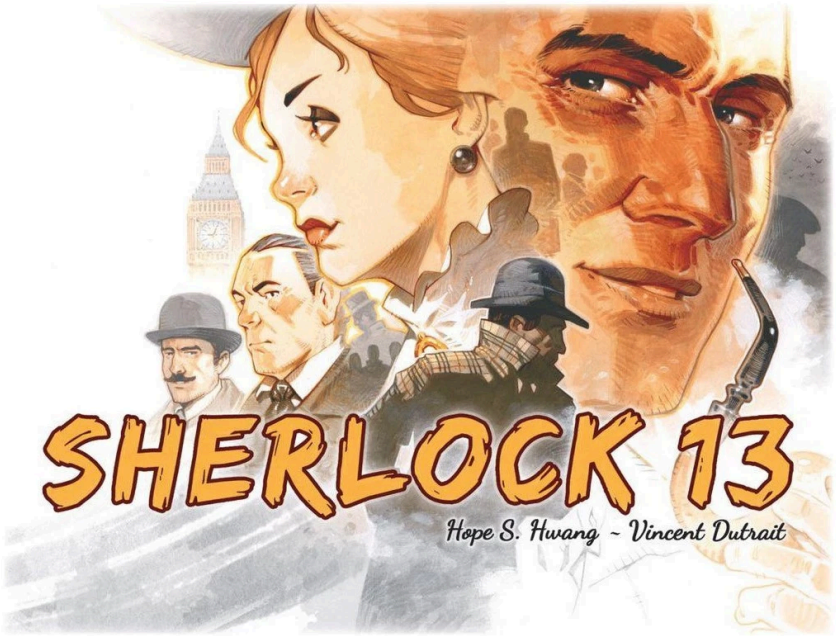
Lam Victor
EI4
TPD

# Rapport réalisation du jeu Sherlock13




Année : 2024-2025
Module : OS User
Encadrant: F. Pecheux

Table des matières :



# Introduction

Ce rapport présente l'implémentation du jeu Sherlock 13 en version réseau et graphique. L'objectif est de permettre à plusieurs joueurs de participer à une partie à distance, chacun via un client doté d'une interface graphique, et coordonnée par un serveur central. Nous décrirons dans un premier temps les règles du jeu, puis l'architecture logicielle mise en place, en détaillant le fonctionnement du serveur et du client, ainsi que les mécanismes de communication et de synchronisation entre les différents composants du programme.

# I. Sherlock 13

## A. Règle du jeu

Dans Sherlock 13, il existe 13 cartes correspondant chacune à un personnage avec 2 à 3 symboles associés. En début de partie, chacun des 4 joueurs reçoit 3 cartes, la dernière devient le criminel que les joueurs devront démasquer.
Puis à tour de rôle, un joueur pourra effectuer une des 3 actions possibles :

- Demander à tous les joueurs s'ils ont un symbole, ils répondent par oui ou par non
- Demander à un joueur combien de fois il possède un symbole
- Accuser le criminel

De plus, les informations demandées sont partagées entre tous les joueurs et que si un joueur se trompe lorsqu'il fait une accusation il ne peut plus jouer.

Enfin, le joueur démasquant le criminel gagne la partie.

## B. Comment lancer le jeu ?

Tout d'abord, il faut lancer le serveur avec

./server numPort

Pour se connecter avec le client

./sh13 ipAdressServer numPortServer ipAdressClient numPortClient

puis sur la fenêtre graphique cliquez sur "connect"

Lorsque c'est le tour d'un joueur, un bouton "Go", lui permettant de confirmer une action, apparaît dans son interface. Il peut alors effectuer l'une des trois actions suivantes :

- Demander à tous les joueurs s'ils possèdent un symbole, en cliquant simplement sur le symbole voulu.
- Demander à un joueur combien de fois il possède un symbole, en sélectionnant à la fois le symbole et le joueur ciblé.
- Accuser un suspect, en cliquant sur l'un des noms des personnages affichés en bas à gauche de la fenêtre.

# II. Architecture du programme

## A. Structure du programme

Le programme sera structuré autour d'un serveur central qui héberge l'ensemble des données du jeu : les cartes, les symboles et le suivi des tours de jeu. C'est sur ce serveur que se déroule toute la partie.

