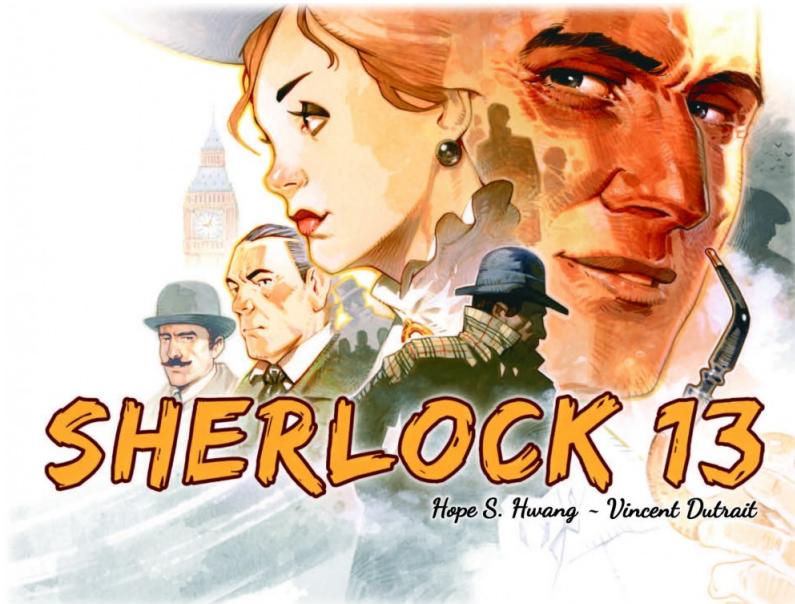


SYSTÈME EXPLOITATION

Projet Sherlock 13



EL ATIA MAJDA

MR. PÊCHEUX

2019-2020

Table des matières

1	Introduction	3
2	Structure de l'ordinateur	3
2.1	Le processeur	3
2.2	La mémoire	3
2.3	Les entrées-sorties	4
3	La notion de cache	4
3.1	Mini-Introduction	4
3.2	Caches de données, caches d'instructions et caches unifiés	5
4	La notion de thread	6
4.1	Une idée de ce que c'est	6
4.2	Cohabitation des threads	7
4.3	Alternatives	7
5	Réseau : Modèle Client-Serveur	7
5.1	Un système à quatre couches	7
5.2	Comment se passe un échange Client - Serveur ?	9
6	La notion de socket : Communication Client-Serveur	10
7	Introduction au jeu Sherlock13	12
8	Règles du jeu	12
9	Comment faire marcher cette interaction client/serveur ?	13
10	Le serveur	14
10.1	Explication des variables	14
10.2	Explication des fonctions	15
10.3	Fonction principale : main	15
11	Le Client	22
11.1	Explication des variables et fonctions	22
11.2	Fonction principale	22
11.3	Synchronisation entre la boucle événementielle et le thread réseau	27
12	Diagramme UML de séquence	28
13	Interface graphique	28

14 Pour aller plus loin...	33
15 Conclusion	33
16 Sources	33

1 Introduction

Dans le cadre de notre formation en MAIN3, nous avons été initiés à la programmation réseau. Le projet Sherlock 13 permet de mettre en pratique diverses notions enseignées telles que l'IPC, les threads, la partie réseau, les appels systèmes...

Un grand merci à Mr.Pêcheux pour avoir pris le temps de m'expliquer certains points par mails !

2 Structure de l'ordinateur

Avant de se précipiter sur l'analyse du code du jeu Sherlock13, il serait judicieux de faire quelques rappels.

Un ordinateur est composé de trois composants importants :

2.1 Le processeur

Il exécute les instructions qui constituent les programmes et le système d'exploitation. Il est composé de plusieurs blocs constructeurs : les unités d'exécution, les fichiers de registres et la logique de contrôle. Le premier bloc contient le dispositif matériel qui exécute les instructions. Dans ce dispositif, on y inclut les sous dispositifs qui passent en revue et décoden les instructions, ainsi que les unités arithmétiques et logiques (UAL) qui réalisent les calculs eux-mêmes.

Il possède un fichier de registres, qui est une petite zone de stockage que le processeur utilise pour stocker des données.

2.2 La mémoire

C'est un réceptacle de stockage pour les données et les programmes utilisés par l'ordinateur. La plupart des ordinateurs possède deux types de mémoire :

- *la mémoire morte* (ROM : Read Only Memory) : stocke un programme qui est exécuté automatiquement par l'ordinateur à chaque fois qu'il est allumé ou réinitialisé (=programme de démarrage). Il donne l'instruction à l'ordinateur de charger son système d'exploitation à partir d'un périphérique.
- *la mémoire vive*(RAM : Random Access Memory) : peut être lue ou écrite. Elle contient les programmes, le système d'exploitation et les données requises par l'ordinateur.

2.3 Les entrées-sorties

Elles contiennent les appareils que l'ordinateur utilise pour communiquer avec le monde extérieur et stocker les données.

3 La notion de cache

3.1 Mini-Introduction

Les caches constituent généralement le ou les niveaux supérieurs de la hiérarchie mémoire et sont toujours construits en SRAM. La principale différence structurelle entre un cache et les autres niveaux de la hiérarchie mémoire tient à ce que les caches sont aidés par un dispositif matériel qui tient registre des adresses mémoires qu'ils contiennent et déplace les données dans ou hors du cache, selon les besoins. Les niveaux inférieurs de la hiérarchie s'appuient généralement sur des fonctions logicielles ou une implémentation hybride (matérielle et logicielle) pour réaliser ces tâches.

Les mémoires caches contiennent généralement un **tableau d'étiquettes** et un **tableau de données**. Le tableau d'étiquettes contient les adresses des données contenues dans le cache, tandis que le tableau de données contient les données elles-mêmes. Le fait de diviser le cache en deux tableaux séparés permet de réduire les temps d'accès, car le tableau d'étiquettes contient généralement beaucoup moins de bits que le tableau de données et peut donc être accédé plus rapidement. Une fois que le tableau d'étiquettes a été accédé, sa sortie doit être comparée à l'adresse de la référence mémoire afin de déterminer si un hit est survenu. La séparation du cache en tableaux de données et d'étiquettes permet d'opérer la détection hit/miss en parallèle avec les recherches dans le tableau de données, ce qui réduit le temps d'accès global.

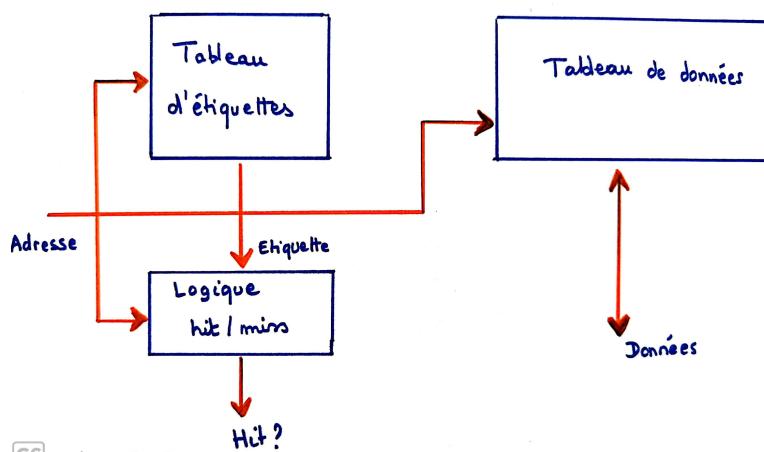


FIGURE 1 – Organigramme du bloc cache

3.2 Caches de données, caches d'instructions et caches unifiés

En expliquant le mécanisme des systèmes mémoires, on a adopté une représentation schématique qui laissait entendre que les instructions et les données se partageaient un même espace dans chaque niveau de hiérarchie mémoire (pour la mémoire principale et virtuelle c'est principalement le cas). Toutefois, dans le cas des caches, les données et les instructions sont souvent stockées dans des caches de données et des caches d'instructions séparés (cf. Figure 2 svp). Cette organisation qui est quelquefois appelée *cache Harvard* ou *architecture Harvard*, est utilisée car elle permet au processeur d'extraire simultanément les instructions du cache d'instructions et les données du cache de données. Quand un cache contient à la fois des instructions et des données, on parle de cache *unifié*.

La séparation entre cache d'instructions et cache de données permet de tirer profit du fait que les programmes ne modifient généralement pas leurs propres instructions. Les caches d'instructions peuvent être alors conçus comme des appareils en lecture seule qui permettent aucune modification des instructions qu'ils contiennent. En conséquence, ces caches peuvent tout simplement se débarasser de tous les blocs qui doivent être évincés sans avoir à les réécrire préalablement dans la mémoire principale, car les données n'auront pas changé depuis le moment où elles ont été amenées dans le cache. De plus, la séparation des caches d'instructions et de données évite les conflits entre les blocs d'instructions et les données qui pourraient être renvoyées aux mêmes emplacements mémoires dans le cas d'un cache unifié.

L'inconvénient de la séparation des caches tient à ce qu'elle complique l'écriture des programmes automofiants...

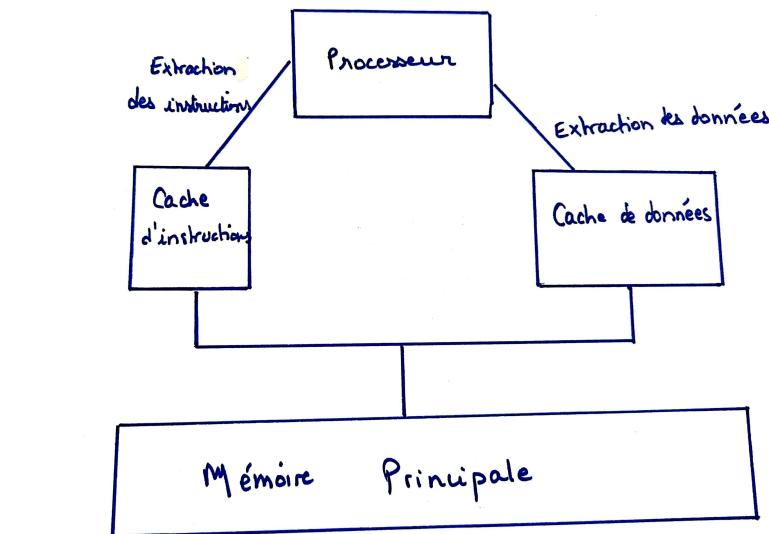


FIGURE 2 – Architecture de cache Harvard

Remarque : Le cache d'instructions sera sensiblement plus petit que son cache de données. Ceci est dû au fait que les instructions d'un programme prennent moins de place en mémoire que ses données. D'ailleurs, la plupart des programmes passent la majorité de leur temps à effectuer des boucles qui utilisent les mêmes instructions plusieurs fois. Du coup, le cache d'instructions du système est plus petit et peut posséder le même taux de hit.

