Rapport de projet - OS USER : Sherlock 13 en réseau local (TCP/client/server)

@Author : Pierre BATAILLE

Ce document présente un rapport sur le projet OS USER, un jeu de déduction multijoueur inspiré de Sherlock 13, développé en réseau local via TCP.
Ce travail m'a pris 16heures du Weekend de Pâques passé en famille

---

1. Introduction

Pour ce projet OS USER, on devait coder un jeu de déduction en réseau inspiré de Sherlock 13, mais version multijoueur local via TCP. L'idée, c'était que 4 joueurs puissent se connecter, poser des questions, faire des déductions et accuser un coupable parmi 13 personnages, avec une vraie interface graphique gérée avec SDL2.

Heureusement, on ne partait pas de zéro. Le code était déjà partiellement écrit, avec une base serveur (`server.c`) et un client SDL (`sh13.c`). Mais pas mal de blocs étaient laissés vides avec des `// RAJOUTER DU CODE ICI`. C'était à nous de compléter la logique réseau et jeu pour que tout fonctionne.

Dans le fichier `server.c`, j'ai surtout complété les parties qui gèrent les échanges entre joueurs :

— les questions sur les symboles (case 'O') envoyées à tout le monde,

— les accusations (case 'G') pour tenter de désigner le coupable,

— les questions ciblées à un seul joueur (case 'S'),

— et la distribution des cartes en début de partie.

Dans `sh13.c`, j'ai ajouté tout ce qu'il fallait pour que le client communique correctement avec le serveur :

— envoie des commandes réseau (connexion, questions, accusations),

— réception et traitement des messages du serveur (case 'I', case 'L', case 'M', case 'D', etc. et un Case 'W' qui est additionnel par rapport au code),

— mise à jour de l'interface graphique (grille, boutons, noms...),

— et surtout, un thread TCP qui tourne en parallèle pour ne pas bloquer SDL quand on reçoit un message.

Ce projet m'a permis de bien revoir pas mal de choses qu'on avait vues en TP :

— **Sockets TCP** : côté serveur, pour accepter les connexions et envoyer les réponses. Et côté client, pour envoyer les commandes des joueurs.

— **Threads** : le client lance un thread pour écouter les messages du serveur sans bloquer l'interface.

— **Mutex** : utilisé pour protéger la variable `synchro`, partagée entre le thread TCP et la boucle SDL.

— **FSM** (machine à états) : la variable `fsmServer` est une petite FSM pour gérer l'état du jeu (attente ou en cours).

2. Règles du jeu Sherlock 13

Rappel des règles du jeu

— Le jeu comprend 13 cartes représentant des suspects, tels que "Sherlock Holmes" ou "Irene Adler"...

— Chaque carte de suspect est associée à un ensemble de symboles distincts (par exemple, une pipe, un carnet, un œil, etc.).

— Au début de la partie, chacun des 4 joueurs reçoit trois cartes. L'une de ces cartes est mise de côté et constitue le coupable (la 13^{e}).

— Les symboles présents sur les cartes de chaque joueur sont visibles uniquement par ce joueur.

— L'objectif pour les joueurs est de déduire quelle carte est le coupable, c'est-à-dire la carte qui n'a été distribuée à aucun joueur.

Pendant leur tour, les joueurs peuvent :

1. Demander combien de fois un symbole apparaît chez les autres joueurs.
2. Demander à un joueur s'il a un symbole spécifique.
3. Accuser directement un personnage. Si c'est correct : on gagne. Sinon, vous ne pouvez plus jouer, mais vous répondez encore aux questions.

Le but est d'utiliser ces informations pour déduire qui est le coupable.

3. Architecture du projet (vue générale)

J'ai mis déjà quelques heures à comprendre l'architecture globale des 2 codes qui est assez longue et compliquée quand c'est pas nous qui avons écrit le code...Mais dans l'ensemble j'ai réussi à comprendre le code, et surtout à comprendre ce qu'il fallait faire pour que le jeu soit fonctionnel ; j'ai également documenté les différentes fonctions et leurs interactions pour faciliter la compréhension et la maintenance du code à l'avenir.

Le projet est divisé en deux programmes principaux : le fichier `server.c` et le fichier `sh13.c`. Ces deux programmes font appel à des bibliothèques SDL2 et TCP sockets, et utilisent les images présentes dans le répertoire du jeu ainsi que le bouton "Connect".

a) Le serveur (`server.c`)

— Mélange les cartes

— Attend que 4 joueurs se connectent.

— Distribue 3 cartes à chaque joueur.

— Calcule les symboles possédés par chaque joueur et les envoie.

— Gère le tour à tour (joueur courant, questions posées, réponses, accusations).

— Envoie des messages aux joueurs pour mettre à jour l'état du jeu.

b) Le client (`sh13.c`)

— Affiche l'interface du jeu avec SDL2 (cartes, boutons, suspects, objets).

