Rapport de Projet OS USER

Date : 20/04/2025
Module: OS USER

Rédigé par : Karlitou Formance MENDY EI4 FISA

1. Introduction

Dans le cadre du module OS USER, j'ai eu l'opportunité de travailler sur un projet captivant : l'implémentation d'un jeu multijoueur inspiré de "Sherlock Holmes 13". Ce projet m'a permis d'appliquer concrètement les concepts théoriques abordés en cours et en TP, notamment la programmation réseau, la gestion des threads et les mécanismes de synchronisation.

Le projet consistait à compléter un squelette de code fourni par l'enseignant pour créer un jeu fonctionnel où des joueurs peuvent se connecter à un serveur, recevoir des cartes et tenter de découvrir l'identité du "coupable".

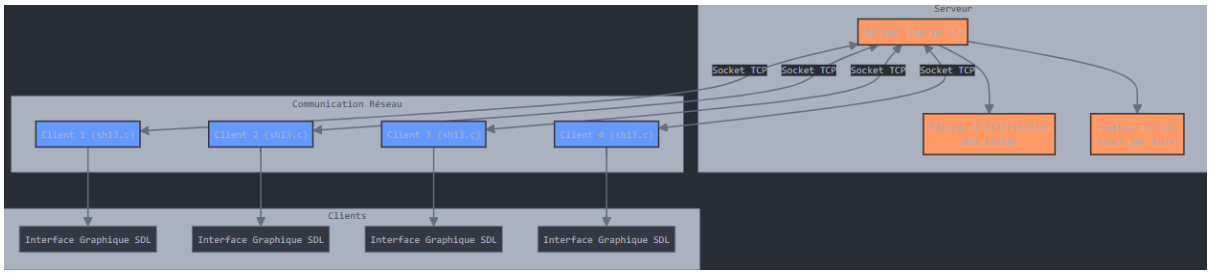
2. Comprendre l'architecture du jeu

Avant de commencer à coder, j'ai pris le temps d'analyser l'architecture du projet. Celui-ci s'articule autour de deux composants principaux :

- Un **serveur** (server.c) chargé de gérer la logique du jeu et les communications
- Un **client graphique** (sh13.c) permettant aux joueurs d'interagir avec le jeu via une interface SDL

Le principe du jeu est le suivant : quatre joueurs se connectent au serveur, chacun reçoit trois cartes parmi douze. La treizième carte est désignée comme "coupable". Chaque carte possède des caractéristiques uniques représentées par des symboles. Les joueurs doivent, à tour de rôle, poser des questions ou faire des observations pour déduire quelle carte est le coupable.

Cette architecture client-serveur est un excellent exemple de communication réseau, un concept fondamental abordé pendant les TP.



Le schéma ci-dessus illustre l'architecture globale du système, montrant comment le serveur central communique avec les quatre clients via des sockets TCP, et comment chaque client gère son interface graphique.

3. Mise en œuvre des concepts de programmation système

3.1 Communication réseau avec les sockets

L'un des aspects les plus intéressants de ce projet a été l'utilisation des sockets TCP pour établir la communication entre le serveur et les clients. J'ai rapidement compris que cela correspondait parfaitement à ce que nous avions vu en TP.

Pour compléter les sections manquantes, j'ai dû comprendre comment :

- Le serveur accepte les connexions des clients

- Les messages sont formatés pour être envoyés et interprétés correctement
- Le serveur diffuse des informations à tous les clients connectés

Pour implémenter les mécanismes d'accusation, j'ai dû analyser le format des messages attendus et comprendre comment le serveur devait réagir à une accusation correcte ou incorrecte. Si l'accusation est correcte, le serveur doit annoncer le gagnant à tous les joueurs. Sinon, il doit passer au joueur suivant.

La fonction `broadcastMessage()` m'a particulièrement intéressé, car elle illustre parfaitement le concept de diffusion dans un environnement réseau, un aspect que j'avais trouvé théorique en cours mais qui prenait tout son sens dans ce projet.

