RAKOTO
Elsa
EI4

RAPPORT PROJET OS USER
SHERLOCK 13

2025

RAKOTO Elsa
EI4

SOMMAIRE

RAKOTO Elsa
EI4

Ce projet avait pour but de coder le jeu Sherlock13, un jeu de cartes destiné à 4 joueurs dont le gagnant est celui qui parvient à identifier laquelle des 13 cartes initiales n'a pas été distribuée.

L'architecture du projet est fondée sur une implémentation client/serveur où chaque client représente un joueur et le serveur à pour rôle celui de maître du jeu, omniscient.

Certaines fonctions utilisées dans ces liaisons de communication étant bloquantes, il a fallu trouver des solutions pour éviter un freeze de l'interface graphique associée à chaque client.

Nous verrons quel cheminement à été mis en place pour réaliser ce projet et comment les outils de réseaux et proches du système ont permis de pallier ce problème.

1) Étapes abordées pour compléter le code C

A) Analyse initiale du code fourni

Ma première tâche a été de comprendre la base de code fournie.
J'ai d'abord analysé les structures et variables déclarées dans les deux fichiers pour comprendre leur rôle dans le jeu.

Côté serveur :

La structure principale tcpClients stocke les informations de connexion des 4 joueurs (adresse IP, port et nom). Plusieurs variables globales importantes gèrent l'état du jeu :

- Le nombre de clients actuellement connectés
- L'état de la machine à états du serveur (attente ou jeu en cours)
- Le jeu de cartes (13 cartes numérotées de 0 à 12)
- Une matrice représentant les caractéristiques des cartes de chaque joueur
- Un tableau des noms des personnages
- L'identifiant du joueur courant

Côté client :

Les variables utiles à la compréhension du programme sont les suivantes :

- Un identifiant de thread pour la communication réseau
- Un mutex pour la synchronisation
- Un buffer pour les messages reçus

- Des variables pour les informations de connexion (IP, port)
- Des variables pour l'état du jeu (cartes du joueur, sélections dans l'interface)
- Une matrice représentant la connaissance du joueur sur les cartes de tous les joueurs

Les messages du jeu régissent :

- La connexion des joueurs
- La distribution des cartes
- Les actions de jeu (accusation, observation, question)
- La transmission d'informations sur les cartes

B) Complétion du code côté serveur

Le serveur s'occupe dans un premier temps de l'enregistrement des informations du client dans la structure tcpClients. Il attribue un identifiant unique à ce client via le message 'I'.

Gestion des actions de jeu

Il a fallu implémenter les trois types d'actions que les joueurs peuvent effectuer dont je vais expliquer l'algorithmique utilisée:

Accusation : Quand un joueur accuse un personnage, on vérifie l'accusation en comparant avec la carte du coupable (deck[12]). Si l'accusation est correcte, le serveur annonce le gagnant à tous. Sinon, on passe au joueur suivant.

Observation : Quand un joueur demande qui n'a pas un certain type de symbole, on parcourt la matrice tableCartes pour trouver quels joueurs n'ont pas ce symbole, puis diffuser cette information.

Question spécifique : Quand un joueur interroge un autre joueur sur un symbole précis, on récupère l'information dans tableCartes et on la diffuse à tous les joueurs.

Pour chaque action, j'ai également implémenté le passage au joueur suivant en incrémentant joueurCourant et en diffusant cette information.

Initialisation du générateur aléatoire :

J'ai ajouté l'initialisation du générateur de nombres aléatoires avec la fonction time() pour garantir une distribution différente des cartes à chaque partie.

Gestion de fin de partie :

J'ai implémenté un message de fin de partie plus informatif qui annonce le gagnant et le personnage correctement accusé.

C) Complétion du code côté client

Le client présentait des défis différents, notamment liés à la gestion des threads et des messages reçus.

Traitement des messages reçus

J'ai complété le switch case qui traite les différents types de messages reçus du serveur. Les voici :

Message 'I' : Stock de l'identifiant du joueur dans gId.

