

---

# Projet Système d'exploitation

---

**Sherlock 13 : Jeu réseau avec  
interface graphique**

**SUMBUL Vedat  
EI4**

Enseignants :

Thibault Hilaire

François Pecheux

## **Table des matières**

Introduction.....	3
1. Ma Démarche de Construction du Programme (Complétion du Code C).....	4
2. Fonctionnement Général du Programme.....	6
3. Utilisation des Concepts Abordés dans le Module.....	14
Conclusion.....	16

## Introduction

Dans le cadre de ce projet, j'ai eu l'opportunité de développer une implémentation fonctionnelle du jeu de déduction Sherlock 13. L'objectif principal était de réaliser une application multijoueur basée sur une architecture client-serveur, permettant à plusieurs participants de s'affronter via le réseau. Partant d'un code C de base, ma mission a consisté à le compléter pour donner vie à la logique du jeu et assurer une communication fiable et réactive entre le serveur central et les différentes instances clientes graphiques.

Dans cette version informatique :

- Un **serveur** (écrit en C) gère la distribution des cartes, la progression du jeu et la communication réseau.
- Des **clients** (eux aussi en C, avec interface SDL2) se connectent au serveur, affichent graphiquement la situation, et permettent aux joueurs de réaliser leurs actions.

Ce rapport détaille ma démarche de construction, explique le fonctionnement du programme dans ses aspects généraux et spécifiques, et met en lumière l'application concrète des concepts clés étudiés durant les travaux pratiques de ce module, notamment la gestion des sockets, des processus, et les mécanismes de multiplexage et de synchronisation. Je décris ici comment j'ai abordé les défis techniques et les choix que j'ai opérés pour réaliser cette application réseau interactive.

## **1. Ma Démarche de Construction du Programme (Complétion du Code C)**

Mon travail a débuté avec un socle de code C fourni, comprenant les bases nécessaires à la compilation et à l'initialisation des composants essentiels comme la bibliothèque graphique SDL côté client. Le défi consistait à transformer ce code embryonnaire en une application client-serveur interactive et pleinement fonctionnelle pour le jeu Sherlock 13.

Ma première étape a été de m'immerger dans les règles spécifiques et la dynamique de jeu de Sherlock 13. Il était fondamental de comprendre comment les informations circulent, comment les actions des joueurs influencent l'état du jeu, et quelles données précises chaque joueur est censé connaître (ou ne pas connaître) à un instant donné. Cette analyse a directement guidé la modélisation des données côté serveur (pour maintenir la vérité du jeu) et côté client (pour afficher l'état perçu par le joueur).

Ensuite, j'ai élaboré le protocole de communication réseau. J'ai listé exhaustivement toutes les interactions nécessaires : l'établissement de la connexion initiale, la manière dont le serveur communique la liste des participants, l'envoi des cartes de départ et des informations initiales sur la table, la notification du joueur dont c'est le tour, l'envoi des différentes actions que le joueur peut entreprendre (faire une suspicion, interroger sur un objet, accuser le coupable), la manière dont le serveur renvoie les résultats de ces actions, et enfin, la signalisation de la conclusion de la partie. Pour chaque type d'échange, j'ai défini un format de message texte simple, concis et structuré autour d'une commande d'un caractère (par exemple, 'C' pour connexion, 'S' pour suspicion, 'G' pour accusation, 'M' pour tour) suivi des arguments nécessaires. Ce protocole est devenu le langage commun entre le client et le serveur.

Fort de ce protocole, j'ai entrepris la complétion du code serveur (`server.c`). Mon travail s'est concentré sur l'implémentation de la boucle principale de gestion des connexions et des messages, et surtout, sur la logique de jeu. J'ai développé la fonction `processClientMessage` pour parser les commandes reçues et déclencher les actions appropriées. J'ai ensuite implémenté les fonctions de gestion des actions des joueurs (`handleGuess`, `handleObjectQuery`, `handleSuspicion`) et du flux du jeu (`advanceTurn`). Cela a inclus la logique de gestion de l'état global : qui possède quelle carte, le contenu de la table des informations (les comptes d'objets), l'identification du coupable, le suivi du joueur courant, et la gestion des joueurs éliminés. J'ai aussi ajouté la logique d'initialisation de la partie une fois les 4 joueurs connectés (mélange des cartes, distribution, calcul initial de la table).

