

Rapport du projet OS : Sherlock 13



Étudiant : Ayoub LADJICI

Spécialité : Électronique – Informatique

Enseignants référents : François PÊCHEUX – Thibault HILAIRE

Table des matières

Introduction	3
1. Analyse de l'architecture Client-Serveur TCP.....	5
1.1 Définition du protocole TCP.....	5
1.2 Mise en œuvre du protocole TCP dans le code.....	5
2. Implémentation du jeu.....	6
2.1 Structure du projet	6
2.2 Complétion du programme	12
2.3 Concepts d'OS utilisés dans l'implémentation	13
3. Test du jeu.....	15

Introduction

Dans le cadre du module Système d'exploitation, nous sommes amenés à réaliser un projet dont le but est d'implémenter une application permettant de jouer au jeu Sherlock 13, en utilisant une architecture client-serveur reposant sur le protocole TCP.

Sherlock 13 est un jeu de déduction et de logique dans lequel les joueurs doivent identifier l'identité d'un suspect unique. Comme son nom l'indique, le jeu met en scène 13 personnages, chacun représenté par une carte associée à deux ou trois symboles.

Dans notre implémentation, le jeu se joue à quatre joueurs. Parmi les 13 cartes, 12 sont distribuées équitablement (3 cartes par joueur), tandis que la carte restante correspond au coupable mystère que les joueurs doivent deviner. À chaque tour, un joueur peut soit :

- Poser une question à l'ensemble des joueurs, portant sur la présence d'un symbole
- Interroger un joueur en particulier sur l'occurrence d'un symbole dans ses cartes
- Accuser un personnage

Une accusation correcte permet de gagner la partie. En revanche, une erreur élimine le joueur de l'enquête, les autres continuent jusqu'à ce quelqu'un trouve la carte mystère.

Architecture Client-Serveur

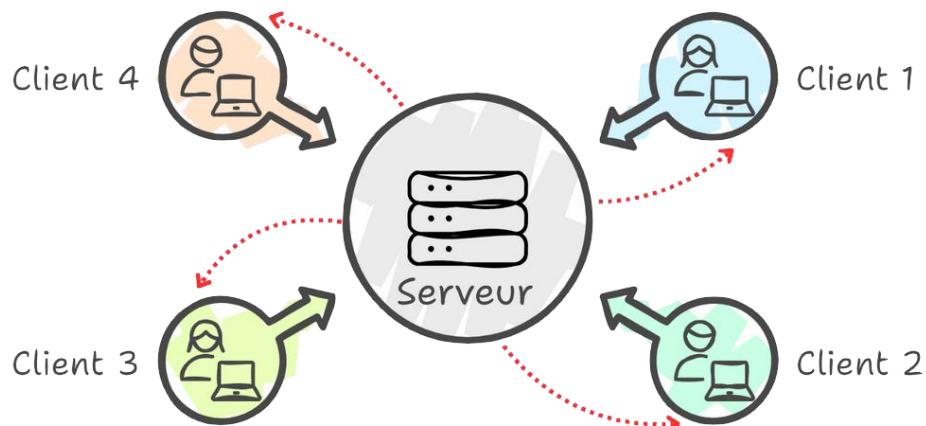


Figure 1 : Schéma illustrant l'architecture Client-Serveur dans le jeu Sherlock 13

Comme illustré dans la figure 1, l'architecture du projet repose sur un modèle client-serveur, dans lequel le serveur joue un rôle central : il gère la logique du jeu, l'état des cartes traite les actions des joueurs et diffuse des messages asynchrones aux clients à travers le réseau. Chaque client incarne un joueur, et interagit avec le serveur à travers un ensemble de commandes précises. Lorsqu'un joueur effectue une action (poser une question, interroger, accuser), elle est transmise au serveur, qui la traite et met à jour l'état du jeu. Le serveur envoie ensuite les informations nécessaires aux autres clients afin de garantir une synchronisation parfaite du déroulement de la partie.

L'interface graphique du jeu a été développée à l'aide de SDL (Simple DirectMedia Layer), une bibliothèque multimédia multiplateforme. Elle nous a permis d'afficher dynamiquement les cartes, les symboles, les objets et les hypothèses de chaque joueur, tout en gérant les événements clavier/souris.

Ce rapport présentera dans un premier temps l'analyse de l'architecture client-serveur TCP, puis l'implémentation du système. Enfin, nous testerons le jeu et commenterons les résultats.

Pour mener à bien ce projet, je me suis appuyé sur mes notes de cours, ainsi que sur plusieurs ressources en ligne pour approfondir les notions abordées. En particulier, le tutoriel sur les sockets de [developpez.com](#) m'a aidé à comprendre la mise en œuvre d'une architecture client/serveur en C. J'ai également consulté le cours sur les sockets disponible à l'adresse [csd.uoc.gr](#) et utilisé des forums comme Stack Overflow. Par ailleurs, j'ai utilisé l'outil [Excalidraw](#) pour réaliser mes schémas.

