

Rapport projet OS USER

Introduction

Ce rapport décrit mon travail pour l'implémentation du jeu de plateau *Sherlock 13* dans une architecture client-serveur en langage C. Il y est expliqué comment je m'y suis pris pour la construction du programme (complétion du code C où c'était demandé), comment le programme fonctionne et quels aspects vus en TP ont été éventuellement utilisés (*socket*, *processus*, *pipe*, *threads*, *mutex*, etc.)

Information importante: je n'ai pas pu aller jusqu'à exécuter deux tours de joueurs consécutifs, sûrement un problème de gestion dans mon code et j'ai souvent rencontré pas mal de latence entre les clics dans une fenêtre client et l'envoi de requêtes entre le client et le serveur (parfois 5 minutes pour connecter un joueur!). Ces raisons font, en partie, que mon projet n'a pu être terminé correctement.

Aperçu de l'architecture client-serveur

Côté serveur, fichier `server.c`

- manipule la logique général du jeu et contrôle l'ordre des actions (connexions, actions de jeu, envois des messages, etc.);
- accepte les connexions et gère la communication avec chaque client (un client étant un joueur et comme le jeu se joue uniquement à quatre, il y a quatre clients);
- permet le mélange des cartes, leur distribution et les requêtes des joueurs (comme deviner quelle est la treizième carte ou qui possède certains objets);
- ordonne la communication TCP à l'aide des fonctions `socket`, `accept`, `connect`, `read`, `write` – communication qui est séquentielle (les tours de jeu déterminent l'ordre d'action des joueurs) et synchrone (on est en attente de l'action d'un joueur pour passer à la suite).

Côté client, fichier `sh13.c`

- interface graphique créée avec la librairie *SDL2*;
- exécute un thread serveur TCP pour rester à l'écoute des messages du serveur principal;
- gère les interactions du joueur avec la fenêtre (clic sur les boutons et les suppositions);
- utilise `pthread` pour gérer les messages entrants (du serveur principal) pour ne pas bloquer la boucle principale qui s'occupe des événements graphiques.

Complétion du code

Côté serveur, fichier `server.c`

- ajout d'une ligne permettant d'avoir un mélange différent à chaque appel de `melangerDeck`:

```
void melangerDeck() {  
    srand(time(NULL)); // Seed différente à chaque exécution  
    //...  
}
```

- ajout d'une fonction `updateTurn` pour mettre à jour explicitement le joueur ayant le tour côté serveur et en informer les joueurs:

```
/*#°#*/  
void updateTurn() {  
    joueurCourant = (joueurCourant + 1) % 4;  
    char reply[256];  
    sprintf(reply, "M %d", joueurCourant);  
    broadcastMessage(reply);  
}  
/*#!#*/
```

- déclaration de variables entières pour stocker le joueur actuel, l'objet sélectionné et la supposition d'un joueur lors des messages correspondants:

```
int joueurSel;  
    int objetSel;  
    int guiltSel;
```

- à la connexion des quatre joueurs, attribution à chacun trois cartes du deck, les blocs d'instructions sont similaires:

```
if (nbClients==4) {  
    // On envoie ses cartes au joueur X, et la ligne qui lui correspond dans tableCartes  
    // Rappel: Le joueur 0 possede les cartes d'indice I,J,K  
    sprintf(reply, "D %d %d %d", deck[I], deck[J], deck[K]);  
    sendMessageToClient(tcpClients[X].ipAddress, tcpClients[X].port, reply);  
    for (int j = 0; j < 8; j++) {  
        sprintf(reply, "V X %d %d", j, tableCartes[0][j]);  
        sendMessageToClient(tcpClients[X].ipAddress, tcpClients[X].port, reply);  
    }  
    // ...  
    // On envoie enfin un message global pour définir le joueur courant  
    sprintf(reply, "M %d", joueurCourant);  
    broadcastMessage(reply);  
    fsmServer=1;  
}
```

