

Rapport de Projet

Implémentation Réseau du Jeu Sherlock 13

Hakan PEKUR
Polytech Sorbonne – Filière EI4
Module OS USER – Année Universitaire 2024/2025

21 avril 2025

Table des matières

1	Introduction	3
1.1	Présentation du Projet	3
1.2	Concepts Abordés	3
1.3	Structure du Rapport	4
2	Règles Détaillées du Jeu Sherlock 13	5
2.1	Mise en Place	5
2.2	Déroulement du Tour	5
2.2.1	Accusation (Message Client vers Serveur : G)	5
2.2.2	Enquête Objet (Message Client vers Serveur : O)	6
2.2.3	Enquête Spécifique (Message Client vers Serveur : S)	6
2.3	Fin de Partie	6
3	Architecture et Implémentation	7
3.1	Architecture Globale	7
3.2	Implémentation du Serveur (<code>server.c</code>)	8
3.2.1	Modèle de Threads et de Communication	8
3.2.2	Machine à États (FSM)	8
3.2.3	Structures de Données Principales	8
3.3	Implémentation du Client (<code>sh13.c</code>)	8
3.3.1	Modèle de Threads et de Communication	8
3.3.2	Synchronisation	9
3.3.3	Interface Graphique (SDL2)	9
4	Protocole de Communication	11
4.1	Synthèse du Protocole	11
4.2	Exemples d'Échanges Typiques	11
4.2.1	Connexion et Démarrage	11
4.2.2	Enquête Objet	12
4.2.3	Enquête Spécifique	12
4.2.4	Accusation Correcte	12
4.2.5	Accusation Incorrecte	12
5	Processus de Développement et Complétion du Code	13
5.1	Analyse Préalable	13
5.2	Complétion du Serveur (<code>server.c</code>)	13
5.3	Robustesse et Nettoyage	14
6	Conclusion	15
	Remerciements	16

A	Structure du Dépôt	17
B	Compilation et Exécution	18
B.1	Prérequis	18
B.2	Compilation	18
B.3	Exécution	18
B.3.1	Serveur	18
B.3.2	Clients	18

Chapitre 1

Introduction

1.1 Présentation du Projet

Ce rapport détaille la conception et l'implémentation d'une version en réseau du jeu de société **Sherlock 13**. Ce projet a été réalisé dans le cadre du module **OS USER** de la filière EI4 de Polytech Sorbonne durant l'année universitaire 2024/2025. L'objectif principal était de partir d'une base de code C incomplète fournie par les encadrants, et de la **compléter** pour obtenir une application client-serveur fonctionnelle. Cette application doit permettre à **quatre joueurs** de s'affronter dans une partie de Sherlock 13, un jeu de déduction où il faut identifier un coupable parmi treize suspects. Le serveur est responsable de la logique centrale du jeu et de la gestion de l'état de la partie, tandis que les clients, dotés d'une interface graphique basée sur la bibliothèque **SDL2**, permettent aux joueurs d'interagir avec le jeu. Au-delà de la simple compléction fonctionnelle, ce projet a été l'occasion d'appliquer et d'approfondir plusieurs concepts fondamentaux de la programmation système et réseau.

1.2 Concepts Abordés

La réalisation de ce projet a permis de mettre en œuvre les aspects suivants de la programmation système :

- **Sockets TCP/IP** : Utilisation intensive des primitives de sockets (API Berkeley Sockets) pour établir une communication fiable et connectée entre le processus serveur et les multiples processus clients. Cela inclut la création (**socket**), la liaison (**bind**), l'écoute (**listen**), l'acceptation (**accept**) et la connexion (**connect**) de sockets, ainsi que l'échange de données structurées sous forme de messages textuels (**read**, **write**).
- **Processus** : L'architecture repose sur des processus distincts (un serveur, quatre clients) qui communiquent uniquement via le réseau, illustrant la communication inter-processus (IPC) distante.
- **Threads** : Le client utilise une architecture multi-threadée (deux threads) pour découpler la gestion de l'interface graphique (potentiellement bloquante sur les événements ou le rendu) de la réception asynchrone des messages réseau (bloquante sur **accept** ou **read**). Cela améliore significativement la réactivité de l'interface utilisateur.
- **Synchronisation Inter-Threads** : Bien qu'un mutex (**pthread_mutex_t**) soit déclaré dans le code client initial, le mécanisme de synchronisation choisi et implémenté repose sur un simple drapeau partagé (**volatile int synchro**). Cette approche, bien que basique, s'est avérée suffisante pour coordonner l'accès au buffer de réception (**gbuffer**) entre le thread d'écoute et le thread principal dans le cadre de l'architecture client fournie. Les mécanismes plus complexes comme les mutex ou sémaphores n'étaient pas nécessaires pour cette interaction spécifique.

