Projet du Dystème d'exploitation

Jeu Sherlock13

Ruxue Zeng 3700218

Mai 2020

Table des matières

1 Introduction **2**

2 Règle du jeu **2**

3 Outils techniques utilisés **2**
3.1 Interface graphique : SDL 2
3.2 Les sockets 3
3.3 Les Thread 3

4 Le serveur **3**
4.1 Structure de données 4
4.2 Initialisation du serveur 4
4.3 Connection des joueurs 5
4.4 Trois actions 6
4.4.1 C'est lui le coupable ! 6
4.4.2 Qui a cet objet ? 7
4.4.3 Combiens as-tu d'objets ? 8
4.5 Ré-initialisation avant le prochain tour 8

5 Le client **9**
5.1 Structure de données 9
5.2 Initialisation 9
5.3 Communication avec le serveur 9
5.4 Interface graphique 10

6 Interactions entre le serveur et les clients **11**

7 Améliorations possibles **12**

8 Conclusion **12**

1 Introduction

Dans le cadre de notre première année du cycle ingénieurs en Mathématiques appliqués et Informatique du Numérique à Polytech Sorbonne, nous avons eu un module d'enseignement : Système d'exploitation, encadré par Monsieur PECHEUX. A la fin de ce module, il nous a été demandé de coder un jeu en C, nommé **Sherlock13**, en mettant oeuvre une interface graphique et des échanges via le réseau.

L'interface graphique du jeu est déja construit par Monsieur PECHEUX, notre travail principal porte sur les aspects réseaux, la communication entre le serveur principal et ses 4 clients. Ainsi, nous devons programmer deux fichier : le fichier *serveur.c* pour le serveur, et le fichier *sh13.c* pour les clients. Comme notre jeu n'a pas besoin d'être rapide, donc nous utilisons le protocole TCP (Transmission Control Protocol) pour notre serveur, qui est fiable, connecté et "lent", et qui permet d'envoyer des données par le réseau.

2 Règle du jeu

Ce jeu consiste à reproduire Sherlock13 avec du code C, qui va permettre aux 4 joueurs de différents serveurs de jouer ensemble, en se connectant au serveur principal. Le jeu est composé de 13 cartes, qui représentent les 13 personnages du Sherlock Holmes. Chaque personnage est caractérisé par une illustration et une série de symboles. Chaque joueur va recevoir 3 cartes au début du jeu, la 13è carte qui reste est l'identité du coupable qu'on cherche. Le but de jeu est de trouver cette carte avant tous les autres. A chaque tour, le joueur peut choisir une action parmi les trois possibilités :

— choisir un symbole et demander qui en possède

— cibler un joueur et lui demander combiens il possède d'un symbole en particulier

— porter une accusation, s'il a raison, le jeu est fini, il a gagné, sinon, il a perdu, il ne peut plus jouer, mais il doit toujours répondre aux questions des autres.

3 Outils techniques utilisés

3.1 Interface graphique : SDL

Dans ce projet, nous n'avons pas besoin de s'en occuper la partie d'interface graphique, mais cela n'empêche pas une petite présentation. Pour construire notre interface graphique, nous avons utilisé la bibliothèque SDL, qui est une bibliothèque particulièrement utilisée pour créer des jeux en 2D. SDL est une

bibliothèque tierce, que nous devons l'installer en cas de besoin. Il est écrit en C, qui est libre, gratuite et multi-plateforme.

3.2 Les sockets

Pour établir une communication entre le client et le serveur, nous utilisons les sockets pour ouvrir une route temporaire (ou permanent) entre eux, qui va permettre d'échanger des informations.

Un socket est une interface logicielle avec les services du système d'exploitation. Il est défini dans la bibliothèque *sys/socket.h*. Pour créer un socket, nous utilisons la fonction *socket()*, qui prend en arguments la famille de la socket que l'on souhaite créer et son type. Pour travailler avec les protocoles TCP, tous les sockets du jeu sont de famille AFINET, qui correspond à l'adresse internet.

Après avoir créé notre socket, la fonction *bind()* va permettre de lier le socket à une structure standard nommée sockaddr. Puis, la fonction *listen()* va définir le nombre de connections pour le socket, et se mettre en attente d'une connexion entrante. Comme on souhaite un serveur et 4 clients, donc on fixe le nombre de connexion à 5. Ensuite, la fonction *connect()* permet de connecter le socket à un serveur.

Une fois le socket est ouvert, il est possible d'échanger des informations via le canal de communication, en faisant des écritures et des lectures. Le développeur doit choisir le format des messages échangés.