4 La notion de thread

4.1 Une idée de ce que c'est

Un thread est une activité, une portion de code avec des données et une pile. Un processus va héberger plusieurs threads. Chaque thread est caractérisé par une structure de données légères : le registre et la pile.

En effet, si on veut arrêter le fil d'exécution du premier thread afin de réaliser le second, on doit sauvegarder la valeur du registre et le pointeur de pile. Il s'exécute en parallèle de d'autres fonctions, notamment en parallèle du programme principal.

Les threads partagent les mêmes données globales (logique car ils partagent le même espace mémoire !) et peuvent donc s'échanger des données et se synchroniser. Si on souhaite qu'un thread communique avec un autre on n'a pas besoin de faire d'IPC.

Dans le langage de programmation C, la gestion de threads se fait en incluant l'interface de la bibliothèque **pthread.h**. Un thread est représenté par une variable de type **p_thread** et est crée à l'aide de la commande suivante :

`int pthread_create(pthread_t * thread, NULL,void *(*start_routine) (void *), void *arg).` Le premier argument est un pointeur vers un thread, le second ne nous intéresse pas tellement (on peut le mettre à **NULL**), le troisième et le quatrième argument sont chacun un pointeur vers la fonction (donc les instructions que devra exécuter le thread) et les arguments pris en entrée par cette fonction. Si la création du thread fonctionne, cette fonction retourne 0, sinon elle retourne n'importe quelle valeur.

Afin d'éviter de se retrouver avec des processus 'zombies', on utilise une fonction appelée **pthread_join()** qui attend que le thread se termine. C'est-à-dire on sort du processus seulement lorsque TOUS les threads sont terminés. La fonction récupère alors la valeur de retour du thread indiquant si sa terminaison est normale ou forcée (c'est une sorte de prévention contre le mauvais fonctionnement du programme).

4.2 Cohabitation des threads

On s'intéresse au problème de synchronisation. On a vu que les threads peuvent avoir accès en même temps à une même variable et donc ils peuvent la modifier simultanément. Ces multlicitures d'une seule et même variable peuvent donner lieu à des valeurs imprévisibles.

Pour pallier à ce problème, on met en place ce qu'on appelle une **exclusion mutuelle**, c'est-à-dire un **mutex**. C'est un moyen pour synchroniser deux threads. Les mutex sont aussi gérés via le include de l'interface de la librairie du thread. Lorsque l'on connaît la zone qui sera accédée par plusieurs threads, on parle de **zone critique**. On la traite en verrouillant le mutex lors de son utilisation par un thread, grâce à la fonction **pthread_mutex_lock** (**pthread_mutex_t * mutex**). Elle permet de protéger l'accès à la variable du programme afin que tous les threads ne puissent pas y accéder simultanément. On fait une demande exclusif pour accéder à cette variable.

Puis, une fois que le thread a fini de modifier la variable, on débloque le mutex avec la fonction **pthread_mutex_unlock** (**pthread_mutex_t * mutex**). Maintenant que l'accès mémoire est protégée, on peut synchroniser en 'endormant' (sleep()) certains threads puis lorsqu'une certaine condition est respectée, ils peuvent se 'réveiller'. Ces conditions sont représentées par les **struct pthread_cond_t** en C. Ainsi, le thread attend le signal envoyé par un autre thread afin de poursuivre son exécution.

4.3 Alternatives

On peut avoir le même résultat en utilisant des forks mis en place vers les années 80/90. En effet, la fonction fork() permet de réaliser une copie conforme d'un processus afin de pouvoir faire fonctionner plusieurs fonctions en parallèle. Quand les processus fils ont terminé leur travail demandé par le processus père alors ...ils disparaissent. Une fois que tous les processus fils sont partis, le père peut décéder à son tour. Cette méthode est moins utilisée que les threads, ce qui se comprend car les threads sont 1000 fois supérieurs aux forks !

5 Réseau : Modèle Client-Serveur

5.1 Un système à quatre couches

On considère **trois couches** sur la machine : les **applications**, le **système d'exploitation** et le **matériel** (contenant éventuellement des périphériques). Lorsqu'une application sollicite le matériel, elle fait un appel système.

Dans Unix, **TOUT** est fichier.

Accès Fichier

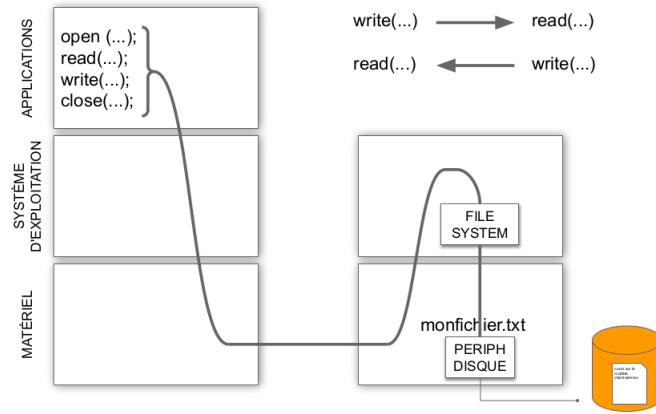


FIGURE 3 – Accès Fichier

Si on souhaite écrire dans un fichier qui se trouve à distance sur une autre machine, on doit étudier la programmation réseau. On s'intéresse plus particulièrement sur l'art et la manière de le faire. On va donc se décaler du système à trois couches vers le système à **quatre couches**. Toutefois dans les cours de programmation, on nous apprend que le **modèle OSI** est constitué de **sept couches**. Mais aucun système informatique n'implémente le système OSI directement !

Modèle OSI – Modèle TCP/IP (unix)

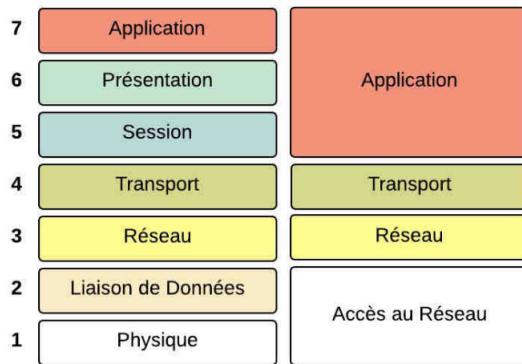


FIGURE 4 – Couches Réseaux

La seule couche qu'on a rajoutée par rapport à notre système à trois couches est la **couche transport** qui joue le rôle de couche intermédiaire. La couche application reste la même, la **couche Réseau** joue le rôle du système d'exploitation qui gère la partie réseau et enfin la **couche Accès au Réseau** qui trouve son équivalence dans la couche matériel du système à trois couches.

5.2 Comment se passe un échange Client - Serveur ?

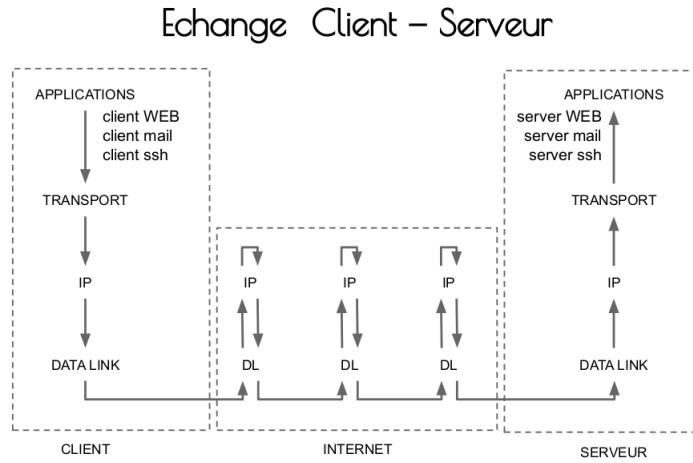


FIGURE 5 – Échange Client - Serveur

On observe bien les quatre couches des deux côtés :

- Applications
- Transport
- Ip \simeq réseau-système exploitation
- Datalink \simeq matériel pour l'émission des bits qu'on souhaite envoyer

Ce qui est intéressant c'est que le client aura l'impression que l'échange se fait horizontalement, i.e un client web aura l'impression de s'adresser à un serveur web etc...Alors qu'en réalité tout se passe de façon vertical (en forme de U) !

Le bloc **INTERNET** permet au message de passer par plusieurs machines intermédiaires (leur couche datalink, ip) puis redescendra pour être réémis jusqu'à une autre machine pour arriver enfin vers la machine de destination.

Le client demande un service et le serveur lui rend ce service puis envoie au client une réponse. Le serveur est toujours disponible. Le client doit donc être concis dans sa requête afin que le serveur lui réponde le plus rapidement possible. Si plusieurs clients émettent des requêtes alors le serveur les traite dans l'ordre qu'elles arrivent. Pour que tout se passe bien, le client doit savoir où se trouve le serveur, i.e son adresse.

Lorsqu'on descend dans le schéma en U, on va venir ajouter des informations supplémentaires à chaque couche.

Par exemple, si à la couche applications je souhaite envoyer des données à la couche suivante, transport, je vais agrémenter mon message précédent par des headers qui permettent de choisir le mode d'encapsulation de données (TCP ou UDP). Arrivé à la couche réseau, on rajoute la couche intermédiaire réseau nécessaire pour que le paquet parte (Ip). Enfin, on envoie un message électronique d'un routeur à un autre via 'Ethernet'.

6 La notion de socket : Communication Client-Serveur

En terme d'appels réalisés par le client et par le serveur, tout cela se passe en parallèle. Si un client souhaite communiquer avec un serveur, il faut que le client et le serveur créent un **socket()** du type adapté (TCP ou UDP, ici dans le jeu Sherlock13 on prendra le protocole TCP). Une fois, que c'est fait, il se passe plus de choses du côté serveur que du côté client. Cela se comprend, maintenant que le serveur a créé une prise, socket, il doit dire au système d'exploitation, que c'est une application réseau et, qu'il souhaite que ses messages arrivent à destination du port n°X.

L'appel système **bind()** associe un numéro de port au socket créé.

Le troisième appel système est **listen()** : le serveur 'écoute' tout ce qui arrive sur le port délivré par bind.

listen() est un appel bloquant. Il attend la connexion d'un client.

connect() : le client va tenter une connexion, en donnant tous les paramètres nécessaires pour le serveur (machine serveur, adresse Ip du serveur, numéro de port du serveur). Si la connexion réussit, un **point de synchronisation** s'établit permettant le déblocage du côté serveur. C'est un passage **simultané** pour le serveur et pour le client. Ce dernier envoie alors une requête au serveur, celui-ci le reçoit **send()** et le traite. Si on est dans une base données, un client attend un message de la part du serveur alors le serveur peut **send()** au client un message.

