Compte-Rendu TP

OS_ USER : Sherlock 13

- **Nom:** Zarroug Amina
- **Groupe TP :** C

- **Date:** 20 avril 2025

Introduction

Dans le cadre de ce projet, nous avons conçu une adaptation numérique du jeu de société Sherlock 13, un jeu d'enquête pour 2 à 4 joueurs où chacun tente de découvrir l'identité d'un mystérieux coupable à l'aide d'indices éparpillés entre les participants. Le concept repose sur l'élimination progressive des suspects à partir des symboles que les joueurs possèdent ou découvrent.

Notre version repose sur une architecture client-serveur et propose une expérience multijoueur via réseau local, enrichie par une interface graphique réalisée en C avec la bibliothèque SDL2. Le développement de cette application a nécessité la mise en place d'une logique réseau robuste, une gestion fine des échanges entre joueurs, ainsi qu'une interface intuitive et dynamique.

Ce document présente dans un premier temps la manière de démarrer le jeu, puis détaille l'utilisation de l'interface, les fonctionnalités développées en plus du jeu d'origine, ainsi que l'organisation interne du code côté serveur et client.

I. Démarrage de l'application

Pour exécuter notre version numérique de Sherlock 13, il faut suivre une procédure simple composée de trois étapes principales : compilation, lancement du serveur, puis connexion des clients.

1. Compilation des sources

Le projet est constitué de deux programmes principaux : un serveur (server.c) qui coordonne la partie, et un client (sh13.c) qui représente un joueur. La compilation est automatisée via un script appelé cmd.sh. Pour générer les exécutables, il suffit d'ouvrir un terminal dans le dossier du projet et de lancer la commande : ***./cmd.sh***

Cela crée deux fichiers exécutables : server et sh13.

2. Initialisation du serveur

Avant de pouvoir jouer, le serveur doit être démarré avec le port sur lequel il écoutera les connexions des clients. Exemple d'utilisation : ***./server 1234***

3. Connexion des joueurs

Chaque joueur lance son client avec les paramètres suivants : adresse IP et port du serveur, IP et port du client, ainsi qu'un nom d'utilisateur. Par exemple :

./sh13 127.0.0.1 1234 127.0.0.1 5678 Alice

L'adresse IP de la machine peut être obtenue via : **hostname -I**

Si tout est exécuté sur la même machine, l'adresse localhost suffit.

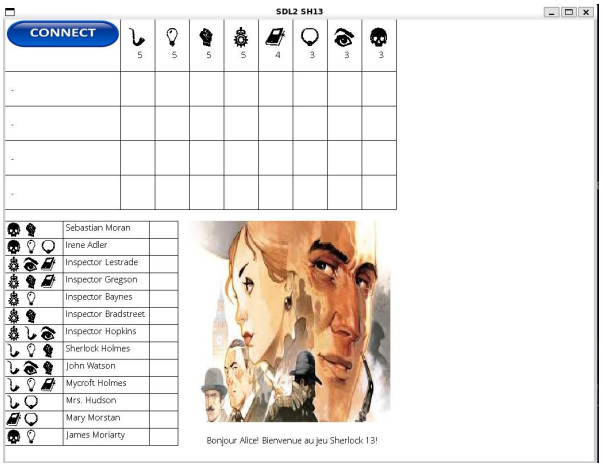

Image 1 - Rendu première connexion

Le jeu commence automatiquement dès que quatre clients sont connectés.

II. Structure et fonctionnement du projet Sherlock13 en langage C

Méthodologie et démarche de développement

Lors de la réalisation de ce projet, nous sommes partis d'un squelette de code incomplet fourni. Celui-ci comportait les bases du client et du serveur, mais de nombreuses fonctions étaient absentes ou seulement esquissées. Nous avons donc procédé par étapes :

- **Analyse du besoin** : comprendre les règles du jeu Sherlock 13 et les traduire en interactions logiques (connexion, tour de jeu, enquêtes, accusations, victoire, élimination).
- **Relecture du code fourni** : repérage des portions utilisables, des fonctions à compléter, des types de messages déjà prévus (G, O, S, etc.)
- **Planification des structures** : clarification des structures de données, comme les tableaux de cartes, la table des symboles, et les listes des joueurs.
- **Développement progressif** : chaque fonctionnalité a été codée, testée de manière isolée, puis intégrée.
- **Tests multi-clients** : validation du fonctionnement du serveur avec quatre clients connectés en simultané.