- **Gestion des Ressources** : Implémentation de mécanismes pour la libération propre des ressources allouées (descripteurs de fichiers de sockets, mémoire, textures SDL, polices TTF) et pour l'arrêt coordonné des threads.

1.3 Structure du Rapport

Ce document est organisé comme suit :

- Le Chapitre 2 rappelle les règles du jeu Sherlock 13 pour quatre joueurs.
- Le Chapitre 3 détaille l'architecture globale client-serveur ainsi que les spécificités d'implémentation du serveur et du client.
- Le Chapitre 4 décrit le protocole de communication textuel utilisé entre le serveur et les clients.
- Le Chapitre 5 explique le processus de développement, en mettant l'accent sur la complétion du code et les choix logiques effectués.
- Le Chapitre 6 conclut ce rapport en résumant les apports du projet et les compétences mises en œuvre.
- Les Annexes fournissent des informations complémentaires sur la structure du dépôt et la compilation.

Chapitre 2

Règles Détailées du Jeu Sherlock 13

Sherlock 13 est un jeu de déduction conçu pour 2 à 4 joueurs. L'implémentation réalisée dans ce projet se concentre exclusivement sur la version à **quatre joueurs**. L'objectif est d'identifier le **Coupable**, un personnage choisi secrètement parmi 13 suspects uniques. Chaque suspect est représenté par une carte et possède une combinaison spécifique de symboles. Il existe 8 types de symboles distincts dans le jeu : Pipe, Ampoule, Poing, Couronne, Carnet, Collier, Oeil, Crâne. La répartition de ces symboles entre les 13 personnages est fixe et connue de tous les joueurs via la feuille d'enquête (ou l'interface graphique dans notre cas).

2.1 Mise en Place

Au début de la partie :

1. Les 13 cartes Suspect sont mélangées.
2. Une carte est tirée au hasard et placée face cachée au centre : c'est le **Coupable**. Personne ne la regarde.
3. Les 12 cartes restantes sont distribuées équitablement entre les 4 joueurs. Chaque joueur reçoit donc **3 cartes** qu'il garde secrètes. Ces cartes représentent des suspects innocents.
4. Chaque joueur reçoit également des informations initiales (calculées par le serveur et transmises) sur le nombre total de chaque type de symbole qu'il possède dans sa main de 3 cartes. Ces informations sont cruciales pour commencer les déductions.

2.2 Déroulement du Tour

Le jeu se joue en tours, dans le sens horaire, en commençant par un premier joueur désigné (le joueur 0 dans notre implémentation). À son tour, le joueur **actif** doit choisir et effectuer **une seule** des trois actions suivantes :

2.2.1 Accusation (Message Client vers Serveur : G)

Le joueur pense avoir identifié le Coupable. Il annonce le nom du suspect qu'il accuse.

- **Si l'accusation est correcte** : Le joueur révèle la carte Coupable. Il a gagné ! La partie se termine immédiatement.
- **Si l'accusation est incorrecte** : Le joueur est informé de son erreur (sans révéler le vrai coupable). Il est **éliminé** de la partie : il ne peut plus effectuer d'action ni gagner. Cependant, il doit continuer à répondre (via le serveur) aux enquêtes des autres joueurs si sa main contient des informations pertinentes. Le jeu continue avec les joueurs restants.

2.2.2 Enquête Objet (Message Client vers Serveur : 0)

Le joueur choisit l'un des 8 symboles et demande : "Quels autres joueurs possèdent au moins une carte avec ce symbole dans leur main ?". Tous les autres joueurs **actifs** qui possèdent ce symbole doivent l'indiquer (dans notre implémentation, le serveur centralise cette information et la renvoie uniquement au joueur enquêteur). Les joueurs n'indiquent pas combien de fois ils possèdent le symbole, juste s'ils le possèdent au moins une fois.