Enfin, on arrête la connexion établie entre le serveur et le client en faisant **close()**.

Du côté serveur, on a un **socket passif** qui attend des requêtes du client.

Du côté client, il y a un **socket actif** qui va initier la communication.

Voici un schéma récapitulatif :

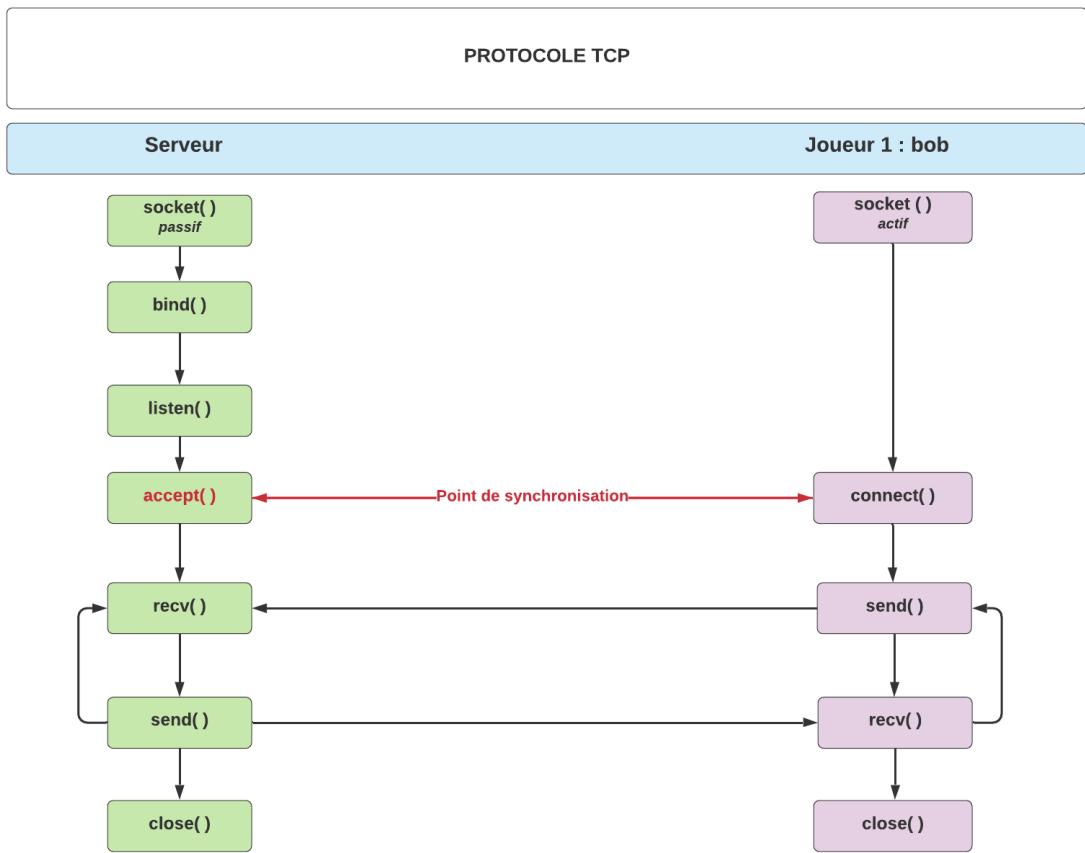


FIGURE 6 – Communication Client-Serveur

7 Introduction au jeu Sherlock13

Le jeu s'articule autour d'un serveur principal (codé en C) et réactif. Celui ci possède deux états :

- État 0 : "on attend que les quatre joueurs se connectent",
- État 1 : "les quatre joueurs sont tous connectés : la partie peut débuter".

Le protocole de communication utilisé est TCP (Transmission Control Protocol). Il est fiable c'est-à-dire que les informations ne seront pas perdues si on arrive à se connecter avec le serveur par contre il est lent (contrairement au protocole UDP qui est rapide mais peu fiable). Concernant les joueurs, on peut les considérer comme des applications qui vont venir 'graviter' autour de ce serveur principal. Il y a donc quatre applications qui vont tourner en concurrence. Pour que le jeu soit réaliste, on a une interaction joueur → serveur. En effet, le joueur peut décider d'envoyer une requête au serveur afin d'effectuer une des trois actions possibles dans le jeu :

- se renseigner sur la présence d'un objet chez les trois autres joueurs (attention les joueurs ne donnent pas le nombre exacte de symboles qu'ils possèdent sauf s'il n'a rien dans ce cas on affiche 0),
- se renseigner sur le nombre exacte d'objet qu'un joueur désigné possède,
- l'accusation c'est-à-dire deviner quelle est la carte coupable.

Sa demande doit être propagée aux autres joueurs via le serveur principal pour qu'ils puissent répondre au joueur courant. Leur réponse sera envoyée au serveur principal qui se chargera de répondre par la suite au joueur courant. Ça devient plus complexe car maintenant le serveur principal est aussi client : il attend les réponses des trois autres joueurs. Par conséquent, ces derniers sont aussi des serveurs !

Ainsi, une fois les réponses venues au serveur principal, ce dernier peut alors envoyer un message personnel au joueur courant. On a une interaction serveur → joueur.

Il y a donc bien des échanges d'informations qui se font entre le serveur (s'il est sollicité) et les quatres clients. Pour que cette communication s'effectue dans de bonnes conditions, les joueurs doivent être munis de l'adresse IP du serveur et du numéro de port du serveur.

8 Règles du jeu

Bienvenue dans l'univers de Sir Conan Doyle ! On dispose de treize cartes. Le serveur va distribuer au hasard trois cartes à chacun des quatre joueurs. Il en restera une de côté, c'est la carte coupable. Le but est de déterminer cette carte là au moyen de trois actions décrites en introduction.

Comme il me restait du temps j'ai eu le loisir d'apporter des modifications au jeu. On a un message de bienvenue qui s'affiche sur les interfaces graphiques. La personne ayant portée une

fausse accusation se voit directement éliminer et on saute son tour durant toute la partie. Une image apparaît alors à la place de bouton *go*, "Vous avez perdu"(en fond transparent). S'il reste un seul joueur à la fin et que tout le reste a été éliminé alors par défaut le joueur restant gagne, quoiqu'il fasse comme **accusation**. Toutefois, j'ai laissé le bouton *go* sur le joueur restant pour qu'il puisse faire les trois actions possibles. Il peut donc demander à tous les joueurs restants la présence de symbole qu'ils possèdent, ou alors désigner un joueur particulier et lui demander le nombre exacte d'objet qu'il possède. Le bouton *go* revient à chaque tour sur le joueur 'survivant'. Jusqu'à ce qu'il fasse une accusation, alors le bouton *go* disparaît et un message s'affiche sur les terminaux de tous les joueurs signalant le (ou la) gagnant(e) et la carte coupable.

9 Comment faire marcher cette intéraction client/serveur ?

Le serveur principale fonctionne à une adresse IP, et à un numéro de port (ex : 32000). On lance le serveur principal avec le numéro de port choisi par exemple :
./serveur 32000 (on se place dans le bon répertoire pour lancer l'exécutable!)

Une fois le serveur crée, les joueurs doivent connaître l'adresse IP ainsi que le port du serveur pour pouvoir se connecter, (la commande ipconfig sur le terminal permet de connaître son adresse IP). On suppose ici que les quatre joueurs sont sur le même ordinateur (ce qui est très peu probable ..!). Le client possède aussi une adresse IP, mais pas de numéro de port ! En effet, le numéro de port est attribué temporairement et dynamiquement par l'appel système **connect()**. Maintenant, pour que le serveur principal sache qu'il peut communiquer avec le serveur du client il faut qu'au moment où le client décide de se connecter il envoie ses informations.

Supposons que le joueur 0 nommé bob souhaite se connecter. Il doit connaître le port qu'il souhaite utiliser pour la connexion. Il devra donc taper la commande suivante :

./sh13 localhost 32000 localhost 32001 bob

Le localhost permet de spécifier que l'adresse IP est au fait l'adresse locale de l'ordinateur courant. Le deuxième localhost permet de spécifier au programme que bob est sur l'ordinateur d'adresse IP locale, donc sur l'ordinateur courant.

Maintenant, si alice décide de rejoindre cette merveilleuse partie elle devra taper la commande :

./sh13 localhost 32000 localhost 32002 alice

Le joueur 2 et 3 ne pourront pas choisir les numéros de port 32001 et 32002 car ils sont déjà réservés respectivement pour bob et alice. Pour un utilisateur étourdi qui tenterait d'accéder à un numéro de port déjà occupé, une erreur s'affichera dans le terminal : **bind error**. Le port étant déjà pris, le socket ne sera pas valide.

Une fois que tous les joueurs sont connectés, la structure de données du serveur principal est mise à jour concernant l'emplacement des serveurs des clients.

```

00 01 01 02 02 01 00 00
Received packet from 127.0.0.1:47428
Data: [C localhost 32001 bob
]

COM=C ipAddress=localhost port=32001 name=bob
0: localhost 32001 bob
id=0
Received packet from 127.0.0.1:47434
Data: [C localhost 32002 alice
]

COM=C ipAddress=localhost port=32002 name=alice
0: localhost 32001 bob
1: localhost 32002 alice
id=1
Received packet from 127.0.0.1:47442
Data: [C localhost 32003 majda
]

COM=C ipAddress=localhost port=32003 name=majda
0: localhost 32001 bob
1: localhost 32002 alice
2: localhost 32003 majda
id=2
Received packet from 127.0.0.1:47452
Data: [C localhost 32004 francois
]

COM=C ipAddress=localhost port=32004 name=francois
0: localhost 32001 bob
1: localhost 32002 alice
2: localhost 32003 majda
3: localhost 32004 francois
id=3

```

FIGURE 7 – Présentation du serveur à l'état 1

10 Le serveur

10.1 Explication des variables

Le code se trouve dans le fichier **serveur.c**. Il est composé d'une structure client très importante qui contient quatre champs : l'adresse **IP** du client (sur 40 caractères), le **numéro de port**, le **nom du client** (sur 40 caractères). Elle définit les 4 clients du jeu, **tcpClients**, à chaque fois qu'un joueur se connecte on rajoute dans cette structure pour l'Id client : 0,1,2,3.

On introduit une variable **fsmServer** (finite state machine). Elle définit l'état dans lequel se trouve le serveur (état 0 ou état 1). Si on est à l'état 1, on passe à la phase de jeu, les messages vont être envoyés entre les différentes parties concernant le jeu et il n'y aura plus d'histoires de connexion.