Constitution du programme et répartition client / serveur

Le programme est constitué de deux fichiers principaux :

- **server.c** : assure la gestion des connexions, la logique du jeu (distribution des cartes, passage de tour, vérification de victoire, redémarrage, etc.), et la communication avec les clients via sockets TCP.
- **sh13.c** : client qui gère l'interface graphique (via SDL2), les entrées utilisateur, et la réception/interprétation des messages en provenance du serveur.

Le serveur conserve la vérité du jeu (qui a quelles cartes, qui est le coupable, etc.) et agit comme un arbitre.

Le client est l'interface du joueur. Il communique ses choix au serveur (via des messages codés) et reçoit les informations de mise à jour de l'état du jeu.

III. Choix de notre architecture : justificatifs et explications

Le choix de l'architecture repose sur une séparation stricte des responsabilités entre le serveur (logique métier et orchestration) et les clients (interface et interactions utilisateur). Cette séparation nous a été imposée, mais nous avons enrichi cette architecture par des choix techniques forts qui ont maximisé la robustesse et la fluidité du jeu.

Justification de l'utilisation des threads

La boucle principale d'un jeu est gourmande en événements utilisateurs (mouvements de souris, clics, rafraîchissement d'écran, etc.). Pour éviter que le client soit bloqué par les opérations réseau (réception des messages du serveur), nous avons utilisé **un thread POSIX dédié à la réception TCP**. Cette solution permet :

Réactivité : l'interface graphique reste fluide, même pendant l'attente de messages.

Traitement asynchrone : le thread secondaire capte en continu les données du serveur.

Découplage logique/UI : les données reçues sont transmises à la boucle SDL par une variable intermédiaire gbuffer, permettant de différer le traitement sans conflit.

Le thread réseau est lancé dès le démarrage du client via pthread_create. Il reste actif pendant toute la session.

Pourquoi ne pas utiliser les processus ou les pipes ?

Les **pipes** sont pertinents dans les communications inter-processus locales, mais inadaptés ici pour une communication à distance. Quant aux **processus**, ils sont plus coûteux en ressources et plus complexes à synchroniser que les **threads**, notamment pour le partage de mémoire (données de jeu communes). Les **threads POSIX** partagent l'espace mémoire et conviennent mieux à notre besoin :

- pas de surcoût de duplication mémoire,
- accès simplifié aux variables globales (comme synchro, gbuffer),
- simplicité de lancement et de contrôle.

Synchronisation avec volatile et mutex

Même si un pthread_mutex_t est initialisé, la synchronisation principale entre le thread réseau et le thread principal se fait via une variable volatile int synchro :

- Quand le thread réseau reçoit un message, il le stocke dans gbuffer puis active synchro.
- Le thread principal détecte cette activation et traite le message en conséquence.
- Une fois le message traité, il remet synchro à 0.

Ce mécanisme simple évite les conflits et garantit que chaque message est lu et interprété une seule fois. Bien que rudimentaire, cette stratégie est efficace pour notre cas, et suffisante en l'absence de traitement concurrent massif.

Lien avec les notions vues en TP

Le projet met en pratique plusieurs notions fondamentales du module de programmation système :

1. Sockets TCP (communication réseau)

Utilisées à la fois côté serveur et côté client.

Le serveur écoute sur un port et accepte les connexions entrantes.

Les clients envoient des messages structurés (ex : C, G, O, etc.)

Les messages reçus sont décodés via sscanf.

2. Threads (gestion de l'écoute client en parallèle)

Le client lance un **thread TCP** (à l'aide de pthread_create) qui écoute en continu les messages envoyés par le serveur.

Cela permet à l'interface graphique de rester réactive pendant que les messages réseau sont traités en arrière-plan.

3. Mutex (synchronisation avec synchro)

Une variable volatile int synchro est utilisée pour synchroniser l'écoute asynchrone du serveur avec le traitement des messages côté client.