Simultanément, j'ai travaillé sur la complétion du code client (`sh13.c`). J'ai implémenté la capacité à se connecter au serveur via les informations fournies en ligne de commande. J'ai développé la fonction `sendMessageToServer` pour formater et envoyer les requêtes du joueur selon le protocole défini. Le cœur du travail client a été la fonction `processServerMessage`, responsable de recevoir et d'interpréter les messages venant du serveur. C'est dans cette fonction que l'état local du jeu côté client est mis à jour en fonction des informations reçues (mise à jour de la liste des joueurs, réception des cartes, mise à jour des valeurs dans la table des informations, changement de joueur dont c'est le tour, réception du résultat final).

Une étape délicate a été l'intégration harmonieuse de la gestion réseau avec la boucle événementielle de la SDL, qui gère l'interface utilisateur. Il était impératif que le programme client puisse simultanément écouter le réseau pour les messages serveur et réagir aux interactions de l'utilisateur (clics de souris, etc.) sans se bloquer. J'ai résolu ce point en utilisant la fonction `select()` pour sonder le socket réseau de manière non bloquante.

Enfin, j'ai complété et affiné la fonction `renderUI` pour qu'elle affiche fidèlement l'état actuel du jeu, en s'adaptant aux différentes phases de la partie (attente, jeu, fin) et en visualisant clairement toutes les informations disponibles pour le joueur (ma main, la table, la liste des suspects barrés, les boutons d'action actifs/inactifs, les messages de statut). J'ai notamment implémenté l'affichage visuel des sélections et du joueur courant.

Ma méthode a été résolument incrémentale et basée sur des tests fréquents. J'ai ajouté et testé une fonctionnalité (un type de message ou une action) à la fois, vérifiant systématiquement la communication entre le client et le serveur et l'impact sur l'état du jeu et l'interface graphique.

## 2. Fonctionnement Général du Programme

Mon programme s'exécute sous la forme d'une architecture client-serveur distribuée. Un processus serveur unique maintient l'état central et la logique du jeu, tandis que plusieurs processus clients (un par joueur) gèrent l'interface utilisateur et interagissent avec le serveur via le réseau.

Voici comment se déroule typiquement une partie, depuis le lancement jusqu'à la fin :

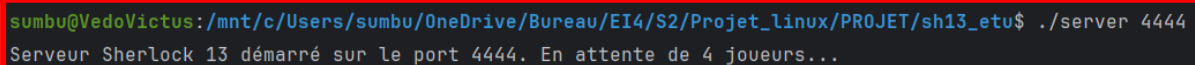
Le processus commence par le lancement du serveur. J'ouvre un terminal et j'exécute le script de compilation, puis le serveur en spécifiant le port d'écoute :

```
./cmd.sh
```

```
./server 4444
```

Le serveur s'initialise et affiche un message indiquant qu'il attend les connexions des joueurs.

Terminal Serveur après lancement, en attente :

A screenshot of a terminal window with a black background and white text. The prompt is 'sumbu@VedoVictus: /mnt/c/Users/sumbu/OneDrive/Bureau/EI4/S2/Projet\_linux/PROJET/sh13\_etu\$'. The command entered is './server 4444'. The output is 'Serveur Sherlock 13 démarré sur le port 4444. En attente de 4 joueurs...'.

```
sumbu@VedoVictus:/mnt/c/Users/sumbu/OneDrive/Bureau/EI4/S2/Projet_linux/PROJET/sh13_etu$ ./server 4444
Serveur Sherlock 13 démarré sur le port 4444. En attente de 4 joueurs...
```