2.2.3 Enquête Spécifique (Message Client vers Serveur : S)

Le joueur choisit un **autre joueur** (même un joueur inactif) et un **symbole spécifiques**. Il demande : "Combien de fois possèdes-tu ce symbole dans ta main ?". Le joueur interrogé est obligé de répondre honnêtement en donnant le nombre exact (0, 1, 2, ou 3). Le serveur fournit cette réponse uniquement au joueur enquêteur.

2.3 Fin de Partie

La partie se termine de deux manières :

1. Un joueur réussit une **Accusation correcte**. Ce joueur est le vainqueur.
2. Tous les joueurs sauf un ont été éliminés suite à des accusations incorrectes. Le **dernier joueur actif** est déclaré vainqueur, sans même avoir besoin de formuler une accusation finale.



FIGURE 2.1 – Exemple de carte suspect : John Watson (Pipe, Oeil, Poing).

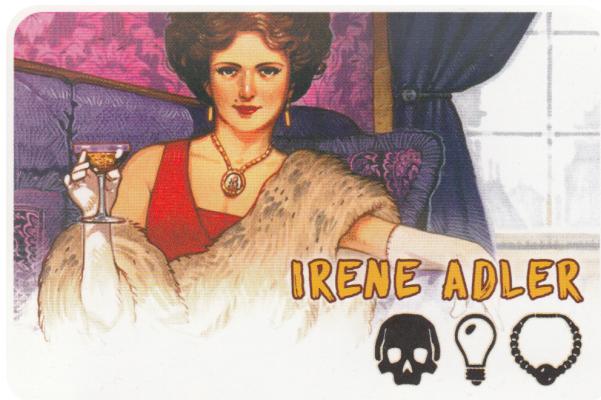


FIGURE 2.2 – Exemple de carte suspect : Irene Adler (Crâne, Ampoule, Collier).

Chapitre 3

Architecture et Implémentation

Le projet Sherlock 13 est structuré selon une architecture client-serveur classique, communiquant via le protocole TCP/IP. Les choix d'implémentation précis ont été en partie guidés par la structure du code squelette fourni.

3.1 Architecture Globale

L'application se compose de deux types de processus :

- **Un processus Serveur (`server.c`)** : Instance unique qui centralise l'état du jeu (cartes des joueurs, coupable, joueur courant, joueurs actifs) et la logique métier (validation des actions, réponses aux enquêtes, détermination du vainqueur).
- **Quatre processus Clients (`sh13.c`)** : Chaque joueur lance une instance du client. Le client gère l'interface graphique (affichage du jeu, interactions utilisateur) et communique avec le serveur pour envoyer les actions du joueur et recevoir les mises à jour de l'état du jeu.

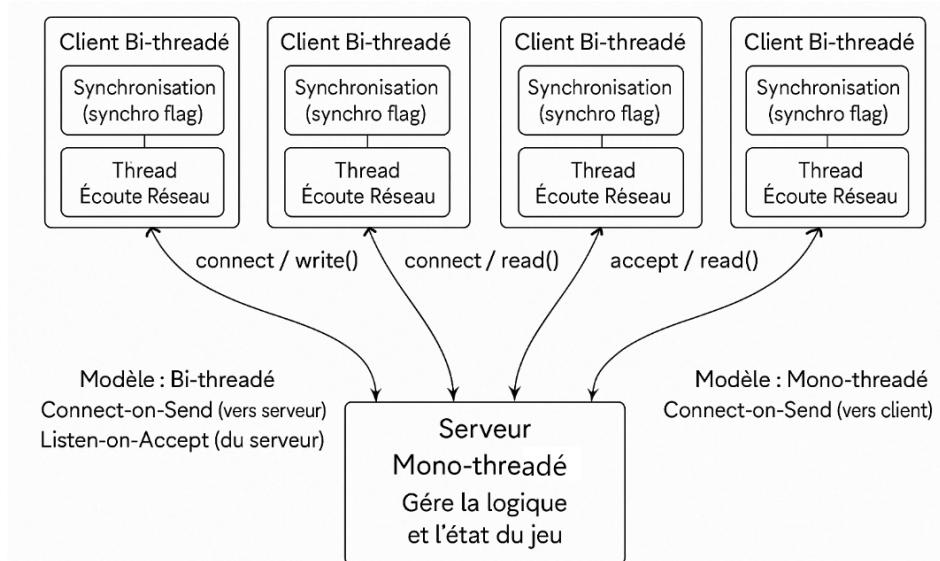


FIGURE 3.1 – Schéma simplifié de l'architecture client-serveur.