— Chaque joueur peut cliquer pour poser une question ou accuser.

— Reçoit les messages du serveur dans un thread à part.

— Met à jour l'affichage en fonction des messages reçus (cartes, noms, joueur courant, etc.).

4. Compilation et exécution

Pour savoir comment compiler les deux fichiers `.c` et le script d'exécution (launch_clients.sh), vous pouvez regarder dans le `readme.md`, ou sinon les commandes sont les suivantes :

```
chmod +x cmd.sh
./cmd.sh
chmod +x launch_clients.sh
```

Puis on lance le serveur :

```
./server 8000
```

Une fois que le serveur est lancé, le deck de cartes est mélangé par le code :

```
melangerDeck();  % cette fonction mélange le deck
```

La fonction est définie ainsi :

```
index1 = rand() % 13;
index2 = rand() % 13;
tmp    = deck[index1];
deck[index1] = deck[index2];
deck[index2] = tmp;
```

On lance ensuite les 4 clients avec :

```
./launch_clients.sh
```

qui exécute par exemple :

```
./sh13 127.0.0.1 8000 127.0.0.1 5677 Pierre
./sh13 127.0.0.1 8000 127.0.0.1 5678 Marie
./sh13 127.0.0.1 8000 127.0.0.1 5679 Leon
./sh13 127.0.0.1 8000 127.0.0.1 5680 Jean
```

Explication des commandes clients :

— **Pierre** : je me connecte au serveur local à l'adresse `127.0.0.1` sur le port `8000`, et j'écoute les réponses sur le port `5677`.

— **Marie** : je me connecte au serveur local à l'adresse `127.0.0.1` sur le port `8000`, et j'écoute les réponses sur le port `5678`.

— **Leon** : je me connecte au serveur local à l'adresse `127.0.0.1` sur le port `8000`, et j'écoute les réponses sur le port `5679`.

— **Jean** : je me connecte au serveur local à l'adresse `127.0.0.1` sur le port `8000`, et j'écoute les réponses sur le port `5680`.

5. Fonctionnement du jeu (étapes détaillées)

Une fois que chaque client est lancé après l'exécution du serveur, une fenêtre s'ouvre avec une interface : il faut cliquer sur "Connect" pour que le client se connecte au jeu. Concrètement :

```
if ((mx<200) && (my<50) && (connectEnabled==1)) {
    sprintf(sendBuffer,"C␣%s␣%d␣%s",gClientIpAddress,gClientPort,gName);
    // RAJOUTER DU CODE ICI
    sendMessageToServer(gServerIpAddress,gServerPort,sendBuffer);
    connectEnabled=0;
}
```

Une fois 4 joueurs connectés (`if (nbClients == 4)`) dans le code, le serveur :

— Distribue 3 cartes à chaque joueur (la carte restante est le coupable). Par exemple, pour le joueur 0 :

```
sprintf(reply, "D␣%d␣%d␣%d", deck[0], deck[1], deck[2]);
sendMessageToClient(tcpClients[0].ipAddress, tcpClients[0].port, reply);
for (int j = 0; j < 8; j++) {
    sprintf(reply, "V␣0␣%d␣%d", j, tableCartes[0][j]);
    sendMessageToClient(tcpClients[0].ipAddress, tcpClients[0].port,
       reply);
}
```

— Remplit une table de symboles (`tableCartes[i][j]`) indiquant les symboles `j` visibles par le joueur `i`.

— Envoie à chaque joueur ses cartes et ses symboles visibles.

À chaque tour, un joueur peut :

— Poser une question générale sur un symbole (O : case 'O'). Le serveur parcourt les autres joueurs et renvoie la somme des présences.

— Poser une question ciblée à un joueur (S : case 'S'). Le serveur renvoie 1 si le joueur visé possède le symbole, 0 sinon.

— Accuser un suspect (G : case 'G'). Le joueur envoie `G <idJoueur> <carte>`, le serveur compare à `deck[12]` et renvoie `W <id>` si accusation correcte.

Exemple :

— Côté serveur : `Data: [G 2 7]` signifie que le joueur 2 a accusé la carte 7.

— Côté client : `consomme |W 2|` signifie que le joueur 2 a trouvé le coupable.

