




















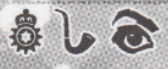



## Rapport : Projet OS USER

		 5	 5	 5	 5	 4	 3	 3	 3
	SEBASTIAN MORAN						SHERLOCK HOLMES		
	IRENE ADLER						JOHN WATSON		
	INSPECTOR LESTRADE						MYCROFT HOLMES		
	INSPECTOR GREGSON						MRS. HUDSON		
	INSPECTOR BAYNES						MARY MORSTAN		
	INSPECTOR BRADSTREET						JAMES MORIARTY		
	INSPECTOR HOPKINS						SHERLOCK 13		

## Architecture générale

Le programme repose sur une architecture client-serveur :

Le serveur (server.c) est responsable de gérer les connexions, d'attribuer les cartes, d'envoyer les données à chaque joueur et de relayer les messages entre les clients une fois la partie commencée.

Le client (sh13.c) affiche une interface graphique SDL et permet à un joueur d'interagir avec le jeu. Il communique avec le serveur via TCP.

## Utilisation des sockets

Le module socket est utilisé dans les deux programmes :

Sur le serveur, un socket TCP est ouvert sur le port donné en argument. Lorsqu'un client se connecte, un nouveau `accept()` est appelé pour lire les données envoyées (messages C, G, O, S).

Sur le client, la fonction `sendMessageToServer(...)` crée un socket TCP, se connecte au serveur, envoie une commande, puis ferme le socket.

fonctions classiques : `socket()`, `connect()`, `bind()`, `listen()`, `accept()`, `read()` et `write()`.

## Utilisation des threads

Le programme client (sh13.c) lance un thread TCP dédié en arrière-plan, via :

```
pthread_create(&thread_serveur_tcp_id, NULL, fn_serveur_tcp, NULL);
```

Ce thread permet au client de recevoir des messages entrants du serveur sans bloquer la boucle principale de l'interface SDL.

## Utilisation d'un mutex

Un mutex global `pthread_mutex_t mutex` est déclaré, même s'il n'est pas strictement requis dans la version minimale du projet (puisque `synchro` sert de signal léger entre thread principal et thread TCP).

Dans un projet plus évolué, il pourrait servir à protéger l'accès au buffer global `gbuffer` lors de lectures concurrentes entre le thread TCP et le thread principal.

## Synchronisation entre thread TCP et SDL

La synchronisation est assurée par une variable globale volatile `int synchro` :

Le thread TCP écrit un message dans `gbuffer` et passe `synchro` à 1.

Le thread principal SDL attend `synchro == 1` dans sa boucle pour traiter les messages, puis le remet à 0.

Cette approche garantit une communication fluide entre les threads sans surcharge, adaptée à un jeu au tour par tour comme Sherlock 13.

# Démarrage de la partie (côté serveur)

Le serveur fonctionne selon une logique de machine à états finis (FSM), où l'état fsmServer == 0 correspond à la phase d'attente des connexions. Chaque fois qu'un client se connecte, ses informations (adresse IP, port, nom) sont stockées dans le tableau tcpClients[], et il reçoit en retour un message "I" lui indiquant son identifiant. À chaque connexion, la liste complète des joueurs est également diffusée à tous les clients via un message L.

Dès que quatre joueurs sont connectés (nbClients == 4), la partie peut commencer. Le serveur passe alors à l'état fsmServer == 1 et effectue les actions suivantes :

## Envoi des cartes personnelles :

Pour chaque joueur i, le serveur envoie un message de type D contenant les trois cartes qui lui ont été attribuées dans le tableau deck.

## Envoi de la ligne d'indices tableCartes[i] :

Ensuite, il envoie au joueur une série de huit messages V, chacun représentant le nombre d'objets de chaque type associés à ses cartes.

Le format de ces messages est V <index\_objet> <valeur>.

## Définition du joueur courant :

Enfin, un message M 0 est envoyé à tous les clients pour indiquer que c'est le joueur 0 qui commence la partie.