3.2 Implémentation du Serveur (server.c)

3.2.1 Modèle de Threads et de Communication

Le serveur, tel que structuré dans le code initial et complété, fonctionne sur un modèle **mono-threadé**.

- **Réception des requêtes** : La boucle principale du serveur attend une connexion entrante sur son socket d'écoute (`accept()`). Lorsqu'un client se connecte, le serveur lit (`read()`) un unique message de ce client, puis **ferme immédiatement ce socket de connexion** (`close()`). Il ne maintient pas de connexion persistante avec les clients pour la réception.
- **Envoi des réponses/mises à jour** : Pour envoyer un message à un client (ou à tous les clients en mode broadcast), le serveur initie une **nouvelle connexion TCP sortante** vers l'adresse IP et le port d'écoute spécifiques du client (`socket()`, `connect()`, `write()`, `close()`).

3.2.2 Machine à États (FSM)

Le déroulement de la partie côté serveur est géré par une machine à états simple, contrôlée par la variable globale `fsmServer` :

- **État 0 (Lobby / Attente)** : Le serveur est en attente de la connexion des 4 joueurs. Il accepte les messages C (Connexion), enregistre les informations des joueurs (IP, port d'écoute, nom), leur attribue un ID (message I), et informe tout le monde de la nouvelle liste de joueurs (message L). Une fois les 4 joueurs connectés, il initialise la partie (mélange via `melangerDeck`, calcul table symboles via `createTable`), envoie les informations initiales (D et V), désigne le premier joueur (M), et passe à l'état 1.
- **État 1 (Jeu en cours)** : Le serveur traite les messages d'action (G, 0, S) envoyés par le joueur courant et actif. Il vérifie la validité de l'action, met à jour l'état du jeu (état `active` d'un joueur si accusation ratée, passage au `joueurCourant` suivant), et envoie les résultats ou mises à jour nécessaires aux clients (V, F, M, W).
- **État 2 (Fin de partie)** : Cet état est atteint lorsqu'un joueur gagne (accusation correcte ou dernier joueur actif). Le serveur a envoyé un message W final. La boucle principale du serveur se termine (ou devrait se terminer proprement).

3.2.3 Structures de Données Principales

L'état du jeu est maintenu dans des variables globales :

- `deck[13]` : Tableau d'entiers (0-12) représentant les cartes suspect. L'ordre est mélangé au début. `deck[12]` contient l'indice de la carte Coupable.
- `coupable` : Variable stockant l'indice (0-12) de la carte coupable (`deck[12]`).
- `tableCartes[4][8]` : Tableau 2D pré-calculé où `tableCartes[i][j]` stocke le nombre d'exemplaires du symbole j possédés par le joueur i dans sa main initiale de 3 cartes.
- `tcpClients[4]` : Tableau de structures contenant les informations de chaque joueur (IP, port d'écoute, nom) et son état (`active` : 1 si en jeu, 0 si éliminé).
- `joueurCourant` : Indice (0-3) du joueur dont c'est actuellement le tour.

3.3 Implémentation du Client (sh13.c)

3.3.1 Modèle de Threads et de Communication

Conformément à l'architecture serveur "connect-on-send", et pour assurer une bonne réactivité, le client utilise une architecture **bi-threadée** :

- **Thread Principal (`main`)** : Responsable de l'initialisation et de la boucle principale SDL2 (gestion des événements, rendu graphique), de la gestion de l'état local du jeu (cartes reçues, informations sur les symboles, sélections de l'utilisateur), et de l'envoi des requêtes au serveur (via `sendMessageToServer`). L'envoi se fait par création d'une connexion TCP sortante éphémère vers le serveur principal.
- **Thread d'Écoute Réseau (`fn_serveur_tcp`)** : Crée au démarrage du client via `pthread_create`, ce thread met en place un **socket d'écoute passif** (`socket`, `bind`, `listen`) sur le port spécifié par le client lors de son lancement. Il attend (`accept`) les connexions initiées par le serveur principal. Lorsqu'une connexion est acceptée, il lit (`read`) le message envoyé par le serveur, le stocke dans un buffer partagé (`gbuffer`), signale sa présence au thread principal via le drapeau `synchro`, attend que le thread principal traite le message (remise de `synchro` à 0), puis ferme la connexion (`close`) et se remet en attente (`accept`).