Le mutex empêche les conflits entre le thread réseau et la boucle d'événements SDL.

4. Processus / pipe (non utilisés ici mais remplacés par threads)

Le projet aurait pu faire appel à des processus ou des tubes nommés (pipes) pour les communications locales, mais dans un contexte client-serveur TCP, les **sockets** et **threads** sont plus adaptés.

IV. Spécificités techniques du code

Gestion des messages : codage simple (première lettre identifiant le type d'action), très utile pour le décodage minimaliste.

Initialisation aléatoire : mélange du paquet par permutation de 1000 paires de cartes.

Affichage SDL : textures chargées pour chaque élément (cartes, objets, boutons, emojis). L'affichage est mis à jour selon les données reçues.

Comment fonctionne le jeu concrètement ?

Le déroulement d'une partie suit un schéma bien défini, encadré par des échanges structurés entre le serveur et les clients :

Connexion initiale :

Le serveur est lancé en premier, en écoute sur un port réseau. Chaque client se connecte en envoyant une commande C avec son IP, port et nom. Lorsque 4 joueurs sont connectés, le serveur leur envoie une commande D contenant leurs 3 cartes, ainsi que des commandes V décrivant leurs symboles associés.

Début de partie :

Un joueur est désigné pour commencer (message M). Il peut alors interagir avec l'interface. Les autres joueurs sont en attente, leurs actions étant désactivées.

Actions disponibles au joueur actif :

Interroger un symbole (commande O) : demande à tous les autres joueurs s'ils possèdent un certain symbole. Réponses globales avec des V.

Cibler un joueur + symbole (commande S) : réponse chiffrée sur le nombre exact de symboles présents.

Accuser un suspect (commande G) : si correct, message W et fin de partie ; sinon, message E et élimination du joueur.

Mécanismes internes :

Chaque action déclenche une réponse du serveur qui met à jour l'état global et le communique à tous les clients.

Le client affiche ces changements visuellement (modification de la grille de symboles, affichage des croix, messages d'élimination).

Fin de partie

Si un joueur accuse correctement ou s'il reste un seul joueur non éliminé, une commande W est émise.

Tous les joueurs peuvent alors cliquer sur « Replay » (commande R). Lorsque les 4 votes sont reçus, la partie est relancée automatiquement.

Interface graphique :

Les éléments cliquables varient selon le tour et l'état du jeu (Go, Connect, Replay).

Les emojis sont affichés grâce à la commande H, ajoutant une dimension ludique.

Ce fonctionnement repose sur des messages courts, compréhensibles et efficaces, assurant une synchronisation parfaite entre le déroulement du jeu et les interfaces des joueurs.

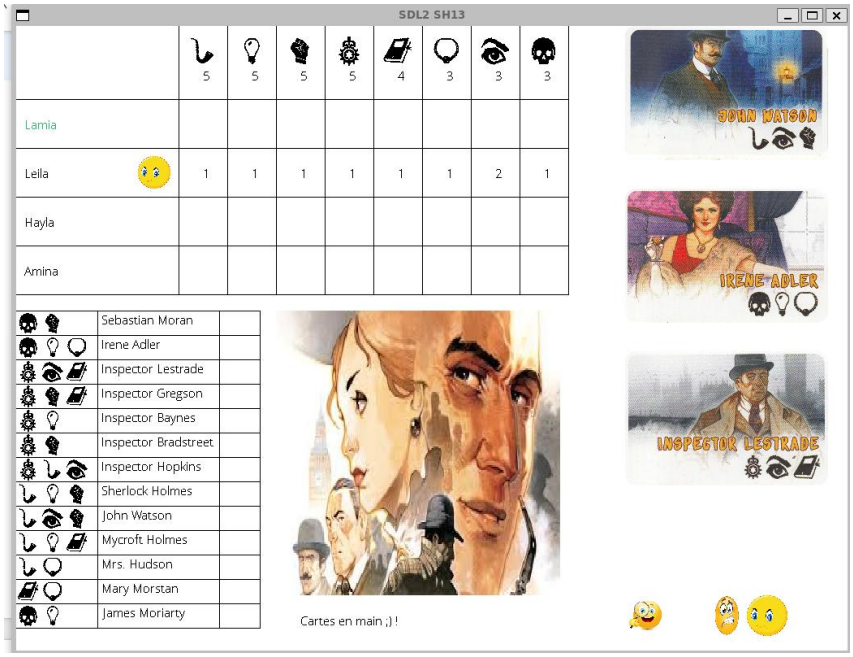

Image 2 - Emoji

Nous avons décidé de rajouter les émoji pour donner plus de convivialité au jeu (mais ils n'ont pas de fonctions spécifiques);