6. Complétion du code expliquée

---

1) La partie `server.c`

case 'O'

```
case 'O':
{
    int senderId, symbole;
    sscanf(buffer, "O␣%d␣%d", &senderId, &symbole);
    for (int i = 0; i < 4; i++) {
        if (i == joueurCourant) continue;
        int presence = (tableCartes[i][symbole] > 0) ? 100 : 0;
        sprintf(reply, "V␣%d␣%d␣%d", i, symbole, presence);
        broadcastMessage(reply);
    }
    joueurCourant = (joueurCourant + 1) % 4;
    sprintf(reply, "M␣%d", joueurCourant);
    broadcastMessage(reply);
    break;
}
```

cette commande permet à un joueur de poser une question ouverte : "Qui parmi vous a ce symbole ?". Le serveur parcourt tous les joueurs (sauf celui qui pose la question) et regarde s'ils ont le symbole. Si oui, il envoie une étoile (100) à tout le monde. Sinon 0. Ce bloc montre bien l'usage du parcours de tableau, de la logique conditionnelle, et de la diffusion via socket (avec un broadcastMessage). j'ai choisi 100 pour symboliser une étoile dans l'affichage SDL (c'est comme ça qu'elle est reconnue dans le client). Et j'ai utilisé broadcastMessage() car tout le monde doit voir cette info sur la grille.

Ensuite, j'ai bien mis à jour joueurCourant pour passer au joueur suivant, ce qui est obligatoire pour garder la logique de tour de jeu. Un oubli ici aurait figé le jeu (à priori ce que je pense).

case 'G'

```
case 'G':
{
    int senderId, guess;
    sscanf(buffer, "G␣%d␣%d", &senderId, &guess);
    if (guess == deck[12]) {
        sprintf(reply, "W␣%d", joueurCourant);
```

```
        broadcastMessage(reply);
        fsmServer = 0;
    } else {
        joueurCourant = (joueurCourant + 1) % 4;
        sprintf(reply, "M␣%d", joueurCourant);
        broadcastMessage(reply);
    }
    break;
}
```

Ici, le joueur tente une accusation directe. On récupère la carte qu'il accuse (guess) et on la compare avec la carte coupable (toujours stockée à deck[12]). Si c'est la bonne, on envoie un message W à tout le monde pour annoncer la victoire, et on arrête la partie (fsmServer = 0). Sinon, la partie continue avec le joueur suivant. J'ai fait en sorte que le message "M" soit toujours envoyé pour garder le tour bien à jour. Ce bloc m'a permis d'appliquer une logique simple mais critique pour gérer la fin de partie, et l'usage de la variable d'état du serveur fsmServer.

case 'S'

```
case 'S':
{
    int senderId, cible, symbole;
    sscanf(buffer, "S␣%d␣%d␣%d", &senderId, &cible, &symbole);
    int quantite = tableCartes[cible][symbole];
    sprintf(reply, "V␣%d␣%d␣%d", cible, symbole, quantite);
    sendMessageToClient(tcpClients[senderId].ipAddress,
                        tcpClients[senderId].port,
                        reply);
    joueurCourant = (joueurCourant + 1) % 4;
    sprintf(reply, "M␣%d", joueurCourant);
    broadcastMessage(reply);
    break;
}
```

Ici, un joueur demande à un autre joueur spécifique combien de fois il a un symbole précis (ex : "est-ce que le joueur 2 a des carnets ?"). Le serveur lit la requête et renvoie directement au demandeur un message "V" avec le résultat. Les autres joueurs ne voient pas cette info (c'est secret). Puis comme d'hab, on passe au joueur suivant avec "M". Ça m'a fait bosser la notion de communication point-à-point via socket. Ce bloc m'a fait bosser la communication unicast (vers un seul client), et ça montre bien la différence entre les messages publics (O) et privés (S) dans le protocole du jeu...M

Distribution des cartes aux joueurs

Code expliqué ci-dessus dans la section 5.

2) La partie `sh13.c`

Envoi de la commande de connexion au clic sur "Connect"

```
sprintf(sendBuffer,"C␣%s␣%d␣%s",gClientIpAddress,gClientPort,gName);
// RAJOUTER DU CODE ICI
printf("%s\n", sendBuffer);
sendMessageToServer(gServerIpAddress,gServerPort,sendBuffer);
```

Quand on clique sur "Connect", le client doit envoyer au serveur son adresse IP, port, et nom. J'ai donc juste ajouté un appel à sendMessageToServer(...) pour envoyer la commande C. Le printf c'est pratique pour déboguer.

Envoi de l'accusation (G) si le joueur clique sur une carte coupable

```
sprintf(sendBuffer,"G␣%d␣%d",gId, guiltSel);
// RAJOUTER DU CODE ICI
printf("%s\n", sendBuffer);
sendMessageToServer(gServerIpAddress,gServerPort,sendBuffer);
```

Quand un joueur pense avoir trouvé le coupable, il clique sur la carte et appuie sur "Go". Le client envoie alors "G <id> <indice_carte > ".Simpleetefficace.

Envoi d'une question ouverte (O)

```
sprintf(sendBuffer,"O␣%d␣%d",gId, objetSel);
// RAJOUTER DU CODE ICI
printf("%s\n", sendBuffer);
sendMessageToServer(gServerIpAddress,gServerPort,sendBuffer);
```

Là, le joueur pose une question ouverte sur un symbole (objet). Le message O est envoyé à tous les joueurs, donc le client le prépare et le balance au serveur..