- au *switch case* sur la première lettre d'une requête de client, exécution de l'action correspondante (G, pour une supposition, un guess, sur la treizième carte; O, pour demander le nombre d'un certain objet que chaque joueur possède; S, pour demander à un joueur spécifique un objet).

```
switch (buffer[0]){
    case 'G': // G pour guilty //ajouté!
        sscanf(buffer, "G %d %d", &id, &guiltSel);
        printf("Joueur %d accuse la carte #%d comme 'Guilty'\n", id, guiltSel);
        if (guiltSel == deck[12]) {
            sprintf(reply, "W %d", id); // W for win
            broadcastMessage(reply);
            fsmServer = 0; // Reset the game
        } else {
            sprintf(reply, "F %d", id); // F for fail
            broadcastMessage(reply);
        }
        break;
    case 'O': // O comme dans Objets (?) - demande à tout le monde //ajouté!
        sscanf(buffer, "O %d %d", &id, &guiltSel);
        printf("Joueur %d demande l'Objet #%d\n", id, objetSel);
        // Compter le nombre d'objets de l'objetSel dans la tableCartes
        // et envoyer la réponse au joueur id
        int count = 0;
        for (int i = 0; i < 4; i++){ count += tableCartes[i][objetSel]; }
        sprintf(reply, "R %d %d", objetSel, count); // R comme dans Réponse
        sendMessageToClient(tcpClients[id].ipAddress, tcpClients[id].port, reply);
        updateTurn();
        break;
    case 'S': // Comme Objet mais on demande à un joueur particulier //ajouté!
        sscanf(buffer, "S %d %d %d", &id, &joueurSel, &objetSel);
        printf("Joueur %d interroge joueur %d à propos de l'Objet #%d\n", id, joueurSel,
objetSel);
        sprintf(reply, "R %d %d %d", joueurSel, objetSel,
tableCartes[joueurSel][objetSel]);
        sendMessageToClient(tcpClients[id].ipAddress, tcpClients[id].port, reply);
        updateTurn();
        break;
    default: break;
}
```

Côté client, fichier sh13.c

- lors d'un clic sur le bouton *Connect*:

```
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;  
}
```

- lors d'un clic sur les différentes options disponibles dans la fenêtre, qui se traduisent par différentes actions (demander un nombre d'objets, demande à un joueur spécifique, accusation):

```
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); //added  
    } else if ((objetSel!=-1) && (joueurSel==--1)){  
        sprintf(sendBuffer,"O %d %d",gId, objetSel);  
        sendMessageToServer(gServerIpAddress, gServerPort, sendBuffer); // added  
    } else if ((objetSel!=-1) && (joueurSel!=-1)){  
        sprintf(sendBuffer,"S %d %d %d",gId, joueurSel,objetSel);  
        sendMessageToServer(gServerIpAddress, gServerPort, sendBuffer); // added  
    }  
}
```

- si le client est synchronisé avec le serveur, le client doit pouvoir gérer les différents types de requêtes gérées par le serveur:

```
if (synchro==1){  
    printf("consomme |%s|\n",gbuffer);  
    int player, object; //ajouté;  
    switch (gbuffer[0]){  
        case 'I': // Message 'I' : le joueur reçoit son Id  
            sscanf(gbuffer, "I %d", &gId);  
            printf("Received ID: %d\n", gId);  
            break;  
        case 'L': // Message 'L' : le joueur reçoit la liste des joueurs  
            sscanf(gbuffer, "L %s %s %s %s", gNames[0], gNames[1], gNames[2], gNames[3]);  
            printf("Player list updated: %s, %s, %s, %s\n", gNames[0], gNames[1],  
gNames[2], gNames[3]);  
            break;  
        case 'D': // Message 'D' : le joueur reçoit ses trois cartes  
            sscanf(gbuffer, "D %d %d %d", &b[0], &b[1], &b[2]);  
            printf("Received cards: %d, %d, %d\n", b[0], b[1], b[2]);  
            break;  
  
        // Cela permet d'affecter goEnabled pour autoriser l'affichage du bouton go  
        case 'M': // Message 'M' : le joueur reçoit le n° du joueur courant  
            int currentPlayer;  
            sscanf(gbuffer, "M %d", &currentPlayer);  
            goEnabled = (currentPlayer == gId);  
            printf("Current player: %d, Go enabled: %d\n", currentPlayer, goEnabled);  
        }  
}
```

```
        break;

    case 'V': // Message 'V' : le joueur recoit une valeur de tableCartes
        int value;
        sscanf(gbuffer, "V %d %d %d", &player, &object, &value);
        tableCartes[player][object] = value;
        printf("Updated tableCartes[%d][%d] = %d\n", player, object, value);
        break;

    case 'R': // Cas des réponses du serveur à une action de joueur
        int count;
        if(sscanf(gbuffer, "R %d %d %d", &player, &object, &count) == 3) {
            // Réponse à 'S': requête à un joueur spécifique
            tableCartes[player][object] = count;
            printf("Player %d has %d of object %d\n", player, count, object);
        } else if(sscanf(gbuffer, "R %d %d", &object, &count) == 2) {
            // Réponse à une requête d'objet
            printf("Total count of object %d: %d\n", object, count);
        }
        break;

    case 'W': // en cas de victoire
        int winner;
        sscanf(gbuffer, "W %d", &winner);
        printf("Player %d won the game!\n", winner);
        goEnabled = 0; // désactiver le bouton go
        break;

    case 'F': // en cas d'échec de supposition
        int failed;
        sscanf(gbuffer, "F %d", &failed);
        printf("Player %d made an incorrect accusation\n", failed);
        break;
}
```