Ensuite, les joueurs lancent l'application client. Dans des terminaux distincts (ou sur différentes machines), chaque joueur exécute le client en lui passant l'adresse IP du serveur, le port du serveur, sa propre adresse IP (bien que moins utilisée dans la version finale côté serveur), un port local (également moins utilisé dans la version finale pour le socket persistant), et son nom. Par exemple, pour 4 joueurs en local :

```
./sh13 127.0.0.1 4444 127.0.0.1 5550 Alice
```

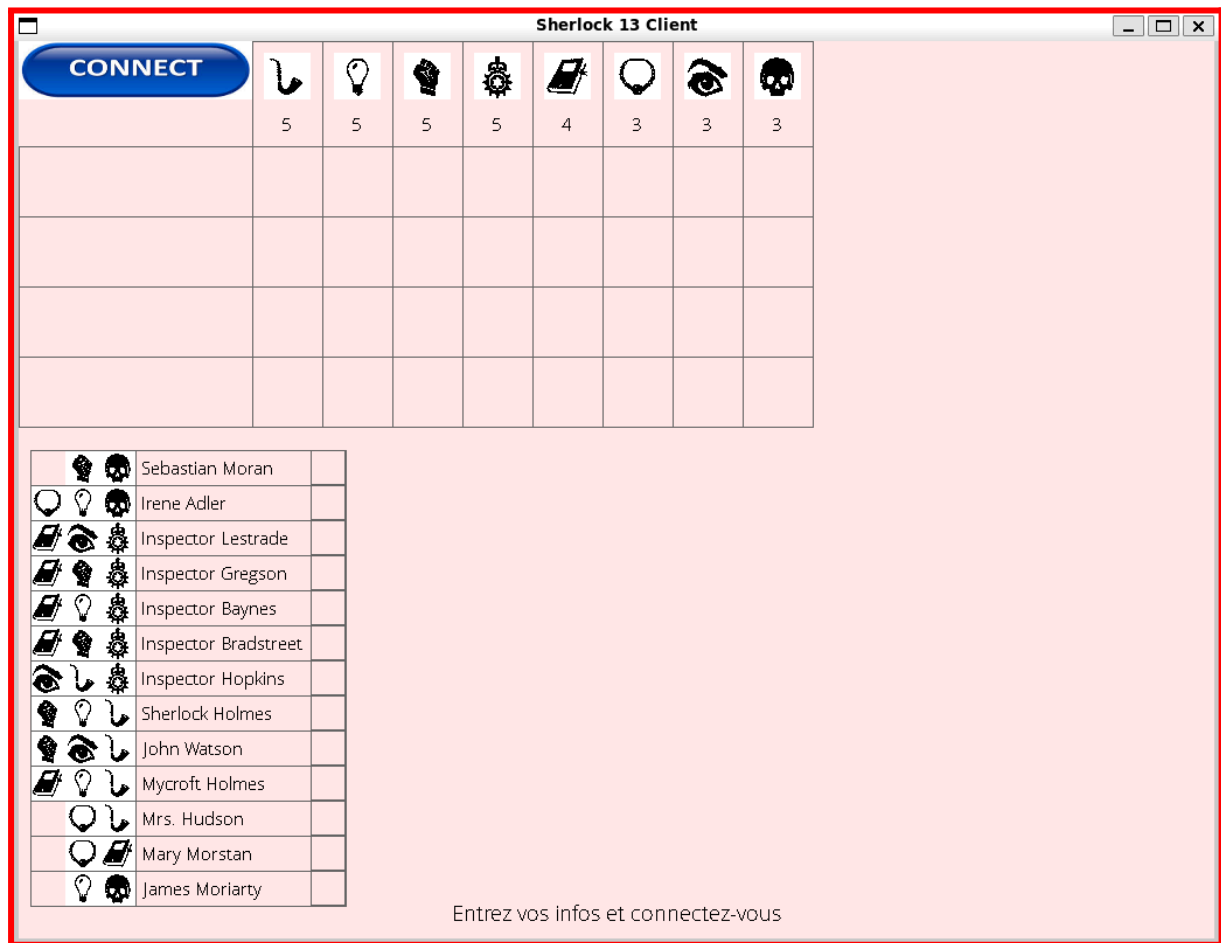
```
./sh13 127.0.0.1 4444 127.0.0.1 5551 Bob
```

```
./sh13 127.0.0.1 4444 127.0.0.1 5552 Carol
```

```
./sh13 127.0.0.1 4444 127.0.0.1 5553 Dave
```









Chaque client s'initialise, configure sa connexion, affiche une fenêtre SDL avec une interface d'attente et un bouton "Connect".

