
Rapport du Jeu Sherlock 13



Etudiants :
BERBAGUI Amine
AMAIRI Tahar

Enseignant :
Mr PECHEUX François

Système d'exploitation

MAIN 3

12 juin 2021

Table des matières

1	Introduction	2
2	Règles du jeu	3
3	Fonctionnement du jeu	4
3.1	Les structures serveurs & clients	4
3.1.1	Serveur TCP	4
3.1.2	Modèle client/serveur	5
3.2	Messages	5
3.2.1	Client ⇒ serveur	6
3.2.2	Serveur ⇒ client	6
4	Diagramme UML	7
5	Analyse des fichiers sources	8
5.1	serveur.c	8
5.1.1	Rôle	8
5.1.2	Description du code	8
5.1.3	Code à compléter	10
5.2	sh13.c	11
5.2.1	Rôle	11
5.2.2	Description du code	11
5.2.3	Code à compléter	13
6	Améliorations	14
6.1	Ajout de pop-ups	14
6.2	Ajout de sons dynamiques	14
7	Lancement du jeu	15
8	Résultats	16
8.1	Jeu de base	16
8.2	Les améliorations	24
8.2.1	Les pop-ups	24
8.2.2	Les sons	26
8.3	Informations sur les terminaux	27
9	Conclusion	28

1 Introduction

Dans le cadre de notre cours de **Système d'exploitation**, nous avons pour objectif la réalisation d'un projet consistant à programmer un jeu en **langage C**. Parmi plusieurs jeux disponibles, celui qui a retenu notre attention est **Sherlock 13**, jeu dont on détaillera le fonctionnement par la suite. Ce jeu doit être créé de manière à pouvoir être joué en réseau (à 4 joueurs) tout en contenant toutes les fonctionnalités nécessaires.

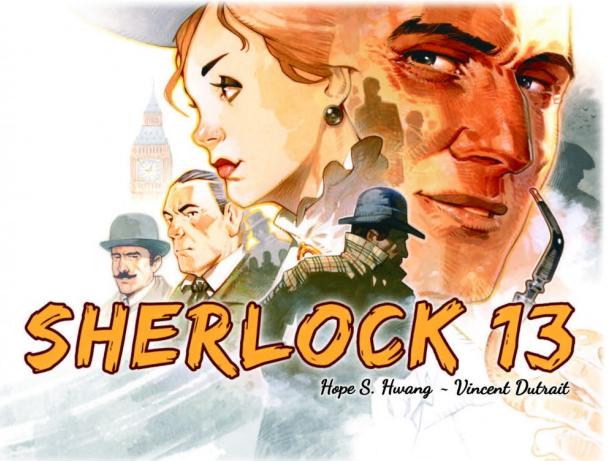
Ainsi, le but de ce projet est de mettre en pratique toute les connaissances acquises en programmation système mais pas que, puisque ce jeu requiert également la création d'une **interface graphique** qui sera gérée par la bibliothèque **SDL**, en plus de la mise en place d'un protocole **TCP**.

L'implémentation du jeu étant déjà réalisée en grande majorité par notre enseignant, le but est donc de compléter les parties de code **manquantes** de deux fichiers source à savoir **server.c** et **sh13.c** de sorte à faire tourner le jeu parfaitement. Ce rapport vient donc expliquer le fonctionnement du jeu, celui du code et à montrer les résultats obtenus au lancement. Enfin, nous verrons aussi les améliorations apportées au code de base.

2 Règles du jeu

Sherlock 13 est un jeu d'enquête et de déduction dans lequel les joueurs prennent le rôle de détective, essayant de démasquer la carte **coupable** qui a commis le crime.

Le jeu se compose de 13 cartes de personnage, avec 2 à 3 caractéristiques. Chaque caractéristique est partagée avec 3 à 5 autres personnages. Les cartes sont mélangées et l'une est mise de côté. Cette carte représente le personnage à deviner, les autres cartes sont réparties parmi les joueurs (3 cartes par joueur).



Les joueurs comptent alors le total des caractéristiques qu'ils ont sur leurs cartes et vont pouvoir poser une de ces deux questions pour mener leur enquête et recueillir de nouvelles preuves :

- Une action d'enquête visant **tous les joueurs** : "Qui a cette caractéristique parmi ses cartes ?"
- ou bien une visant **uniquement un joueur spécifique** : "Combien de cette caractéristique avez-vous ?"

En utilisant ces indices, les joueurs tentent de déterminer quelle est la carte coupable. Au lieu d'une question, un joueur peut annoncer quel personnage il soupçonnera d'être la carte coupable. Ce joueur vérifie en secret la carte cachée. Si le soupçon était correct, ce joueur gagne le jeu, sinon le joueur est éliminé et les autres joueurs continuent la partie.



3 Fonctionnement du jeu

Dans cette partie, nous abordons d'un point de vue technique le fonctionnement de notre jeu, nous expliquerons dans un premier temps comment fonctionnent les structures du serveur et clients, puis comment communiquent-elles dans un second temps. Enfin, nous terminerons avec le **diagramme UML** du jeu résumant le fonctionnement réseau du jeu.

3.1 Les structures serveurs & clients

3.1.1 Serveur TCP

Voyons tout d'abord comment fonctionne le jeu d'un point de vue réseau avec nos structures serveurs et clients. Notre jeu utilise le protocole **TCP** (*Transmission control protocol*). Développé dans les années 70, ce protocole est un des principaux de la couche transport dans le modèle **TCP/IP**. Il permet, au niveau des applications, de gérer les données en provenance (ou à destination) de la couche internet inférieur du même modèle.

Lors d'une communication à travers le protocole TCP, les deux machines doivent établir une connexion. La machine émettrice (celle qui demande la connexion) est appelée client, tandis que la machine réceptrice est appelée serveur. On dit qu'on est alors dans un environnement Client-Serveur. Ainsi le rôle de notre serveur est très important et doit posséder les caractéristiques suivantes :

- Il doit tourner en permanence en attendant les requêtes du client.
- Il peut répondre à plusieurs clients en même temps.
- Il est le garant du jeu (gère les joueurs).
- Il permet l'initialisation et la fin d'une communication de manière courtoise.

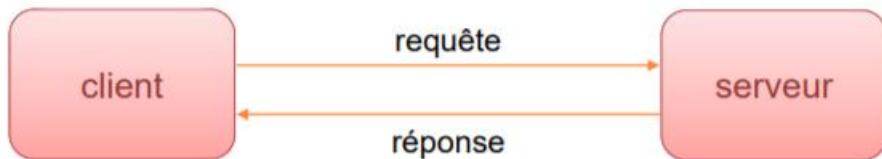


FIGURE 1 – Schéma décrivant la communication client/serveur

3.1.2 Modèle client/serveur

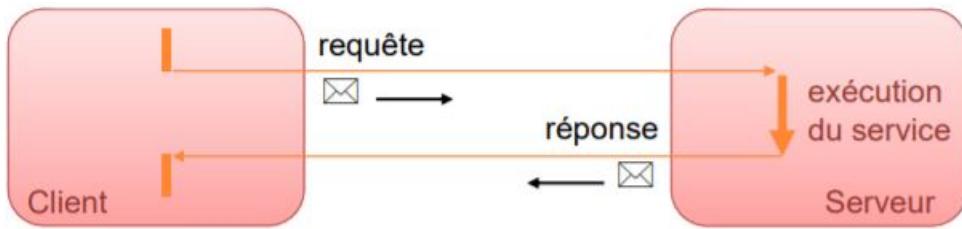


FIGURE 2 – Schéma simplifié du modèle serveur/client

Le schéma ci-dessus illustre la communication entre un client et un serveur. Or, dans notre jeu en réseau, notre configuration sera de 4 clients/serveur et 1 serveur, ce qui signifie que les joueurs seront à la fois client **et** serveur. Ainsi, ils doivent pouvoir traiter deux événements en parallèle :

- Les événements **réseaux** (réception/envoie de messages de manière asynchrone).
- Les événements **graphiques** (au niveau de l'interface SDL).

