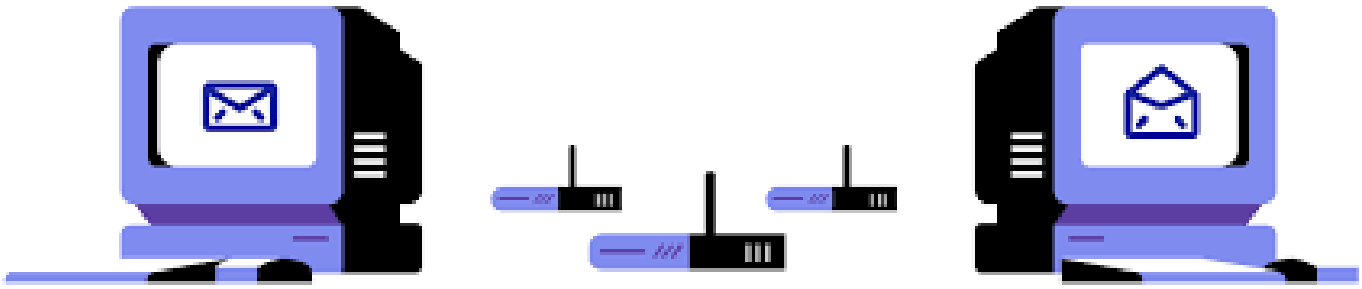
POLYTECH SORBONNE

SORBONNE UNIVERSITÉ

Projet : OS USER



Etudiant : Michel Vandar

Groupe : TP B

Année : 2024/2025

Table des matières

1 - Introduction

Dans ce projet, on devait compléter un jeu inspiré de "Sherlock 13" en langage C, avec une interface graphique et un fonctionnement en réseau. Le but était de permettre à plusieurs joueurs de se connecter à un serveur, recevoir leurs cartes, et jouer à tour de rôle selon les règles du jeu. On a utilisé les bibliothèques SDL2 pour l'interface et les sockets pour la communication. Ce rapport explique comment on s'y est pris, comment fonctionne le jeu, et les concepts vus en TP qu'on a utilisés.

2 - Methode de développement

On a commencé par relire attentivement les fichiers donnés (sh13.c) pour comprendre quelles fonctions étaient déjà prêtes et où il fallait rajouter du code. Ensuite, on a décidé de suivre l'ordre logique du jeu : d'abord faire marcher la connexion au serveur, ensuite gérer les cartes, puis les échanges de messages. On a fait les tests étape par étape, en ajoutant beaucoup de printf pour suivre l'exécution et repérer les bugs. Le travail a été divisé entre l'interface (chargement des images, affichage) et la communication réseau (messages entre client et serveur). Le serveur qu'on a utilisé était une version qu'on avait construite et testée en parallèle.

Quand on a commencé, le fichier sh13.c pour le client avait des parties vides mais bien balisées avec des commentaires comme // RAJOUTER DU CODE ICI, ce qui aidait à voir où coder. De même, dans le fichier server.c, le professeur avait laissé une partie de la logique de jeu à compléter.

Voici comment on a procédé et où on a dû intervenir dans le code :

Côté client (sh13.c)

On a rajouté le code dans la boucle principale, dans le switch (gbuffer[0]), pour traiter les messages reçus du serveur :

- **Message I** : récupérer l'ID du joueur avec sscanf(gbuffer, "I %d", &gId);
- **Message L** : récupérer les noms des 4 joueurs avec sscanf(gbuffer, "L %s %s %s %s", ...)
- **Message D** : stocker les 3 cartes du joueur avec sscanf(gbuffer, "D %d %d %d", ...)
- **Message M** : mettre à jour goEnabled selon si c'est à notre tour
- **Message V** : mettre à jour le tableau tableCartes localement

Tous ces traitements étaient à compléter pour que le jeu fonctionne correctement.

Côté serveur (server.c)

Le serveur devait gérer deux états : avant et après le lancement de la partie (variable fsmServer).

- **Quand fsmServer == 0** :
    - On devait **remplir les parties "RAJOUTER DU CODE ICI"** pour envoyer les cartes à chaque joueur :

```
sprintf(reply, "D %d %d %d", ...);
sendMessageToClient(...);
```

- Ensuite, on devait envoyer la **ligne correspondante de tableCartes** :

```
for (int j = 0; j < 8; j++) {
  sprintf(reply, "V %d %d", j, tableCartes[i][j]);
  sendMessageToClient(...);
}
```

- Et on devait **envoyer le joueur courant** à tout le monde :

```
sprintf(reply, "M %d", joueurCourant);
broadcastMessage(reply);
```

3 - Fonctionnement du programme

Le programme commence par lancer une fenêtre SDL et initialise la police, les textures (cartes, objets, boutons). Ensuite, il crée un thread serveur local qui écoute sur le port du client. Lorsqu'on clique sur "connect", le client envoie un message au serveur principal pour s'enregistrer. Le serveur répond en envoyant un ID, les cartes du joueur, et une partie du tableau du jeu. Chaque message du serveur (type I, D, L, M, V) est interprété dans le programme et mis à jour dans les bonnes variables. L'interface permet ensuite d'interagir avec les objets, de faire des accusations, et d'envoyer les actions au serveur principal

4 - Utilisation des concepts vus en TP

Dans ce projet, on a mis en pratique plusieurs notions vues pendant les TPs du module. Voici les plus importantes avec des exemples concrets tirés de notre code :

- **Sockets (communication réseau)**

On a utilisé les **sockets TCP** pour établir la communication entre les clients (joueurs) et le serveur (le maître du jeu). C'est la base de tout le fonctionnement du jeu.