Communication

Initialisation

- dans un terminal, le serveur démarre à l'aide de la commande:

```
./server <server_port>
```

- dans un autre terminal (un pour chaque client), chaque client démarre avec:

```
./sh13 <server_ip> <server_port> <client_ip> <client_port> <player_name>
```

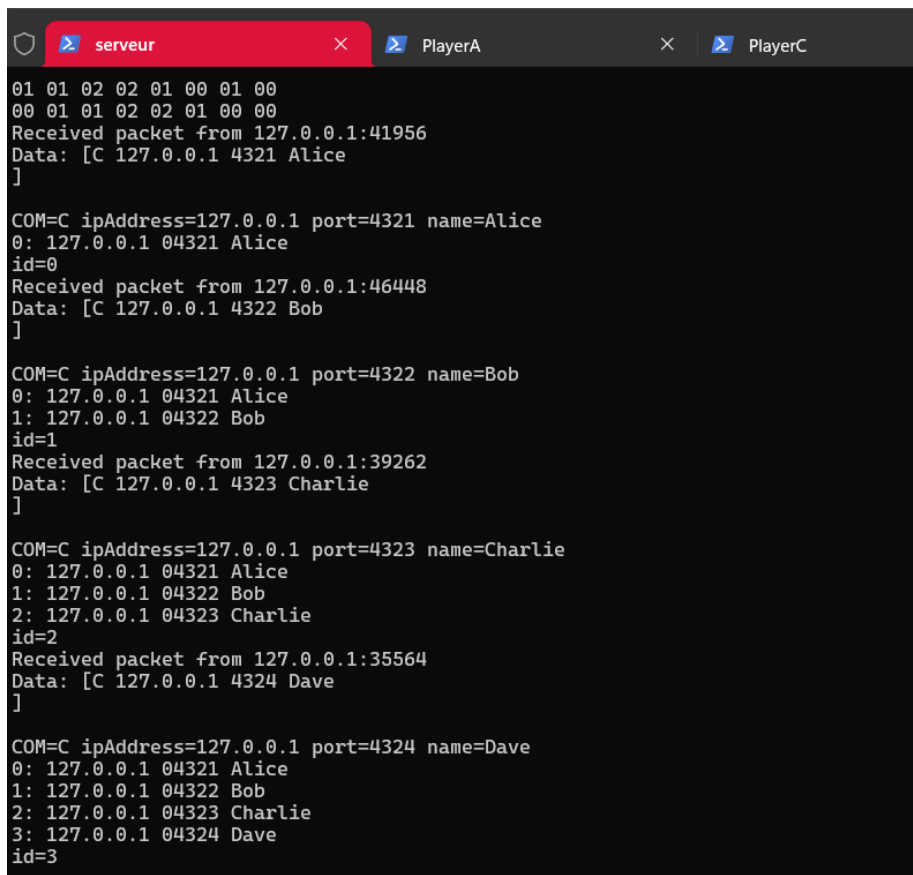
- un client envoie **C** (connect) au serveur pour amorcer une connexion:

```
C <client_ip> <client_port> <name>
```

- le serveur stocke cette information dans son tableau `tcpClients[]` et répond:

- **I** <id> pour assigner à chaque joueur son ID;
 - **L** <name1> <name2> <name3> <name4> pour communiquer à tous (broadcast) la liste des joueurs.
- quand quatre joueurs sont connectés:
 - il envoie trois cartes: **D** <card1> <card2> <card3> à chaque joueur;
 - il envoie un compte pour chaque objet: **V** <player_id> <object_id> <count>;
 - commence la partie: **M** <currentPlayer_id>