3.3.2 Synchronisation

La communication entre le thread d'écoute et le thread principal se fait via :

- Le buffer partagé `char gbuffer[256]`
- Le drapeau `volatile int synchro`

3.3.3 Interface Graphique (SDL2)

Le thread principal utilise SDL2, `SDL_image` et `SDL_ttf` pour afficher :

- La grille principale affichant les informations connues (`tableCartes`) sur les symboles possédés par chaque joueur.
- La liste des 13 suspects avec leurs symboles et une case à cocher (`guiltGuess`) pour que le joueur marque ses déductions.
- Les 3 cartes (`b[0]`, `b[1]`, `b[2]`) détenues par le joueur.
- Les boutons "Connect" et "GO", dont l'affichage dépend de l'état (`connectEnabled`, `goEnabled`).
- Les noms des joueurs (`gNames`), avec une indication visuelle pour le joueur courant (`joueurCourantId`).
- Un bandeau de statut (`statusMessage`) informant le joueur de l'état actuel ou des derniers événements.

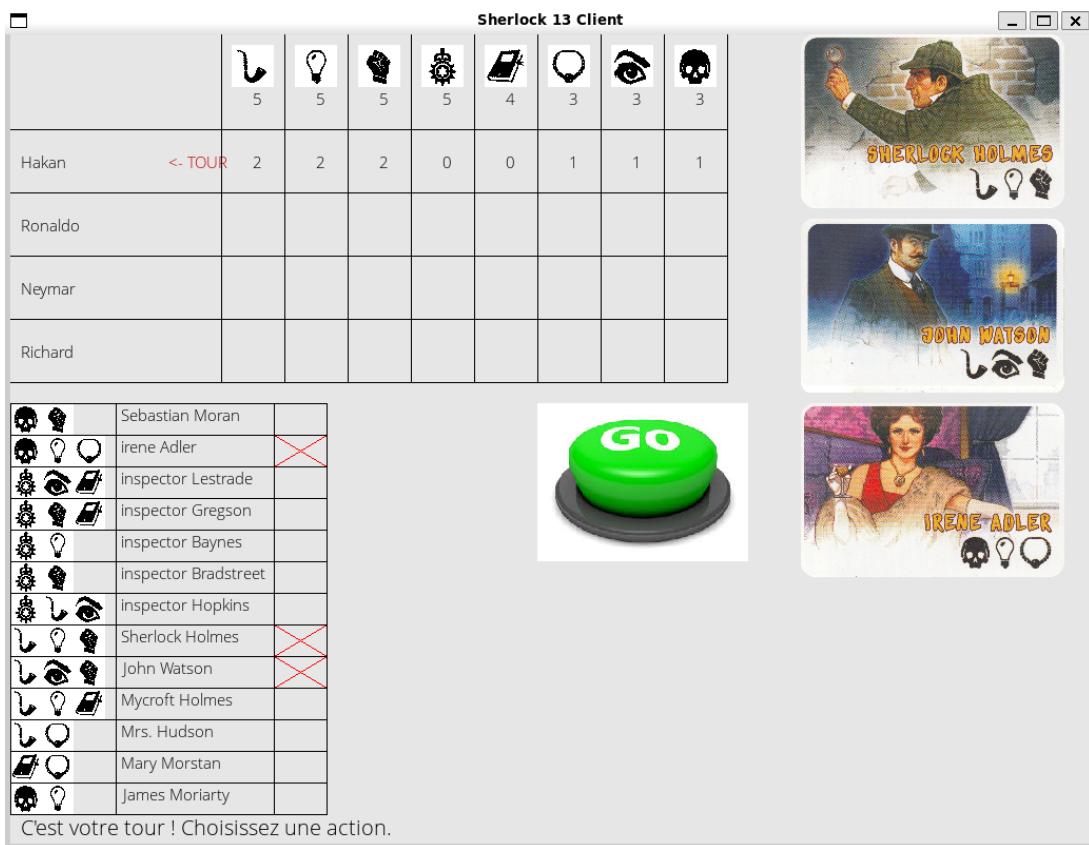


FIGURE 3.2 – Aperçu de l'interface graphique du client Sherlock 13.