Chaque joueur sera représenté par un client, qui se connectera au serveur central pour envoyer ses actions (comme poser une question ou faire une accusation) et gérer l'interface graphique. En parallèle, chaque joueur disposera également d'un serveur local capable de recevoir les mises à jour du jeu envoyées par le serveur central (par exemple : réponse à une question, changement de tour, etc.).

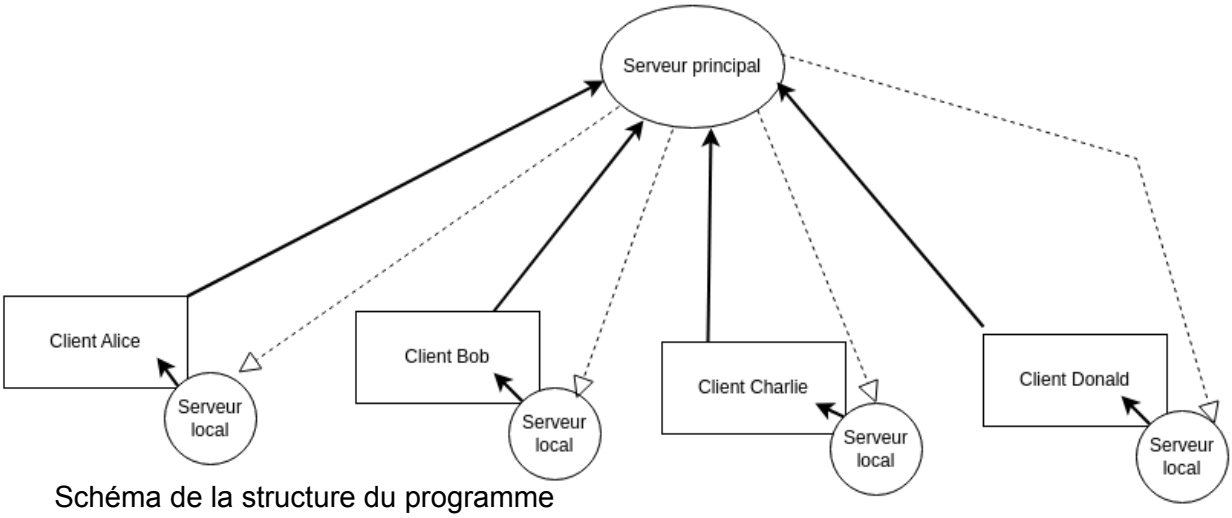


Schéma de la structure du programme

## B. Structures de donnée

Les cartes du jeu sont stockées dans un tableau. Après un mélange, les cartes sont attribuées aux joueurs : les trois premières vont au premier joueur, les trois suivantes au second, etc. La 13ème carte, qui reste non distribuée, correspond au criminel à découvrir.

Parallèlement, une matrice est utilisée pour représenter la répartition des symboles entre les joueurs : chaque ligne correspond à un joueur, chaque colonne à un symbole, et chaque case contient le nombre d'occurrences de ce symbole pour ce joueur.

### C. Diagramme de séquence

#### Phase de connexion

Une fois lancé, le serveur central attendra que les joueurs se connectent avant de démarrer la partie. Pour rejoindre une partie, un client devra envoyer un message au serveur sous la forme :
"C adresseIPClient portClient nomJoueur"

En réponse, le serveur attribue un identifiant unique au joueur et lui enverra :
"I idJoueur"

Ensuite, il diffusera à tous les joueurs connectés la liste complète des participants via un message de type :
"L joueur1 joueur2 …"

Lorsque 4 joueurs ont rejoint la partie, le jeu peut commencer.