Une variable **deck** qui contient les indices des 13 cartes du jeu.

tableCartes : matrice de la table des cartes 4 lignes (pour 4 joueurs) et 8 colonnes (pour 8 symboles). Pour faciliter le debogging, on ajoute un tableau **nomcartes** qui contient le noms des différents personnages du jeu.

Puis une variable **joueurCourant** qui permet de déterminer qui doit jouer.

10.2 Explication des fonctions

melangerDeck() : mélange 1000 fois le deck, un échange aléatoire entre deux positions d'indice. En fait, pour chaque indice on associe une carte dans la variable nomcartes et ce qu'on mélange est au final l'ensemble des indices.

createTable() : dans un premier temps tout mon tableau est initialisé à 0 puis on le remplit avec les éléments. On incrémente correctement la table des symboles.

printDeck() : affiche le deck pour savoir que le tableau a bien été mélangé.

printClient() : affiche la structure de données des clients.

findClientByName () : à partir du nom d'un joueur on retrouve son indice.

sendMessageToClient() : permet à l'intérieur du serveur de se faire passer pour un client et d'envoyer un message à quelqu'un d'autre. On y trouve des appels de systèmes fondamentaux tels que **socket()**, **connect()**, **write()**.

broadcastMessage() : on fait quatre appels à la fonction **sendMessageToClient()** en précisant grâce à la structure **tcpClient** si on veut envoyer au client 0,1,2,3.

10.3 Fonction principale : main

On s'assure que les toutes les informations sont bien données, que le client a bien donné son numéro de port etc, sinon on affiche une erreur. On crée le socket, on fait le bind, c'est-à-dire qu'on associe un numéro de port au socket qu'on vient de créer, puis on fait listen() et on entre dans la boucle d'attente du serveur **while(1)**. Entre temps, on fait **printDeck()**, **melangerDeck()**, **createTable()**, **printDeck()**, on met **joueurCourant à 0** car c'est le joueur 0 qui commence à jouer en premier.

printDeck() affiche sur le terminal le numéro des cartes qui correspondent aux noms des personnages, puis on affiche le tableau **tableCartes** le nombre d'objets qu'une carte possède. L'affichage des personnages s'effectue dans l'ordre dans lequel on a entré les personnages dans le **nomcartes**. Quant à l'affichage du tableau, il est initialisé avec des zéros.

On mélange aléatoirement les cartes avec la fonction **melangerDeck()**. Puis, on attribue les trois premiers personnages (affichés sur le terminal serveur) au joueur 0, les trois suivants au joueur 1 etc. La **tableCartes** est mise à jour grâce à **createTable()** avec le nombre d'objets (sept au total, suivant les colonnes) indiqués sur les trois cartes distribuées en fonction du joueur (ligne de la matrice).

La dernière carte correspond au coupable (ici dans cet exemple c'est Mycroft Holmes). On affiche avec **printDeck()** sur le terminal le nouveau tableau avec le nombre d'objets correspondant à chaque joueur.

```

majda@majda-VirtualBox:~/Bureau$ ./serveur
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
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 8 – Initialisation du serveur

Juste après avoir affiché ces renseignements, on entre dans le while(1), l'appel système accept() est bloquant car il attend la connexion d'un client (cf. **6.La notion de socket : Communication Client-Serveur** svp). S'il se connecte pour envoyer un message, on récupère un **nouveau file descriptor de socket**. Puis, on lit le message que le client a envoyé et, on affiche les informations.

On utilise la variable **fsmServer** pour savoir dans quel état est le serveur. Si **fsmServer=0**, on regarde le message que le client envoie, s'il commence par '**C**', il y a une connexion donc on récupère dans le buffer l'adresse Ip du client, le numéro de port du client et le nom du client. On remplit la table **tcpClient** au bon endroit pour chaque client, on incrémente **nbClients** car il y a un joueur en plus qui vient de se connecter.

On cherche grâce **findClientByName()** l'identifiant correspondant à la personne qui vient de se connecter et le serveur envoie un message personnel, grâce à la fonction utilitaire **sendMessageToClient()** pour lui communiquer son Id, via le message '**I 0**'. La fonction prend en argument l'adresse IP du serveur sous forme de chaîne de caractères, le numéro de port du serveur et le message que le client veut envoyer. Puis, on fait un **write()** qui permet d'envoyer au socket le message entré en argument suivi d'un saut à la ligne. Pour recevoir le message, le serveur doit appliquer l'appel système **read()**. Puis, en récupérant toutes les données dans la table **tcpClients** on lui envoie une reply .

On envoie un deuxième message pour mettre à jour la liste des joueurs connectés.

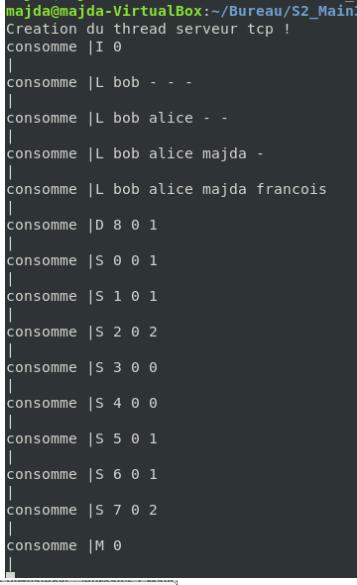
Un fois que le nombre de clients vaut 4, **fsmServer=1** et on a le message '**L bob alice majda francois**' (avec le nom de TOUS les joueurs) qui s'affiche.

Puis, le serveur distribue alors les trois cartes pour chacun des quatre joueurs. Ces derniers reçoivent un **message personnel de type 'D X Y Z'**. D pour Deal et X,Y,Z, trois cartes distribuées au joueur, ici bob a reçu les cartes 8 (John Watson),0 (Sebastian Moran) et 1 (Irene Adler) (cf Figure 8 9).

Puis, sur leur terminal respectif, un message de **type 'S'** (pour Symboles plutôt que V) s'affiche. L'interface graphique est mise à jour avec les valeurs des symboles (pipe, ampoule, poing...). Sur la Figure 6, le message "**consomme | S 0 0 1**" (objet, joueur, nombre), signifie que l'objet 0, la pipe, est détenu par le joueur n°0, bob, et il en possède 1. (on peut vérifier ses informations sur l'interface graphique et c'est juste cf.Figure 9 !)

Puis, on envoie un message à tout le monde pour dire quel est le joueur courant : ici c'est le joueur n°0, car c'est le début de la partie. C'est un message broadcast c'est-à-dire que tous les autres joueurs sont prévenus de la personne qui doit jouer. La fonction **broadcastMessage()** parcourt la liste des clients (`tcpClients`) et applique la fonction **SendMessageToClient()** à chaque joueur. Le message "**consomme | M 0**" indique que c'est au joueur n°0 de jouer et il s'avère que c'est bob. Le bouton *go* nous indique aussi qui doit jouer.

Voici un exemple de la présentation du terminal et de l'interface graphique de bob :



```

majda@majda-VirtualBox:~/Bureau/S2_Main$ Creation du thread serveur tcp !
consomme |I 0
|
consomme |L bob - - -
|
consomme |L bob alice - -
|
consomme |L bob alice majda -
|
consomme |L bob alice majda francois
|
consomme |D 8 0 1
|
consomme |S 0 0 1
|
consomme |S 1 0 1
|
consomme |S 2 0 2
|
consomme |S 3 0 0
|
consomme |S 4 0 0
|
consomme |S 5 0 1
|
consomme |S 6 0 1
|
consomme |S 7 0 2
|
consomme |M 0
|

```



The screenshot shows the game's graphical interface. At the top, it says "SDL2 SH13". Below is a grid of player statistics:

	5	5	5	5	4	3	3	3
bob	1	1	2	0	0	1	1	2
alice								
majda								
francois								

Below the grid is a list of characters with their icons:

Sebastian Moran	
Irene Adler	
Inspector Lestrade	
Inspector Gregson	
Inspector Baynes	
Inspector Bradstreet	
Inspector Hopkins	
Sherlock Holmes	
John Watson	
Mycroft Holmes	
Mrs. Hudson	
Mary Morstan	
James Moriarty	

On the right side, there are three cards displayed:

- JOHN WATSON**: Shows John Watson in a suit.
- SEBASTIAN MORAN**: Shows Sebastian Moran in a suit.
- IRENE ADLER**: Shows Irene Adler in a red dress.

A green "GO" button is centered at the bottom. A message "Bienvenue sur Sherlock 13 !" is visible at the bottom left.

FIGURE 9 – Code dans server.c, Terminal et Interface graphique du joueur n°0

On envoie des messages de jeu de type :

'G' : guilty lorsqu'on fait une accusation. Ici la personne qui accuse à tort est éliminée. Un message s'affiche pour tout le monde (sur les terminaux via le **broadcastMessage()**) à la fin de la partie en indiquant le gagnant et la carte coupable. Dans l'exemple ci-dessous, francois qui possède l'id n° 3, a fait l'hypothèse que c'est la carte n° 9 (Mycroft Holmes) qui est la carte coupable. Donc un message '**G 3 9**' s'affiche sur le terminal serveur. Il s'avère que francois ...a raison ! Donc un message s'affiche sur tous les terminaux, indiquant à tout le monde qu'il a gagné.

```
case 'G':
    int accusateur, accusee;
    sscanf(buffer, "G %d %d", &accusateur, &accusee);
    if(accusee==deck[12]){
        winner=10;
        loser=-1;
        //Fin de la partie .C'est gagné.
        char msg_fin_winner[100];
        char msg_fin_guilt[100];

        sprintf(msg_fin_winner,"W Winner is %s !",tcpClients[accusateur].name);
        sprintf(msg_fin_guilt,"G Guilty is %s !", nomcartes[deck[12]]);
        //broadcastMessage("M 8");
        broadcastMessage(msg_fin_winner);
        broadcastMessage(msg_fin_guilt);

        exit(0);
    }
    else{
        //C'est perdu, le joueur doit être éliminé
        nbRestant--;

        //Si tout le monde est éliminé
        if( nbRestant==0){
            char msg_fin_winner[100];
            char msg_fin_guilt[100];

            sprintf(msg_fin_winner,"W Winner is %s !",tcpClients[accusateur].name);
            sprintf(msg_fin_guilt,"G Guilty is %s !", nomcartes[deck[12]]);

            //broadcastMessage("M 8");
            broadcastMessage(msg_fin_winner);
            broadcastMessage(msg_fin_guilt);

            exit(0);
        }

        //Sinon, on élimine l'accusateur
        sendMessageToClient(tcpClients[accusateur].ipAddress,tcpClients[accusateur].port,"M 9");
    }
}

Received packet from 127.0.0.1:47758
Data: [G 3 9
]
|
go! joueur=-1 objet=-1 guilt=9
consomme |W Winner is francois !
|
10
consomme |G Guilty is Mycroft Holmes !
|
```

FIGURE 10 – Code dans server.c, Terminal et Interface graphique du joueur n° 3

'O' : le joueur se renseigne sur un objet en s'adressant à tous les joueurs. Si un joueur possède l'objet, le serveur envoie un message personnel à chacun des joueurs de type '**V joueur_ qui_ possede_ objet, objet, 100**' (si le joueur possède l'objet) **sinon** on a un message de type '**V joueur_ qui_ possede_ pas_ objet, objet, 0**'. Le serveur de son coté affiche sur son terminal un message de type '**O 0 7**'. Dans l'exemple ci-dessous, cela signifie que bob (id n° 0) se renseigne sur l'objet n° 7, le crâne. Seul le joueur n° 1 (alice) en possède, les deux autres n'en n'ont pas. On passe ensuite au joueur suivant (alice) : '**M 1**'.