V. Résultat obtenu

Le jeu fonctionne en mode réseau local, avec une interface stable, la synchronisation des joueurs, la gestion des tours, des messages, des éliminations, et la possibilité de relancer une partie en fin de jeu. Les principales difficultés ont concerné la gestion d'événements simultanés, notamment entre affichage et traitement réseau.

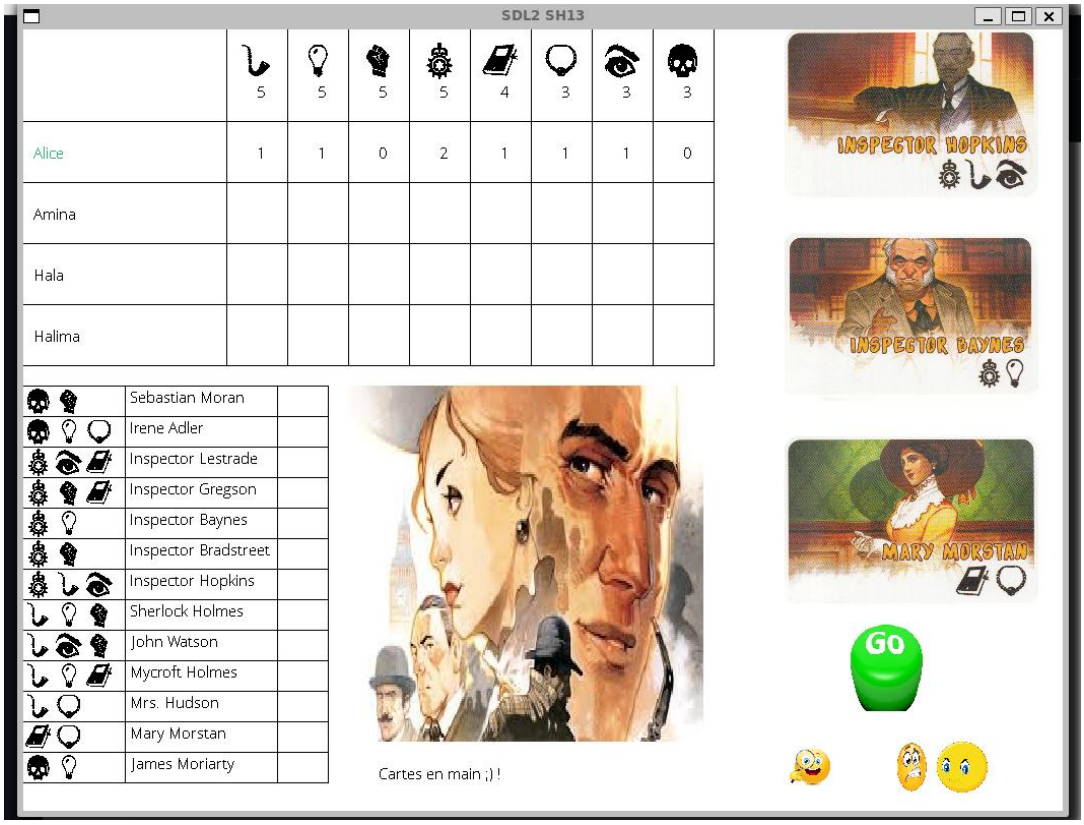

Image 3 Pendant le rôle

VI. Conclusion

Ce projet nous a permis de mettre en application concrète des notions parfois abstraites : les communications par sockets, la programmation multi-threadée, la synchronisation, et le rendu graphique. Il a surtout permis de comprendre les enjeux d'une architecture client-serveur et la nécessité de bien structurer l'information à transmettre dans un contexte réel de jeu multi-utilisateurs.