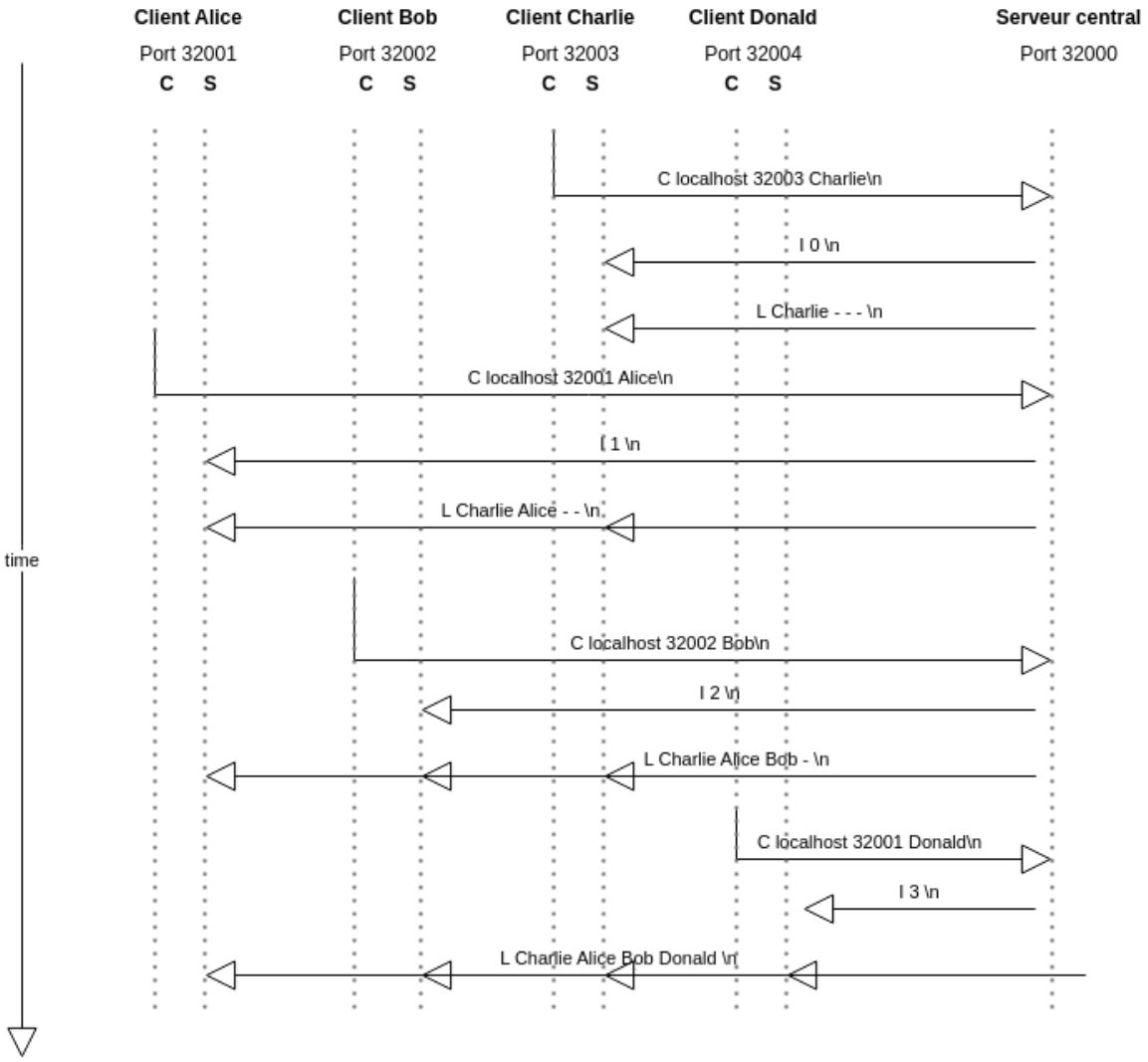



Diagramme UML de séquence : Phase de connexion

Phase de jeu

Une fois la partie lancée, le serveur central envoie à chaque joueur ses cartes personnelles ainsi que le nombre de symboles qu'il possède, sous la forme suivante : "D carte1 carte2 carte3 nbSymbole1 nbSymbole2 …"

Ensuite, la boucle de jeu commence.

Annonce du tour de jeu :

Le serveur central informe tous les joueurs de l'identité du joueur dont c'est le tour, en respectant l'ordre de connexion initial :
"M idJoueur"

Le joueur actif peut alors choisir l'une des trois actions suivantes :

1. **Demander à tous les joueurs s'ils possèdent un symbole donné**

Le joueur envoie :
"O idJoueur idSymbole"

Le serveur répond pour chaque joueur par :
"V val idJoueur idSymbole", où :

val = 0 : le joueur ne possède pas le symbole

val = 42 : le joueur possède le symbole

2. **Demander à un joueur combien de fois il possède un symbole**

Le joueur envoie :
"S idJoueur idJoueurCible idSymbole"

Le serveur répond par :
"V val idJoueurCible idSymbole", où val est le nombre exact de fois que le joueur ciblé possède ce symbole.