1. Analyse de l'architecture Client-Serveur TCP

1.1 Définition du protocole TCP

Le protocole TCP (Transmission Control Protocol) permet une communication fiable entre deux entités d'un réseau. Une connexion entre le client et le serveur est nécessaire avant de démarrer l'échange des données. Ensuite, les données sont découpées en plusieurs segments plus petits et numérotés puis envoyés au destinataire. A l'arrivée, les segments sont réassemblés dans le bon ordre. TCP vérifie également le flux de données pour s'assurer que l'expéditeur n'envoie pas plus de données que le récepteur ne peut en traiter. Donc, TCP garantit que les données échangées entre le serveur et le client sont correctement envoyées, reçues et ordonnées.

1.2 Mise en œuvre du protocole TCP dans le code

Tout d'abord, le code client permet d'établir une connexion fiable avec le serveur, d'envoyer des commandes et de recevoir des réponses du serveur. Le fonctionnement général suit les étapes classiques illustrées dans la Figure 2 : création d'un socket, connexion, envoi/réception de messages, puis fermeture de la connexion.

En effet, la première étape pour établir une communication entre 2 programmes est de créer un socket (prise de communication). Cela se fait à l'aide de la fonction `socket()` :

```
socket(domain = AF_INET, type = SOCK_STREAM, protocol = 0)
```

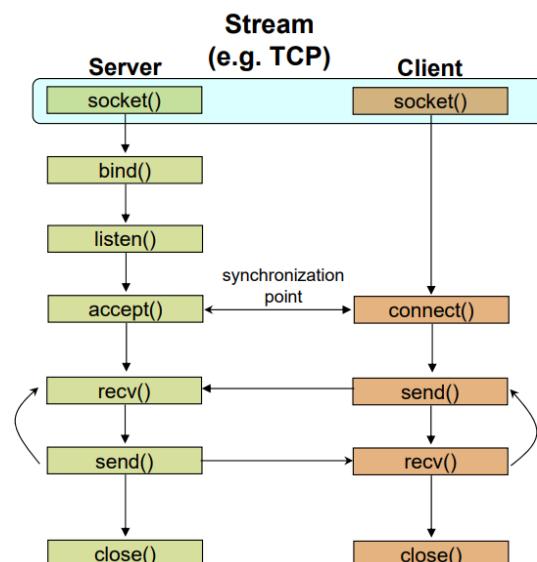


Figure 2 : Schéma illustrant le processus de connexion TCP entre un serveur et un client

Le paramètre `AF_INET` indique que le socket utilise la famille de `SOCK_STREAM` garantit une connexion fiable et ordonnée et le dernier paramètre laisse le système d'exploitation choisir automatiquement le protocole approprié, en l'occurrence `IPPROTO_TCP`.

Une fois le socket créé, le client utilise une structure `SOCKADDR_IN` pour définir les informations nécessaires à la connexion, notamment l'adresse IP du serveur et le numéro de port. Ensuite, le client tente de se connecter au serveur avec la fonction `connect()`. Si cette étape est réussie, le client pourra envoyer un message au serveur à l'aide de la `write()` puis ferme immédiatement la connexion avec `close()`.

Dans le code du projet, cette séquence est encapsulée dans la fonction `sendMessageToServer()`. À chaque fois qu'un message doit être envoyé, cette fonction est appelée : elle ouvre une connexion temporaire, envoie le message, puis ferme le socket. Le client ne conserve pas de connexion ouverte en permanence.

La réception des messages en provenance du serveur se fait via un thread séparé (`fn_serveur_tcp`). Ce thread joue le rôle d'un petit serveur TCP local sur le client. Il attend les

connexions provenant du serveur principal, lit les messages avec `read()` et les place dans un buffer global (`gbuffer`). Un indicateur (`synchro`) permet ensuite au thread principal d'en prendre connaissance dans la boucle SDL.

Enfin, lorsqu'un joueur ferme la fenêtre du jeu (en cliquant sur la croix de la fenêtre), le programme se termine simplement. Il ne signale pas explicitement sa déconnexion au serveur (aucun message de déconnexion n'est envoyé).

Le code serveur est responsable de la gestion des connexions clients et du traitement des requêtes. Il suit aussi les grandes étapes du protocole TCP, comme illustré dans la Figure 2 : création du socket (`socket()`), association à un port avec `bind()`, puis mise en écoute avec `listen()`.

Comme pour le client, la première étape du serveur consiste à créer un socket avec les mêmes paramètres et à le lier via la fonction `bind()` avec une structure `STOCKADDR` où sont renseignés des informations comme le numéro de port du serveur et est défini avec `INADDR_ANY` pour accepter les connexions de n'importe quelle adresse IP. Le socket est ensuite mis en mode écoute pour gérer les connexions entrantes avec `listen()` et définir également la taille maximale de la file d'attente des connexions entrantes.