Message 'L' : Mise à jour du tableau des noms des joueurs gNames.

Message 'D' : Stock des trois cartes reçues dans le tableau b, puis initialisation de tableCartes avec la fonction initCards().

Message 'M' : Mise à jour du joueur courant, activation/désactivation du bouton "Go" si c'est au tour du joueur ou non.

Message 'V' : Mise à jour de tableCartes avec les informations reçues sur les cartes des joueurs.

Implémentation de initCards()

J'ai également implémenté la fonction initCards() qui traduit les cartes de personnages en symboles. Cette fonction parcourt les trois cartes du joueur et, pour chaque carte, incrémente les compteurs des symboles correspondants dans tableCartes.

2) Utilisation des sockets

Le code fourni utilisait des sockets TCP pour la communication réseau.

Architecture client-serveur

Le jeu utilise une architecture client-serveur où le serveur crée un socket d'écoute sur un port spécifié et les clients se connectent à ce socket pour envoyer des commandes.

Le serveur traite les commandes et répond aux clients

Le serveur doit également pouvoir envoyer des messages aux clients à tout moment. Pour cela, chaque client crée son propre socket d'écoute. Le client informe le serveur de son adresse IP et de son port lors de la connexion. Le serveur utilise ces informations pour envoyer des messages aux clients.

Les fonctions utilisées pour la communication sont :

- sendMessageToClient() qui envoie un message à un client spécifique
- broadcastMessage() qui envoie un message à tous les clients
- sendMessageToServer() qui permet au client d'envoyer un message au serveur

Ces fonctions créent une connexion TCP temporaire pour chaque message.