Enfin, lorsque la connection est terminée, il faut utiliser la fonction *close()* pour fermer le socket.

3.3 Les Thread

L'exécution du jeu nécessite lancement de plusieurs fonctions en même temps. Pour cela, nous allons utiliser la fonction *thread()*, une fonction qui s'exécute en parallèle du programme principal. Tous les threads partagent le même espace mémoire, ils ont donc tous accès aux variables. Pour éviter qu'il y a une confusion d'écriture et de lecture lors de plusieurs demande en accès en parallèle, on met en place une exclusion mutuelle, qui permet un seul acteur à la fois. Pour cela, on utilise la fonction *pthread.mutex.lock()*, qui permet de protéger l'accès à un acteur, afin d'empêcher tous les autres threads y accéder.

4 Le serveur

Le serveur permet de contrôler le jeu, il établit les connexions entre les joueurs.

4.1 Structure de données

Le serveur définit une structure **client** qui possède comme arguments l'adresse IP du client, son numéro de port et son nom. Il a également un tableau d'entier **deck**, dont les nombres correspondent à un personnage dans le tableau **nomcartes**. Le dernier entier de **deck** est le coupable. Le serveur possède aussi un tableu **tablecarte**, de dimension 2, qui permet de mémoriser les symboles attribués à chaque joueur. En plus, un tableau **perdant[4]** et un entier **cpt** sont introduit, pour mémoriser les perdants et compter le nombre.

4.2 Initialisation du serveur

Le serveur a besoin un socket pour établir des connexions entre les différents machines via le protocole TCP. Ainsi, pour l'initialiser, nous devons d'abord créer un socket.

Après avoir créé le socket, tous les données du jeu sont initialisées. On appelle la fonction *printDeck()* pour afficher sur le terminal le nom des personnages dans l'ordre. Puis, on appelle la fonction *MelangerDeck()* pour mélanger les ordres. On a ajouté l'instruction srand(time(NULL)) dans cette fonction, afin de rendre la mélange soit différent à chaque fois qu'on relance le jeu. Une affichage du tableau **tableCartes** est fait après chaque *printDeck()*.

Maintenant, on attend la connexion des 4 joueurs.

```
ZENGdeMacBook-Pro:sh13_etu ruxue$ ./server 32000
0 Sebastian Moran
1 irene Adler
2 inspector Lestrade
3 inspector Gregson
4 inspector Baynes
5 inspector Bradstreet
6 inspector Hopkins
7 Sherlock Holmes
8 John Watson
9 Mycroft Holmes
10 Mrs. Hudson
11 Mary Morstan
12 James Moriarty
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
8 John Watson
11 Mary Morstan
2 inspector Lestrade
3 inspector Gregson
0 Sebastian Moran
6 inspector Hopkins
5 inspector Bradstreet
1 irene Adler
10 Mrs. Hudson
12 James Moriarty
9 Mycroft Holmes
4 inspector Baynes
7 Sherlock Holmes
01 00 01 01 02 01 02 00
01 00 02 02 01 00 01 01
01 01 01 01 00 02 00 01
01 03 00 01 01 00 00 01
```

FIGURE 1 – Terminal du serveur

4.3 Connection des joueurs

Nous créons une boucle infini while(1), où on regardera l'état du serveur. Nous avons introduit une variable **fsmServer** pour définir l'état du serveur : 0 veut dire qu'il est en attente des connexions des 4 joueurs ; 1 veut dire qu'il est prêt pour lancer la partie.

Lorsqu'un client est connecté, il envoie un message de type 'C adresseIP numéroPort nom' au serveur. Et lorsque le serveur reçoit ce message, il récupère toutes ces données, et envoie un message personnel au joueur via *sendMessageToClient()*, pour lui communiquer son identifiant à travers un message de type 'I id'. Puis il envoie un message de type 'L' à tous les joueurs connectés via *broadcastMessage()*, pour leur communiquer la liste des joueurs. En fin, on incrémente le nombre de clients. Lorsque'il vaut 4, on peut lancer le jeu.

Lorsque le jeu a commencé, le serveur envoie un message personnel de type 'D carte1 carte2 carte3' à chaque client, pour communiquer les numéros de leurs trois cartes. Puis, il leur envoie des messages de type 'V' qui contiennent chacun le nombre de symboles possédés par le joueur. Le client peut ensuite voir le nombre de symboles qu'il possède dans son écran.