Fenêtre Client avant connexion, avec le bouton Connect :



Lorsque le joueur clique sur "Connect", le client tente d'établir une connexion TCP avec le serveur. Si la connexion réussit, le client envoie un message de connexion ('C') contenant son nom. Le serveur reçoit la connexion, attribue un ID au client (0, 1, 2, ou 3), stocke ses informations et lui envoie son ID ('I'). Le serveur met ensuite à jour sa liste interne des joueurs et diffuse cette liste ('L') à *tous* les clients connectés. L'interface client se met à jour pour afficher les noms des joueurs connectés. Le bouton "Connect" se désactive.

Fenêtre Client après connexion, affichant les noms des joueurs connectés, bouton Connect désactivé :

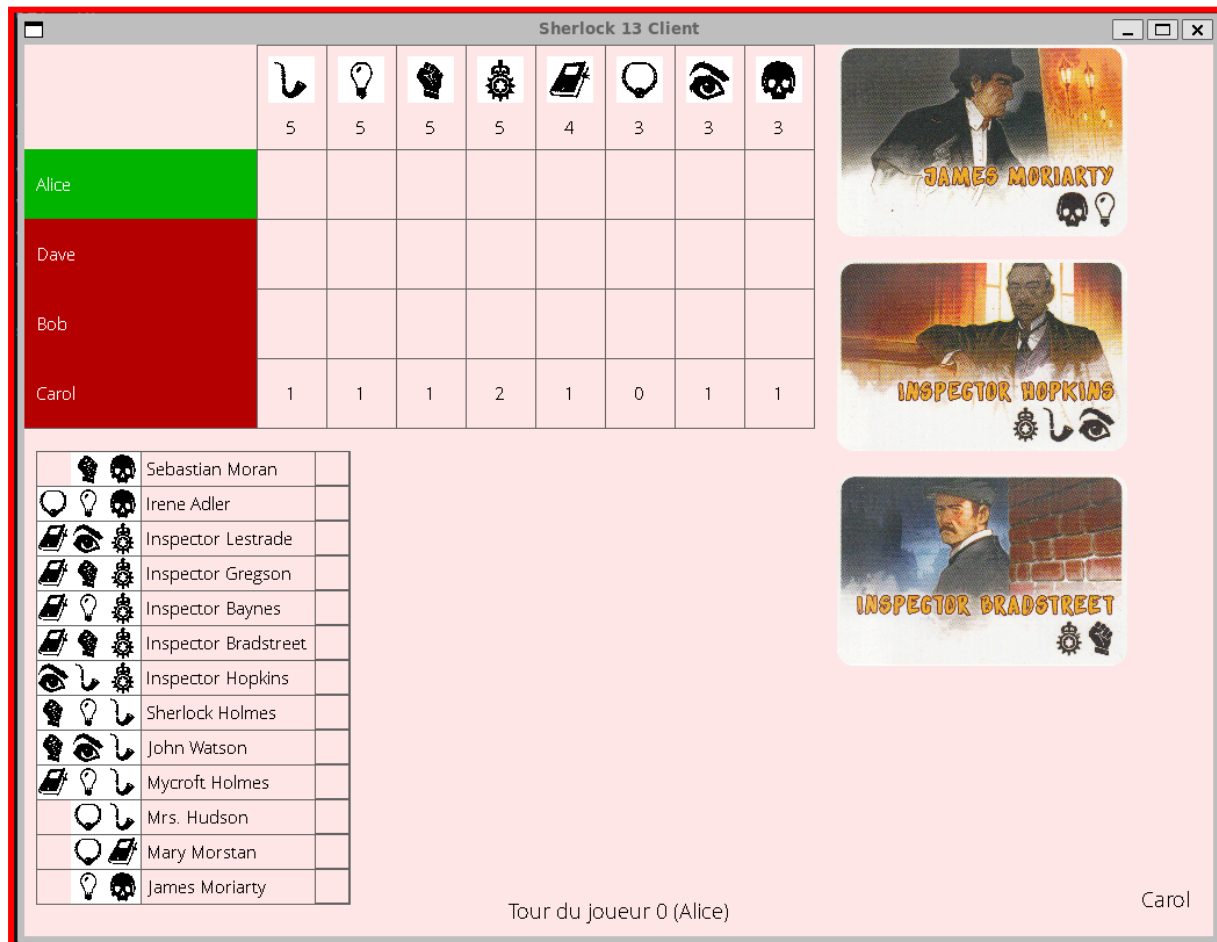
Sherlock 13 Client								
								
	5	5	5	5	4	3	3	3
Alice								
Dave								
Bob								
Carol	1	1	1	2	1	0	1	1

Quand le quatrième joueur se connecte, le serveur détecte que le nombre maximum de joueurs est atteint et déclenche le début de la partie. Il mélange le paquet de cartes, choisit le coupable (la dernière carte du deck mélangé), distribue les cartes de la main à chaque joueur (les 3 premières cartes du deck pour chaque joueur), et calcule les informations initiales de la table (le nombre d'objets que chaque joueur *ne possède pas* sur les cartes qui lui ont été distribuées). Le serveur envoie ensuite à chaque client leurs 3 cartes ('D') et toutes les valeurs initiales de la table ('V' pour chaque paire (joueur, objet)). Enfin, il désigne le premier joueur dont c'est le tour (généralement le joueur 0) et en informe tous les clients ('M').