```

case 'O':
    // Retourne la liste des gens qui ont l'objet demandé
    int demandeur, objet;
    sscanf(buffer,"%d %d",&demandeur,&objet);
    char message[15];
    for (int i=0; i<4; i++){
        if (i==demandeur){
            if (tableCartes[i][objet]==0)
            {
                sprintf(reply, "V %d %d %d",i,objet, tableCartes[i][objet]);
                broadcastMessage(reply);
            }
            else
            {
                sprintf(reply, "V %d %d 100",i,objet);
                for (int j=0; j<4; j++){
                    if (j!=i){
                        sendMessageToClient(tcpClients[j].ipAddress, tcpClients[j].port, reply);
                    }
                }
            }
        }
    }

    // Joueur suivant
    joueurCourant=(joueurCourant+1)%4;
    sprintf(message,"M %d",joueurCourant);
    broadcastMessage(message);
}

3: localhost 32004 francois
id=3
Received packet from 127.0.0.1:47558
Data: [0 0 7
]

consomme |M 0
|
go! joueur=-1 objet=7 guilt=-1
consomme |V 1 7 100
|
consomme |V 2 7 0
|
consomme |V 3 7 0
|
consomme |M 1
|

```

FIGURE 11 – Case 'O' : Demande de crâne du joueur n° 0, bob

'S' : le joueur désigne un joueur particulier et se renseigne sur le nombre exacte d'objets qu'il possède. Puis, on affiche la réponse de cette personne sur tous les terminaux via la fonction **broadcastMessage()**. Ensuite, on fait un deuxième **broadcastMessage()** pour désigner le nouveau joueur courant.

```

case 'S': ;
    // Retourne le nombre d'objet demandé qu'une personne a
    int obj,joueur;
    sscanf(buffer,"%d %d %d",&demandeur,&joueur,&obj);
    char mess[10];
    sprintf(mess,"%d %d %d",obj,joueur,tableCartes[joueur][obj]);
    broadcastMessage(mess);

    //Joueur suivant
    joueurCourant=(joueurCourant+1)%4;
    sprintf(mess,"M %d",joueurCourant);
    broadcastMessage(mess);
k;

```

FIGURE 12 – Case 'S'

'E' : la personne est éliminée, on saute son tour si elle est sollicitée. Puis on avertit tous les joueurs du nouveau joueur courant par l'intermédiaire d'un **broadcastMessage()**. Dans l'exemple ci-dessous, majda, id n° 2, a fait l'hypothèse que le coupable est l'inspecteur Bradstreet...et elle a faux ! Donc majda est éliminée de la partie. Un message '**E**' s'affiche sur le terminal du serveur. Et, un message '**'Vous avez perdu'**' s'affiche sur son interface graphique.

```

case 'E': ;
    //Cas où il faut passer au joueur suivant ci l'autre est éliminé
    char msg[10];
    joueurCourant = (joueurCourant+1)%4;
    sprintf(msg,"M %d",joueurCourant);
    broadcastMessage(msg);
break;

```

Received packet from 127.0.0.1:47994
Data: [E]

	5	5	5	5	4	3	3	3
bob								
alice								
majda	1	1	2	2	1	0	1	0
francois					2			0
Sebastian Moran								
Irene Adler								
Inspector Lestrade								
Inspector Gregson								
Inspector Baynes								
Inspector Bradstreet								
Inspector Hopkins								
Sherlock Holmes								
John Watson								
Mycroft Holmes								
Mrs. Hudson								
Mary Morstan								
James Moriarty								

Vous avez
PERDU

Benvenue sur Sherlock 13 !

FIGURE 13 – Élimination du joueur n° 2 : majda

11 Le Client

Concernant cette section, **cf.sh13.c** svp.

Un client exigeant attend deux choses :

- avoir un affichage graphique qui marche bien
- recevoir à tout moment des messages réseaux en provenance du serveur principal.

On traitera plusieurs aspects très importants :

- comment gérer les évènements graphiques
- comment gérer le côté réseau
- comment synchroniser la boucle graphique avec le thread.

On a les includes des interfaces des bibliothèques (cf.SDL).

11.1 Explication des variables et fonctions

On a un thread dont l'identifiant est **thread_server_tcp**. On va avoir besoin d'une fonction thread avec le prototype suivant **void *fn_server_tcp (void *arg)**. On y trouve comme expliqué à la partie **4. La notion de thread** les appels de systèmes fondamentaux comme **socket()**, **bind()**, **listen()**, une boucle infinie **while(1)** avec un **accept()** et un **read()**.

Une variable **volatile int synchro** qu'on expliquera plus tard

J'ai ajouté une variable **elimine** qui nous indique si le joueur est interdit de jouer ou non (suite à une fausse accusation).

sendMessageToServer() : envoie un message à destination du serveur principal. Il prend comme arguments l'adresse Ip , le numéro de port et le message du serveur.

On y trouve les appels systèmes fondamentaux tels que **socket**, **connect()**, **write()** puis **close()**. À chaque fois que le client veut envoyer un message, il rétablira une nouvelle connexion.

11.2 Fonction principale

main : on initialise la SDL , on crée la Window, le renderer, les surfaces, les textues correspondantes, on ouvre si besoin les polices de caractères. Le renderer gère les double buffer (un buffer qu'on affiche et un autre dans lequel on met à jour (REDRAW)). Puis, on crée le thread avec **ret=pthread_create (&thread_server_tcp_id, NULL, fn_server_tcp,NULL)**

qui représente le **server TCP**. Le troisième paramètre contient le pointeur vers la fonction qui représente ce qu'on veut que le thread exécute.

Boucle des évènements graphiques : On commence par regarder s'il y a des évènements (ex cliquer sur la souris). En fonction de là où on a cliqué sur l'écran, on envoie un message en destination du serveur principal.

```

case SDL_MOUSEBUTTONDOWN:
    SDL_GetMouseState( &mx, &my );
    //printf("mx=%d my=%d\n",mx,my);
    if ((mx<200) && (my<50) && (connectEnabled==1))
    {
        sprintf(sendBuffer,"C %s %d %s",gClientIpAddress,gClientPort,gName);

        // Creation du message retour
        sendMessageToServer(gServerIpAddress, gServerPort, sendBuffer);
        connectEnabled=0;
    }
    else if ((mx>=0) && (mx<200) && (my>=90) && (my<330))
    {
        joueurSel=(my-90)/60;
        guiltSel=-1;
    }
    else if ((mx>=200) && (mx<680) && (my>=0) && (my<90))
    {
        objetSel=(mx-200)/60;
        guiltSel=-1;
    }
    else if ((mx>=100) && (mx<250) && (my>=350) && (my<740))
    {
        joueurSel=-1;
        objetSel=-1;
        guiltSel=(my-350)/30;
    }
    else if ((mx>=250) && (mx<300) && (my>=350) && (my<740))
    {
        int ind=(my-350)/30;
        guiltGuess[ind]=1-guiltGuess[ind];
    }
    else if ((mx>=500) && (mx<700) && (my>=350) && (my<450) && (goEnabled==1))
    {
        printf("go! joueur=%d objet=%d guilt=%d\n",joueurSel, objetSel, guiltSel);
        if (guiltSel!=-1)
        {
            sprintf(sendBuffer,"G %d %d",gId, guiltSel);
            sendMessageToServer(gServerIpAddress, gServerPort, sendBuffer);
        }
        else if ((objetSel!=-1) && (joueurSel==-1))
        {
            sprintf(sendBuffer,"O %d %d",gId, objetSel);
            sendMessageToServer(gServerIpAddress, gServerPort, sendBuffer);
        }
        else if ((objetSel!=-1) && (joueurSel!=-1))
        {
            sprintf(sendBuffer,"S %d %d %d",gId, joueurSel,objetSel);
            sendMessageToServer(gServerIpAddress, gServerPort, sendBuffer);
        }
    }
}

```

FIGURE 14 – Boucle des évènements graphiques

Si on ne sélectionne ni une colonne (symbole), ni une ligne (joueur ou coupable) on reset tout c'est-à-dire qu'on met **joueurSel = -1**, **objetSel = -1** et **guiltSel = -1**.

Pour gérer la partie réseau, si on met un **accept ()** on **brisera la dynamique de la boucle graphique** (car c'est un appel bloquant).

Par conséquent, après avoir testé tous les évènements graphiques, on regarde s'il n'y a pas une connexion d'un client qui se fait. Y a t-il un évènement réseau ? Si la variable **volatile synchro=1**, on vérifie le mutex :

```

if (synchro==1)
{
    pthread_mutex_lock( &mutex );
    printf("consomme |%s|\n",gbuffer);
    switch (gbuffer[0])

```

FIGURE 15 – Boucle évènements graphiques

On a donc reçu un message, on va donc consulter le gbuffer (variable globale) remplit par le thread réseau. Et, on regarde quels types de messages nous a été envoyés.

- **Message 'I'** : permet au joueur d'avoir identifiant
- **Message 'L'** : Liste des joueurs connectés, elle est mise à jour sur l'interface graphique de tous les joueurs
- **Message 'D'** : Deal. Une fois que les quatre joueurs sont connectés, on distribue les trois cartes.

```

// Message 'I' : le joueur reçoit son Id
case 'I':
    sscanf(gbuffer,"I %d",&gId);
    break;
// Message 'L' : le joueur reçoit la liste des joueurs
case 'L':
    sscanf(gbuffer,"L %s %s %s %s",gNames[0],gNames[1],gNames[2],gNames[3]);
    break;
// Message 'D' : le joueur reçoit ses trois cartes
case 'D':
    sscanf(gbuffer,"D %d %d %d",&b[0],&b[1],&b[2]);
    break;

```

FIGURE 16 – Messages 'I','L','D'

- **Message 'M'** : Chaque joueur reçoit le numéro du joueur courant. Si le numéro du joueur courant correspond à son identifiant, on affiche le bouton *go* sur l'interface du joueur en mettant **goEnabled à 1**. Le bouton *go* s'affiche et le joueur courant peut jouer. Si la personne est éliminée, on la saute. On affiche une image ("Vous avez perdu") à la place du bouton *go*. Si c'est au tour du joueur suivant de jouer (s'il n'a pas été éliminé), on enlève le bouton *go* de l'interface du joueur précédent. Si c'est la fin de la partie, alors on enlève le bouton *go*.