Envoi d'une question ciblée (S)

```
sprintf(sendBuffer,"S␣%d␣%d␣%d",gId, joueurSel, objetSel);
// RAJOUTER DU CODE ICI
printf("%s\n", sendBuffer);
sendMessageToServer(gServerIpAddress,gServerPort,sendBuffer);
```

Réception des messages du serveur (dans `switch(gbuffer[0])`)

- ***case 'I'*** *– réception de l'identifiant* `sscanf(gbuffer, "I %d", &gId);` C'est le tout premier message qu'on reçoit après connexion. Il nous donne notre identifiant gId, utilisé pour toutes les actions suivantes. Elementaire mon cher Watson n
- ***case 'L'*** *– réception de la liste des joueurs* `sscanf(gbuffer, "L %s %s %s %s", gNames[0], gNames[1],gNames[2],gNames[3]);` Le serveur nous envoie les pseudos des 4 joueurs connectés. Je les stocke dans gNames pour qu'ils s'affichent à côté de la grille.
- ***case 'D'*** *– réception des 3 cartes* `sscanf(gbuffer, "D %d %d %d", &b[0], &b[1], &b[2]);` C'est ici qu'on récupère nos 3 carte de départ. Le serveur nous envoie leurs indices dans le deck, et on les stocke pour pouvoir les afficher.
- ***case 'M'*** *– mise à jour du joueur courant*

```
int joueur;
sscanf(gbuffer, "M␣%d", &joueur);
goEnabled = (joueur == gId) ? 1 : 0;
```

Le serveur envoie le numéro du joueur dont c'est le tour. Si c'est nous, on active le bouton "Go". Sinon on attend. J'ai fait simple : une comparaison.

— ***case 'V'*** *– mise à jour de la grille de symboles*

```
int row = 0, col = 0, n = -1;
sscanf(gbuffer, "V␣%d␣%d␣%d", &row, &col, &n);
if (n == 100 && tableCartes[row][col] < 0)
    tableCartes[row][col] = n;
else
    tableCartes[row][col] = n;
```

Ce message met à jour une case de la grille (symbole pour un joueur donné). Si n == 100, ça veut dire "étoile" (vu dans les questions ouvertes), mais je n'écrase pas une valeur déjà connue. J'ai géré ce cas en évitant de remettre 100 si une vraie valeur était déjà présente.

— ***case 'W'*** *– fque j'ai créé alor qu'il était pas dans la structure du code de base :*

```
int winner;
sscanf(gbuffer, "W␣%d", &winner);
char msg[128];
sprintf(msg, "Le␣joueur␣%s␣a␣gagné␣la␣partie␣!", gNames[winner]);
SDL_ShowSimpleMessageBox(SDL_MESSAGEBOX_INFORMATION,
                         "Fin␣de␣la␣partie", msg, NULL);
goEnabled = 0;
```

Ce message indique qu'un joueur a gagné et saffiche 4 fois , pour prévenir chaqun des 4 joueurs. J'affiche une boîte de message SDL pour prévenir, puis je désactive le bouton "Go" pour bloquer les actions.

7. Bilan et conclusion

Ce projet m'a vraiment permis de mettre en pratique plusieurs notions qu'on avait vues en TP, mais dans un cadre beaucoup plus concret et motivant : créer un vrai jeu en réseau avec interface graphique. Ça m'a forcé à bien comprendre le fonctionnement des ***sockets TCP****, la gestion des* ***threads****, l'usage des* ***mutex****, et même un peu la structure d'un protocole personnalisé entre client et serveur.*

Même si beaucoup de code était déjà fourni, les parties à compléter étaient quand même critiques pour le bon déroulement du jeu. Il fallait comprendre comment chaque message circulait entre le serveur et les clients, et comment chaque action du joueur déclenchait une mise à jour visible côté interface. Ce n'était pas juste « ajouter du code pour que ça compile », mais vraiment intégrer les blocs dans une logique de communication réseau + affichage interactif.

J'ai pris le temps de bien commenter les zones que j'ai complétées, en essayant de garder un code propre, cohérent, et facile à relire. J'ai aussi appris à mieux déboguer les échanges TCP (ce qui n'est pas toujours fun ...), et à tester chaque interaction entre plusieurs clients.

Au final, ce projet m'a aidé à consolider mes bases en C, en réseau, et à découvrir SDL que je ne connaissais pas avant (j'ai surtout découvert que ça ne fonctionnait pas bien sur Windows, même avec un noyau Linux comme WSL). Et le fait de travailler sur un vrai petit jeu, ça rend les choses beaucoup plus motivantes. Franchement j'ai appris pas mal, même si ça n'a pas toujours été facile à mettre au point.