Pour cela, on utilisera la notion de **thread** : on aura donc un thread qui s'occupera du côté réseau et le main qui s'occupera de l'interface graphique.

3.2 Messages

Maintenant que nous avons vu ce qu'était un serveur TCP, ainsi que le modèle client/serveur, intéressons nous aux interactions entre les deux dans notre jeu et comment chaque message est interprété par le serveur/client. La communication se fait via l'ouverture d'une **socket**, il s'agit d'un modèle permettant la communication **IPC** (*Inter Process Communication*) afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'à travers un protocole TCP. Pour cela, nous allons utiliser la communication serveur/client d'Unix :

1. Le serveur ouvre une socket avec la fonction **socket()**.
2. Il associe à cette socket un port avec la fonction **bind()**.
3. Il se met en écoute, i.e qu'il est prêt à accepter des connexions, avec la fonction **listen()**.
4. Finalement, il se bloque en attente d'une connexion à l'aide de la fonction **accept()**.

5. Lorsqu'un client se connecte à l'aide de la fonction `connect()`, on utilise le principe fondamental de "**tout est fichier dans Unix**" pour communiquer :

- Pour envoyer un message, on utilise l'interface `write`.
- Pour lire un message, on utilise l'interface `read`.

Pour communiquer on utilisera un encodage bien spécifique qu'on présentera dans la prochaine section.

3.2.1 Client ⇒ serveur

Voyons à présent quels sont les messages que notre serveur est susceptible de recevoir et à quoi correspondent-ils :

Encodage	Données	Information
C : Connexion	Adresse IP, port, nom	Connexion avec le serveur
G : Guilty	ID, n°suspect	Accusation d'une carte
O : Others	ID, n°symbole	Enquête globale
S : Solo	ID, n°symbole, joueur	Enquête spécifique

3.2.2 Serveur ⇒ client

De la même manière voyons les messages envoyés par le serveur vers le client :

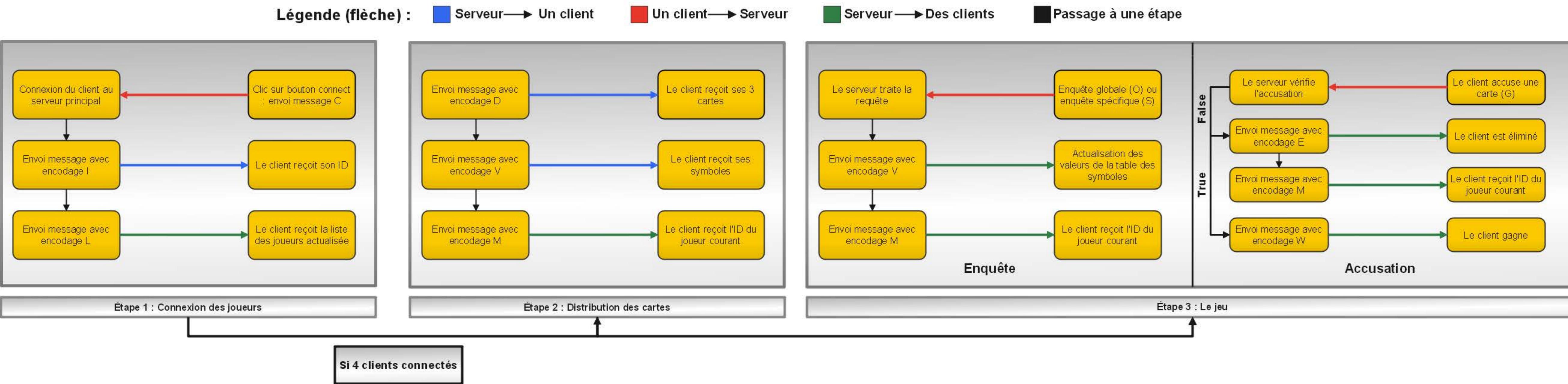
Encodage	Données	Information
I : ID	ID	Identifiant du joueur
L : List	Tableau d'ID	Liste des noms des joueurs
D : Deck	Tableau d'indice	Les 3 cartes du joueur
M	ID	Numéro du joueur courant
V : Value	ID, n°symbole, valeur	Valeur de la table des symboles
E : Eliminated	ID, n°suspect	ID de l'éliminé et de la carte suspectée
W : Winner	ID, n°suspect	ID du gagnant et du coupable

N.B :

- **ID** correspond au numéro du joueur envoyant le message.
- **Enquête globale** correspond à la demande visant tous les joueurs : "Qui a cette caractéristique parmi ses cartes ?".
- **Enquête spécifique** correspond à la demande visant un joueur : "Combien de cette caractéristique avez-vous ?".

4 Diagramme UML

Diagramme UML



5 Analyse des fichiers sources

Pour pouvoir lancer le jeu, nous avons à notre disposition deux fichiers sources écrits en C : `server.c` et `sh13.c`. Cependant, ils sont incomplets et afin de pouvoir les compléter il est nécessaire de connaître leurs fonctionnements.

5.1 serveur.c

5.1.1 Rôle

Le fichier `server.c`, comme son nom l'indique, permet d'ouvrir le serveur principal en utilisant le protocole **TCP** décrit au-dessus. Bien plus, il s'occupe du mélange des cartes et de leur distribution aux clients (i.e les joueurs). Finalement, il gère les messages envoyés par les clients (voir la partie 3) en leur renvoyant une réponse adaptée selon la requête reçue.

5.1.2 Description du code

Avant de s'intéresser au **main**, il est important de noter que nous avons des **variables globales**, une **structure** et finalement des **fonctions** :

- Structure
 - * **_client** : une structure permettant de définir un client avec trois paramètres : son adresse IP (**ipAddress**), son numéro de **port** et finalement son nom (**name**).
- Variables globales
 - * **tcpClients** : un tableau de 4 clients.
 - * **nbClients** : variable permettant de compter le nombre de clients connectés au serveur.
 - * **fsmServer** : variable d'état du serveur. Si celle-ci vaut 0 alors celui-ci ne peut initier le jeu car il manque des joueurs et lorsque nbClients vaut 4, elle passe à 1 et le serveur peut lancer le jeu.
 - * **deck** : tableau d'index des cartes.
 - * **tableCartes** : tableau de taille 4×8 permettant de stocker pour chacun des joueurs les symboles qu'ils ont selon la distribution des cartes.
 - * **nomcartes** : tableau contenant le nom des cartes.
 - * **joueurCourant** : variable permettant de définir le joueur courant en train de jouer via son ID.
 - * **TrackEliminatedPlayer** : tableau de taille 4 initié à 0 permettant de suivre l'élimination des joueurs. Par exemple, si le joueur d'ID 0 est éliminé alors `TrackEliminatedPlayer[0]` vaut 1.

- Fonctions

- * **error** : permet de gérer les erreurs selon le message en paramètre.
- * **melangerDeck** : permet de mélanger les cartes aléatoirement à l'aide de la variable deck.
- * **createTable** : Initialise le tableau des symboles tableCartes. On choisit comme configuration pour la distribution des cartes :
 - Le joueur d'ID 0 possède les cartes d'indice 0, 1, 2.
 - Le joueur d'ID 1 possède les cartes d'indice 3, 4, 5.
 - Le joueur d'ID 2 possède les cartes d'indice 6, 7, 8.
 - Le joueur d'ID 3 possède les cartes d'indice 9, 10, 11.
 - Le coupable est la carte d'indice 12.
- * **pr Deck** : permet d'afficher le deck.
- * **pr Clients** : permet d'afficher les informations des clients.
- * **findClientByName** : permet d'obtenir l'ID d'un client avec son nom.
- * **sendMessageToClient** : permet d'envoyer un message à un client spécifique.
- * **broadcastMessage** : permet d'envoyer un message à tous les clients.