```

// Message 'M' : le joueur reçoit le n° du joueur courant
// Cela permet d'affecter goEnabled pour autoriser l'affichage du bouton go
case 'M':
    int chiffre=-1;
    sscanf(gbuffer, "M %d", &chiffre);
    if(chiffre==10){
        elimine=0;
        goEnabled=0;

        // Je passe au joueur suivant
        sendMessageToServer(gServerIpAddress, gServerPort, "E");
    }

    // Si je me suis trompé, je suis interdit !
    else if(chiffre==9){

        elimine=1;
        loserEnabled=1;
        goEnabled=0;

        // Je passe au joueur suivant
        sendMessageToServer(gServerIpAddress, gServerPort, "E");
    }

    // Suis-je sollicité ?
    goEnabled = (chiffre==gId);

    //Si on me sollicite alors que je suis interdit :
    if(goEnabled && elimine){

        // Je ne joue pas et je passe à la suite
        goEnabled=0;
        sendMessageToServer(gServerIpAddress, gServerPort, "E");
    }
}
break;

```

FIGURE 17 – Message 'M'

— **Message 'V'** : indique la présence (ou non) d'un objet précis détenu par un joueur particulier. En effet, lorsque le joueur courant fait la demande, le serveur lui répond via le if et on affiche sur l'interface graphique une '*' quand les joueurs ont l'objet en question, 0 sinon. Toutes les informations sont affichées sur les terminaux et les interfaces graphiques des joueurs. Ce qui est compréhensible dans la mesure où dans un jeu 'réel', les autres joueurs entendent les réponses données. Toutefois aucune information détenue par le demandeur n'est divulguée.

<pre> case 'V': int i,obj,val; sscanf(gbuffer+2, "%d %d %d", &i, &obj, &val); if (tableCartes[i][obj]==-1 tableCartes[i][obj]==100){ tableCartes[i][obj]=val; } break; </pre>	<pre> consomme M 0 go! joueur=-1 objet=7 guilt=-1 consomme V 1 7 100 consomme V 2 7 0 consomme V 3 7 0 consomme M 1 </pre>
--	---

FIGURE 18 – Terminal de bob 'I 0'

On remarque ici que bob (id n° 0) s'est renseigné sur l'objet n° 7 : le crâne. Seul l'identifiant 1 (alice) en possède '**V 1 7 100**', majda et francois n'en possède pas d'où le 0 à la fin du message 'V'.

— **Message 'S'** : affiche le nombre exacte d'un symbole précis détenu par une personne désignée.

```
majda@majda-VirtualBox:~/Bureau/S2_Main3/
Creation du thread serveur tcp !
consomme |I 1
|
consomme |L bob alice - -
|
consomme |L bob alice majda -
|
consomme |L bob alice majda francois
|
consomme |D 10 12 6
|
consomme |S 0 1 2
|
consomme |S 1 1 1
|
consomme |S 2 1 0
|
consomme |S 3 1 1
|
consomme |S 4 1 0
|
consomme |S 5 1 1
|
consomme |S 6 1 1
|
consomme |S 7 1 1
|
consomme |M 0
|
consomme |V 2 7 0
|
consomme |V 3 7 0
|
consomme |M 1
|
go! joueur=3 objet=4 guilt=-1
consomme |S 4 3 2
|
consomme |M 2
|
k;
```

FIGURE 19 – Terminal d'Alice 'I 1'

Sur cet exemple, on voit '**S 4 3 2**' (tout en bas), cela signifie qu'alice a demandé combien de symbole n° 4 (cahier) à francois (id n° 3), et il en a 2.

Une fois qu'on a traité le message, on met **synchro à 0** pour signifier qu'on a traité le message. Puis on débloque le mutex.

```
synchro=0;
|
|     pthread_mutex_unlock( &mutex );
}
```

FIGURE 20 – Traitement du message réseau, synchro à zéro

11.3 Synchronisation entre la boucle évènementielle et le thread réseau

On va maintenant s'intéresser à une variable très importante. L'indication **volatile** pour désigner **synchro**, permet l'accès systématique à la véritable valeur de la variable **synchro**. Avec la notion de cache (cf **3. La notion de cache**), on va avoir de multiples occurrences de **synchro** dans le cache, dans la vraie mémoire et/ou dans le cache d'un autre processseur... Il se peut que le compilateur fasse des optimisations... Pis encore, il se peut que différents threads aient modifiés la valeur de **synchro**! En plus, d'apparaître dans la boucle des évènements graphiques, elle y est aussi dans le thread réseau.

```
void *fn_serveur_tcp(void *arg)
{
    int sockfd, newsockfd, portno;
    socklen_t clilen;
    struct sockaddr_in serv_addr, cli_addr;
    int n;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd<0)
    {
        printf("socket error\n");
        exit(1);
    }

    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = gClientPort;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    if(bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
    {
        printf("bind error\n");
        exit(1);
    }

    listen(sockfd,5);
    clilen = sizeof(cli_addr);
    while (1)
    {
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
        if (newsockfd < 0)
        {
            printf("accept error\n");
            exit(1);
        }

        bzero(gbuffer,256);
        n = read(newsockfd,gbuffer,255);
        if (n < 0)
        {
            printf("read error\n");
            exit(1);
        }
        //printf("%s",gbuffer);

        pthread_mutex_lock( &mutex );
        synchro=1;
        pthread_mutex_unlock( &mutex );

        while (synchro);
    }
}
```

FIGURE 21 – Thread Réseau

À la fin, on met **synchro à 1**, il y a un nouveau message réseau qui vient d'arriver, il faut le traiter. Avant de recevoir un autre message réseau, on attend que **synchro** passe à zéro d'où le **while (synchro)**, avant de revenir sur le `while(1)`.

Cela explique la **synchronisation entre la boucle des évènements graphiques et le serveur TCP**.

12 Diagramme UML de séquence

Voici un schéma récapitulatif de tout ce qui a été cité :

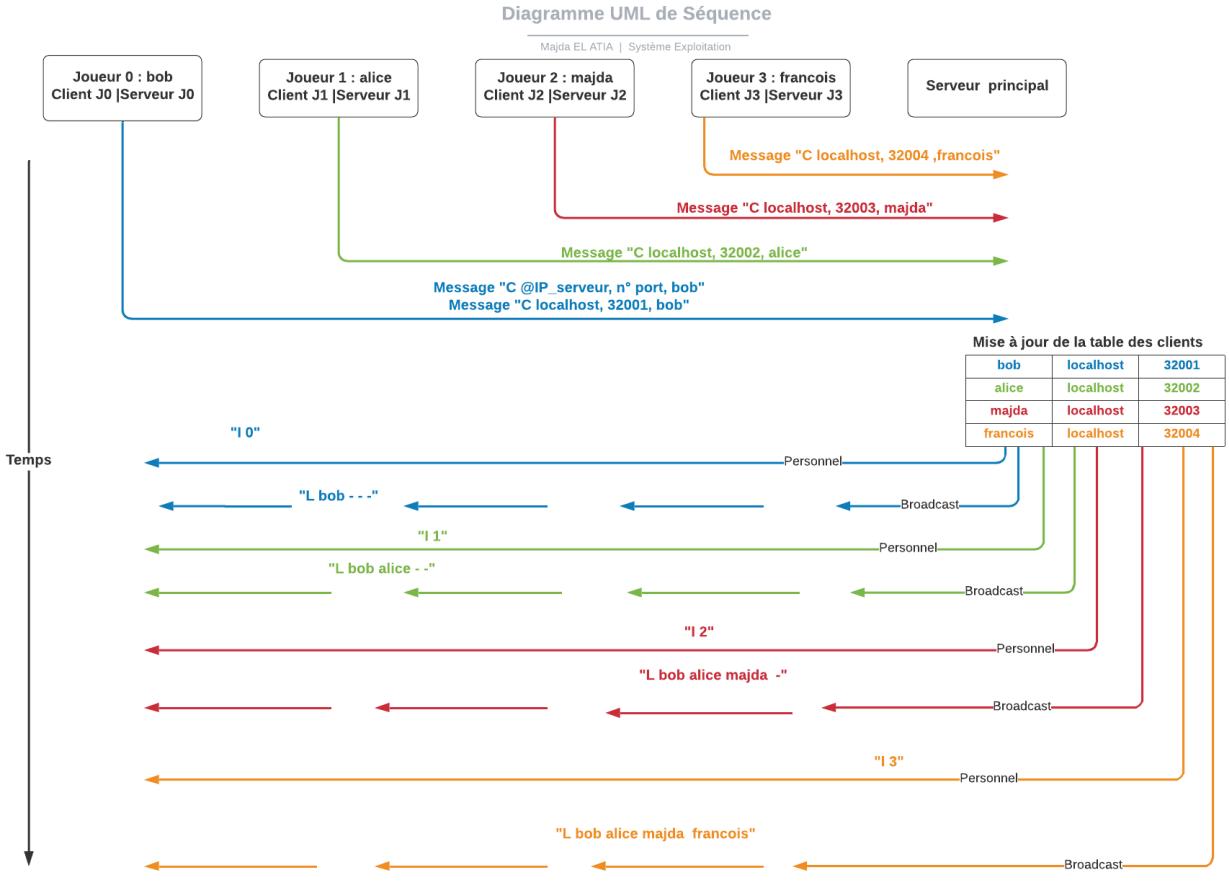


FIGURE 22 – Diagramme UML de séquence

13 Interface graphique

Dans la suite du code **sh13.c**, on refait la mise à jour de l'affichage.
Le bout de code nous a été fourni, il est intéressant de l'analyser. On utilise la bibliothèque **SDL**. On initialise tout le système graphique avec **SDL_Init(SDL_INIT_VIDEO)**, puis **TTF_Init()** (true type font, pour les polices de caractères). J'ai ajouté une image pour le perdant (celui qui a accusé à tort). J'ai modifié la couleur de la fenêtre en gris et parce qu'on est poli, on souhaite au joueur un message (en rose) de bienvenue '**Bienvenue sur le jeu de Sherlock 13 !**' sur chacune des interfaces graphiques des quatre joueurs.