3. **Accuser un suspect d'être le criminel**

Le joueur envoie :
"G idJoueur idCarte"

Si l'accusation est juste, le serveur met fin à la partie avec :
"E …"

Si l'accusation est fausse, le joueur est éliminé mais la partie continue tant qu'il reste au moins un joueur actif.

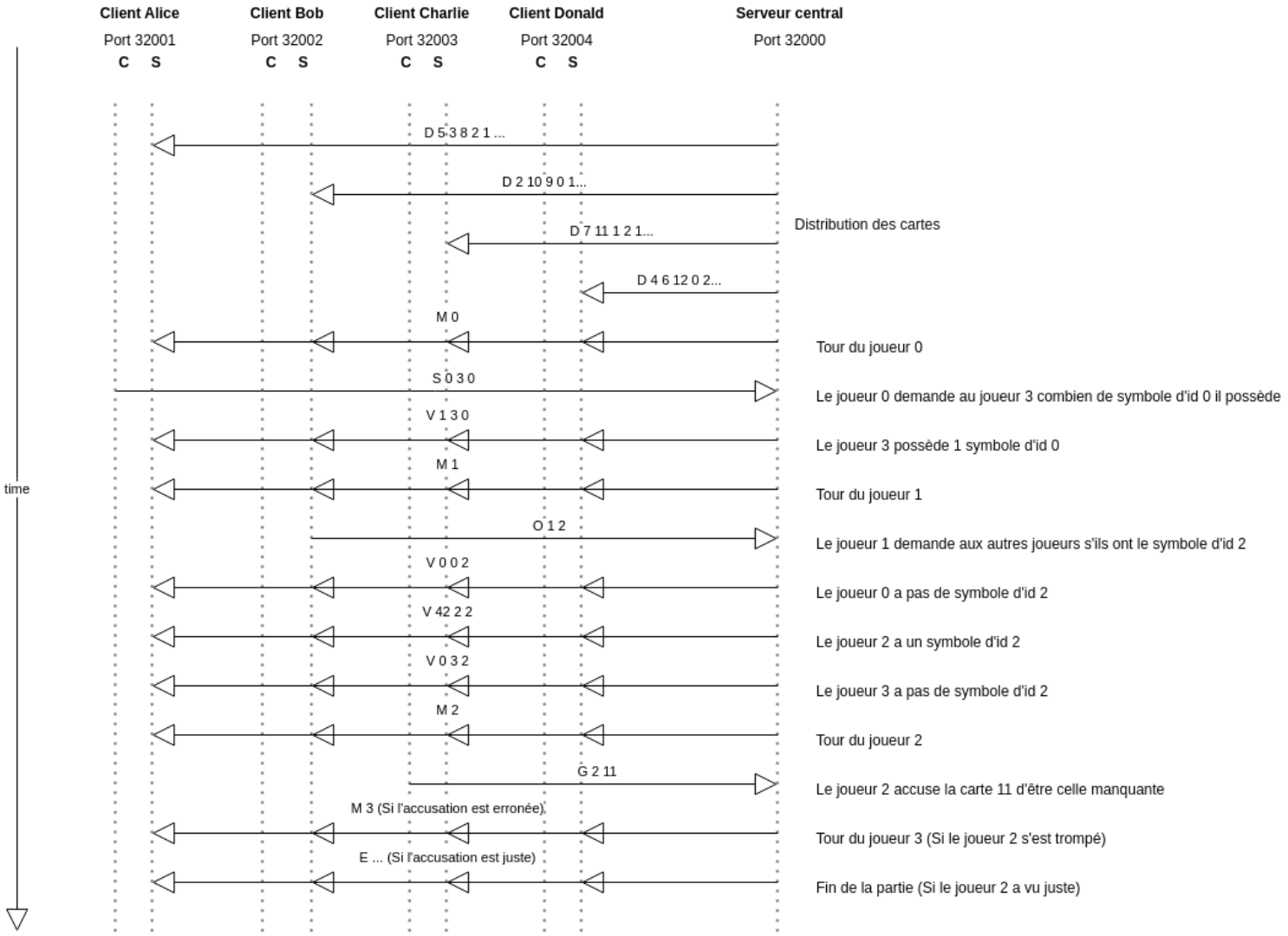



Diagramme UML de la phase de jeu

# III. Partie serveur - server.c

Le serveur central est chargé de gérer l'ensemble des données de la partie : les cartes, les joueurs connectés et le nombre de symboles détenus par chacun. Au lancement de la partie, il mélange les cartes puis les distribue aux joueurs. L'implémentation repose sur l'utilisation de sockets TCP, assurant une communication fiable avec les clients/joueurs pour recevoir leurs actions. Un second socket est également utilisé pour

transmettre des messages vers le serveur local d'un client via la fonction sendMessageToClient().

Le fonctionnement du serveur est divisé en deux phases principales :

Phase de connexion, durant laquelle le serveur attend que tous les joueurs soient connectés.

Phase de jeu, où il traite les actions des joueurs à tour de rôle.

Parmi les parties à compléter dans le code, il y avait notamment la distribution des cartes : cela consistait à remplir un buffer reply avec les cartes et les symboles associés à chaque joueur, puis à l'envoyer via sendMessageToClient().

Ensuite, l'implémentation des actions possibles en jeu a été réalisée. Pour chaque action reçue, le serveur traite la demande : soit il révèle les informations requises (comme les symboles possédés), soit il vérifie la validité d'une accusation. Une fois l'action terminée, il passe la main au joueur suivant.
Une variable playing a également été ajoutée à la structure de chaque joueur, permettant de vérifier s'il est toujours en jeu lors du changement de tour.

# IV. Partie client - sh13.c

Le client gère l'interface graphique et envoie les actions du joueur au serveur central. Pour recevoir les messages du serveur en parallèle, un thread secondaire est lancé : il crée un serveur local dédié à la réception des messages. Contrairement à un fork, l'utilisation d'un thread permet de partager des variables avec le thread principal (celui de l'interface graphique), notamment la variable synchro, utilisée pour la synchronisation entre les deux.