3) Utilisation des threads

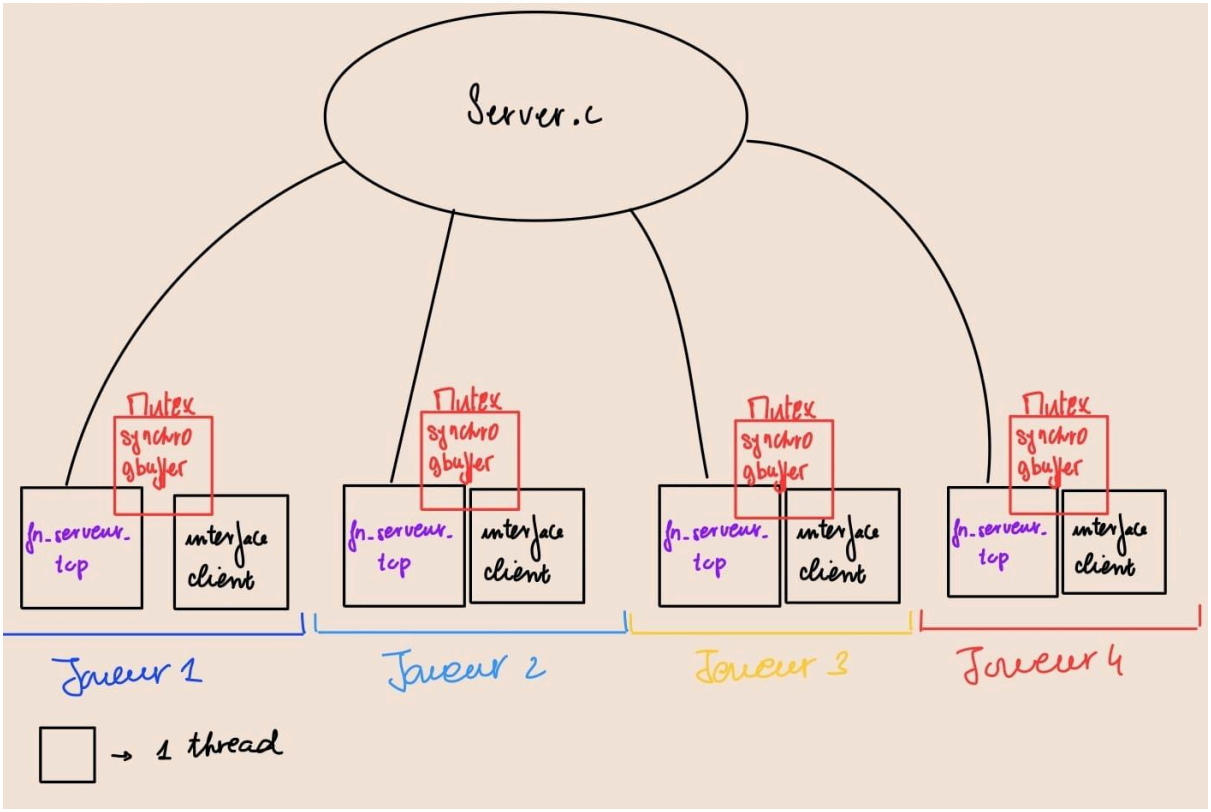

Schéma de l'architecture du projet SH13

Le client utilise une architecture à deux threads car il doit à la fois gérer l'interface et ses évènements en continu, mais aussi les messages reçus de la part du serveur.

Thread principal

Le thread principal gère l'interface graphique SDL. Il traite donc les événements utilisateurs, à savoir clic ou sélection. Il gère l'envoi de messages au serveur et traite d'un autre côté les messages reçus par le thread serveur TCP.

Thread serveur TCP

Un thread secondaire est créé, il exécute la fonction fn_serveur_tcp(). Il permet donc d'écouter en permanence les connexions entrantes du serveur, de stocker les messages reçus dans un buffer partagé (gbuffer). Il s'assure également de signaler au thread principal qu'un message est disponible (synchro = 1). Il attend de la part du thread principal qu'il traite le message lorsque Synchro=0.

J'ai dans un premier temps modifié la boucle d'attente active de la fonction fn_seveur_tcp() . Elle vérifie maintenant l'état de la variable synchro et fait ensuite une pause de quelques microsecondes.

De plus, j'ai complété le code qui traite les messages reçus par le thread serveur TCP, en veillant à respecter la synchronisation entre les threads.

4) Utilisation des mutex

A) Rôle des mutex dans le code

Le mutex est utilisé pour protéger l'accès concurrent aux ressources partagées entre les deux threads, en l'occurrence :

- Le buffer gbuffer qui contient les messages reçus
- La variable synchro qui indique si un message est disponible de la part du serveur

Sans le mutex, on aurait un risque de corruption des données si les deux threads souhaitaient accéder simultanément à ces ressources : l'un en lecture pendant que l'autre la modifie par exemple.

B) Localisation d'utilisation du mutex

J'ai utilisé le mutex à deux endroits :

Le thread serveur TCP verrouille le mutex avant d'écrire dans gbuffer et modifier synchro.

- Le thread principal verrouille le mutex avant de lire gbuffer et de traiter le message, puis il réinitialise synchro
- Le thread tcp verrouille le mutex lorsqu'il s'apprête à recevoir un message du serveur.

Conclusion

Ce projet m'a permis de mettre en application les notions vues en cours. Au niveau de l'architecture d'un système client serveur notamment mais aussi et surtout sur notions orientées plutôt OS user.

Ayant pratiqué le C depuis des années, un langage bas niveau, c'est une des premières fois cette année que j'ai l'occasion de toucher à des paramètres aussi proches du système à savoir l'utilisation d'un mutex et de threads.

En plus du côté ludique et très concret du projet, celui-ci était particulièrement bien conçu pour introduire des notions difficiles à comprendre aux premiers abords. Le fait que la majorité du code qui ne correspondait pas aux nouvelles notions apportées par la matière à été un réel avantage, car cela a permis de se concentrer sur l'acquisition des concepts OS user et ne pas se retrouver submergé par des artifices.

En fin de compte, j'ai pu rendre le jeu Sherlock 13 fonctionnel et robuste.