le bouton "Go".
Après chaque action (enquête ou accusation), le serveur change de joueur en sautant ceux qui sont éliminés.

• Actions du joueur

- **Enquête ciblée (S)** : Un joueur demande à un autre s'il possède un symbole. Le serveur lui répond directement.
- **Enquête globale (O)** : Tous les autres joueurs lèvent la main si le symbole est présent dans leurs cartes.
- **Accusation (G)** : Le joueur tente de désigner le coupable. Si c'est correct, il gagne. Sinon, il est éliminé.

- **Gestion des états**

Chaque joueur connaît :

- ses propres cartes (message D),
- les noms des joueurs (message L),
- le contenu partiel de la table des symboles (messages V),
- qui joue (M), qui lève la main (O), qui est éliminé (X), et si la partie est terminée (F).

- **Interface graphique**

L'interface affiche tous les éléments du jeu :

- les noms des joueurs avec leurs objets,
- les suspects, leurs symboles, et les cartes détenues,
- les indicateurs visuels pour les levées de main, l'élimination, ou la sélection,
- les boutons d'actions et un message de fin avec un bouton "Quitter".

- **Synchronisation et threads**

Le client crée un thread TCP (fn_serveur_tcp) qui reçoit les messages du serveur sans bloquer la SDL.
Un mutex protège les accès à la mémoire partagée (gbuffer, synchro) entre la boucle principale et ce thread.
Cela garantit que les messages sont traités correctement sans interférence avec l'affichage.

Conclusion

Ce projet m'a permis de mobiliser de manière concrète l'ensemble des notions abordées en TP tout au long du module OS USER.
Grâce à l'approche progressive et au code partiellement fourni, j'ai appris à analyser un projet existant, à comprendre sa structure, et à y ajouter des fonctionnalités complexes de manière cohérente.

J'ai utilisé les **sockets TCP** pour gérer la communication réseau entre le serveur et les clients, les **threads** pour assurer la réactivité du client sans bloquer l'interface SDL, et les **mutex** pour protéger l'accès aux données partagées. L'usage de **SDL2** a permis de rendre le jeu graphique, interactif et plus engageant.

Ce projet a représenté un véritable défi, notamment dans la coordination entre les échanges réseau, la logique du jeu et l'interface graphique.
Il m'a appris à gérer des situations concrètes de concurrence, de synchronisation et de gestion d'événements, tout en respectant une architecture client-serveur robuste.

Avec plus de temps, j'aurais aimé enrichir l'interface avec des animations, un affichage plus moderne, ou encore ajouter des logs réseau pour faciliter le débogage. J'ai tout de même tenu à améliorer l'interface pour la rendre plus interactive, même si certaines idées n'ont pas pu être intégrées.

Malgré cela, le projet est **fonctionnel**, **stable**, et respecte l'ensemble des consignes du sujet. Il m'a offert une expérience complète de développement **système** et **graphique en C**, en conditions quasi réelles.