Dans le **main**, on utilise le protocole TCP pour initier le serveur. En effet, nous avons :

- A la ligne 249, l'ouverture du **socket**.
- A la ligne 260, le **bind** permettant d'associer au socket le port en paramètre du main.
- A la ligne 263, le **listen** permettant d'initier l'écoute de connexions entrantes.

Après cela nous rentrons dans une boucle infinie qui est partagé en deux sections :

- Si $fsmServer = 0$, alors à l'aide de la fonction **accept** on accepte les connexions entrantes. A chaque connexion, on incrémente nbClients de 1 puis on envoie au client connecté son ID (à l'aide de la fonction **sendMessageToClient**) et la liste des joueurs connectés actuellement (à l'aide de la fonction **broadcastMessage**). Dès lorsque $nbClients = 4$, alors le jeu peut commencer : on envoie à chacun des clients le joueur courant et individuellement, on envoie les cartes et les symboles à chacun des clients. Finalement, **fsmServer** passe à 1.
- Si $fsmServer = 1$, alors on sait que le jeu a commencé et le serveur se met à disposition des clients en répondant à leurs requêtes. Dès lorsqu'un joueur gagne, on sort de la boucle infinie et le serveur rompt avec toutes les connexions et celui-ci est fermé.

- Il est intéressant de noter que pour l'envoi de message nous utiliserons la variable **reply** et que pour la réception de message nous utiliserons la variable **buffer**. L'encodage sera par la suite écrit/lu dans le **premier caractère** de ces deux variables.

5.1.3 Code à compléter

Nous devons compléter deux parties du code présentes chacune dans la boucle infinie :

- La première partie se situe lorsque $nbClients = 4$. Comme souligné auparavant, à ce niveau nous devons : distribuer individuellement les cartes et les symboles à chacun des joueurs et annoncer le joueur qui va commencer à jouer. Pour cela, on utilise l'encodage décrit à la partie 3 :
 - On envoie à chacun des joueurs les cartes à l'aide de l'encodage **D** en utilisant la variable **reply** et la fonction **spr f**. On rappelle qu'un joueur d'ID k possède les cartes d'indice $deck[0 + 3 \times k]$, $deck[1 + 3 \times k]$ et $deck[2 + 3 \times k]$, il suffit donc d'effectuer une boucle **for** pour k allant de 0 à 3 et d'envoyer au joueur d'ID k le message **reply** à l'aide de la fonction **sendMessageToClient**.
 - Au sein de la même boucle for, on envoie le nombre de symboles au joueur d'ID k selon les cartes qu'il a obtenu en utilisant l'encodage **V** de la même manière qu'auparavant avec **reply** et **spr f**.
 - Finalement, on envoie à tous les joueurs le joueur courant avec l'encodage **M** en utilisant la fonction **broadcastMessage**.
- La seconde partie se situe lorsque le serveur passe à l'état 1, i.e, lorsqu'il se met à disposition des clients afin de satisfaire leurs requêtes. Pour cela, on utilise un **switch** pour chaque encodage reçu et on lit la variable **buffer** à l'aide de la fonction **sscanf** :
 - **Encodage G** : on vérifie à l'aide du buffer et **sscanf** si la carte proposée par le joueur est coupable :
 - * Si c'est le cas, alors celui-ci gagne la partie et on envoie à tous les joueurs le message ayant un encodage **W**.
 - * Si ce n'est pas le cas, alors le joueur est éliminé et la partie continue avec un joueur en moins. Par conséquent, on annonce l'élimination de celui-ci et la carte qu'il a choisi à tous les autres joueurs avec un message ayant un encodage **E**. Bien plus, on annonce le prochain joueur qui va jouer à l'aide d'un message ayant un encodage **M**.
 - * Lorsqu'un joueur est éliminé et qu'il n'en reste qu'un alors ce dernier gagne aussi la partie, on en revient donc au premier cas.
 - * Il est important de rappeler que lorsqu'un joueur gagne une partie, alors on sort de la boucle infinie et on ferme le socket. Le programme finit donc son exécution.

- **Encodage O** : on lit à l'aide de sscanf dans le buffer, l'ID du client et l'indice du symbole demandé par celui-ci. Puis nous vérifions pour chacun des autres joueurs s'ils possèdent ce dit symbole à l'aide du tableau tableCartes : si c'est le cas alors on écrit dans reply une valeur de 100 ou 0 sinon puis on broadcast à tous les clients reply. Bien plus, celui-ci utilisera un encodage **V**. Juste après, on annonce le prochain joueur qui va jouer à l'aide d'un message ayant un encodage **M**.
- **Encodage S** : De même que pour l'encodage O, mais ici on écrit dans reply la quantité du symbole que possède le joueur visé par l'enquête. Le résultat est ensuite envoyé uniquement au client ayant initié la requête. Finalement, on annonce le prochain joueur qui va jouer.

5.2 sh13.c

5.2.1 Rôle

Le fichier sh13.c contient le code permettant d'exécuter le jeu côté client, i.e, le joueur. On rappelle que le joueur est à la fois un serveur et un client, il doit donc être capable de recevoir et d'envoyer des requêtes. Bien plus, l'exécution de l'interface graphique se fait du côté du client. Par conséquent, sh13.c permet d'initier le serveur client et s'occupe de l'affichage graphique en parallèle, il est donc nécessaire d'utiliser la notion de **thread**.

5.2.2 Description du code

Tout comme pour le fichier server.c, nous allons nous intéresser en premier aux différentes variables globales et aux fonctions avant de passer à la description du main :

- **Variables globales**
 - * **thread_serveur_tcp_id** : ID du thread initialisant le serveur client TCP.
 - * **mutex** : mutex thread.
 - * **gbuffer** : variable contenant les messages reçus par le serveur principal.
 - * **gServerIpAddress** : adresse IP du serveur principal.
 - * **gServerPort** : port du serveur principal.
 - * **gClientIpAddress** : adresse IP du client exécutant sh13.c.
 - * **gClientPort** : port du client exécutant sh13.c.
 - * **gName** : nom du joueur exécutant sh13.c.
 - * **gNames** : tableau contenant le nom de tous les joueurs par ID, e.g, gNames[**ID = 0**] donnera le nom du joueur d'ID 0. Les champs du tableau seront initialisés à "-".
 - * **gId** : ID joueur exécutant sh13.c

- * **joueurSel** : variable permettant de savoir si le joueur à sélectionner un joueur pour une enquête. Elle sera initialisée à -1.
- * **objetSel** : variable permettant de savoir si le joueur à sélectionner un symbole pour une enquête. Elle sera initialisée à -1.
- * **guiltSel** : variable permettant de savoir si le joueur à sélectionner une carte coupable.
- * **guiltGuess** : variable permettant de gérer les croix rouges contenues dans le tableau des suspects. Elle sera initialisée à -1.
- * **tableCartes** : tableau de taille 4×8 permettant de stocker pour chacun des joueurs les symboles qu'ils ont selon la distribution des cartes.
- * **b** : tableau contenant les cartes possédées par le joueur.
- * **goEnabled** : variable d'état permettant l'affichage du bouton "Go" pour initier un tour de jeu (1 si c'est le cas, 0 sinon).
- * **connectEnabled** : variable d'état permettant l'affichage du bouton "Connect" pour initier une connexion (1 si c'est le cas, 0 sinon).
- * **nobjets** : tableau contenant le nombre de chaque symbole.
- * **bnoms** : tableau contenant les noms des cartes.
- * **synchro** : variable permettant d'indiquer si le client a reçu un message de la part du serveur.
- Fonctions
 - * **fn_serveur_tcp** : fonction permettant d'initier le thread du serveur client. Elle utilise un protocole TCP, tout comme dans le fichier server.c.
 - * **sendMessageToServer** : fonction permettant d'envoyer un message au serveur principal.

