Rapport Projet Systèmes & Réseaux

KUT Kemal

2023/2024

Contents

1 Introduction **3**
1.1 Lancer le programme . . . 3

2 Gestion des cartes et des joueurs **3**
2.1 Une carte . . . 3
2.2 Un joueur . . . 3
2.3 Stockage des cartes . . . 4
2.4 Stockage des joueurs . . . 5

3 Côté Serveur **5**
3.1 Initialisation du serveur . . . 5
3.2 Gestionnaire de signaux . . . 6

4 Côté Client **6**

5 Communication Serveur/Client **6**

6 Le jeu **7**
6.1 Menu du jeu . . . 7
6.2 Ajouter un joueur . . . 7
6.3 Voir la liste des joueurs connectés . . . 7
6.4 Jouer : lancer une partie . . . 7
6.5 Déroulement de la partie . . . 8
6.6 Poser une carte . . . 8
6.7 Quitter . . . 9

7 Le robot **9**

8 La génération de statistiques **9**
8.1 Fichier Joueur . . . 9
8.2 Fichier scores . . . 9
8.3 Classement en PDF . . . 10

9 Démonstration du menu **10**

10 Mono machine vs multi machine **11**

11 Les améliorations à ajouter **11**

1 Introduction

Rapport du projet du cours de Systèmes & Réseaux du S5 de la L3. Le projet consistait à faire un jeu de cartes : 6 qui prend, en utilisant les méthodes, fonctionnalités vus en cours. **Suite à une incompréhension du jeu, certaines règles ont été modifiées. J'en ai discuté avec monsieur Lalou lors de ma présentation de projet, et il a été convenu de noter les différences au niveau des règles du jeu.**

- Le joueur choisit lui même la carte à poser et la rangée.
- Le jeu continue tant qu'un joueur n'atteint pas le score 66.

Même si les règles du jeu diffèrent un peu, les fonctionnalités que j'ai implémentées restent les mêmes, quelles que soient les règles du jeu.

Pour la gestion des joueurs et des cartes, j'ai opté pour l'utilisation de tableaux dynamiques. Cela me permet de manipuler les tableaux en fonction des situations, en adaptant leur taille en conséquence.

1.1 Lancer le programme

Il faut lancer le programme depuis le répertoire exec, tout d'abord jeu et ensuite joueur et robot autant de fois que vous voulez ajouter des joueurs (max 10).

2 Gestion des cartes et des joueurs

2.1 Une carte

Une carte est représentée par une structure qui comporte 2 membres, une variable de type int pour la valeur numérique de la carte et une autre variable de type int pour la valeur de tête de boeuf (attribué aléatoirement).

```
typedef struct{
    int valNum;
    int valTete;
} Carte;
```

2.2 Un joueur

Un joueur est représenté par une structure qui comporte plusieurs membres :

```
typedef struct {
    char pseudo[512];
    int socket;
    Carte *main;
    int nbCartes;
    int score;
} Joueur;
```

Le joueur a un pseudo, un descripteur de socket qu'il lui est assigné, un tableau de cartes (initalement composé de 10 cartes mais ensuite la taille du tableau varie en fonction des situations de la partie), un compteur de cartes, et un score (qui ne doit pas dépasser 66).

2.3 Stockage des cartes

Les 104 cartes du jeu sont initialement stockées dans un tableau dynamique nommé `cartes`. Au cours de la partie, les cartes sont distribuées au joueur ou au plateau et c'est pour cette raison que j'ai choisi de stocker les cartes dans un tableau dynamique (pour pouvoir manier la taille du tableau, enlever les cartes déjà distribuées).
Quand un carte est distribuée, je la marque en modifiant ces valeurs à 0, ensuite j'appelle une méthode pour "arranger" les cartes, cette méthode supprime toutes les cartes à 0 du paquet et réduit la taille du paquet. Cela permet d'avoir un tableau propre avec seulement les cartes restantes à jouer. **Attention : si une carte est supprimée du paquet de cartes, cela veut dire qu'elle est forcément soit posée dans le plateau de jeu, soit elle est dans le paquet du joueur. Aucune carte n'est perdue.** Le tableau cartes est initialisé au début de la partie dans la méthode `initManche`. Le tableau contient au début du jeu, les cartes ayant la valeur numérique allant de 1 à 104 avec des valeurs de têtes de boeufs générés aléatoirement.
J'ai choisi d'allouer dynamiquement de la mémoire pour stocker ces cartes car au fil de la partie, des cartes sont distribuées, sont placées dans le plateau de jeu et donc la taille du tableau varie. Avec une variable globale que j'utilise en guise de compteur de carte m'aide à gérer la taille du tableau tout au long de la partie (cette variable s'appelle `nbCartesTotal`).

