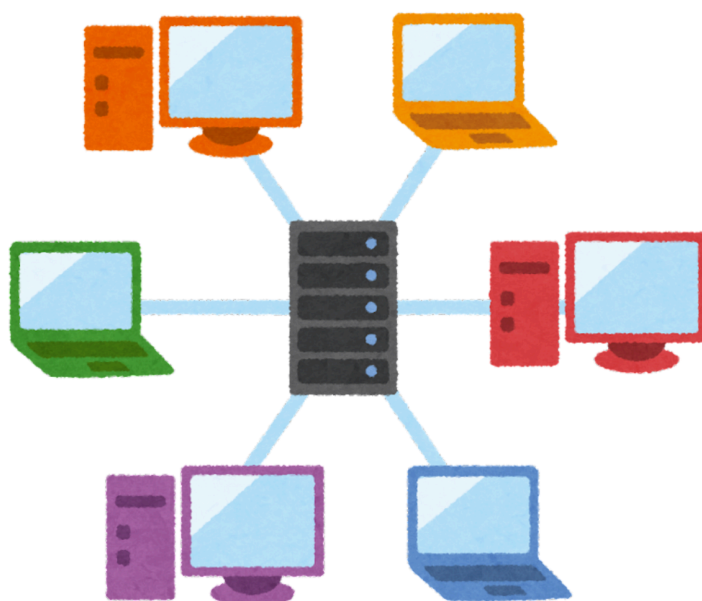


## **Rapport du Projet OS USER**

### **Jeu Sherlock 13**



**CACAU Ioana**  
**EI4**  
**TPA**

## Table des matières

<b>Table des matières.....</b>	<b>2</b>
<b>1. Introduction.....</b>	<b>3</b>
<b>2. Architecture et Concepts Systèmes.....</b>	<b>3</b>
2.1 Sockets.....	3
2.2 Threads et Mutex.....	4
2.3 Gestion Séparée des Clients.....	6
<b>3. Implémentation des Mécanismes de Jeu.....</b>	<b>8</b>
3.1 Connexion et Initialisation (commandes C, I, N, L).....	8
3.2 Commandes pendant la Partie.....	9
Commande G – Deviner le coupable.....	9
Commande O – Deviner un objet.....	10
Commande S – Poser une question à un joueur.....	11
<b>Conclusion.....</b>	<b>13</b>

## 1. Introduction

Ce projet consiste en l'implémentation en réseau du jeu "Sherlock 13", en utilisant différents concepts systèmes abordés en cours et lors des séances de travaux pratiques.

## 2. Architecture et Concepts Systèmes

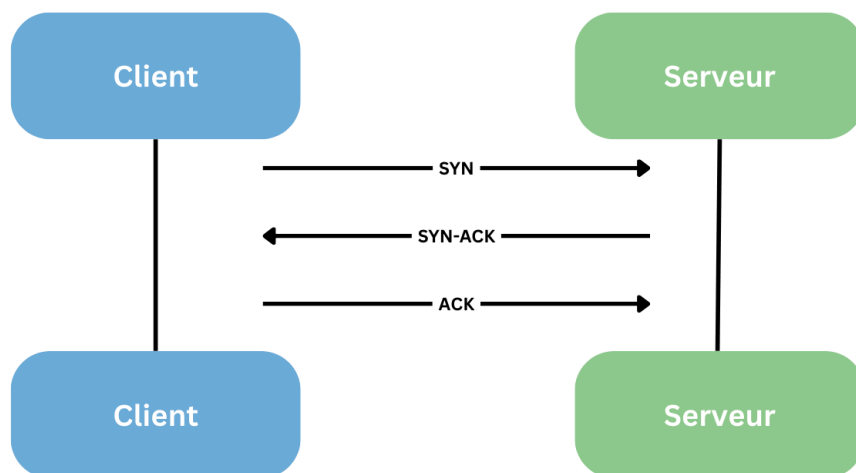
### 2.1 Sockets

Les sockets permettent d'établir une communication bidirectionnelle entre deux programmes, généralement entre un serveur et un ou plusieurs clients. Dans notre projet, nous avons utilisé des sockets TCP, c'est-à-dire basés sur le protocole TCP/IP, afin d'assurer une transmission fiable des données entre le serveur et les clients.