Lorsque le socket du serveur est prêt, il entre alors dans une boucle infinie `while(1)` où il gère les connexions et traite les messages reçus des clients. Lorsqu'un client se connecte, la connexion est acceptée avec `accept()`, le message du client est lu via `read()`, puis le socket est fermé immédiatement après traitement. Chaque message est traité séquentiellement.

Le serveur possède une liste des clients connectés dans un tableau (`tcpClients[]`), et gère les états du jeu via une machine à états (`fsmServer`) en fonction des messages reçus (C, G, O, S, etc.). Lorsqu'un client envoie une requête (par exemple pour se connecter ou pour jouer), le serveur la lit et envoie la réponse soit directement au client concerné, soit à tous les clients (diffusion via `broadcastMessage()`).

2. Implémentation du jeu

2.1 Structure du projet

Le projet *Sherlock 13* est structuré autour de deux fichiers principaux : un pour le serveur et un pour le client. Cela va nous permettre de bien comprendre leurs rôles dans l'architecture client-serveur.

Le fichier `server.c` implémente le serveur central du jeu. Il est chargé de :

- Gérer les connexions des quatre clients via TCP
- Attribuer à chaque joueur un identifiant
- Mélanger et distribuer les cartes
- Envoyer les cartes aux clients

- Maintenir l'état du jeu à travers une machine à états (si fsmServer=0, nous sommes en attente de jour, si fsmServer=1 on joue)
- Répondre aux différentes commandes (C, G, O, S) reçues des clients
- Diffuser les messages (broadcast) à tous les joueurs ou répondre à un joueur en particulier

Il utilise une structure `tcpClients[]` pour stocker les informations de chaque joueur connecté (IP, port, nom), ainsi qu'une matrice `tableCartes[][]` pour représenter les symboles présents sur les cartes de chaque joueur.

Le déroulement global suit une logique simple :

1. Attente de la connexion des 4 joueurs
2. Envoi des informations initiales à chacun
3. Lancement de la partie et gestion des tours
4. Traitement des messages des joueurs selon leur nature (question, accusation)

Voici un schéma simplifié du fonctionnement du serveur :