Type de Message	Format	Description	Émetteur
Connexion	`C adresse port nom`	Client demande à se connecter	Client
Identifiant	`I id`	Attribution d'un identifiant au client	Serveur
Liste Joueurs	`L nom1 nom2 nom3 nom4`	Liste des joueurs connectés	Serveur
Distribution	`D carte1 carte2 carte3`	Distribution des cartes au client	Serveur
Tour Joueur	`M id`	Indique quel joueur peut jouer	Serveur
Observation	`O id symbole`	Demande d'observation d'un symbole	Client
Question	`S id cible symbole`	Question à un joueur spécifique	Client
Accusation	`G id carte`	Accusation d'une carte	Client
Résultat	`V joueur symbole valeur`	Résultat d'observation ou de question	Serveur

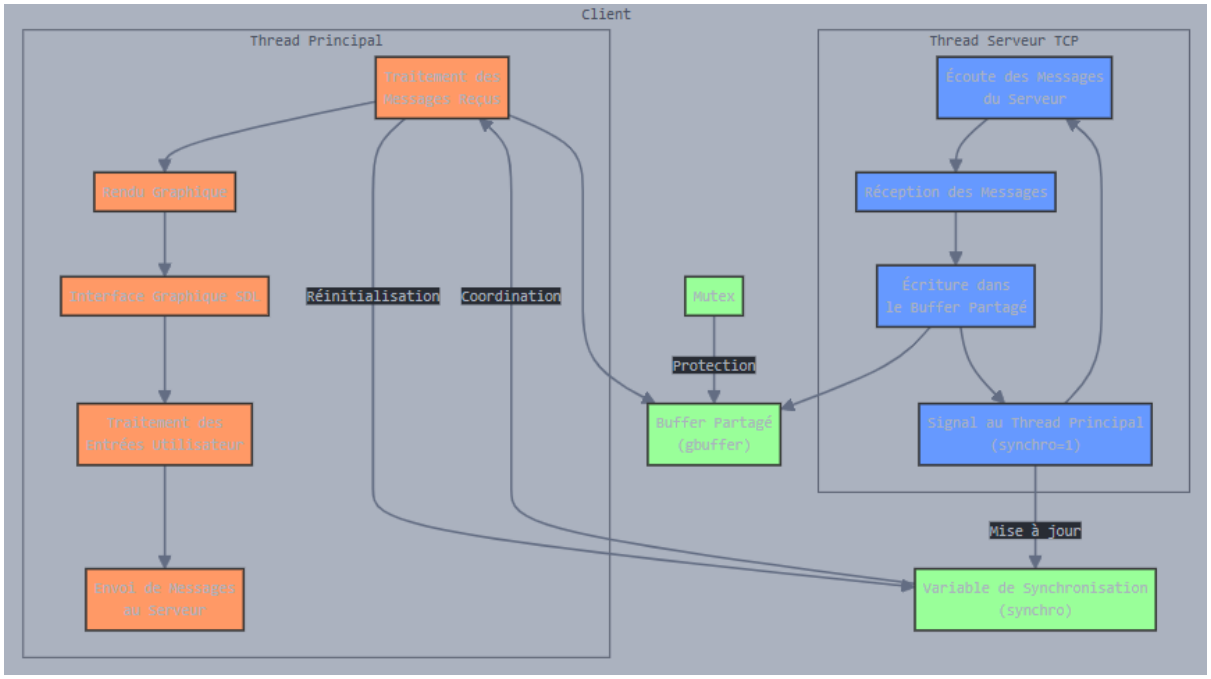
Le diagramme de séquence ci-dessus illustre le protocole de communication que j'ai implémenté, montrant les échanges de messages entre le serveur et les clients lors d'une partie typique.

3.2 Parallélisme et threads

Le client utilise un thread dédié pour recevoir les messages du serveur, ce qui m'a permis d'approfondir ma compréhension de la programmation concurrente.