Dans le **main**, on commence par initier l'affichage graphique grâce à la librairie SDL2 : initialisation de la fenêtre d'affichage, son ouverture, chargement de la police, des images, du fond, des textures etc... (de la ligne 165 à la ligne 225). Après cela, on crée le thread qui s'occupera de la partie réseau avec la fonction **fn_serveur_tcp** (ligne 231) et on rentre dans une boucle infinie (ligne 233) similaire à celle du fichier server.c mais celle-ci dépend de la variable **quit** :

- Lorsqu'un **événement graphique** se produit on rentre dans le if de la ligne 235 et on regarde à quoi cela correspond (avec l'emplacement du clic effectué par le joueur) en actualisant les variables **joueurSel**, **objetSel** et **guiltSel**. Ensuite, on envoie le message correspondant au serveur avec la fonction **sendMessageToServer**. Par ailleurs, pour savoir l'endroit exact du clic du joueur on vérifie les variables **mx** et **my**.

- Lorsqu'un **événement réseau** se produit on rentre dans le if de la ligne 321 grâce à la variable synchro. On lit ensuite le message reçu dans le gbuffer et on agit en conséquence selon l'encodage lu.

Finalement, à partir de la ligne 392 le programme réactualise l'interface graphique avec les informations obtenues suite à l'un des événements.

5.2.3 Code à compléter

Nous avons deux parties à compléter, chacune se trouvant dans une section d'un des événements décrits auparavant :

- **Évènement graphique**

- * Lorsque **connectEnabled = 1** : cela correspond au clic du bouton "Connect", ce qui signifie que le joueur essaye d'établir une connexion avec le serveur principal. Pour cela, on écrit dans la variable **sendBuffer** les variables globales **gClientIpAddress**, **gClientPort** et **gName** puis on envoie sendBuffer au serveur principal en utilisant un encodage **C**.
- * Lorsque **goEnabled = 1** : cela correspond au clic du bouton "Go", ce qui signifie que le joueur vient de jouer son tour. Pour connaître quelle action vient d'être menée par celui-ci on regarde les valeurs de **guiltSel**, **guiltGuess** et **objetSel** :
 - Si $guiltSel \neq -1 \Rightarrow$ proposition d'une carte coupable. Pour cela on envoie, en utilisant un encodage **G**, un message au serveur principal avec l'indice de la carte coupable.
 - Si $objetSel \neq -1$ et $joueurSel = -1 \Rightarrow$ enquête chez tous les joueurs. Pour cela on envoie, en utilisant un encodage **O**, un message au serveur principal.
 - Si $objetSel \neq -1$ et $joueurSel \neq -1 \Rightarrow$ enquête chez un joueur. Pour cela on envoie, en utilisant un encodage **S**, un message au serveur principal avec l'indice du symbole.

- **Évènement réseau**

- * Lorsque **synchro = 1** : cela correspond au moment auquel le client reçoit une réponse de la part du serveur principal. Pour filtrer les messages, on se base sur l'encodage décrit à la partie 3 en utilisant un **switch** (on rappelle que les réponses du serveur sont écrites dans la variable gbuffer) :
 - **Encodage I** : l'ID du joueur est enregistré dans **gID**.
 - **Encodage L** : on actualise le tableau **gNames** avec le nouveau joueur connecté.
 - **Encodage D** : on enregistre l'indice des cartes que le joueur a obtenu dans le tableau **b**.

- **Encodage M** : on vérifie si l'ID du joueur qui va jouer est celle du client. Si c'est le cas, alors goEnabled passe à 1.
- **Encodage V** : on actualise le tableau tableCartes avec les réponses obtenues suite aux enquêtes.
- **Encodage W** : on annonce le joueur gagnant et la carte coupable. La variable quit passe à 1 afin de sortir de la boucle infinie.
- **Encodage E** : on annonce le joueur qui a été éliminé.

Bien plus, pour chacun des messages reçus, on a une description personnalisée dans le terminal du client afin qu'il puisse mieux comprendre la signification de chacun des encodages.

6 Améliorations

Dans cette section nous discuterons des améliorations apportées au jeu de base. Celles-ci sont toutes présentes dans le fichier `sh13.c`.

6.1 Ajout de pop-ups

La première amélioration ajoutée est l'ouverture de pop-up durant certaines phases de jeu à l'aide de SDL. Ces fenêtres sont au nombre de trois :

- Lorsqu'un joueur est éliminé suite à une mauvaise accusation : on utilise la fonction `disp_elim`. Celle-ci sera placée dans le **switch** au cas **E**.
- Lorsqu'un joueur a trouvé la carte coupable et gagne donc la partie : on utilise la fonction `disp_win`. Celle-ci sera placée à la sortie de la boucle while.
- De façon analogue, lorsqu'un joueur a perdu la partie, soit parce qu'il n'a pas été assez rapide ou bien car il a été éliminé : on utilise la fonction `disp_lose`. De même, celle-ci sera placée à la sortie de la boucle while.

6.2 Ajout de sons dynamiques

La seconde amélioration serait de mettre du son au sein de notre jeu à des moments bien précis. En effet, l'ajout de fichiers audio dans une fenêtre graphique peut se faire avec la bibliothèque **SDL.h** mais celle-ci n'offre pas la possibilité de jouer plusieurs sons en même temps, etc... Pour cela, une bibliothèque plus complète existe, **SDL_mixer.h** qui s'installe très facilement sous UNIX à l'aide de cette commande :

```
sudo apt-get install libSDL2-mixer-dev
```

Cette librairie offre beaucoup de fonctionnalités, elle prend en charge tout un tas de formats : **.wav .mp3 .voc .mid .ogg** et permet de gérer la fréquence des sons, le volume, etc...

Ainsi, l'idée serait de mettre un son bien spécifique à chaque évènement du jeu. Par exemple, lorsqu'un joueur est éliminé, avoir un son représentant cela, pareil lorsqu'un joueur gagne. Tout ceci aura pour but d'améliorer l'immersion du joueur dans le jeu.

Cette implémentation se fera dans le fichier **sh13.c** à l'aide de la fonction **playsound()** qui prend en paramètre le chemin vers la musique à jouer et le temps durant lequel la musique jouera. Il suffit ainsi d'appeler au bon endroit cette fonction :

- Au niveau des fonctions **disp_elim**, **disp_win** et **disp_lose** : ici on aura des musiques appropriées pour chaque cas.
- Lorsqu'un joueur est éliminé alors tous les autres joueurs seront avertis par une voix-off : "A player has been eliminated".
- De même, lorsque c'est au tour du joueur de jouer ("Your turn to pick") ou bien lorsque c'est à l'adversaire ("Enemy's turn to pick").
- Finalement, nous aurons une voix-off marquant le début de la partie ("The battle begins!").

7 Lancement du jeu

Concernant le lancement du jeu, cela se fait plutôt aisément. En effet, il faut suivre quelques étapes sur un terminal UNIX, à savoir :

1. **Compilation** : pour cela on exécute la commande **./cmd.sh**, qui est fichier **shell** contenant les commandes de compilation des fichiers sources permettant de générer nos deux exécutables.
2. **Lancement du server** : On lance le serveur en exécutant la commande :

./server <Numéro de port>

3. **Connexions joueurs** : A présent que le serveur est lancé, chaque joueur doit rejoindre le serveur et se connecter pour rejoindre la partie. Pour cela, on exécute la commande suivante :

**./sh13 <Adresse IP serveur>
<Port serveur> <Adresse IP client> <Port client> <Nom client>**

N.B : vous pouvez obtenir votre adresse IP en tapant la commande : `hostname -I`. Remplacez les adresses IP par `localhost` si vous voulez exécuter le serveur et les clients sur le même ordinateur. Il est important de noter qu'il faut choisir des ports différents et disponibles !