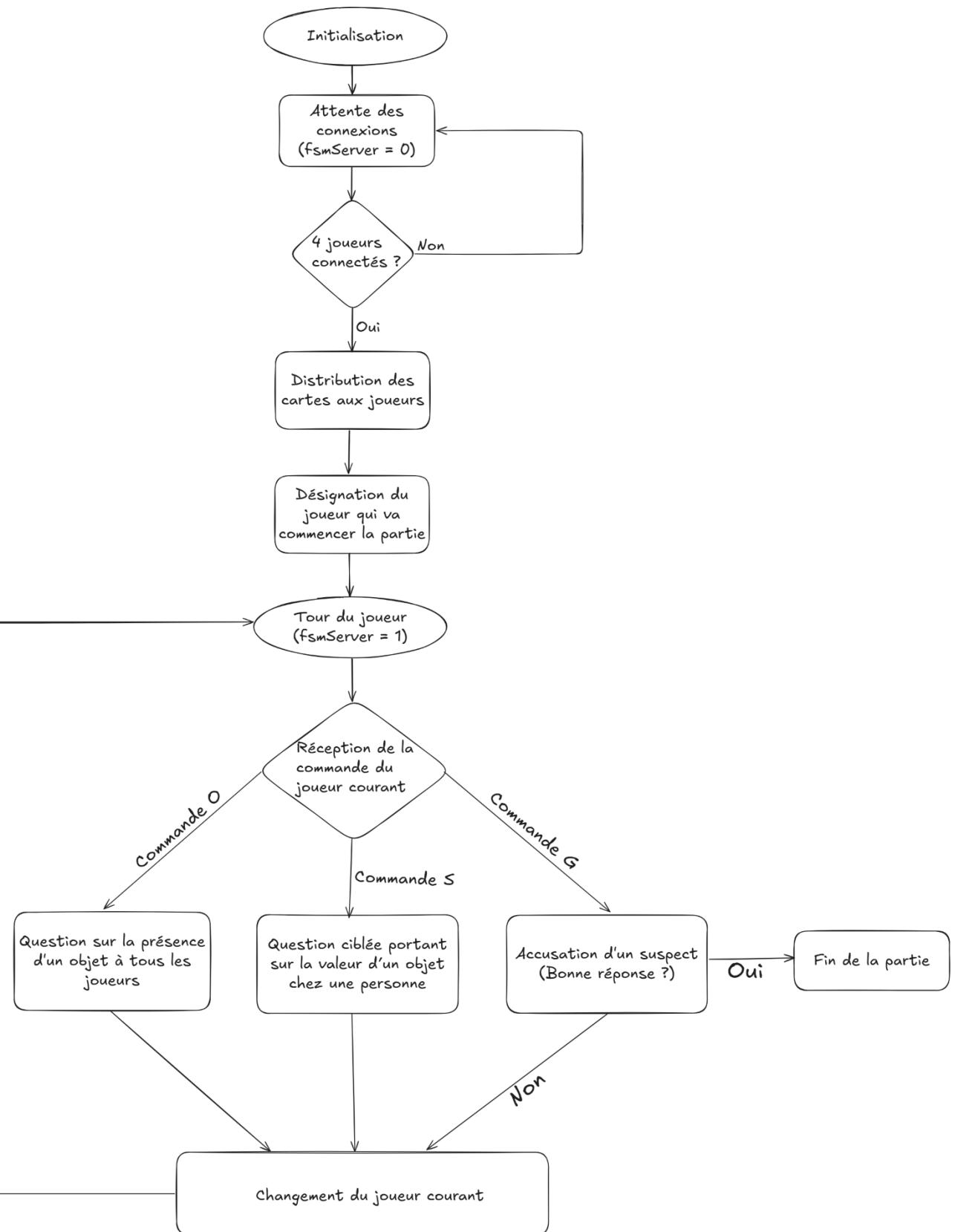


Diagramme de la logique serveur

Voici le tableau des commandes que le serveur peut envoyer :

Commande	Signification	Exemple	Explication
I	Attribution de l'identifiant	I 0	Le serveur attribue l'identifiant 0 au client.
L	Liste des joueurs	L Ayoub Ronaldo Mbappe Zidane	Le serveur envoie la liste des joueurs actuellement connectés.
D	Distribution des cartes	D 9 3 6	Le serveur envoie au joueur ses trois cartes (ici, indices 9, 3 et 6).
V	Réponse à une question (valeur d'un objet)	V 0 9 1	Le joueur 0 possède 1 fois l'objet d'indice 9 dans ses cartes.
M	Désignation du joueur courant dans la partie	M 1	Le serveur indique que c'est au tour du joueur d'indice 1 de jouer
W (facultative)	Victoire	W 0	Le joueur 2 a gagné la partie.
E (facultative)	Élimination	E 1	Le joueur 1 s'est trompé et est éliminé.

Le fichier sh13.c est un fichier client. Il est chargé de :

- Se connecter au serveur en envoyant son IP, port local et prénom
- Réceptionner les messages du serveur
- Afficher une interface graphique (via SDL2) permettant au joueur d'interagir avec le jeu
- Interpréter et afficher les réponses du serveur dans le terminal du client
- Gérer les événements (clics, commandes clavier) et envoyer les commandes appropriées (O, S, G) au serveur

Le déroulement global côté client suit cette logique :

1. L'utilisateur exécute la commande `./sh13 [ip_serveur] [port_serveur] [ip_client] [port_client] [nom_joueur]`
2. Une fenêtre s'ouvre et le joueur peut se connecter au serveur en cliquant sur le bouton « Connect »
3. Le joueur reçoit son identifiant, ses cartes, et les données initiales du jeu

4. L'utilisateur interagit via l'IHM : il peut sélectionner un joueur ou un objet, formuler une accusation ou une question
5. Les messages sont envoyés au serveur, qui répond avec des informations (via V, M, W, E, etc.)

Voici un schéma simplifié du fonctionnement du client :