Le protocole TCP (Transmission Control Protocol) joue un rôle fondamental dans cette communication : il garantit que les messages envoyés arrivent dans le bon ordre et sans erreurs. Il gère aussi le contrôle de flux entre les deux extrémités, ainsi que la congestion du réseau.

Avant d'échanger des données, le protocole TCP établit une connexion persistante entre les deux parties à l'aide du three-way handshake. Ce mécanisme se déroule en trois étapes : dans la première étape client envoie un paquet SYN pour initier la connexion, puis le serveur répond avec un paquet SYN-ACK pour confirmer la réception. Finalement, le client renvoie un paquet ACK pour finaliser l'établissement de la connexion. Une fois cette phase terminée, les deux programmes peuvent communiquer de manière stable jusqu'à la fermeture de la socket.

Celle-ci peut être visualisée sur le schéma ci-dessous:



Dans notre projet, cette mise en place se retrouve dans le fichier `server.c`:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0); // Création de la socket TCP
bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
listen(sockfd, 5); // Écoute des connexions
```

Puis chaque connexion entrante est acceptée avec :

```
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
```

Côté client, la fonction `sendMessageToServer` dans `sh13.c` se charge de la connexion au serveur et de l'envoi du message :

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
write(sockfd, sendbuffer, strlen(sendbuffer));
```

Tous ces appels permettent d'établir la communication TCP entre le client et le serveur à chaque interaction dans notre jeu.

## 2.2 Threads et Mutex

Dans notre projet, l'utilisation des threads permet de maintenir une interface fluide tout en recevant des messages du serveur. Comme chaque client peut recevoir des informations à tout moment (nouveau tour, données de `tableCartes..`), il est important que ces messages soient gérés en parallèle sans bloquer l'interface SDL. Pour cela, on utilise deux threads : un pour l'interface graphique (thread principal), et un second pour la réception réseau.

Ce second thread est lancé au début du programme dans sh13.c avec le code suivant :

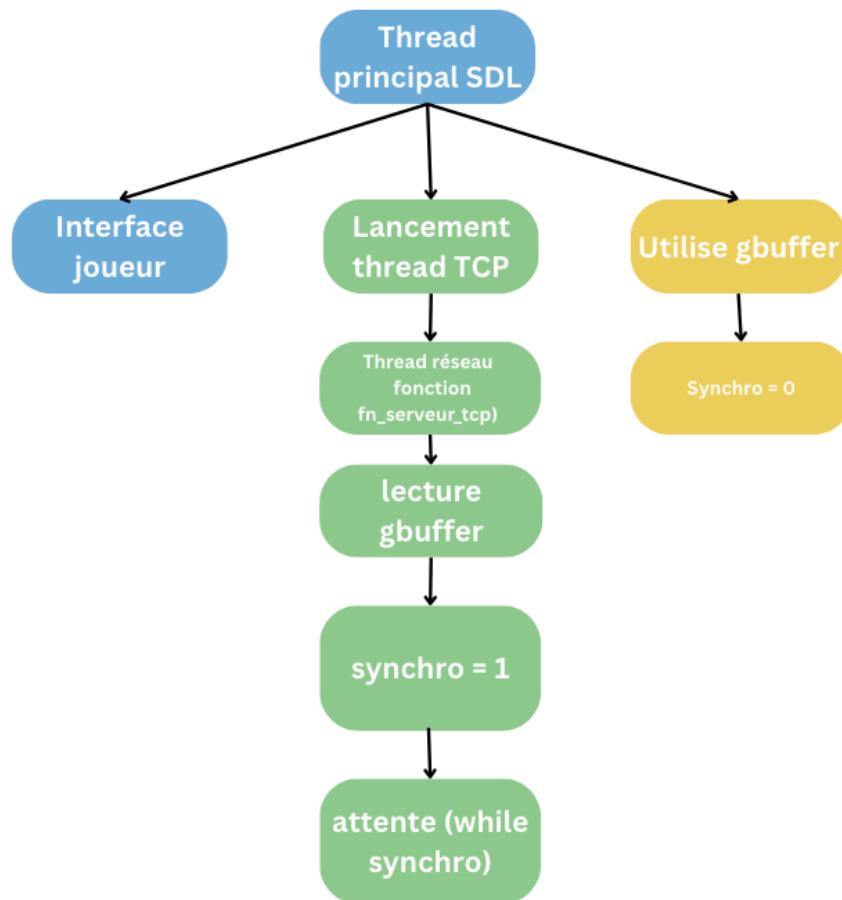
```
pthread_t thread_serveur_tcp_id;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // Mutex statique
...
ret = pthread_create ( & thread_serveur_tcp_id, NULL, fn_serveur_tcp,
NULL);
```