Ce thread fonctionne en parallèle de la boucle principale qui gère l'interface graphique, permettant ainsi de recevoir des messages du serveur sans bloquer l'interaction de l'utilisateur. Cette séparation des préoccupations est un concept puissant que j'ai vraiment apprécié mettre en pratique.

L'implémentation du thread serveur côté client m'a fait réaliser l'importance d'une bonne conception pour gérer efficacement les opérations asynchrones. Pendant que l'interface graphique reste réactive aux actions de l'utilisateur, le thread de réception peut attendre passivement les messages du serveur.



Ce schéma détaille l'architecture multithreadée du client, montrant comment les deux threads (principal et serveur TCP) interagissent via un buffer partagé protégé par un mutex et coordonné par une variable de synchronisation.

3.3 Synchronisation avec les mutex

Un défi majeur dans tout système concurrent est la synchronisation des accès aux ressources partagées. Dans ce projet, j'ai utilisé des mutex pour protéger l'accès au buffer de réception des messages.

En complétant la partie client, j'ai dû veiller à ce que les accès au buffer partagé soient correctement protégés par des mutex. Ces mécanismes de verrouillage assurent que les modifications du buffer sont atomiques, évitant ainsi les problèmes de concurrence entre le thread principal et le thread de réception.

J'ai particulièrement apprécié voir comment la théorie des sections critiques vue en cours se traduisait en pratique. Sans cette protection, des corruptions de données auraient pu se produire si les deux threads avaient accédé simultanément au buffer.

3.4 Communication inter-threads

Pour coordonner le thread de réception et le thread principal, j'ai implémenté un mécanisme de signalisation basé sur une variable partagée (`synchro`). Ce mécanisme permet au thread de réception de signaler qu'un nouveau message est disponible, et au thread principal de signaler qu'il a traité le message.

Cette approche, bien que simple, m'a permis de comprendre les principes fondamentaux de la communication inter-threads sans recourir à des mécanismes plus complexes comme les variables de condition.

4. Défis rencontrés lors de la complétion du code

4.1 Implémentation des mécanismes d'action dans le serveur

Un des premiers défis que j'ai rencontrés a été de comprendre et d'implémenter correctement les différentes actions du jeu dans le serveur (accusation, observation, question spécifique).

Pour la partie accusation (`case 'G'`), j'ai d'abord eu du mal à identifier comment vérifier si l'accusation était correcte. Après analyse du code existant, j'ai compris que le coupable était stocké à l'indice 12 du tableau `deck`. J'ai donc implémenté une comparaison entre la carte accusée et cette valeur.

Pour les observations (`case 'O'`), j'ai rencontré un problème particulier : je devais déterminer quelles informations partager avec les joueurs. Après réflexion, j'ai réalisé que le jeu ne devait révéler que les joueurs qui n'avaient *pas* de symboles du type demandé. J'ai donc implémenté une boucle qui vérifie cette condition pour chaque joueur et n'envoie des informations que lorsque la valeur est 0.

Pour les questions spécifiques (`case 'S'`), le défi était de récupérer correctement les paramètres du message et de construire une réponse appropriée sans divulguer trop d'informations.

Dans les trois cas, je devais aussi m'assurer de passer correctement au joueur suivant en mettant à jour la variable `joueurCourant` et en diffusant ce changement.

J'ai également remarqué un problème lors des premiers tests : le coupable était toujours le même d'une partie à l'autre ! En examinant le code, j'ai constaté que bien que la fonction `melangerDeck()` était appelée, le générateur de nombres aléatoires n'était pas correctement initialisé. J'ai donc ajouté une instruction `srand(time(NULL))` pour initialiser le générateur avec l'heure actuelle, assurant ainsi une distribution vraiment aléatoire des cartes à chaque partie. Ce petit détail a fait une grande différence dans l'expérience de jeu, rendant les parties beaucoup plus intéressantes et imprévisibles.