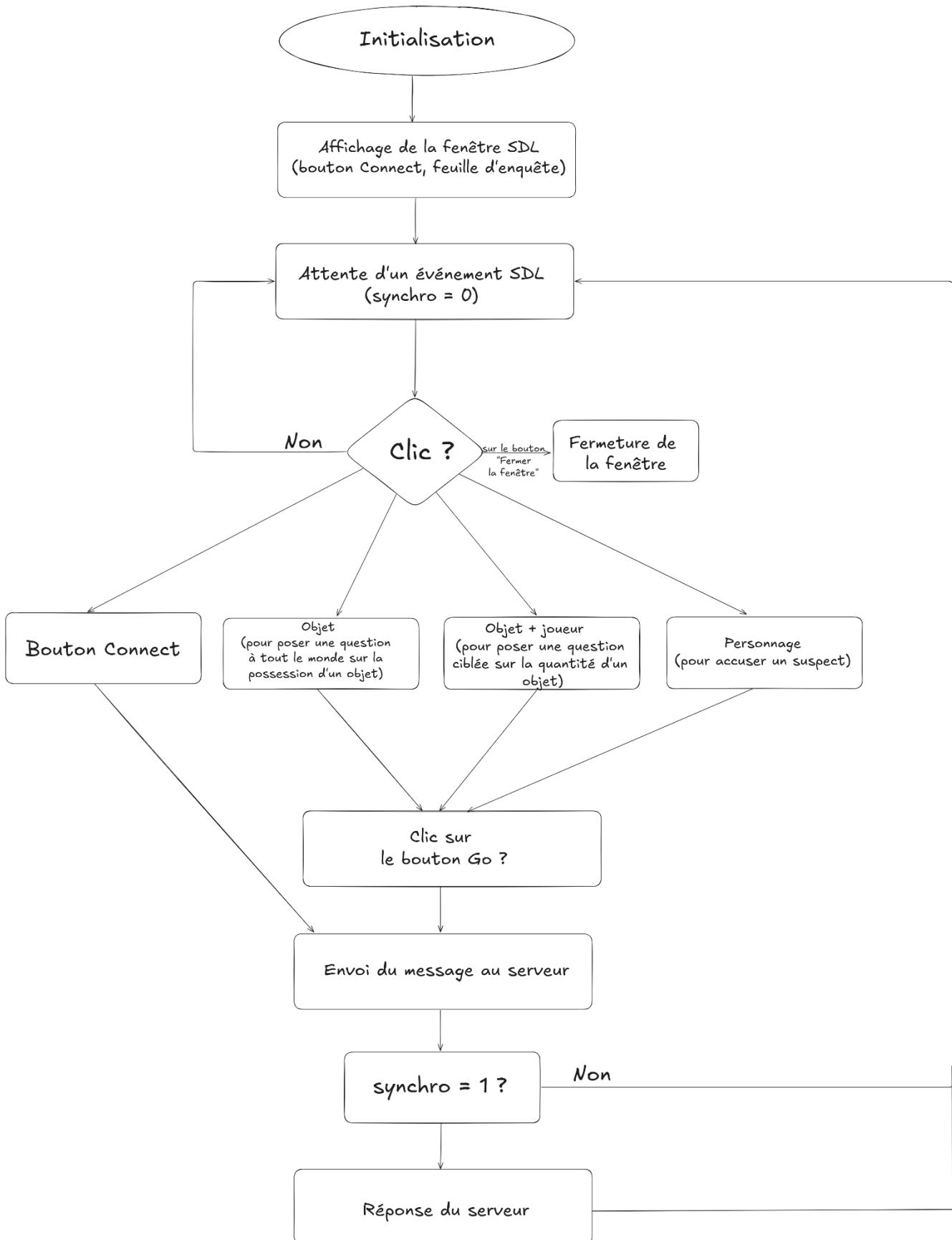


Diagramme de la logique client

Voici le tableau des commandes que le client peut envoyer :

Commande	Signification	Exemple	Explication
C	Connexion d'un client	C 127.0.0.1 2025 Ayoub	Le joueur « Ayoub » se connecte au serveur en fournissant son IP, son port d'écoute, et son prénom.
G	Accusation d'un suspect	G 0 10	Le joueur 0 accuse le personnage d'indice 10.
O	Question à tous portant sur la présence d'un objet dans leurs cartes	O 0 6	Le joueur 0 demande si l'objet 6 est présent dans les cartes des autres joueurs.
S	Question ciblée portant sur la valeur d'un objet chez une personne	S 1 2 5	Le joueur 1 demande au joueur 2 combien de fois il possède l'objet 5.

2.2 Complétion du programme

Le code fourni initialement comportait plusieurs zones à compléter, aussi bien côté serveur que côté client.

Tout d'abord, le code client fourni comportait plusieurs zones à compléter, notamment pour la gestion des échanges avec le serveur via TCP. D'une part, il fallait gérer l'envoi de commandes au serveur selon les actions effectuées par l'utilisateur dans l'interface SDL. Les commandes (C, G, O, S) étaient déjà préformatées sous forme de chaînes de caractères, la tâche consistait principalement à appeler la fonction `sendMessageToServer()` avec les bons arguments (IP et port du serveur) pour transmettre ces messages.

D'autre part, le client devait également gérer la réception des messages en provenance du serveur. Ces messages étaient reçus dans un thread dédié (`fn_serveur_tcp`) via un socket TCP, puis stockés dans un buffer (`gbuffer`) en activant la variable de synchronisation (`synchro = 1`) pour signaler qu'un message est prêt à être traité dans la boucle. Le but était de décoder le contenu du message reçu de la part du serveur d'une part en identifiant le caractère en début de message, s'agissant du type de commande, cela a été possible grâce à `sscanf`. Puis de gérer la logique pour certaines commandes notamment que lorsque c'était au joueur de jouer, d'activer le bouton Go sur la fenêtre graphique.

Ensuite, l'objectif principal dans le fichier serveur était de gérer correctement le démarrage de la partie dès que les quatre joueurs sont connectés, puis de traiter les commandes envoyées par les clients pendant le jeu.

Dans un premier temps, dès que tout le monde est connecté, il fallait distribuer à chaque joueur ses 3 cartes, envoyer à tous la ligne de symboles correspondante à chaque joueur dans la matrice tableCartes et définir le joueur qui commence la partie.

Dans un second temps, pendant le déroulement du jeu (fsmServer == 1), le serveur reçoit et traite les commandes envoyées par les clients. Si c'est une commande G alors le joueur tente de deviner le coupable donc le serveur compare le personnage accusé avec le dernier élément du tableau deck[], qui représente le coupable à trouver. Personnellement, j'ai choisi d'implémenter deux nouvelles commandes permettant de gérer les deux issues de cette accusation, en effet, si c'est correct, j'ai choisi d'envoyer une nouvelle commande W à tous les joueurs pour signaler la victoire du joueur concerné. Sinon, si c'est incorrect, j'ai ajouté la commande E pour informer que ce joueur est éliminé et ensuite M pour indiquer le prochain joueur.