Nous avons aussi mis à disposition un fichier `test.sh` permettant d'automatiser ces différentes étapes afin de pouvoir tester le code rapidement. Il est important de noter que ce fichier shell a été testé avec le terminal d'origine d'Ubuntu 20.04, i.e, **gnome terminal**. Il est donc nécessaire de l'installer si vous utilisez une différente distribution ou terminal.

8 Résultats

Il est maintenant temps de vous présenter le jeu !

8.1 Jeu de base

- La première fenêtre est celle pour se connecter, sur laquelle tous les joueurs vont tombés au départ :

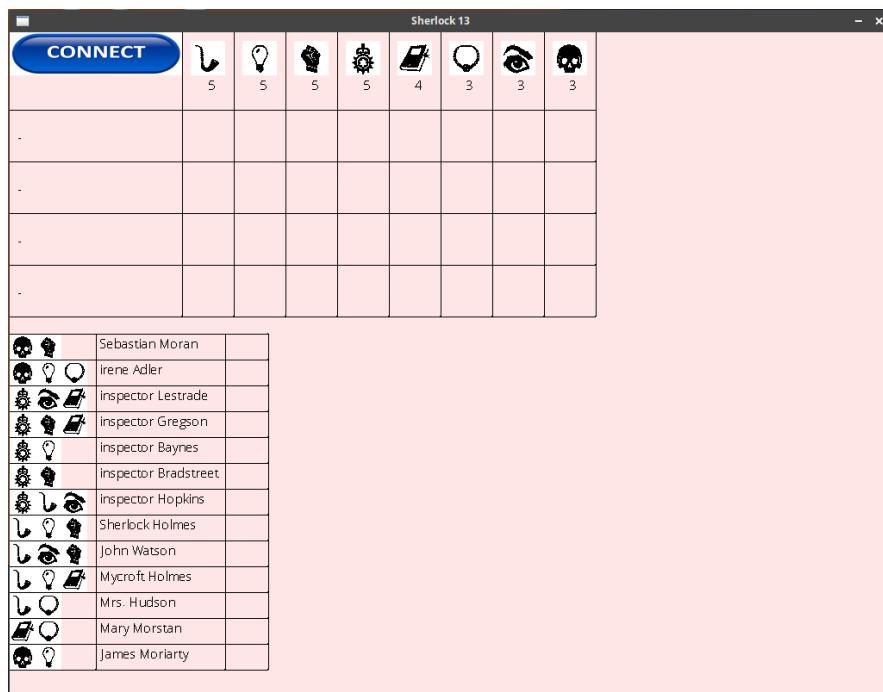


FIGURE 3 – Fenêtre de connexion au serveur

2. Une fois connecté, le joueur patiente jusqu'à la connexion des joueurs manquants sur le serveur :



FIGURE 4 – Joueur connecté en attente des joueurs manquants

On remarque sur cette figure qu'il y a déjà des joueurs connectés !

3. Lorsque les 4 joueurs se sont connectés au serveur, la partie peut commencer. Chaque joueur reçoit ses 3 cartes et peut les marquer d'une croix rouge car ils ne peuvent pas être coupable. Bien plus, chaque joueur dispose d'un tableau récapitulant le décompte de chaque symbole. Le bouton **GO** sert à indiquer au joueur que c'est à son tour de jouer.

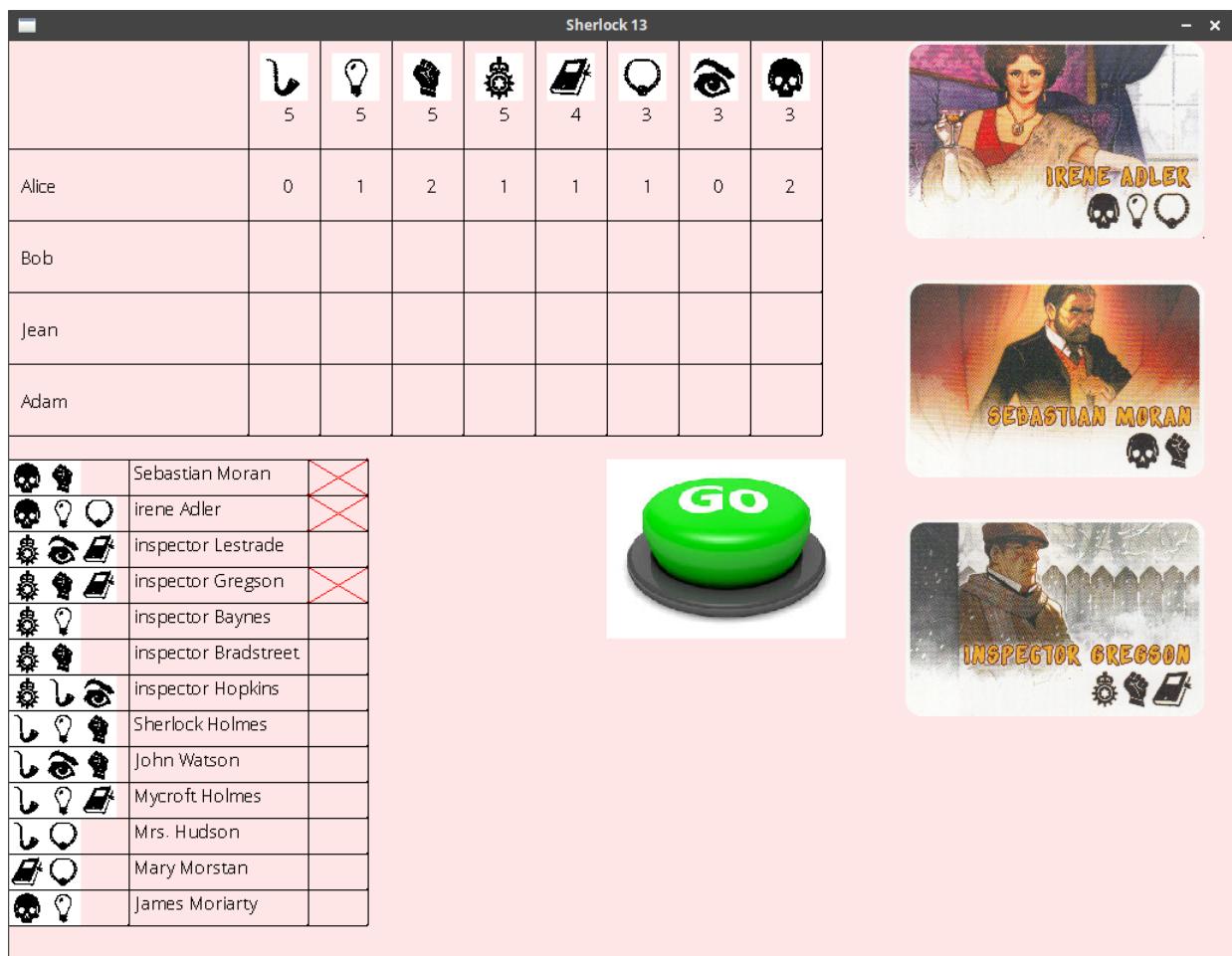


FIGURE 5 – Début de la partie

4. A présent, le but du jeu est d'enquêter au près des adversaires pour tenter de trouver la carte coupable. Le joueur dispose de deux moyens comme vu auparavant :

- Enquête globale :