```

SDL_Rect dstrect_grille = { 512-250, 10, 500, 350 };
SDL_Rect dstrect_image = { 0, 0, 500, 330 };
SDL_Rect dstrect_imagine = { 0, 340, 250, 330/2 };

SDL_SetRenderDrawColor(renderer, 230, 230, 230, 230);
SDL_Rect rect = {0, 0, 1024, 768};
SDL_RenderFillRect(renderer, &rect);

if (joueurSel!=-1)
{
    SDL_SetRenderDrawColor(renderer, 255, 180, 180, 255);
    SDL_Rect rect1 = {0, 90+joueurSel*60, 200 , 60};
    SDL_RenderFillRect(renderer, &rect1);
}

if (objetSel!=-1)
{
    SDL_SetRenderDrawColor(renderer, 180, 255, 180, 255);
    SDL_Rect rect1 = {200+objetSel*60, 0 , 60 , 90};
    SDL_RenderFillRect(renderer, &rect1);
}

if (guiltSel!=-1)
{
    SDL_SetRenderDrawColor(renderer, 180, 180, 255, 255);
    SDL_Rect rect1 = {100, 350+guiltSel*30, 150 , 30};
    SDL_RenderFillRect(renderer, &rect1);
}

{
    SDL_Rect dstrect_pipe = { 210, 10, 40, 40 };
    SDL_RenderCopy(renderer, texture_objet[0], NULL, &dstrect_pipe);
    SDL_Rect dstrect_ampoule = { 270, 10, 40, 40 };
    SDL_RenderCopy(renderer, texture_objet[1], NULL, &dstrect_ampoule);
    SDL_Rect dstrect_poing = { 330, 10, 40, 40 };
    SDL_RenderCopy(renderer, texture_objet[2], NULL, &dstrect_poing);
    SDL_Rect dstrect_couronne = { 390, 10, 40, 40 };
    SDL_RenderCopy(renderer, texture_objet[3], NULL, &dstrect_couronne);
    SDL_Rect dstrect_carnet = { 450, 10, 40, 40 };
    SDL_RenderCopy(renderer, texture_objet[4], NULL, &dstrect_carnet);
    SDL_Rect dstrect_collier = { 510, 10, 40, 40 };
    SDL_RenderCopy(renderer, texture_objet[5], NULL, &dstrect_collier);
    SDL_Rect dstrect_oeil = { 570, 10, 40, 40 };
    SDL_RenderCopy(renderer, texture_objet[6], NULL, &dstrect_oeil);
    SDL_Rect dstrect_crane = { 630, 10, 40, 40 };
    SDL_RenderCopy(renderer, texture_objet[7], NULL, &dstrect_crane);
}

```

FIGURE 23 – Thread Réseau

On affiche la grille des images, puis la couleur de la Window. Dans les if, lorsqu'on clique sur un joueur, un objet ou un coupable la case est coloriée suivant le code RGB. Puis, on dit où on souhaite que les images des objets s'affichent.

La fonction **SDL_SetRenderDrawColor(SDL_Renderer* renderer, Uint8 r, Uint8 g, Uint8 b, Uint8 a)**) permet de redessiner l'image perpétuellement comme si on repartait de zéro. Ensuite, on fait un **SDL_RenderFillRect(SDL_Renderer* renderer, const SDL_Rect* rect)**, on couvre la fenêtre avec la couleur de fond sur toute la grandeur de l'écran. On ramène les images dans un rectangle qui a une largeur de X et une hauteur de Y. Avec la fonction **SDL_RenderCopy(SDL_Renderer* renderer, SDL_Texture* texture, const SDL_Rect* srcrect, const SDL_Rect* dstrect)**, connaissant le renderer et le pointeur vers la texture, je vais afficher l'image dans la fenêtre globale à la position indiquée par le rectangle précédent. C'est exactement la même logique pour tout le reste.

```

SDL_Color col1 = {0, 0, 0};
for (i=0;i<8;i++)
{
    SDL_Surface* surfaceMessage = TTF_RenderText_Solid(Sans, nbobjets[i], col1);
    SDL_Texture* Message = SDL_CreateTextureFromSurface(renderer, surfaceMessage);

    SDL_Rect Message_rect; //create a rect
    Message_rect.x = 230+i*60; //controls the rect's x coordinate
    Message_rect.y = 50; // controls the rect's y coordinate
    Message_rect.w = surfaceMessage->w; // controls the width of the rect
    Message_rect.h = surfaceMessage->h; // controls the height of the rect

    SDL_RenderCopy(renderer, Message, NULL, &Message_rect);
    SDL_DestroyTexture(Message);
    SDL_FreeSurface(surfaceMessage);
}

for (i=0;i<13;i++)
{
    SDL_Surface* surfaceMessage = TTF_RenderText_Solid(Sans, nbnoms[i], col1);
    SDL_Texture* Message = SDL_CreateTextureFromSurface(renderer, surfaceMessage);

    SDL_Rect Message_rect;
    Message_rect.x = 105;
    Message_rect.y = 350+i*30;
    Message_rect.w = surfaceMessage->w;
    Message_rect.h = surfaceMessage->h;

    SDL_RenderCopy(renderer, Message, NULL, &Message_rect);
    SDL_DestroyTexture(Message);
    SDL_FreeSurface(surfaceMessage);
}
//Parce qu'on est poli on met un message de bienvenue !

char gbuffer3[256];
SDL_Color col3 = {189,116,236};
for (i=0;i<4;i++){
    sprintf(gbuffer3,"Bienvenue sur Sherlock 13 !");
    SDL_Surface* surfaceMessage = TTF_RenderText_Solid(Sans, gbuffer3, col3);
    SDL_Texture* Message = SDL_CreateTextureFromSurface(renderer, surfaceMessage);

    SDL_Rect Message_rect; //create a rect
    Message_rect.x = 500; //controls the rect's x coordinate
    Message_rect.y = 600; // controls the rect's y coordinate
    Message_rect.w = surfaceMessage->w; // controls the width of the rect
    Message_rect.h = surfaceMessage->h; // controls the height of the rect

    SDL_RenderCopy(renderer, Message, NULL, &Message_rect);
    SDL_DestroyTexture(Message);
    SDL_FreeSurface(surfaceMessage);
}

```

FIGURE 24 – Création du message de Bienvenue

On met les objets affichés dans une grille, puis dans le second for, on affiche les joueurs dans la seconde grille. On crée une surface avec **SDL_CreateTextureFromSurface (renderer, surfaceMessage)**, où surfaceMessage est un **pointeur vers la surface créée**. On a un objet texture sur lequel on récupère un pointeur.

On crée une texture, un renderer pour pouvoir afficher le message. Attention, on affiche des **SDL textures et non pas des **SDL surfaces**** !

```

        for (i=0;i<4;i++)
            for (j=0;j<8;j++)
            {
                if (tableCartes[i][j]!=-1)
                {
                    char mess[10];
                    if (tableCartes[i][j]==100)
                        sprintf(mess,"%*");
                    else
                        sprintf(mess,"%d",tableCartes[i][j]);
                    SDL_Surface* surfaceMessage = TTF_RenderText_Solid(Sans, mess, col1);
                    SDL_Texture* Message = SDL_CreateTextureFromSurface(renderer, surfaceMessage);

                    SDL_Rect Message_rect;
                    Message_rect.x = 230+i*60;
                    Message_rect.y = 110+i*60;
                    Message_rect.w = surfaceMessage->w;
                    Message_rect.h = surfaceMessage->h;

                    SDL_RenderCopy(renderer, Message, NULL, &Message_rect);
                    SDL_DestroyTexture(Message);
                    SDL_FreeSurface(surfaceMessage);
                }
            }

        // Sebastian Moran
    {
        SDL_Rect dstrect_crane = { 0, 350, 30, 30 };
        SDL_RenderCopy(renderer, texture_objet[], NULL, &dstrect_crane);
    }
    {
        SDL_Rect dstrect_poing = { 30, 350, 30, 30 };
        SDL_RenderCopy(renderer, texture_objet[2], NULL, &dstrect_poing);
    }
    // Irene Adler
    {
        SDL_Rect dstrect_crane = { 0, 380, 30, 30 };
        SDL_RenderCopy(renderer, texture_objet[], NULL, &dstrect_crane);
    }

```

FIGURE 25 – Table des suppositions

Pour chacun des joueurs, on met dans la grille les valeurs des objets qu'ils possèdent.
Pour chaque personnage (13 au total), on va venir lui associer les symboles présents sur sa carte.

```

// Afficher les suppositions
for (i=0;i<13;i++)
{
    if (guiltGuess[i])
    {
        SDL_RenderDrawLine(renderer, 250,350+i*30,300,380+i*30);
        SDL_RenderDrawLine(renderer, 250,380+i*30,300,350+i*30);

        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
        SDL_RenderDrawLine(renderer, 0,30+60,680,30+60);
        SDL_RenderDrawLine(renderer, 0,30+120,680,30+120);
        SDL_RenderDrawLine(renderer, 0,30+180,680,30+180);
        SDL_RenderDrawLine(renderer, 0,30+240,680,30+240);
        SDL_RenderDrawLine(renderer, 0,30+300,680,30+300);

        SDL_RenderDrawLine(renderer, 200,0,200,330);
        SDL_RenderDrawLine(renderer, 260,0,260,330);
        SDL_RenderDrawLine(renderer, 320,0,320,330);
        SDL_RenderDrawLine(renderer, 380,0,380,330);
        SDL_RenderDrawLine(renderer, 440,0,440,330);
        SDL_RenderDrawLine(renderer, 500,0,500,330);
        SDL_RenderDrawLine(renderer, 560,0,560,330);
        SDL_RenderDrawLine(renderer, 620,0,620,330);
        SDL_RenderDrawLine(renderer, 680,0,680,330);

        for (i=0;i<14;i++)
            SDL_RenderDrawLine(renderer, 0,350+i*30,300,350+i*30);
        SDL_RenderDrawLine(renderer, 100,350,100,740);
        SDL_RenderDrawLine(renderer, 250,350,250,740);
        SDL_RenderDrawLine(renderer, 300,350,300,740);
    }
}

```

FIGURE 26 – Aide pour trouver le coupable

Ici, on affiche les deux grilles qui contiennent tous les noms des 13 personnages pour aider l'utilisateur à trouver le coupable.