Au final, il envoie un message 'M joueurCourant' à tout le monde, pour communiquer qui est le joueur courant (qui est par défaut 0), qui permet donc l'affichage du bouton 'GO'. On passe la variable **fsmServer** à 1 pour indiquer la partie peut commencer.

FIGURE 2 – Terminal du serveur et du premier joueur lorsque les 4 joueurs sont connectés

4.4 Trois actions

4.4.1 C'est lui le coupable !

Lorsque le joueur annonce une accusation, le serveur reçoit de son part un message de type 'G id coupable', qui contient l'identifiant du joueur et le numéro de la carte accusée.

Deux cas possible après une accusation : soit il a trouvé le coupable et le jeu s'arrête, il a gagné ; soit il s'est trompé et il ne peut plus jouer, mais il doit répondre aux questions des autres.

FIGURE 3 – Message du terminal lorsque l'accusation est fausse

S'il a trouvé le coupable, le serveur va envoyer à tout le monde un message de type 'F', pour dire que ce joueur a gagné. Puis il affiche dans son propre terminal le gagant et le coupable.

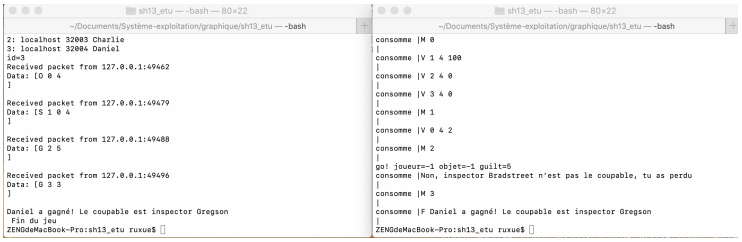

FIGURE 4 – Message du terminal lorsque l'accusation est vraie (Gauche : Serveur Droite : Joueur)

Si le joueur s'est trompé, on incrémente le nombre de perdant **cpt**, et on ajoute son id dans le tableau **perdant**, ce qui permettra de sauter son tour par la suite.

```
case 'G': //Mise en accusation
{
    int id, guiltsel;
    sscanf(buffer, "G %d %d", &id, &guiltsel);
    if (guiltsel == deck[12]){
        sprintf(reply, "F %s a gagné! Le coupable est %s", tcpClients[id].name, nomcartes[guiltsel]);
        broadcastMessage(reply);
        printf("%s a gagné! Le coupable est %s\n Fin du jeu\n", tcpClients[id].name, nomcartes[guiltsel]);
        return 0;
    }
    else{
        sprintf(reply, "Non, %s n'est pas le coupable, tu as perdu", nomcartes[guiltsel]);
        sendMessageToClient(tcpClients[id].ipAddress, tcpClients[id].port, reply);
        perdant[cpt]=id;
        cpt++;
```

FIGURE 5 – Code pour message du type 'G'

4.4.2 Qui a cet objet ?

La deuxième action possible est de demander à tout le monde : qui possède un objet en particulier. Dans ce cas là, le serveur recoit un message de type 'O' qui contient l'identifiant du joueur et le numéro correspondant à l'objet.

Le serveur va ensuite regarder le nombre dans la case correspondante dans le tableau **tableCartes** pour chaque joueur (sauf le joueur courant). Si le nombre est supérieur à 0, le serveur envoie un message de type 'V' qui se terminera par 100, et qui permettra d'afficher une étoile dans la case correspondante de l'interface, à condition que la case ne soit pas remplit par un nombre différent que 100. Sinon, le serveur envoie un message de type 'V' qui se terminera par 0.

FIGURE 6 – Qui a cet objet Livre ?

4.4.3 Combiens as-tu d'objets ?

La dernière action possible est de demander à un joueur en particulier : combiens il en possède un objet particulier. Pour cela, le serveur va recevoir un message de type 'S' contenant l'identifiant du joueur courant, l'identifiant du joueur ciblé, et le numéro d'objet sélectionné.

Ensuite, le serveur regarde la case correspondante dans le tableau **tableCartes**, et envoie un message de type 'V' qui contient le nombre de symbole que ce joueur possède à tout le monde.

FIGURE 7 – Alice, combiens as-tu Livre ?

4.5 Ré-initialisation avant le prochain tour

Après avoir traité l'action du joueur courant, nous devons incrémenter **joueurCourant**, et ondoit vérifier si ce nouvel joueur ne soit pas dans le tableau **perdant**, si c'est le cas, on passe au joueur suivant. Lorsque on atteint 4, la variable **joueurCourant** repasse à 0.

```
joueurCourant = (joueurCourant+1)%4;
int i=0;
while (i < cpt){
    if (joueurCourant == perdant[i]){
        joueurCourant = (joueurCourant+1)%4;
        i=0;
    }
    else{
        i++;
    }
}
sprintf(reply, "M %d", joueurCourant);
```

FIGURE 8 – Code pour incrémenter joueurCourant

Une fois que le nouvel joueur courant est fixé, le serveur envoie un message de type 'M' à tout le monde pour afficher le bouton GO dans l'interface de celui qui va jouer.