Chapitre 4

Protocole de Communication

Les échanges d'informations entre le serveur et les clients reposent sur un protocole tex-tuel simple via TCP. Chaque message est une ligne de texte terminée par un retour à la ligne (\n), commençant par une lettre majuscule identifiant le type de message, suivie des paramètres nécessaires séparés par des espaces.

4.1 Synthèse du Protocole

Message	Sens	Format et Sémantique
C	C → S	C <ip> <port> <name>
I	S → C _i	I <id>
L	S → C*	L <n0> <n1> <n2> <n3>
D	S → C _i	D <c0> <c1> <c2>
V	S → C _i	V <p> <o> <v>
M	S → C*	M <id>
G	C → S	G <id> <suspectId>
O	C → S	O <id> <objectId>
S	C → S	S <id> <cibleId> <objectId>
F	S → C _i	F <couvableId>
W	S → C*	W <winnerId> <couvableName> <couvableId>

C = Client, S = Serveur, C_i = un client, C* = tous les clients ; paramètres entre <> sont numériques (ID, port, indices) ou textuels (IP, nom).

4.2 Exemples d'Échanges Typiques

4.2.1 Connexion et Démarrage

1. Client 1 envoie : C 192.168.1.10 12346 Joueur1
2. Serveur répond à Client 1 : I O
3. Serveur broadcast : L Joueur1 - - -
4. ... (idem pour Joueur2, Joueur3, Joueur4) ...
5. Serveur broadcast : L Joueur1 Joueur2 Joueur3 Joueur4
6. Serveur → C0 : D 7 1 10 et 8× V 0 j v
7. Serveur broadcast : M O

4.2.2 Enquête Objet

1. C0 → S : 0 0 0
2. S → C0 : v 2 0 100 et v 3 0 100
3. S broadcast : m 1

4.2.3 Enquête Spécifique

1. C1 → S : s 1 3 7
2. S → C1 : v 3 7 1
3. S broadcast : m 2

4.2.4 Accusation Correcte

1. C2 → S : g 2 12
2. S broadcast : w 2 James Moriarty 12

4.2.5 Accusation Incorrecte

1. C3 → S : g 3 7
2. S → C3 : f 12
3. S broadcast : m 0

Chapitre 5

Processus de Développement et Complétion du Code

Le cœur de ce projet n'était pas de développer l'application de zéro, mais de **compléter** une base de code C existante pour le serveur (`server.c`) et le client (`sh13.c`). Le travail a donc consisté à analyser le code fourni, identifier les sections manquantes (marquées par `// RAJOUTER DU CODE ICI`), et implémenter la logique nécessaire en respectant l'architecture et le protocole préexistants.

5.1 Analyse Préalable

Avant d'écrire du code, une phase d'analyse a été essentielle pour comprendre :

- L'architecture réseau imposée (serveur mono-thread acceptant des connexions éphémères, client bi-thread avec un thread écouteur passif).
- Le format des messages déjà partiellement gérés ou attendus.
- Les structures de données globales (`deck`, `tableCartes`, `tcpClients` côté serveur ; `gId`, `gNames`, `b`, `tableCartes`, `guiltGuess` côté client) et leur rôle.
- Le mécanisme de synchronisation client (`synchro`).
- Les fonctions utilitaires fournies (`error`, `sendMessageToClient`, `broadcastMessage`, `findClientByName` côté serveur ; `sendMessageToServer`, `fn_serveur_tcp` côté client).