Les captures suivantes représentent certains de ces messages:



```
01 01 02 02 01 00 01 00
00 01 01 02 02 01 00 00
Received packet from 127.0.0.1:41956
Data: [C 127.0.0.1 4321 Alice
]

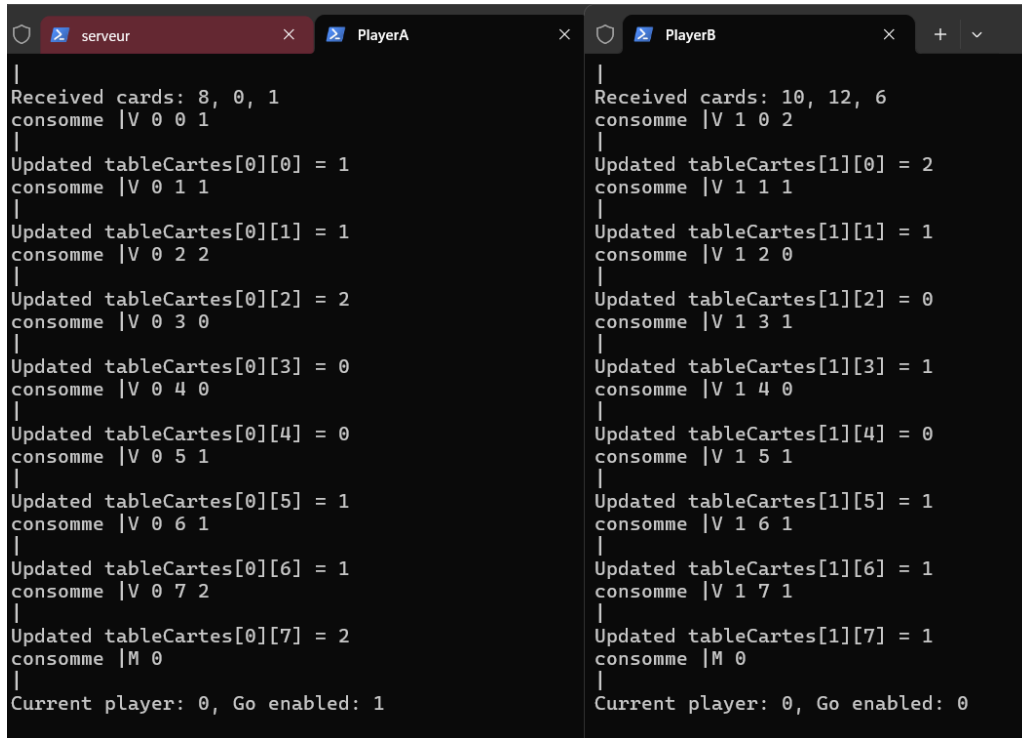
COM=C ipAddress=127.0.0.1 port=4321 name=Alice
0: 127.0.0.1 04321 Alice
id=0
Received packet from 127.0.0.1:46448
Data: [C 127.0.0.1 4322 Bob
]

COM=C ipAddress=127.0.0.1 port=4322 name=Bob
0: 127.0.0.1 04321 Alice
1: 127.0.0.1 04322 Bob
id=1
Received packet from 127.0.0.1:39262
Data: [C 127.0.0.1 4323 Charlie
]

COM=C ipAddress=127.0.0.1 port=4323 name=Charlie
0: 127.0.0.1 04321 Alice
1: 127.0.0.1 04322 Bob
2: 127.0.0.1 04323 Charlie
id=2
Received packet from 127.0.0.1:35564
Data: [C 127.0.0.1 4324 Dave
]

COM=C ipAddress=127.0.0.1 port=4324 name=Dave
0: 127.0.0.1 04321 Alice
1: 127.0.0.1 04322 Bob
2: 127.0.0.1 04323 Charlie
3: 127.0.0.1 04324 Dave
id=3
```

Figure 1:
connexion des
clients au serveur
et échange de
messages



```
|
Received cards: 8, 0, 1
consomme |V 0 0 1
|
Updated tableCartes[0][0] = 1
consomme |V 0 1 1
|
Updated tableCartes[0][1] = 1
consomme |V 0 2 2
|
Updated tableCartes[0][2] = 2
consomme |V 0 3 0
|
Updated tableCartes[0][3] = 0
consomme |V 0 4 0
|
Updated tableCartes[0][4] = 0
consomme |V 0 5 1
|
Updated tableCartes[0][5] = 1
consomme |V 0 6 1
|
Updated tableCartes[0][6] = 1
consomme |V 0 7 2
|
Updated tableCartes[0][7] = 2
consomme |M 0
|
Current player: 0, Go enabled: 1

|
Received cards: 10, 12, 6
consomme |V 1 0 2
|
Updated tableCartes[1][0] = 2
consomme |V 1 1 1
|
Updated tableCartes[1][1] = 1
consomme |V 1 2 0
|
Updated tableCartes[1][2] = 0
consomme |V 1 3 1
|
Updated tableCartes[1][3] = 1
consomme |V 1 4 0
|
Updated tableCartes[1][4] = 0
consomme |V 1 5 1
|
Updated tableCartes[1][5] = 1
consomme |V 1 6 1
|
Updated tableCartes[1][6] = 1
consomme |V 1 7 1
|
Updated tableCartes[1][7] = 1
consomme |M 0
|
Current player: 0, Go enabled: 0
```