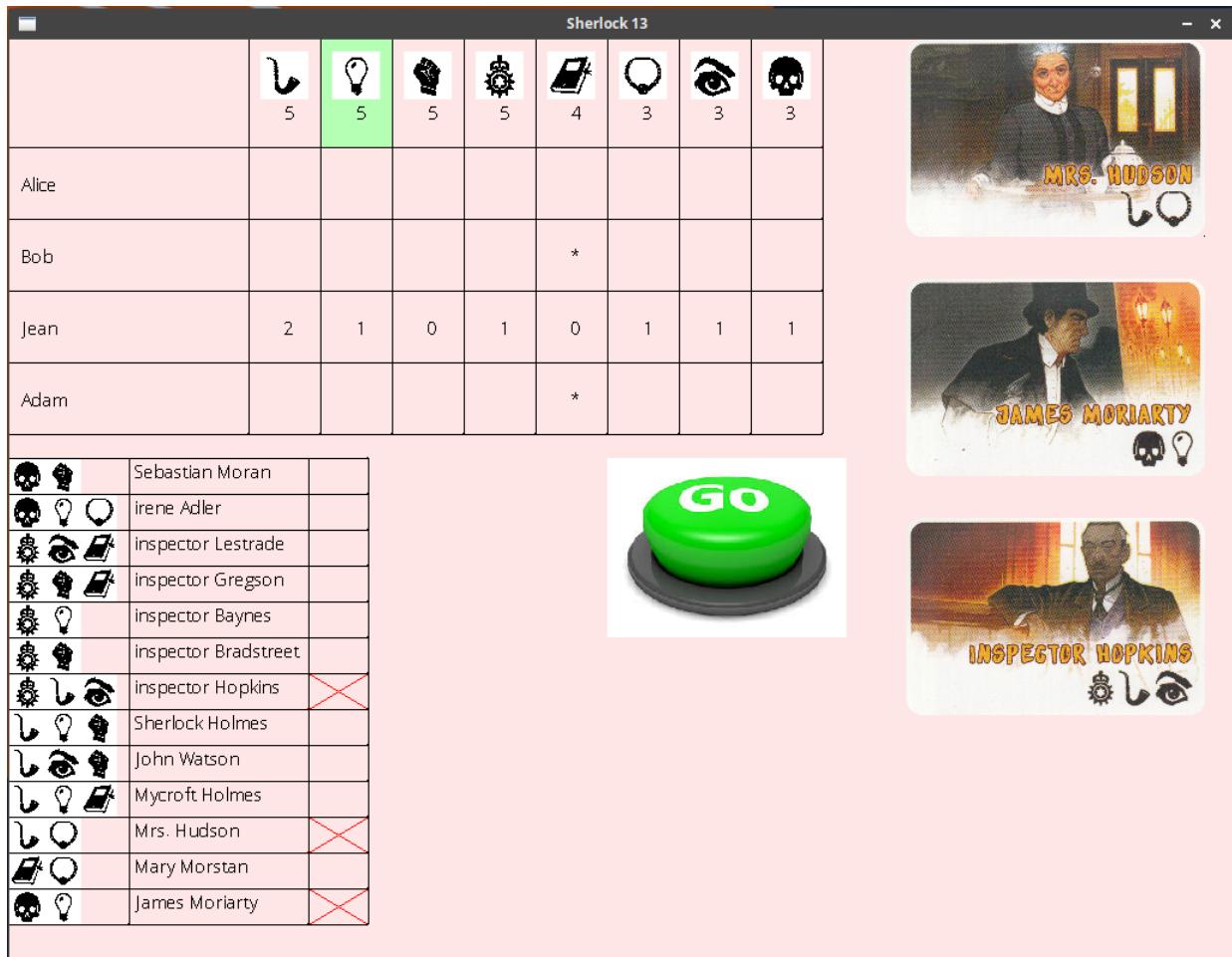


FIGURE 6 – Exemple d'enquête globale

On choisit le symbole souhaité (voir couleur verte sur l'ampoule) dans le tableau des symboles puis on demande aux autres joueurs s'ils possèdent ce symbole en cliquant sur le bouton **GO**. Tous les joueurs reçoivent les réponses. Si un joueur possède ce symbole, alors une étoile apparaît dans la case correspondante, 0 sinon.

- Enquête spécifique visant un joueur :



FIGURE 7 – Exemple d'enquête spécifique

On choisit le symbole souhaité (pipe verte) puis le joueur concerné (Alice en rose). Ensuite, on valide en cliquant sur le bouton **GO**. Le joueur désigné doit alors dire le nombre exact de symbole qu'il possède.

5. Comme indiqué plus haut, lors d'une enquête globale, une étoile * apparaît dans le tableau à la case correspondante lorsqu'un joueur possède un symbole. Pour l'enquête spécifique, un chiffre apparaît correspondant au nombre de symbole qu'il possède.

Par exemple :

	Λ	💡	✊	⚙️	📝	⌚	👁️	💀
	5	5	5	5	4	3	3	3
Alice		*						
Bob		*			*			
Jean	2	1	0	1	0	1	1	1
Adam		*			*			

💀💡	Sebastian Moran	
💀💡⌚	Irene Adler	
💀👁️📝	inspector Lestrade	
💀💡📝	Inspector Gregson	
💀💡	inspector Baynes	
💀💡	inspector Bradstreet	
💀💡👁️	inspector Hopkins	✗
Λ💡💀	Sherlock Holmes	
Λ👁️💀	John Watson	
Λ💡📝	Mycroft Holmes	
Λ⌚💀	Mrs. Hudson	✗
📝⌚💀	Mary Morstan	
💀💡	James Moriarty	✗

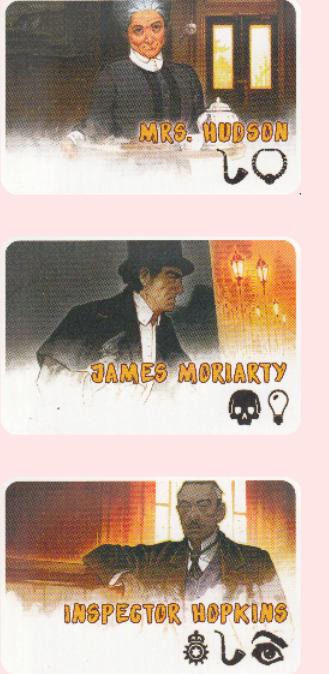


FIGURE 8 – Résultat enquête globale

On remarque ici que tous les joueurs possède le symbole **ampoule**.