À la réception des messages 'D', 'V', et 'M', l'état interne du client se met à jour (gameStarted = 1, remplissage de myCards et tableCartes, mise à jour de joueurCourant). L'interface graphique bascule : la zone de jeu principale apparaît, affichant la grille, la liste des suspects, et les cartes du joueur. Le message de statut indique qui joue, et le bouton "GO" s'active uniquement si c'est le tour de ce client.



Interface client au début de la partie, montrant la grille, la liste des suspects, les cartes, le joueur courant indiqué :



Le jeu procède par tours. Le joueur dont c'est le tour (indiqué visuellement dans la grille) peut effectuer une action. L'interface graphique guide le joueur :

- La Grille Joueurs/Objets :** C'est le cœur visuel des informations partagées. Les lignes représentent les joueurs (y compris moi-même), et les colonnes représentent les 8 types d'objets.
  - Les cellules affichent un nombre : c'est le compte d'objets de ce type que le joueur correspondant *ne possède pas*, tel que déduit des cartes distribuées. Une valeur de -1 signifie "inconnu" en début de partie ou si l'information n'a pas été révélée. Une valeur de 100, affichée comme '\*', indique que le joueur *possède* cet objet (révélé par une interrogation d'objet 'O').
  - Couleurs des lignes :** La ligne correspondant au joueur dont c'est le tour est mise en évidence avec une couleur (j'ai utilisé le vert). Les lignes des autres joueurs sont dans une autre couleur (rouge). Les joueurs éliminés sont grisés. Cela permet d'identifier rapidement qui doit jouer.
  - Je peux cliquer sur une ligne de joueur et une colonne d'objet dans la grille pour sélectionner un joueur et un objet en vue d'une suspicion ('S'). Les sélections sont mises en évidence (rose pour le joueur, vert pour l'objet).