Figure 2: distribution des cartes et déclaration du joueur qui commence

Actions de jeu en tour par tour

Chaque client, lorsque c'est son tour de jouer (déterminé par la variable `goEnabled`) peut envoyer comme message au serveur:

- G <id> <card> → "J'accuse cette carte."
- O <id> <object_id> → "Combien de fois cet objet apparaît-il?"
- S <id> <target_player_id> <object_id> → "Ce joueur possède-t-il cet objet?"

Le serveur peut répondre:

- W <id> → Victoire et fin du jeu.
- F <id> → Échec d'accusation et élimination.
- R <object_id> <count> ou R <player> <object> <count> → Réponse.
- M <current_player> → Indique le prochain tour.

Notions du cours utilisées

Les notions utilisées sont celles de la liste suivante.

- le *threading*, en partie, avec: `pthread_create` pour qu'un client lance un thread de serveur TCP qui est à l'écoute en permanence de messages entrants (`fn_serveur_tcp`).
- les sockets:
 - côté serveur
création de socket TCP: `socket(AF_INET, SOCK_STREAM, 0)`,
binding à un port: `bind()`,
écoute des messages de clients: `listen()`,
accepter les connexions de clients: `accept()`,
lire les message de clients: `read()`,
répond via une nouvelle connexion par message:
`sendMessageToClient()` est utilisée pour créer et fermer une socket pour chaque réponse.
 - côté client
agit comme un mini-serveur avec la fonction: `fn_serveur_tcp()` qui accepte les messages provenant du serveur principal,
`sendMessageToServer()` est utilisée pour communiquer avec ledit serveur principal.

Lorsque les exécutables sont lancés correctement, la partie démarre comme montré dans l'image ci-dessous (avec une fenêtre SDL et un terminal pour chaque client):

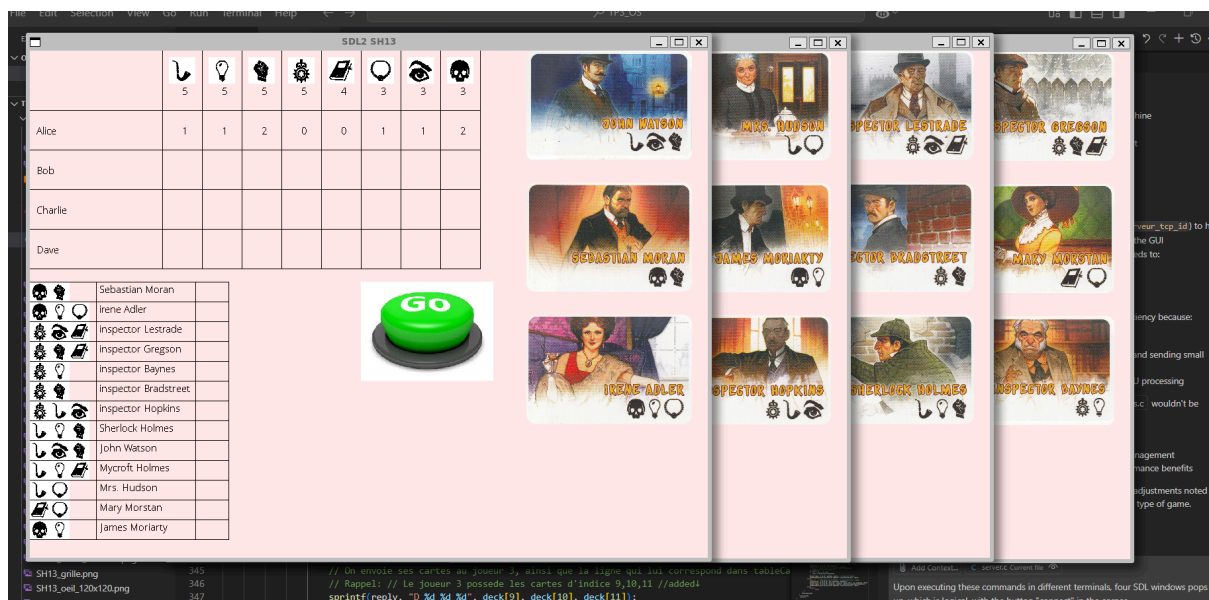


Figure 3: vue du jeu côté client