La fonction `fn_serveur_tcp` contient une boucle infinie qui attend les messages envoyés par le serveur :

```
void *fn_serveur_tcp(void *arg)
{
    ...
    while (1)
    {
        newsockfd = accept(sockfd, (struct sockaddr *)
&cli_addr, &clilen);
        ...
        bzero(gbuffer,256);
        n = read(newsockfd,gbuffer,255);
        ...
        synchro=1;
        while (synchro);
    }
}
```

La synchronisation est assurée par la variable `synchro`, qui empêche le thread de réseau de modifier `gbuffer` tant que le thread principal n'a pas terminé de le lire.

Comme illustré sur le schéma ci-dessous, ce mécanisme permet d'éviter les conflits entre les deux threads :



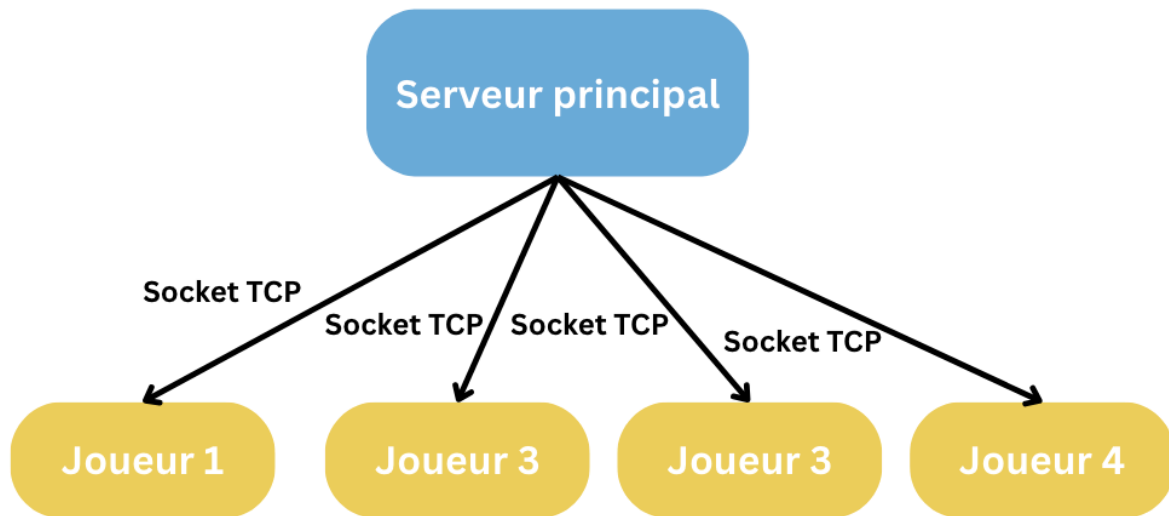
Ce fonctionnement à deux threads permet donc d'éviter les conflits d'accès à gbuffer tout en assurant une bonne réactivité de l'interface. Comme la réception des messages se fait en parallèle, l'utilisateur peut interagir normalement sans que les clics soient ralentis, même lorsque le serveur envoie des données.