4.2 Traitement des messages côté client

L'implémentation de la partie client m'a confronté à des défis différents. Je devais compléter le code pour traiter cinq types de messages différents (identifiant, liste des joueurs, distribution des cartes, tour de jeu, et valeurs de cartes).

Pour le message 'I' (identifiant), j'ai dû comprendre comment extraire l'identifiant du message et le stocker correctement dans la variable globale `gId`.

Pour le message 'L' (liste des joueurs), le défi était de parser correctement une chaîne contenant quatre noms de joueurs.

Pour le message 'D' (distribution des cartes), j'ai dû non seulement extraire les identifiants des cartes, mais aussi appeler la fonction `initCards()` pour initialiser la table de jeu du client.

Le message 'M' (joueur courant) m'a posé un problème particulier : je devais activer le bouton "Go" uniquement pour le joueur dont c'était le tour. J'ai implémenté cette logique en comparant l'identifiant reçu avec l'identifiant du client.

Enfin, pour le message 'V' (valeur d'une carte), j'ai dû faire attention à ne pas écraser des informations déjà connues. J'ai ajouté une condition pour ne mettre à jour la table que si la valeur n'était pas déjà connue.

4.3 Synchronisation entre threads côté client

Un défi majeur a été de comprendre et d'utiliser correctement les mécanismes de synchronisation pour la communication entre le thread principal et le thread de réception des messages.

Dans la partie "consomme message" du client, j'ai dû faire particulièrement attention à respecter le protocole de synchronisation existant. La variable `synchro` est utilisée comme signal entre les threads, et je devais m'assurer de la remettre à 0 après avoir traité un message.

Au début, j'ai rencontré un problème où certains messages étaient perdus. Après débogage, j'ai réalisé que j'oubliais parfois de déverrouiller le mutex après avoir traité certains messages, ce qui bloquait le thread de réception. Cette expérience m'a fait comprendre l'importance d'une gestion rigoureuse des mutex dans une application multithreadée.

5. Réflexion sur les concepts du module

Ce projet a été une excellente occasion d'appliquer et d'approfondir ma compréhension des concepts clés du module OS USER :

- **Programmation réseau** : L'utilisation des sockets TCP pour la communication client-serveur m'a permis de comprendre comment les applications distribuées fonctionnent en pratique.
- **Programmation concurrente** : La gestion de threads multiples m'a fait apprécier la puissance du parallélisme pour créer des applications réactives.
- **Synchronisation** : L'utilisation des mutex m'a montré comment protéger les données partagées dans un environnement multithreadé.
- **Communications inter-processus** : Bien que ce projet utilise principalement des threads plutôt que des processus séparés, les principes de communication sont similaires et m'ont aidé à consolider ma compréhension.

En particulier, j'ai apprécié comment ces concepts s'intègrent pour former un système cohérent. Ce projet m'a montré que ces notions ne sont pas seulement des concepts théoriques, mais des outils pratiques pour résoudre des problèmes réels de programmation.

6. Conclusion

Ce projet a été une expérience enrichissante qui m'a permis de transformer des concepts théoriques en compétences pratiques. J'ai non seulement appris à programmer avec des sockets, des threads et des mutex, mais j'ai aussi développé une intuition pour la conception de systèmes distribués.

La partie la plus gratifiante a été de voir le jeu fonctionner correctement, avec plusieurs clients connectés simultanément, échangeant des informations et interagissant de manière fluide. Cela m'a donné un aperçu de la satisfaction que peuvent apporter des projets plus complexes en programmation système.

Je suis maintenant plus confiant dans ma capacité à concevoir et implémenter des applications qui utilisent la communication réseau et la programmation concurrente, des compétences qui seront certainement précieuses dans ma future carrière.

Par souci de clarté et de lisibilité du rapport, j'ai choisi de ne pas inclure les extraits de code source que j'ai implémentée, préférant me concentrer sur l'explication des concepts, des mécanismes et des choix de conception qui ont guidé mon travail de complétion du projet.