Cette variable est essentielle pour assurer la synchronisation entre l'interface graphique et le thread réseau. Comme son nom l'indique, elle permet de coordonner l'échange d'informations entre les deux threads. Concrètement, le thread réseau, via la fonction fn_serveur_tcp, reçoit un message par socket, puis met la variable synchro à 1 pour signaler qu'un message est prêt à être traité. Il se bloque ensuite dans une boucle while, en attendant que le thread principal (graphique) remette synchro à 0.

De son côté, le thread graphique vérifie périodiquement la valeur de synchro. Lorsqu'elle vaut 1, il interprète le message reçu selon son format, met à jour l'état du jeu, puis remet synchro à 0.
Ce mécanisme agit comme une interruption, garantissant que le thread réseau ne lira pas un nouveau message tant que le précédent n'a pas été traité.

De plus, la variable synchro est déclarée avec le mot-clé volatile, ce qui indique au compilateur que sa valeur peut être modifiée à tout moment, notamment par les deux threads. Cela empêche toute mise en cache et garantit un accès direct en mémoire.

Concernant la partie du code à compléter, elle portait sur l'envoi des actions du joueur vers le serveur central ainsi que sur la réception des messages en retour. Cela consistait à remplir un buffer avec le message à envoyer, ou à lire les données reçues dans un buffer pour les interpréter ensuite. De plus un cas dans le switch case gérant la mise à jour de l'état du jeu a été ajouté pour terminer le processus quand la partie est terminée.

## Conclusion

L'implémentation de Sherlock 13 a permis de mettre en œuvre des concepts clés du développement réseau et de la programmation concurrente, notamment l'utilisation de sockets TCP et de threads pour assurer une communication fluide entre le serveur central et les clients. La synchronisation entre les threads via la variable synchro, l'utilisation d'une interface graphique interactive, ainsi que la gestion structurée des données du jeu, ont constitué les principaux défis relevés dans ce projet. Cette réalisation offre ainsi une version fonctionnelle, interactive et distribuée du jeu, respectant à la fois ses règles originales et les contraintes techniques d'une architecture client-serveur.