```

    //SDL_RenderCopy(renderer, texture_grille, NULL, &dstrect_grille);
if (b[0]!=-1)
{
    SDL_Rect dstrect = { 750, 0, 1000/4, 660/4 };
    SDL_RenderCopy(renderer, texture_deck[b[0]], NULL, &dstrect);
}
if (b[1]!=-1)
{
    SDL_Rect dstrect = { 750, 200, 1000/4, 660/4 };
    SDL_RenderCopy(renderer, texture_deck[b[1]], NULL, &dstrect);
}
if (b[2]!=-1)
{
    SDL_Rect dstrect = { 750, 400, 1000/4, 660/4 };
    SDL_RenderCopy(renderer, texture_deck[b[2]], NULL, &dstrect);
}

// Le bouton go
if (goEnabled==1)
{
    SDL_Rect dstrect = { 500, 350, 200, 150 };
    SDL_RenderCopy(renderer, texture_gobutton, NULL, &dstrect);
}
// Le bouton connect
if (connectEnabled==1)
{
    SDL_Rect dstrect = { 0, 0, 200, 50 };
    SDL_RenderCopy(renderer, texture_connectbutton, NULL, &dstrect);
}

//L'image loser
if (loserEnabled==1)
{
    SDL_Rect dstrect = { 500, 350, 200, 150 };
    SDL_RenderCopy(renderer, texture_loser, NULL, &dstrect);
}

```

FIGURE 27 – Positions et affichage des images

On affiche les images *go*, *connect*, *loser*. D'ailleurs, pour obtenir le fond transparent de l'image j'ai utilisé la commande :

SDL_SetColorKey(loser,SDL_TRUE,SDL_MapRGB(loser->format,0,0,0))

```

SDL_Color col = {0, 0, 0};
for (i=0;i<4;i++)
{
    if (strlen(gNames[i])>0)
    {
        SDL_Surface* surfaceMessage = TTF_RenderText_Solid(Sans, gNames[i], col);
        SDL_Texture* Message = SDL_CreateTextureFromSurface(renderer, surfaceMessage);

        SDL_Rect Message_rect; //create a rect
        Message_rect.x = 10; //controls the rect's x coordinate
        Message_rect.y = 110+i*60; // controls the rect's y coordinate
        Message_rect.w = surfaceMessage->w; // controls the width of the rect
        Message_rect.h = surfaceMessage->h; // controls the height of the rect

        SDL_RenderCopy(renderer, Message, NULL, &Message_rect);
        SDL_DestroyTexture(Message);
        SDL_FreeSurface(surfaceMessage);
    }
}

SDL_RenderPresent(renderer);

}

SDL_DestroyTexture(texture_deck[0]);
SDL_DestroyTexture(texture_deck[1]);
SDL_FreeSurface(deck[0]);
SDL_FreeSurface(deck[1]);
SDL_DestroyRenderer(renderer);
SDL_DestroyWindow(window);

SDL_Quit();

return 0;

```

FIGURE 28 – Fin de l'interface graphique

Après avoir copié toutes les images, on fait **SDL_RenderPresent(renderer)** ce qui permet de **lancer la commutation du double buffer**.

Pour finir, parce qu'on fait du travail propre, on détruit toutes les surfaces, textures, window créées.

14 Pour aller plus loin...

J'ai remarqué un fait tout à fait amusant dans ce jeu. Il est possible de demander à soi-même combien on possède de symbole X et le résultat s'affiche sur toutes les interfaces...C'est amusant dans la mesure où dans la réalité, on ne va jamais demander à soi-même combien d'objet on a...à moins qu'on ait pris plus de mojitos que nécessaire! En découvrant ça (vers la fin, il ne me restait plus beaucoup de temps), je m'étais dit qu'il aurait été bien de régler ce point là, en interdisant au joueur courant de se sélectionner.

L'autre point que j'aurai aimé avoir le temps d'aborder, c'est de gérer la sortie des joueurs. En effet, si un joueur quitte la partie alors il y a un bug. L'idéal aurait été de laisser la partie se dérouler sans le joueur qui nous a quitté.

Toutefois, j'ai eu le temps de réaliser les étapes importantes du projet et rajouter quelques options : changement de la couleur de la fenêtre, affichage d'un message (en rose) de bienvenue sur l'interface graphique de tous les joueurs, élimination du joueur qui a porté une fausse accusation, la partie continue sans lui et on lui affiche un message 'Vous avez perdu' en fond transparent (et oui, on s'amuse comme on peut!). On gère aussi la fin de partie en affichant sur tous les terminaux, le gagnant et la carte coupable.

15 Conclusion

Ce projet nous a permis de bien comprendre et de mettre en pratique toutes les notions théoriques enseignées. Cette découverte sur la programmation réseau et graphique s'est faite de façon très ludique! Bien que le code de départ nous soit déjà fourni, il a été essentiel d'en comprendre toutes les étapes afin de bien s'approprier le jeu et y ajouter autant d'options qu'on souhaite.

16 Sources

Les figures 3,4,5 sont extraites du polycopié de cours *Modèle Client-Serveur* réalisé par Mr.Wajbürt et Mr. Pécheux.

Annexes

Voici les interfaces graphiques des quatres joueurs afin de bien comprendre tous les points abordés à travers cet exemple.



FIGURE 29 – Interface graphique de bob

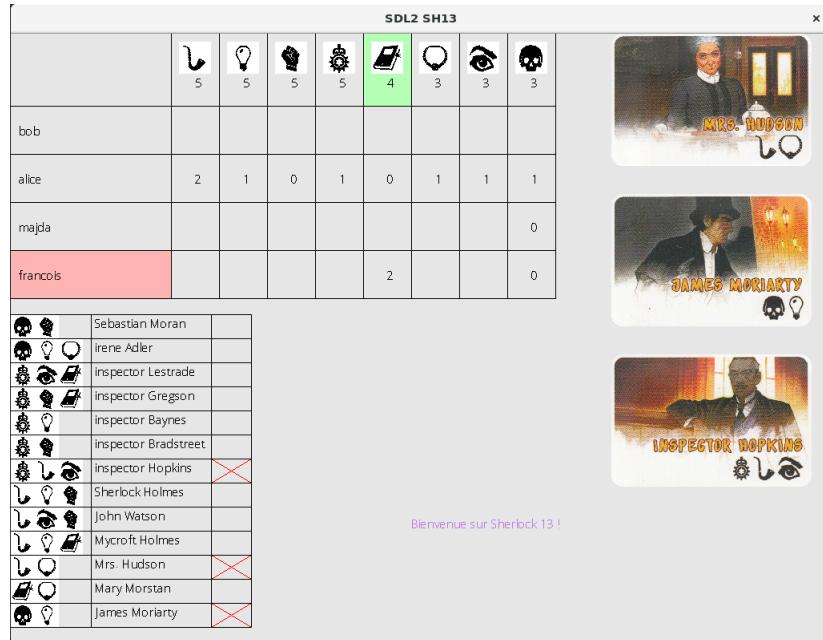


FIGURE 30 – Interface graphique d'alice

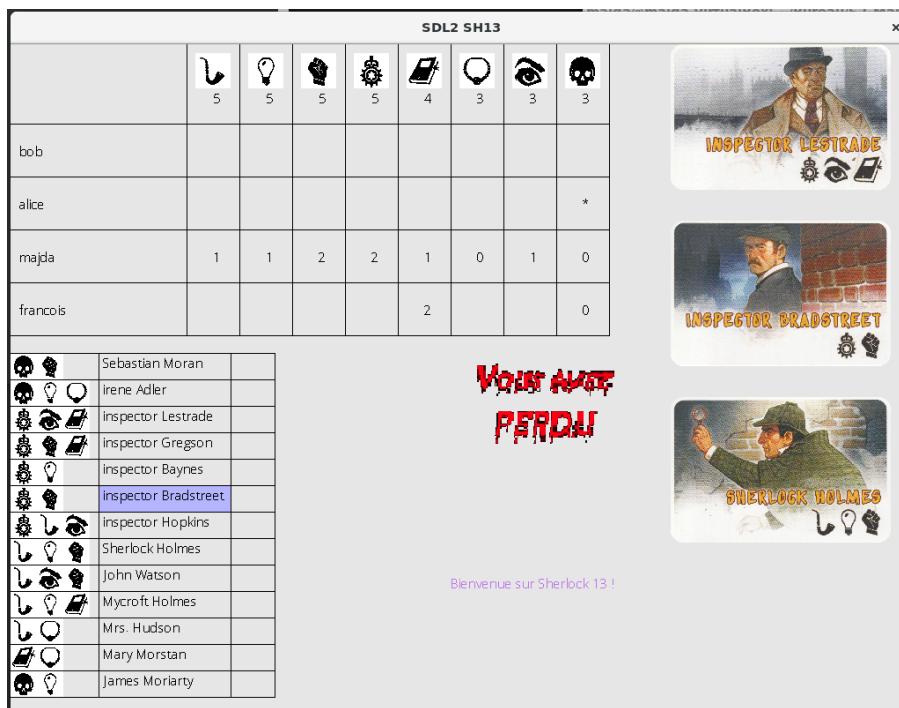


FIGURE 31 – Interface graphique de majda

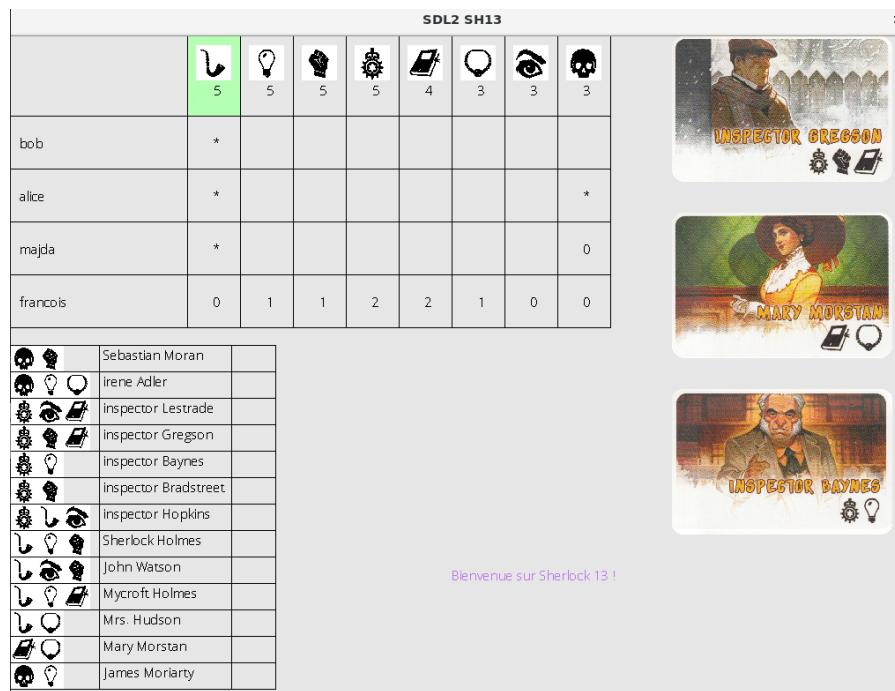


FIGURE 32 – Interface graphique de francois