## 2.3 Gestion Séparée des Clients

Dans notre projet, chaque client (sh13.c) est lancé séparément dans son propre terminal, ce qui fait que chaque instance fonctionne comme un processus indépendant. Ces processus clients n'ont aucune mémoire partagée entre eux, et toutes les communications passent par des sockets TCP avec le serveur. Cela garantit une vraie séparation des données, ce qui est un des principes vus dans le TP1 sur les processus.

Ce fonctionnement permet au serveur de continuer à tourner normalement même si un client se déconnecte ou rencontre une erreur. Il n'existe aucune dépendance directe entre les clients : chacun fonctionne de manière autonome, avec sa propre mémoire, sa propre socket et son propre thread d'exécution.

Comme illustré dans le schéma suivant, les clients sont isolés les uns des autres et communiquent uniquement via le serveur :



Le lancement manuel dans des terminaux différents est une preuve claire de ce comportement/

```

./sh13 localhost 32000 localhost 32001 Joueur1
./sh13 localhost 32000 localhost 32002 Joueur2
./sh13 localhost 32000 localhost 32003 Joueur3
./sh13 localhost 32000 localhost 32004 Joueur4
  
```

Chaque appel démarre un exécutable indépendant. Le serveur, lui, reste actif même si un client se déconnecte, car il n'en dépend pas directement :

```

//dans server.c
while(1) {
    newsockfd = accept(...); // Attend une nouvelle connexion
}
  
```

On aurait pu aussi utiliser `fork()` pour créer un processus par client côté serveur, comme on l'a vu en TP, mais cela n'était pas nécessaire ici. L'utilisation de threads dans `sh13.c`, combinée avec les sockets TCP, fournit déjà une bonne séparation entre les utilisateurs. De plus, la bibliothèque SDL2 impose d'utiliser un seul thread principal pour le rendu graphique, ce qui aurait compliqué une structure avec des processus multiples.

### 3. Implémentation des Mécanismes de Jeu

Le fonctionnement du serveur repose sur deux grandes phases principales : d'abord, la phase d'attente des connexions, puis, une fois que tous les joueurs sont présents, la gestion active des échanges pendant la partie. Chaque commande envoyée par les clients est traitée dans le serveur via un switch, qui permet d'adapter la réponse en fonction de l'action du joueur.

#### 3.1 Connexion et Initialisation (commandes C, I, N, L)

Tout commence par la réception de la commande C, envoyée par chaque client au moment de la connexion. Elle contient l'adresse IP, le port et le nom du joueur. Le serveur enregistre ces informations dans le tableau `tcpClients`, en incrémentant le nombre total de joueurs connectés.

Une fois que les quatre joueurs sont connectés, le serveur passe automatiquement à la distribution des cartes. CLe serveur envoie à chaque joueur ses trois cartes via une commande D, suivie de l'envoi de la ligne correspondante dans la table de jeu (`tableCartes`) avec des messages

```
sprintf(reply, "D %d %d %d", deck[0], deck[1], deck[2]);  
sendMessageToClient(tcpClients[0].ipAddress, tcpClients[0].port, reply);
```

Chaque message D est suivi de plusieurs messages V pour indiquer les objets visibles pour ce joueur. Enfin, lorsque tous les joueurs ont reçu leurs cartes, le serveur annonce le début de la partie en envoyant un message M avec `joueurCourant = 0`, c'est-à-dire que le joueur 0 commence.





capture d'écran du jeu après la distribution des cartes

Côté client, à la réception du message I, le joueur récupère son identifiant personnel et le renvoie au serveur accompagné de son nom avec la commande N. Ensuite, le message L reçu contient la liste complète des joueurs connectés, qui est utilisée pour afficher leurs noms dans l'interface.