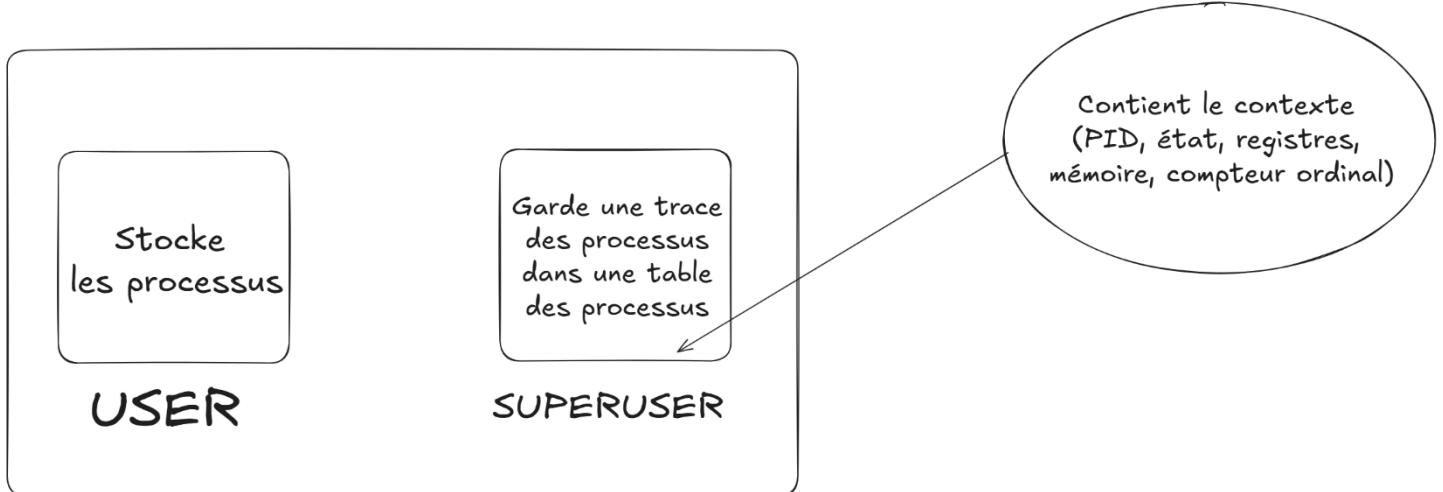
Si c'est une commande O alors le joueur demande combien de fois un symbole donné apparaît chez les autres joueurs. Le serveur parcourt la matrice tableCartes verticalement et vérifie si la valeur de l'objet dans leurs cartes est supérieure à 0, si c'est le cas, il envoie un message V avec la valeur 100 à tout le monde signifiant la présence de l'objet, sinon 0.

Si c'est une Commande S alors le joueur interroge un joueur précis sur un symbole. Le serveur regarde combien de fois le symbole apparaît dans les cartes du joueur ciblé, puis envoie un message V uniquement au joueur qui a posé la question.

2.3 Concepts d'OS utilisés dans l'implémentation

Le développement du jeu Sherlock 13 en architecture client-serveur nous a permis de réinvestir plusieurs notions clés abordées durant les TPs du module.

- Un socket est une interface logicielle qui permet à un programme d'exploiter les services d'un protocole réseau, en particulier le protocole TCP, afin d'échanger des données avec un autre programme distant.
Son utilisation dans le projet est détaillée dans la section [1.2 - Mise en œuvre du protocole TCP dans le code](#) de ce rapport.
- Un processus est un programme en cours d'exécution. Par exemple, lorsqu'on exécute une commande comme ls ou gcc, un processus est créé. C'est le système d'exploitation (OS) qui est responsable de la gestion des processus (partage des ressources (CPU, mémoire...), priorité d'exécution, interruptions...). Elle gère également la table des processus, dans laquelle sont stockées les informations sur chaque processus (PID, état, registre, espace mémoire, compteur ordinal...).
Comme on peut le voir sur le schéma ci-dessous, l'OS divise la mémoire en deux espaces (USER, SUPERUSER).



En mode USER, on doit faire un appel système (ex : fork est une instruction privilégiée qui permet de créer un nouveau processus (enfant) en dupliquant le processus appelant (parent) où l'enfant hérite de la plupart des attributs du parent à l'exception de l'ID.) si on souhaite modifier la table des processus. A l'aide de fork, on interrompt l'exécution de l'application et passe temporairement en mode SUPERUSER. L'OS ajoute une nouvelle entrée dans la table des processus et copie le contexte du processus parent dans le nouveau processus. Enfin, l'OS retourne en mode USER et reprend l'exécution du processus parent et du nouveau processus (enfant).

Toutefois, dans ce projet nous n'avons pas utilisé l'appel fork mais on a vu en TP qu'il était possible de faire un fork() après un accept() afin que le processus enfant gère la communication avec le client, pendant que le parent écoute d'autres connexions.