- **La Liste des Suspects** : À gauche, la liste affiche les 13 personnages suspects avec les icônes des objets qui leur sont associés. À droite de chaque nom, il y a une petite case à cocher/barrer.
  - Je peux cliquer sur le nom d'un suspect pour le sélectionner en vue d'une accusation ('G'). La sélection est mise en évidence (bleu).
  - Je peux cliquer sur la petite case pour barrer un suspect que j'ai éliminé par déduction (guiltGuess). Une croix rouge apparaît. C'est une aide visuelle locale, non partagée avec le serveur.
- **Mes Cartes** : En bas à droite (ou dans une zone dédiée), mes 3 cartes sont affichées. Celles-ci sont mes informations secrètes.
- **Le Bouton "GO"** : Ce bouton n'est actif que lorsque c'est mon tour (goEnabled = 1). Une fois que j'ai fait une sélection valide (joueur+objet pour suspicion, ou suspect pour accusation, ou juste un objet pour interrogation si aucune autre sélection n'est faite), je clique sur ce bouton pour soumettre mon action au serveur.
- **Le Message de Statut** : Un texte affiché en bas de l'écran (centré) me fournit des informations sur l'état du jeu : "Entrez vos informations...", "Connecté...", "Tour du joueur X...", "C'est votre tour...", ou les messages de fin de partie.
- **Mon Nom** : Mon propre nom est affiché en bas à droite pour me rappeler mon identité dans la partie.

Lorsqu'un joueur clique sur "GO" pendant son tour après avoir fait des sélections, le client envoie le message approprié au serveur ('S', 'O', ou 'G').

Interface client pendant le tour d'un joueur, montrant une sélection de joueur(en rose) et d'objet(en vert clair) pour suspicion, bouton GO actif :



Interface client pendant le tour d'un joueur, montrant une sélection de suspect(en violet clair) pour accusation, bouton GO actif :



Le serveur traite l'action. Si c'est une suspicion ('S') ou une interrogation ('O'), il calcule le résultat (le nombre d'objets ou le statut de possession) et diffuse cette information à tous les clients via un message 'V' (pour Suspicion) ou 'K' (pour Interrogation d'Objet). Tous les clients mettent à jour leur tableCartes locale en conséquence, et l'interface graphique se rafraîchit.

Interface client après une action de suspicion ou d'interrogation, montrant la table des informations mise à jour :

The screenshot shows the 'Sherlock 13 Client' window. It features a game board with a grid of numbers and icons. The board is divided into sections for players Alice, Dave, Bob, and Carol. A list of suspects is shown on the left, each with a set of icons. On the right, there are three character cards: Mycroft Holmes, Mrs. Hudson, and Sherlock Holmes. The interface also displays the current turn (Tour du joueur 1 (Dave)) and the player Alice.

	5	5	5	5	4	3	3	3
Alice	3	2	1	0	1	1	0	0
Dave								
Bob					2			
Carol								

Sebastian Moran	
Irene Adler	
Inspector Lestrade	
Inspector Gregson	
Inspector Baynes	
Inspector Bradstreet	
Inspector Hopkins	
Sherlock Holmes	
John Watson	
Mycroft Holmes	
Mrs. Hudson	
Mary Morstan	
James Moriarty	

Tour du joueur 1 (Dave)

Alice

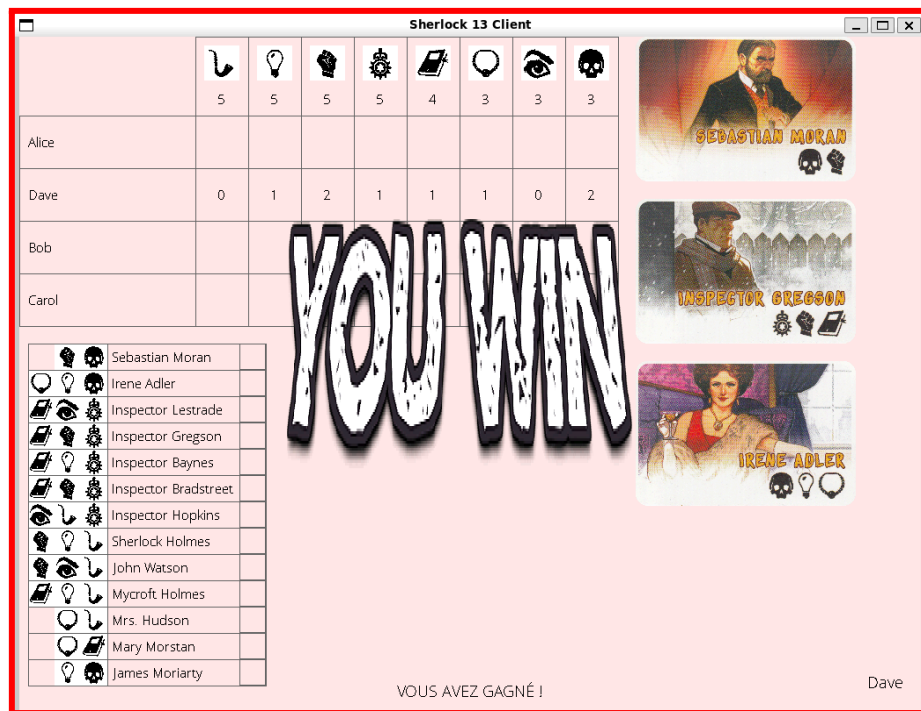
Si l'action est une accusation ('G'), le serveur vérifie si le suspect désigné est bien le coupable (la carte deck[CULPRIT\_CARD\_INDEX]).