## Gestion des commandes pendant la partie

Une fois la partie lancée (fsmServer == 1), le serveur écoute les commandes envoyées par les clients. Trois types de messages sont attendus :

G : le joueur tente de désigner le coupable (guess)

O : le joueur interroge sur un objet donné

S : le joueur interroge un autre joueur sur un objet donné

Conformément aux consignes du projet, le serveur ne traite pas le contenu de ces messages. Il se contente de les relayer à tous les autres clients via un broadcastMessage.

## Respect des consignes

Le serveur a été complété conformément aux indications du sujet et aux commentaires laissés dans le code :

Lors de la connexion du quatrième joueur, il passe automatiquement à l'état de jeu.

Il envoie à chaque joueur :

un message D contenant ses cartes,

huit messages V correspondant à sa ligne personnelle du tableau `tableCartes`.

Il envoie un message M 0 pour définir le joueur courant.

Enfin, pendant la partie, il se contente de relayer les messages G, O, et S à tous les clients, sans en analyser la logique.

## Partie client : sh13.c

Le fichier sh13.c implémente le client graphique du jeu Sherlock 13, basé sur la bibliothèque SDL2. Il permet à un joueur d'interagir avec l'interface, de recevoir des informations du serveur et d'envoyer ses actions pendant la partie.

### Envoi des actions client vers serveur

Lorsqu'un joueur clique sur le bouton "Go", le client analyse la sélection effectuée par l'utilisateur et construit un message adapté à la situation :

Si un personnage est sélectionné (`guiltSel != -1`), un message de type G (guess) est généré.

Si un objet seul est sélectionné (`objetSel != -1` et `joueurSel == -1`), un message de type O est envoyé.

Si un objet et un joueur sont sélectionnés, un message de type S est construit.

Ces messages sont ensuite transmis au serveur via la fonction `sendMessageToServer(...)`.

### Réception des messages venant du serveur

Le client a été complété pour réagir aux messages envoyés par le serveur selon les consignes données dans les commentaires du code :

Message	Effet sur le client
I <id>	Stocke l'identifiant du joueur dans gld
L <nom1> <nom2> <nom3> <nom4>	Met à jour les noms des joueurs dans gNames[]
D <a> <b> <c>	Stocke les trois cartes du joueur dans b[0..2]
M <id>	Si id == gld, active le bouton <b>Go</b> (goEnabled = 1)
V <col> <val>	Remplit la table tableCartes[gld][col] avec la valeur reçue

Les messages sont reçus via un thread TCP parallèle, qui écoute sur le port du client. Lorsqu'un message arrive, synchro lance le traitement dans la boucle principale du jeu.

## Conclusion

Côté serveur, la logique de la machine à états (FSM) permet une gestion simple et claire de la phase de connexion des joueurs (fsmServer == 0) et du déroulement de la partie (fsmServer == 1). Le serveur assure la distribution des cartes (D), des indices (V), ainsi que la synchronisation du jeu (M). Les actions des joueurs (G, O, S) sont simplement relayées sans interprétation, comme demandé.

Côté client, le programme sh13.c reçoit et traite les messages envoyés par le serveur pour mettre à jour l'interface et réagir aux actions de l'utilisateur. Lorsqu'un joueur effectue une action, un message adapté est généré (G, O ou S) et transmis au serveur.

Limite observée : bien que l'ensemble du projet ait été complété uniquement dans les blocs indiqués par les commentaires du code, le jeu ne peut pas fonctionner totalement (ex : enchaînement des tours, validation des actions) sans ajouter une logique supplémentaire côté serveur. Cette logique (comme changer le joueur courant après un clic sur "Go") n'était pas explicitement demandée dans les consignes, et n'a donc pas été implémentée ici. Le jeu est donc fidèle à la consigne, mais reste à l'état de prototype relayant les messages, sans gestion complète du gameplay.

L'ensemble du projet respecte la séparation des rôles entre le client (affichage, interactions) et le serveur (communication, synchronisation), tout en respectant les contraintes de complétion minimale imposées par le sujet.