### 3.2 Commandes pendant la Partie

Une fois la partie lancée, le serveur passe à un autre switch (quand fsmServer == 1) où il écoute les actions des joueurs. C'est à ce moment que les commandes G, O et S deviennent essentielles pour le déroulement de la partie.

#### Commande G – Deviner le coupable

La commande G permet à un joueur de tenter de deviner directement l'identité du coupable. Si le joueur a raison, le serveur envoie un message W à tous les clients pour annoncer la victoire. Sinon, la main passe au joueur suivant. Le serveur vérifie d'abord que c'est bien le tour du joueur concerné. Ensuite, il compare la valeur devinée à deck[12], qui contient la carte du coupable.

```
if (reponse == deck[12]) {
    sprintf(reply, "W %d", id);
```

```
broadcastMessage(reply);
}
```

En cas d'échec, la main passe automatiquement au joueur suivant, avec un message M.

The screenshot shows the SDL2 SH13 game interface. It features a grid of player scores and a list of characters. The grid has 9 columns representing different objects (represented by icons: a hook, a lightbulb, a hand, a gear, a book, a necklace, an eye, and a skull) and 4 rows representing players (J3, J1, J4, J2). The scores are as follows:

	Hook	Lightbulb	Hand	Gear	Book	Necklace	Eye	Skull
J3	1	2	0	1	0	2	0	1
J1								
J4								
J2								

Below the grid is a list of characters with their associated icons and a status column:

Character	Icons	Status
Sebastian Moran	Skull, Hand	
Irene Adler	Skull, Lightbulb, Necklace	
Inspector Lestrade	Gear, Eye, Book	
Inspector Gregson	Gear, Hand, Book	
Inspector Baynes	Gear, Lightbulb	
Inspector Bradstreet	Gear, Hand	X
Inspector Hopkins	Gear, Hook, Eye	
Sherlock Holmes	Hook, Lightbulb, Hand	
John Watson	Hook, Eye, Hand	
Mycroft Holmes	Hook, Lightbulb, Book	
Mrs. Hudson	Hook, Necklace	
Mary Morstan	Book, Necklace	
James Moriarty	Skull, Lightbulb	

On the right side, there are three character cards: Inspector Baynes, Irene Adler, and Mrs. Hudson, each with their respective icons.

Le joueur 0 a deviné 5 (le coupable est 2)  
Mauvaise réponse. Tour suivant.

```
id=2 ind=7 val=0
consomme | M 0
|
gId=0
consomme | M 1
|
gId=1
|
```

capture d'écran d'une tentative de devin incorrecte









## Commande O – Deviner un objet

Avec la commande O, un joueur peut interroger un objet, pour savoir combien de fois il est visible chez les autres. Le serveur additionne la présence de l'objet dans les grilles des trois adversaires, met à jour localement les cellules dans tableCartes, puis envoie un message O avec le total. Cela permet aux joueurs d'éliminer certaines hypothèses.

Ce type de question ne permet pas de gagner directement mais fait partie de la logique de déduction.

```
sprintf(reply, "O %d %d %d", id, objetSel, total);
```

```
broadcastMessage(reply);
```

	 5	 5	 5	 5	 4	 3	 3	 3
J3					0			
J1	0	0	2	2	2	1	0	0
J4					1			
J2					0			

capture d'écran après une question O









### Commande S – Poser une question à un joueur

Enfin, la commande S permet à un joueur de demander à un autre joueur s'il possède un objet spécifique. Le serveur regarde si la cellule `tableCartes[joueurSel][objetSel]` est non nulle et envoie un message V à tous les clients pour mettre à jour cette information.