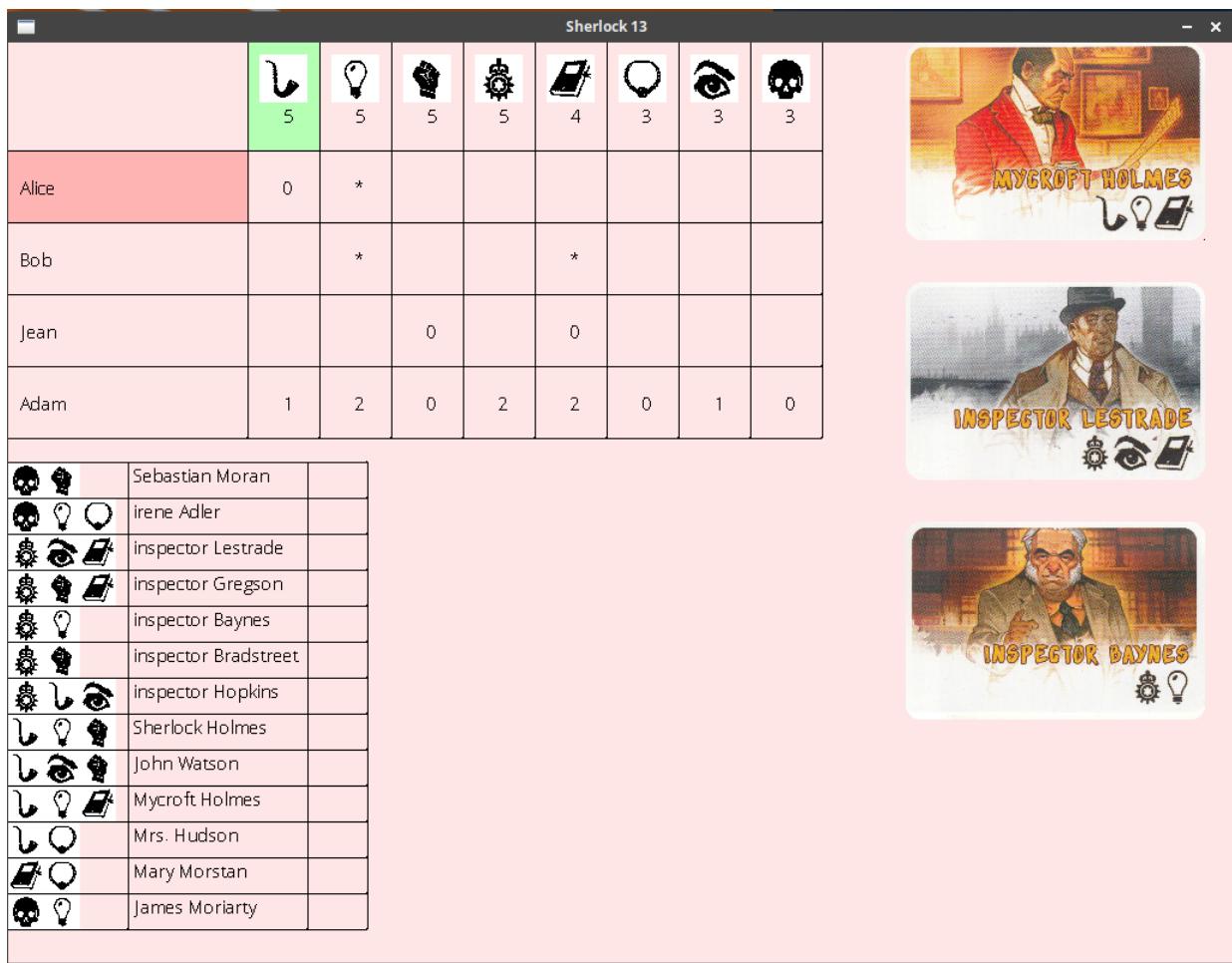


FIGURE 9 – Résultat enquête spécifique

Sur cet exemple, on voit qu'Alice ne possède aucun symbole pipe.

6. Après avoir enquêté, si un joueur pense avoir trouvé le coupable, il peut procéder à une accusation. Pour cela, il sélectionne un personnage en cliquant sur son nom dans la table des suspects (couleur bleu) et puis clique sur **Go** pour l'accuser :

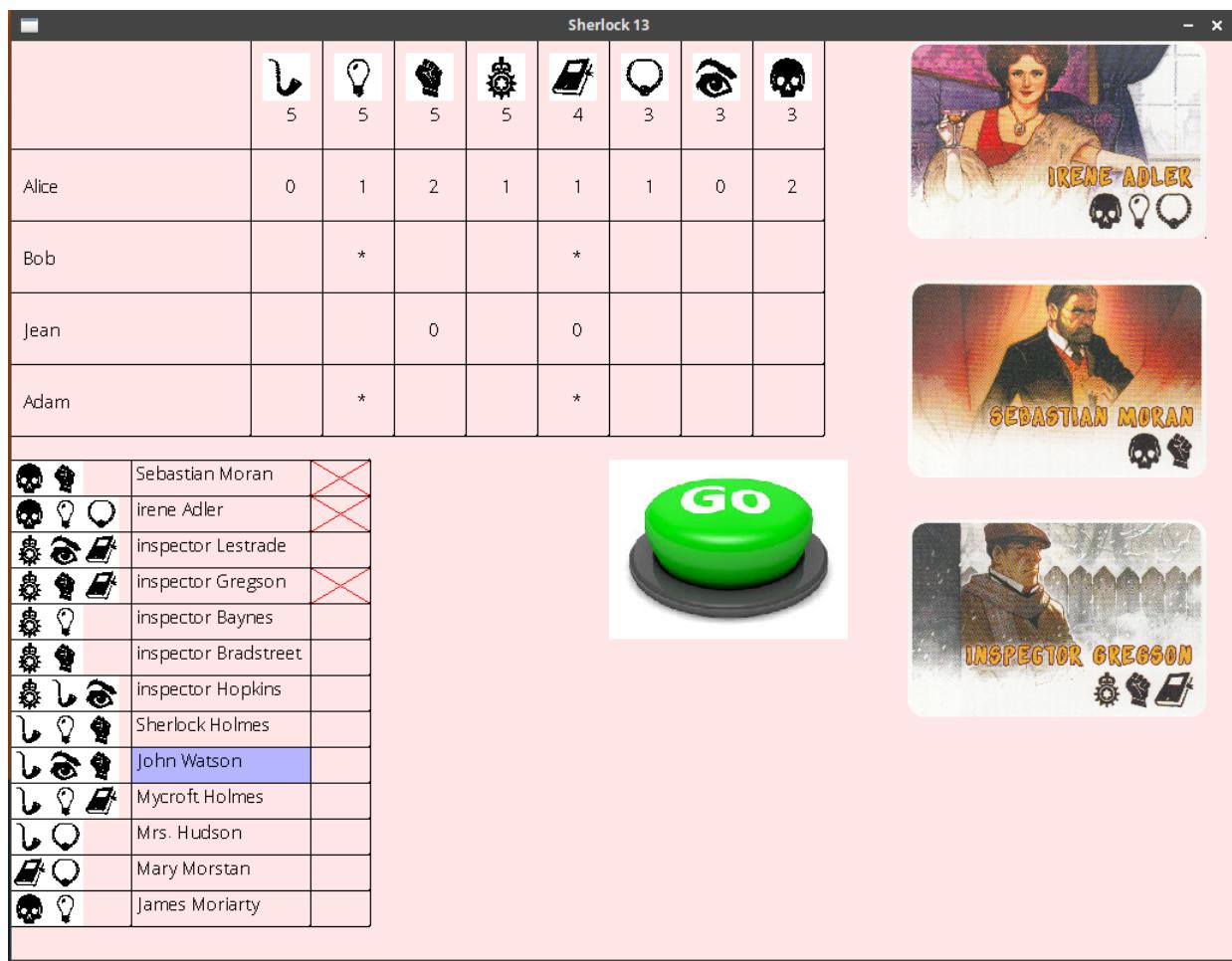


FIGURE 10 – Accusation sur John Watson

8.2 Les améliorations

8.2.1 Les pop-ups

- Si le personnage accusé est bien le coupable, alors le joueur gagne la partie et une fenêtre le félicitant apparaît sur son écran. Les 3 joueurs perdants sont informés du gagnant et du coupable sur une fenêtre différente et la partie est finie !



FIGURE 11 – Fenêtre félicitant le gagnant de la partie

- Si le personnage accusé est innocent, alors le joueur s'est trompé et est éliminé. Une fenêtre lui apprenant son élimination apparaît sur son écran. Les 3 autres joueurs apprennent son élimination via le terminal et un commentaire sonore. Cependant, il continue de répondre aux questions des autres joueurs.