- **Côté client**, la fonction sendMessageToServer() permet d'envoyer un message formaté au serveur via une socket :

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
connect(sockfd, ...);
write(sockfd, sendbuffer, strlen(sendbuffer));
```

- **Côté serveur**, on écoute en permanence les connexions entrantes :

sockfd = socket(AF_INET, SOCK_STREAM, 0);

bind(sockfd, ...);

listen(sockfd, 5);

newsockfd = accept(sockfd, ...);

read(newsockfd, buffer, 255);

On a aussi utilisé les sockets pour faire des **envois personnalisés** à un joueur (sendMessageToClient) ou des **diffusions globales** (broadcastMessage) à tous les clients.

- **Threads (multi-tâche)**

Dans le fichier sh13.c (client), on a utilisé un **thread dédié** pour recevoir les messages réseau sans bloquer l'interface graphique.

- Le thread fn_serveur_tcp() tourne en boucle, attend les messages du serveur et les stocke dans un buffer global (gbuffer) :

pthread_create(&thread_serveur_tcp_id, NULL, fn_serveur_tcp, NULL);

Ce thread nous permet de séparer la logique réseau du reste du programme (interface SDL + événements utilisateur).

- **Mutex**

On a utilisé un **mutex** pour protéger la variable gbuffer qui est partagée entre deux threads :

- Le **thread TCP** (fn_serveur_tcp) qui reçoit les messages du serveur et les écrit dans gbuffer
- La **boucle principale SDL** (dans main) qui lit le contenu de gbuffer pour le traiter et mettre à jour l'affichage

On a utilisé pthread_mutex_lock() avant d'écrire dans gbuffer, et pthread_mutex_unlock() juste après, pour éviter qu'un autre thread ne modifie ou lise la variable en même temps. Ça permet d'éviter les conflits d'accès et les bugs liés à des données corrompues.

- Exemple dans notre code :

pthread_mutex_lock(&mutex);

strcpy(gbuffer, "message reçu");

pthread_mutex_unlock(&mutex);

Ce mécanisme de verrouillage assure que les accès sont faits un par un, ce qui est indispensable quand plusieurs threads communiquent entre eux.

- **Traitement de messages (protocole simple)**

Le jeu repose sur des **messages textes codés avec une lettre** pour indiquer l'action :

- C ip port name → connexion d'un joueur
- I id → envoi de l'identifiant du joueur
- L n1 n2 n3 n4 → liste des joueurs
- D c1 c2 c3 → cartes du joueur
- V col val → valeurs du tableau tableCartes
- M x → joueur courant

Ces messages sont traités par des blocs switch(gbuffer[0]) dans sh13.c.

- **État du jeu (fsmServer)**

On a aussi utilisé une **machine à états** très simple côté serveur pour différencier deux phases :

- fsmServer == 0 : attente des connexions des joueurs
- fsmServer == 1 : partie en cours

Ça permet de séparer les comportements selon où on en est dans le déroulement du jeu.

- **Aléatoire et logique de jeu**

Dans le serveur, on a utilisé la fonction melanger_cartes() pour mélanger les cartes, avec :

srand(time(NULL));

Ensuite, les cartes sont distribuées aux joueurs avec des messages D + V, et la table tableCartes[][] est remplie automatiquement avec la fonction createTable() selon les symboles des personnages.

5 - Problèmes rencontrés et solutions

Le plus gros problème a été les **segmentation faults** au démarrage, souvent causées par des tableaux non initialisés ou des pointeurs mal utilisés. On a aussi eu des soucis de chargement d'images (chemins incorrects ou fichiers manquants). Pour résoudre tout ça, on a mis plein de printf de debug pour savoir où ça plantait. Ensuite, il y avait parfois des bugs sur les messages reçus : mauvaise interprétation, ou valeurs pas mises à jour. À chaque fois, on relisait le format du message côté serveur et on vérifiait bien le sscanf côté client

6 - Conclusion

Ce projet nous a vraiment aidés à comprendre comment fonctionne un vrai programme client-serveur en C. On a bien revu les threads, les sockets, et la gestion d'une interface graphique avec SDL. Même si on a eu des bugs, on a réussi à les corriger et à faire marcher un jeu fonctionnel en

réseau. Le code est simple mais complet, et on a pu tester plusieurs cas. C’était un bon exercice pratique pour appliquer les notions du module.