- Un thread, on peut le voir comme une partie d'un processus qui travaille en parallèle avec d'autre parties à l'intérieur du même programme. Dans notre projet, on a créé un thread pour gérer la réception des messages de la part du serveur sans bloquer l'interface graphique SDL
- Un mutex est comme un verrou qui garantit que seul un thread à la fois accède à une ressource partagée comme une variable globale pour pouvoir écrire ou lire par exemple. Il est très utile pour éviter une corruption de données. On l'a initialisé dans notre projet mais nous ne l'avons pas utilisé. On l'a « compensé » par une variable de synchronisation qui nous permet de savoir quand un message a été reçu pour être traiter

3. Test du jeu

Le serveur est lancé est la commande :

```
./server [num_port_server]
```

num_port_server : Port TCP sur lequel le serveur attend les connexions des clients

Ce port doit être le même que celui fourni aux clients pour se connecter. Par ailleurs, il y a $2^{16} = 65536$ ports existants, ceux en dessous de 1024 sont réservées. Le numéro de port du serveur est fixe tandis que celui du client est fixe.

Chaque client (joueur) peut se connecter au serveur avec la commande :

```
./sh13 127.0.0.1 [num_port_server] 127.0.0.1 [num_port_client] [prenom]
```

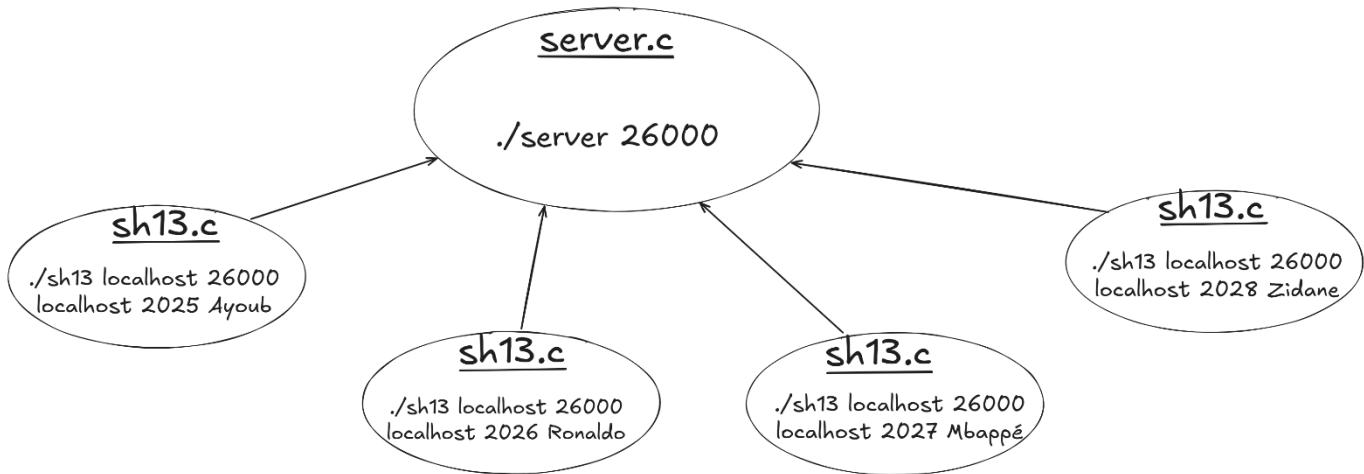
127.0.0.1 (1^{er} argument) : Adresse IP du serveur principal

num_port_server : Port du serveur principal

127.0.0.1 (2^{ème} argument) : Adresse IP locale du client

num_port_client: Port TCP sur lequel le client va écouter les messages du serveur

prenom : Nom du joueur



Une fois que les 4 clients ont lancé la commande sur leur propre terminal, cela leur ouvrira une fenêtre SDL et il faudra cliquer sur le bouton Connect pour pouvoir s'inscrire et jouer au jeu.

Dans cette simulation, on aura :

- Joueur 0 : Ayoub (port 2025)
- Joueur 1 : Ronaldo (port 2026)
- Joueur 2 : Mbappé (port 2027)
- Joueur 3 : Zidane (port 2028)

	5	5	5	5	4	3	3	3
Ayoub	1	1	2	0	0	1	1	2
Ronaldo								
Mbappe								
Zidane								

SDL2 SH13

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

	5	5	5	5	4	3	3	3
Ayoub								
Ronaldo								
Mbappe	1	1	2	2	1	0	1	0
Zidane								

SDL2 SH13

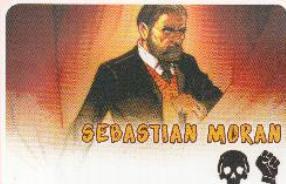
Sebastian Moran	Irene Adler	Inspector Lestrade	Inspector Gregson	Inspector Baynes	Inspector Bradstreet	Inspector Hopkins	Sherlock Holmes	John Watson	Mycroft Holmes	Mrs. Hudson	Mary Morstan	James Moriarty

On peut voir que les cartes ont bien été mélangées et distribués à tous les joueurs. Chaque joueur possède bien 3 cartes. On peut commencer à jouer.