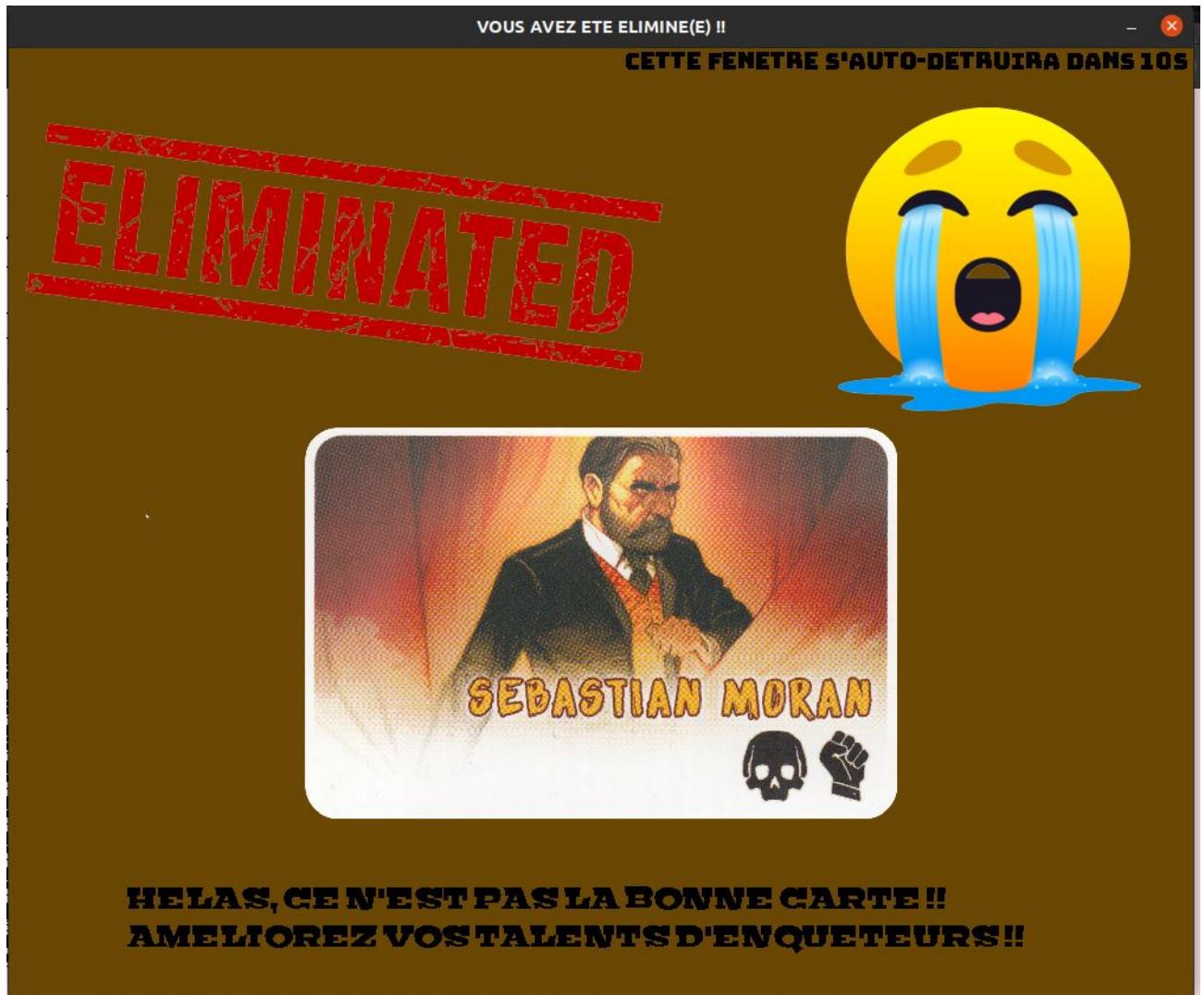


FIGURE 12 – Fenêtre pour un joueur éliminé

- Finalement pour les joueurs perdants :



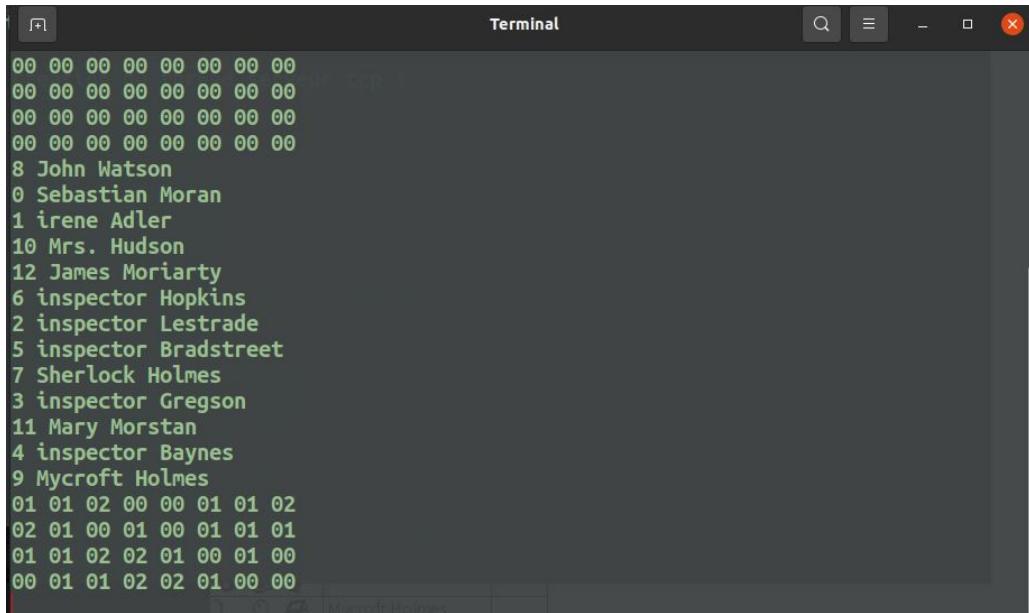
FIGURE 13 – Fenêtre des perdants

8.2.2 Les sons

On ne peut malheureusement présenter les résultats concernant les sons sur le rapport mais durant nos tests, sur une même machine, cela fonctionnait parfaitement. Néanmoins, c'était très désagréable car les sons se chevauchaient car on était sur une même machine. Ainsi, pour une meilleure expérience, il faudra privilégier une simulation sur différentes machines.

8.3 Informations sur les terminaux

- Sur le terminal du serveur, nous pouvons voir les valeurs de tableCartes, les cartes avant/après le mélange, les paquets envoyés/reçus, la connexion des joueurs etc... :



```

Terminal
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 00
8 John Watson
0 Sebastian Moran
1 irene Adler
10 Mrs. Hudson
12 James Moriarty
6 inspector Hopkins
2 inspector Lestrade
5 inspector Bradstreet
7 Sherlock Holmes
3 inspector Gregson
11 Mary Morstan
4 inspector Baynes
9 Mycroft Holmes
01 01 02 00 00 01 01 02
02 01 00 01 00 01 01 01
01 01 02 02 01 00 01 00
00 01 01 02 02 01 00 00

```

FIGURE 14 – Terminal serveur

- Concernant le terminal du client, on peut y remarquer le déroulement de la partie : les joueurs éliminés, le joueur courant, le gagnant etc... :



```

Terminal
Creation du thread serveur tcp !
Votre ID est : 3.
La liste des joueurs est : Adam, Jean, Bob et Alice.
Vos cartes sont : inspector Gregson, Mary Morstan et inspector Baynes.
C'est au tour de Adam de jouer.
Adam a été éliminé !
C'est au tour de Jean de jouer.
C'est au tour de Bob de jouer.
C'est à votre tour de jouer !
C'est au tour de Jean de jouer.
C'est au tour de Bob de jouer.
Bob a gagné ! La carte coupable était : Mycroft Holmes.


```

FIGURE 15 – Terminal client

9 Conclusion

En conclusion, nous pouvons dire que nous avons pris un très grand plaisir à réaliser ce projet. En plus d'être un jeu que nous trouvons amusant, comprendre la manière dont on le programme à travers toutes les notions acquises durant ce module nous rend très satisfaits. Malgré les quelques difficultés rencontrées durant le remplissage des bouts de code manquants, les vidéos de notre enseignant nous ont grandement aidées, ainsi que des recherches sur internet évidemment. En plus de ce qui était demandé, nous avons réussi à ajouter quelques améliorations qui rendent l'expérience de jeu encore meilleure et dont nous sommes très fiers.

De plus, au delà d'avoir réussi à faire fonctionner le jeu correctement, ce sont les compétences acquises que l'on retient le plus, à savoir la compréhension du fonctionnement d'un processeur, des différents appels systèmes ou de la programmation multi-threads. Sans oublier la manipulation des différentes bibliothèques de SDL. Tout ceci vient enrichir et consolider nos acquis informatiques.