- Si l'accusation est correcte, le joueur gagne. Le serveur marque le joueur comme gagnant, la partie se termine.
- Si l'accusation est incorrecte, le joueur est éliminé. Le serveur marque le joueur comme perdu. La partie continue si au moins deux joueurs actifs restent.

Dans les deux cas (accusation correcte ou incorrecte, ou si une action normale a été effectuée), si la partie n'est pas terminée, le serveur passe le tour au joueur actif suivant et en informe tous les clients via un nouveau message 'M'.

La partie se termine soit lorsqu'un joueur accuse correctement le coupable (ce joueur gagne), soit lorsque tous les autres joueurs sont éliminés (le dernier joueur actif gagne). Le serveur envoie alors des messages de fin : 'Z' au gagnant et 'P' aux perdants. L'interface client affiche un message de fin ("VOUS AVEZ GAGNÉ !" ou "VOUS AVEZ PERDU !") et une image appropriée. Le bouton "GO" se désactive pour tout le monde.

Interface client en fin de partie, montrant l'écran de victoire :



Interface client en fin de partie, montrant l'écran de défaite :



Le programme client reste actif en affichant l'écran de fin jusqu'à ce que le joueur ferme la fenêtre. Le serveur continue de fonctionner mais refuse les nouvelles connexions et n'attend plus d'actions de jeu (il peut potentiellement être arrêté manuellement).



### 3. Utilisation des Concepts Abordés dans le Module

Dans la construction de ce jeu, j'ai mis en pratique plusieurs concepts fondamentaux de la programmation système et réseau vus durant le module :