On remarque aussi qu'on a ajouté une condition sur la variable **joueurCourant** pour terminer le jeu lorsque le nombre de perdant vaut 3, donc le joueur qui reste gagne le jeu.

```
if (cpt == 3){
    sprintf(reply, "F %s a gagné! Le coupable est %s", tcpClients[joueurCourant].name, nomcartes[deck[12]]);
    broadcastMessage(reply);
    printf("%s a gagné! Le coupable est %s\n Fin du jeu\n", tcpClients[joueurCourant].name, nomcartes[guiltsel]);
    return 0;
}
```

FIGURE 9 – Code pour cpt==3 : Fin du jeu

5 Le client

Le fichier **sh13.c** utilise les bibliothèques SDL, pthread et des sockets.

5.1 Structure de données

Le client utilise différentes structures de données, comme par exemple **pthread_t** et **pthread_mutex_t** qui permettent de stocker l'identifiant du server tcp et le mutex.

De plus, il utilise des chaînes de caractères pour stocker ses propres informations et celles du serveur : gbuffer (message recç provient du serveur), gServerIpAddress (l'adresse IP du serveur), gServerPort (le numéro de port du server), gClientIpAddress (l'adresse IP du client) et gName (le nom du client).

Il utilise également des tableaux : gNames pour stocker les noms des 4 joueurs, et tableCartes pour stocker les cartes.

5.2 Initialisation

Le fichier **sh13.c** utilise des threads pour des fonctions en parallèle. Pour rassurer la mise à jour des données à tout le monde, on déclare la variable globale **synchro** comme volatile, ce qui force le programme à relire la valeur de cette variable à chaque utilisation, donc la fonction interrompue, et découvrira que la valeur est modifiée par un autre intervenant.

5.3 Communication avec le serveur

Le client doit écrire des messages au serveur, et être capable de recevoir les réponses, donc il a un comportement de "serveur", qui utilise aussi *bind* et *listen*

de la librairie socket.

Une fois que le port de client est prêt à écouter, on rentre dans une boucle infini. Dans un premier temps, on attend que le serveur nous annonce le départ du jeu, puis on récupère le message. Puis, on signale l'arrivée d'un message par l'intermédiaire du sémaphore *mutex*. Il permet de gérer l'interface graphique, tant qu'il n'aura pas finie de traité le message, on ne passera pas à la suite.

Comme disait précedemment dans la partie du serveur, le client reçoit des messages du serveur, puis les interprétera pour modifier son interface graphique. Avant toute lecture, on regarde la variable *synchro* afin de déterminer si le client a reçu un message.
On observe les types de message suivants :

- **Type I** : qui permet au joueur de savoir son identifiant
- **Type L** : qui mettre à jour la liste des joueurs connectés
- **Type D** : qui permet au joueur de savoir ses trois cartes
- **Type V** : qui contient l'information d'un symbole pour un joueur particulier, qui permet de mettre à jour le tableau des cartes
- **Type M** : qui informe le joueur courant. Avec une boucle if et la variable goEnabled, si c'est son tour, le bouton "GO" s'affiche dans l'interface graphique
- **Type F** : qui annonce la fin du jeu, le gagnant et le coupable.

Une fois le message lu, on remet *synchro* à 0 pour pouvoir à nouveau être capable de lire des messages du serveur.

5.4 Interface graphique

Pour chaque joueur, en utilisant la bibliothèque SDl, on crée une fenêtre de taille 1024 sur 768 pixels. Cette fenêtre contient une liste des 13 personnages (avec les symboles à côté), et un tableau de taille 4*8 pour afficher les informations de symboles pour chaque joueur.

Le client utilise une boucle pour gérer les évènements graphiques, qui prennent en compte les massages du serveur.

L'affichage de bouton 'GO' signifie que c'est à son tour. Pour déclencher un évènement, il suffit de cliquer sur "GO". Un boucle if va vérifie s'il y a eu un évènement, puis on regardera de quel évènement s'agit-il.