Ce type de question est très utilisé dans les premières étapes du jeu pour affiner les déductions. À la suite de cette commande, la main passe également au joueur suivant.

```
int hasObject = tableCartes[joueurSel][objetSel] > 0;
sprintf(reply, "V %d %d %d", joueurSel, objetSel, hasObject ? 1 : 0);
broadcastMessage(reply);
```

Ce type de question est très utilisé dans les premières étapes du jeu pour affiner les déductions. À la suite de cette commande, la main passe également au joueur suivant.

	 5	 5	 5	 5	 4	 3	 3	 3
J3					0			
J1								
J4	3	2	2	0	1	0	1	0
J2	1				0			

capture d'écran avec une réponse V à une question

D'autres commandes secondaires interviennent automatiquement côté serveur : D pour la distribution des cartes, V pour la mise à jour des grilles, M pour le changement de tour, et W pour annoncer la victoire. Celles-ci ne sont pas envoyées manuellement par les joueurs mais générées suite aux actions principales. Voici un tableau récapitulatif des commandes :

Commande	Rôle	Comportement dans le code
C	Connexion d'un joueur	Le client envoie son IP, port et nom. Le serveur l'enregistre, lui attribue un ID et broadcaste la liste mise à jour. Si 4 joueurs sont connectés, la partie démarre.
D	Distribution des cartes	Générée par le serveur : chaque joueur reçoit ses 3 cartes (deck[x]). Envoyée après la connexion du 4e joueur.
V	Mise à jour des grilles	Générée par le serveur : envoie les nouvelles valeurs de la table tableCartes après certaines actions (S, O).
M	Changement de tour	Générée automatiquement après chaque tour (échec G, action S, ou question O). Permet de savoir à qui c'est le tour.
G	Deviner le coupable	Le joueur tente une accusation directe (deck[12]). Si c'est correct, W est envoyé. Sinon, le tour passe (M).
O	Deviner un objet	Le joueur interroge un objet. Le serveur compte les occurrences chez les autres et broadcast la réponse. Si ça correspond au coupable, W peut être envoyé.
S	Interroger un joueur	Le joueur demande si un adversaire a un objet. Le serveur répond avec V, puis passe au joueur suivant (M).
W	Victoire	Générée par le serveur si une accusation (G) ou hypothèse (O) est correcte. Elle termine la partie.

Grâce à cette structure en commandes bien définies, le serveur gère le déroulement du jeu de manière fluide. Chaque action d'un joueur a une conséquence directe visible pour tous les participants, ce qui permet un jeu à la fois interactif et équitable.

## Conclusion

Ce projet a été une opportunité concrète d'appliquer les notions théoriques étudiées dans le cadre du module OS USER. L'implémentation du jeu Sherlock 13 en réseau nous a permis d'explorer de manière approfondie les mécanismes de communication via sockets TCP, la gestion multi-threadée avec synchronisation à l'aide de mutex, ainsi que l'architecture client-serveur basée sur des processus isolés.

La construction progressive du jeu, en partant d'un code partiellement fourni et en complétant les différentes sections, a renforcé notre compréhension du déroulement des

échanges entre les joueurs et le serveur. Nous avons ainsi pu observer l'importance de la séparation des responsabilités entre l'interface graphique (SDL), la logique réseau et la logique de jeu.

L'utilisation des commandes C, I, N, L pour l'initialisation, et G, O, S pour les actions pendant la partie, a permis une gestion structurée et réactive des interactions. Chaque ajout de code a été réfléchi pour assurer à la fois la cohérence fonctionnelle du jeu et la stabilité du système, même en cas de déconnexion ou de comportement inattendu.

En conclusion, ce projet nous a permis de consolider de manière concrète les notions vues en TP, tout en confrontant les contraintes réelles de la programmation réseau et multi-threadée. Il nous a également permis de comprendre l'importance d'un code bien organisé, ce qui facilite à la fois le développement et le bon fonctionnement de l'application.