Tour 1 (Ayoub) :

	5	5	5	5	4	3	3	3
Ayoub	1	1	2	0	0	1	1	2
Ronaldo								
Mbappé			1					
Zidane								

	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	


Si on décrypte le message du serveur, on comprend que Ayoub demande au joueur d'indice 2 (Mbappé) combien de fois possède-t-il l'objet d'indice 1 (ampoule) ?

```
Received packet from 127.0.0.1:47678
Data: [S 0 2 1
]
```

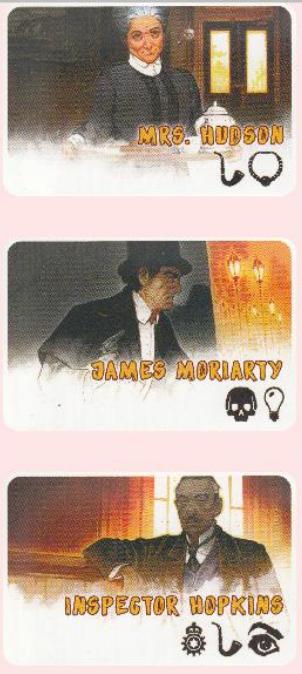
La réponse reçue est que Mbappé en possède seulement 1 exemplaire. Puis le serveur passe la main à joueur 1 (Ronaldo).

```
joueurCourant=0
go! joueur=-1 objet=-1 guilt=-1
go! joueur=2 objet=-1 guilt=-1
go! joueur=2 objet=-1 guilt=-1
go! joueur=2 objet=-1 guilt=-1
go! joueur=2 objet=1 guilt=-1
consomme |V 2 1 1
|
joueur=2 objet=1 valeur=1
consomme |M 1
|
joueurCourant=1
```

Tour 2 (Ronaldo) :

	5	5	5	5	4	3	3	3
Ayoub								*
Ronaldo	2	1	0	1	0	1	1	1
Mbappe								0
Zidane								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	



Ici, Ronaldo demande « Combien d'entre vous (sauf moi) possèdent au moins un crâne dans leurs cartes ? »

Le serveur répond que le joueur 0 (Ayoub) possède au moins crâne, et que les autres n'en ont pas.

```
Received packet from 127.0.0.1:57140
Data: [0 1 7
]
```

```
id=1 objet=7
```

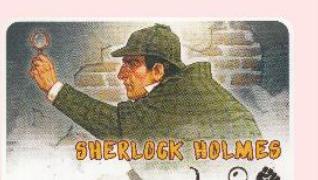
```
joueurCourant=1
go! joueur=-1 objet=7 guilt=-1
consomme |V 0 7 100
|
joueur=0 objet=7 valeur=100
consomme |V 2 7 0
|
joueur=2 objet=7 valeur=0
consomme |V 3 7 0
|
joueur=3 objet=7 valeur=0
consomme |M 2
|
joueurCourant=2
```

Tour 3 (Mbappé) :

					SDL2 SH13			
	5	5	5	5	4	3	3	3
Ayoub								*
Ronaldo								
Mbappe	1	1	2	2	1	0	1	0
Zidane				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	





Le joueur 2 (Mbappé) interroge le joueur 3 (Zidane) à propos de l'objet 3 (la couronne).

```
Received packet from 127.0.0.1:49424
Data: [S 2 3 3
]
```

Le serveur répond que le joueur 3 en possède 2.

```
joueurCourant=2
go! joueur=3 objet=3 guilt=-1
consomme |V 3 3 2
|
joueur=3 objet=3 valeur=2
consomme |M 3
|
joueurCourant=3
```

Tour 4 (Zidane) :

	5	5	5	5	4	3	3	3
Ayoub	*							*
Ronaldo	*							
Mbappe	*							0
Zidane	0	1	1	2	2	1	0	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	



Le joueur 3 (Zidane) demande à tout le monde qui possède l'objet 0 (la pipe).

```
Received packet from 127.0.0.1:37880
Data: [0 3 0
]
id=3 objet=0
```

Tous les autres joueurs (0, 1 et 2) ont au moins une pipe.

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

Tour 5 (Ayoub) :

	5	5	5	5	4	3	3	3	SDL2 SH13
Ayoub	*	1	2	0	0	1	1	*	
Ronaldo	*								
Mbappe	*	1						0	
Zidane								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	

Le joueur Ayoub a envoyé la commande pour accuser Mycroft Holmes d'être le coupable.

Par l'intermédiaire de l'indicateur 'W', le serveur confirme qu'Ayoub a bien gagné la partie.

```
Received packet from 127.0.0.1:54986
Data: [G 0 9
]
```

```
joueurCourant=0
go! joueur=-1 objet=-1 guilt=9
consomme |W 0
|
go! joueur=-1 objet=-1 guilt=9
consomme |M 1
|
```