5.2 Complétion du Serveur (`server.c`)

1. **Initialisation** (dans `if (nbClients == 4)`) :
 - Ajout des boucles `for` pour envoyer D et les 8 V à chaque joueur, puis broadcast de M 0 et passage à `fsmServer=1`.
2. **Gestion de l'Accusation** (case '`G`') :
 - Vérification du joueur courant et actif, comparaison avec `coupable`, envoi de W ou F, mise à jour de l'état des joueurs, et passage au prochain tour ou fin.
3. **Gestion de l'Enquête Objet** (case '`O`') :
 - Boucle sur les autres joueurs actifs, envoi de V i `objectId` 100 à l'enquêteur, puis broadcast M.
4. **Gestion de l'Enquête Spécifique** (case '`S`') :
 - Vérification et envoi de V `targetPlayerId` `objectId` `count`, puis broadcast M.
5. **getNextActivePlayer** :
 - Parcours circulaire pour trouver le prochain joueur actif, retourne -1 si aucun.

5.3 Robustesse et Nettoyage

- **Sécurité des Chaînes** : `snprintf`, `strncpy`.
- **Validation des Indices** : Tests avant accès aux tableaux.
- **Gestion des Erreurs** : Vérification des retours systèmes, `perror`, `fprintf(stderr, ...)`.
- **Nettoyage des Ressources** : Fermeture des sockets, `SDL_Quit`, `TTF_Quit`, `IMG_Quit`.
- **Arrêt Propre du Thread Client** : Connexion locale pour débloquer `accept`, puis `pthread_join`.

Chapitre 6

Conclusion

Ce projet, centré sur la complétion d'une base de code C existante, a permis de mettre en pratique de manière concrète et approfondie plusieurs concepts clés de la programmation système et réseau sous environnement Unix/Linux. La transformation d'un squelette applicatif en un jeu réseau fonctionnel "Sherlock 13" pour quatre joueurs a nécessité la maîtrise de :

- La programmation de sockets TCP/IP, en particulier dans un modèle de communication où le serveur initie également des connexions vers les clients.
- La gestion de processus multiples communiquant via le réseau.
- L'utilisation de threads côté client pour séparer les tâches d'interface graphique et de communication réseau, améliorant la réactivité.
- La mise en œuvre d'un mécanisme de synchronisation simple mais efficace entre threads via un drapeau partagé.
- La gestion de l'état distribué du jeu et sa mise à jour cohérente via un protocole de communication défini.
- Les bonnes pratiques de programmation C pour la robustesse (gestion des erreurs, sécurité des buffers) et le nettoyage des ressources.

Remerciements

Je tiens à remercier chaleureusement mes professeurs **François Pecheux** et **Thibault Hilaire** pour la qualité de leurs enseignements, leur disponibilité et leurs conseils avisés tout au long de la réalisation de ce projet dans le cadre du module OS USER.

Annexe A

Structure du Dépôt

```
1 .
2     assets/           # Images PNG + police TTF sans.ttf
3         connectbutton.png
4         gobutton.png
5         SH13_0.png
6         ... (SH13_1.png     SH13_12.png)
7         SH13_ampoule_120x120.png
8         SH13_carnet_120x120.png
9         SH13_collier_120x120.png
10        SH13_couronne_120x120.png
11        SH13_crane_120x120.png
12        SH13_oeil_120x120.png
13        SH13_pipe_120x120.png
14        SH13_poing_120x120.png
15        sans.ttf
16     server.c          # Code source du serveur
17     sh13.c            # Code source du client graphique SDL2
18     Makefile          # Fichier pour la compilation (all / clean)
19     README.md         # Documentation du projet
20     rapport/          # Rapport du projet
21     Projet_OSUSER_HakanPEKUR.pdf
```

Annexe B

Compilation et Exécution

B.1 Prérequis

```
1 sudo apt update
2 sudo apt install build-essential libsdl2-dev libsdl2-image-dev libsdl2-ttf-dev
```

B.2 Compilation

```
1 make      # Compile server et sh13
2 make clean # Nettoyage
```

B.3 Exécution

B.3.1 Serveur

```
1 ./server 12345
```

B.3.2 Clients

```
1 ./sh13 127.0.0.1 12345 127.0.0.1 12346 Joueur1
2 ./sh13 127.0.0.1 12345 127.0.0.1 12347 Joueur2
3 ./sh13 127.0.0.1 12345 127.0.0.1 12348 Joueur3
4 ./sh13 127.0.0.1 12345 127.0.0.1 12349 Joueur4
```