- **Sockets** : J'ai utilisé les sockets comme l'outil de base pour la communication réseau entre les processus client et serveur. Le serveur ouvre un socket d'écoute (AF\_INET, SOCK\_STREAM) sur un port spécifié, auquel les clients se connectent. Chaque connexion acceptée par le serveur lui fournit un nouveau socket dédié pour communiquer *uniquement* avec ce client particulier. Côté client, j'ouvre un socket pour initier une connexion sortante vers l'adresse et le port du serveur. Les échanges de données (les messages du protocole que j'ai défini) transitent par ces canaux fiables (TCP), en utilisant les appels `read()` pour recevoir et `write()` pour envoyer des données. Les sockets encapsulent les complexités du réseau, me permettant de me concentrer sur le protocole d'application.
- **`select()`** : J'ai trouvé la fonction `select()` particulièrement utile, tant côté serveur que client, pour gérer l'efficacité des entrées/sorties réseau. Côté serveur, `select()` me permet de surveiller simultanément l'activité sur plusieurs sockets : le socket d'écoute (pour savoir si un nouveau client veut se connecter) et les sockets de communication avec tous les clients connectés (pour savoir si l'un d'eux a envoyé un message). Cela évite d'utiliser des appels bloquants comme `read()` sur un seul socket à la fois, ce qui rendrait le serveur incapable de gérer d'autres clients en attendant. `select()` signale quels sockets sont prêts à être lus, me permettant de traiter les messages au fur et à mesure. Côté client, son utilisation est différente mais tout aussi importante : je l'utilise avec un délai d'attente nul ou très court (`select_timeout`). Intégré dans la boucle principale de la SDL, cela me permet de vérifier de manière non bloquante s'il y a des données en attente sur le socket connecté au serveur. Si `select()` indique que le socket est prêt, je lis le message ; sinon, la boucle continue pour gérer les événements graphiques. Cela garantit que l'interface utilisateur reste fluide et réactive, même en attendant un message du serveur.
- **Processus** : L'architecture du jeu repose sur plusieurs processus distincts s'exécutant potentiellement sur différentes machines : un processus pour le serveur et un processus séparé pour chaque client. La communication entre ces processus est réalisée exclusivement par le biais du réseau et des sockets TCP, le modèle d'IPC le plus adapté pour des applications distribuées.
- **Threads** : Dans la conception initiale ou potentielle du programme client, j'ai envisagé l'utilisation d'un thread dédié pour la réception des messages réseau afin de ne pas bloquer l'interface graphique. Cependant, j'ai finalement opté pour l'approche basée sur `select()` dans le thread principal. La raison principale de ce choix est d'éviter les complications liées à la synchronisation entre un thread réseau et le thread principal de la SDL. La modification concurrente de variables partagées (l'état du jeu côté client, par exemple la table des informations ou le joueur courant)

par un thread réseau et la lecture de ces mêmes variables par le thread SDL pour le rendu peut entraîner des erreurs difficiles à déboguer. `select()` m'a fourni une solution efficace pour gérer l'attente réseau de manière non bloquante au sein du thread unique de l'interface graphique, simplifiant ainsi grandement le code et évitant le besoin de mécanismes de synchronisation complexes.

- **Pipes** : Je n'ai pas fait usage de pipes dans ce projet. Les pipes sont un mécanisme de communication inter-processus (IPC) couramment utilisé entre des processus ayant une relation de parenté ou s'exécutant sur la même machine. Mon application étant conçue pour une architecture réseau où clients et serveurs peuvent être distribués, les sockets réseau étaient l'outil approprié pour la communication.
- **Mutexes** : L'utilisation de mutexes est essentielle dans les programmes multithreadés pour protéger l'accès aux données partagées et prévenir les conditions de course. Étant donné que j'ai choisi une architecture non multithreadée pour la gestion réseau (en utilisant `select()` dans le thread principal de la SDL), et que le serveur gère la concurrence via les E/S plutôt qu'avec des threads par client, il n'y avait pas de scénarios où plusieurs threads auraient accédé simultanément aux mêmes variables partagées nécessitant une protection par mutex.

## Conclusion

En achevant ce projet de développement du jeu Sherlock 13, j'ai pu concrétiser l'ensemble des concepts théoriques et pratiques abordés durant le module, en particulier ceux relatifs à la communication réseau et à la programmation système. Partir d'un code incomplet m'a forcé à analyser en profondeur les besoins fonctionnels et les contraintes techniques pour bâtir une solution robuste et interactive.

L'implémentation d'une architecture client-serveur, la définition d'un protocole de communication simple et efficace, et surtout, l'utilisation judicieuse des sockets et de la fonction `select()` pour gérer les entrées/sorties non bloquantes, ont constitué le cœur technique de cette réalisation.

J'ai particulièrement apprécié la manière dont `select()` a permis d'intégrer la gestion réseau de manière fluide dans la boucle principale de l'interface graphique SDL, un défi courant dans le développement d'applications interactives en réseau. Bien que d'autres concepts comme les threads, mutexes ou pipes aient été envisagés ou étudiés en théorie, le choix de les écarter au profit d'une solution plus simple et adaptée à cette architecture spécifique démontre une compréhension des compromis de conception.

Ce projet m'a fourni une expérience précieuse dans le développement d'applications distribuées et a renforcé ma maîtrise des outils fondamentaux de la programmation système réseau en C.