— Si le joueur clique sur le bouton connect pour connecter au jeu, le client envoie un message de type C au serveur

— Si le joueur sélectionne un objet pour demander qui en possède, il va envoyer un message de type O au serveur

— Si le joueur sélectionne un objet et un joueur pour lui demander combien il en possède cet objet, il va envoyer un message de type S au serveur

— Si le joueur sélectionne un personnage du jeu pour l'accuser, il va envoyer un message de type G au serveur.

6 Interactions entre le serveur et les clients

Pour que vous pouvez mieux comprendre, voici un protocole

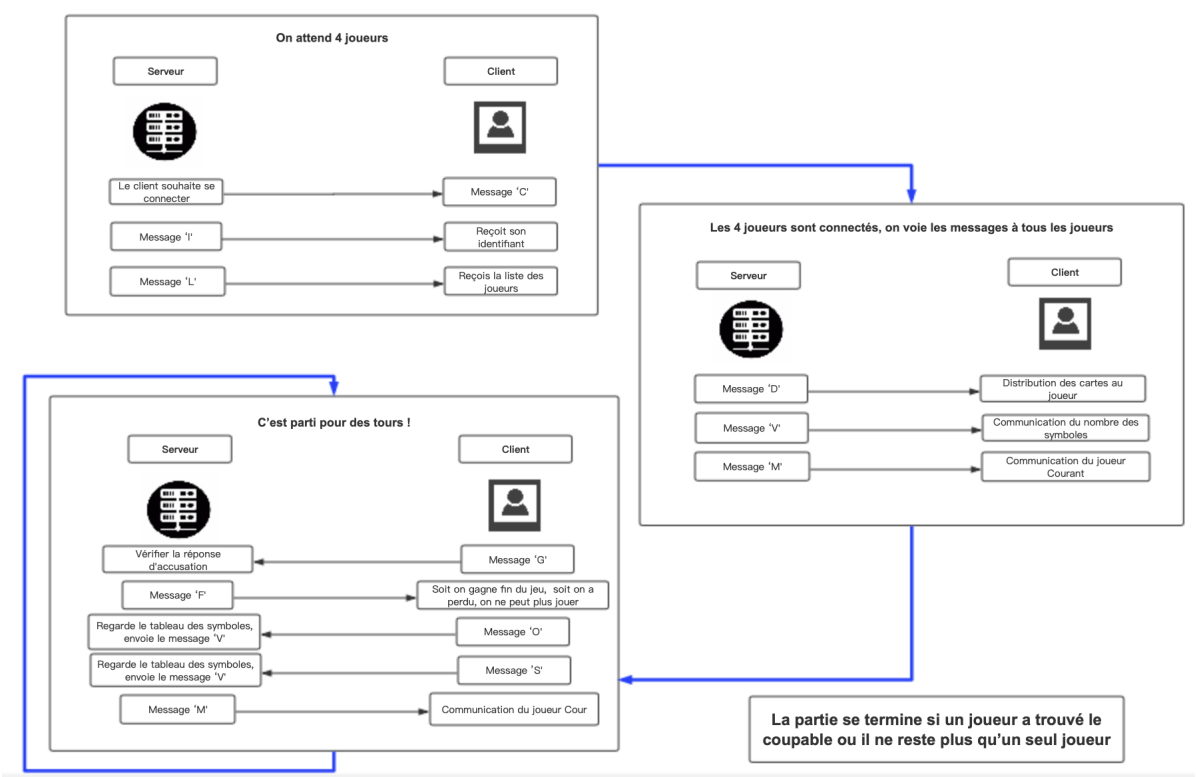

FIGURE 10 – Protocole du jeu

7 Améliorations possibles

Pour ce projet, nous avons rencontré des difficultés pour réaliser certaines règles. Tour d'abord, lors que une accusation est fausse, comment mettre le joueur inactif ? J'ai créé une liste perdant, qui nécessite beaucoup de comparaisons pour la mise à jour du joueur courant, je me suis rendu compte que il est mieux d'introduire une variable **actif** dans la structure cleint pour désigner l'état du joueur.

Ensuite, concernant l'interface graphique, en cas de perte ou de victoire, une simple message affiche dans le terminal, nous pouvons l'améliorer en affichant une image dans l'interface graphique.

8 Conclusion

Ce projet nous a permis de mieux comprendre les notions du réseau et de l'interface graphique. Nous avons appris à utiliser des thread et les sockets, d'établir une communication réseau entre un serveur et plusieurs clients.

Enfin, je remercie à monsieur PECHEUX de fournir un début de code complet, qui me permet d'avancer rapidemment. Ce projet me donne une première expérience pour réaliser un jeu du réseau, qui est très intéressant.