Il y a également un tableau de cartes pour le plateau de jeu. Ce tableau n'a pas besoin d'être dynamique puisque sa taille ne varie jamais. C'est un tableau de 4 lignes/rangées avec 6 colonnes. Quand dans les cases du plateau, il n'y a pas de carte, les valeurs sont à 0. C'est comme ça que je différencie une case disponible dans le plateau.

- Tableau du paquet de cartes du jeu : cartes.
- Tableau du plateau de jeu : plateau.

```
1 Carte *cartes; //Tableau principal pour les cartes du jeu
2 int nbCartesTotal; //Compteur de cartes pour le tableau dynamique
3 Carte plateau[4][6]; //Tableau de jeu
```

J'ai également implémenté différentes méthodes pour gérer les cartes au cours de la partie :

- retirerCartesDistribuees
- retirerCartesJoueur
- redonnerCartes

La méthode **retirerCartesDistribuees** a pour objectif de retirer les cartes distribuées, c'est-à-dire les cartes qui ont déjà été utilisées ou attribuées à un joueur ou au plateau, du tableau initial de cartes. Pour ce faire, elle crée un tableau temporaire tab de type Carte avec une taille égale au nombre total de cartes. Ce tableau temporaire sera utilisé pour trier et stocker les cartes non distribuées.
Ensuite, la méthode parcourt le tableau `cartes` initial, vérifiant si la valeur numérique de chaque carte est différente de zéro. Les cartes dont la valeur numérique est non nulle sont considérées comme non distribuées. Ces cartes sont ensuite copiées dans le tableau temporaire tab, et le compteur cptCartes est incrémenté.
Une fois toutes les cartes non distribuées identifiées et copiées dans le tableau tab, la méthode **libère l'espace mémoire occupé** par le tableau initial `cartes` à l'aide de la fonction free.
Ensuite la méthode met à jour la variable nbCartesTotal pour refléter le nombre de cartes non distribuées, puis alloue un nouvel espace mémoire pour le tableau `cartes` avec la taille mise à jour. Les cartes non distribuées sont ensuite copiées depuis le tableau temporaire tab dans le tableau cartes. Enfin, la méthode libère l'espace mémoire utilisé par le tableau temporaire tab, car il n'est plus nécessaire.

```
1 void retirerCartesDistribuees(){
2     Carte tab = malloc(nbCartesTotal sizeof(Carte)); //tableau temporaire pour trier
3     int cptCartes = 0; //cpt de cartes non distribu es
4
```

```
    for (int i = 0; i < nbCartesTotal; i++){
        if (cartes[i].valNum != 0){
            //les cartes != 0 sont les cartes non distribu es
            tab[cptCartes].valNum = cartes[i].valNum;
            tab[cptCartes].valTete = cartes[i].valTete;
            cptCartes++;
        }
    }
    //on vide notre tableau initial
    free(cartes);
    nbCartesTotal = cptCartes;
    cartes = malloc(nbCartesTotal * sizeof(Carte));
    //on remet dans notre tab initial nos cartes non distribu es
    for (int i = 0; i < nbCartesTotal; i++){
        cartes[i].valNum = tab[i].valNum;
        cartes[i].valTete = tab[i].valTete;
    }
    free(tab); //plus besoin de ce tab
}
```

La méthode **retirerCartesJoueur** part du même principe que la méthode `retirerCartesDistribuees` mais elle s'applique aux cartes dans la main du joueur, je ne vais pas expliquer le code de cette méthode car c'est le même principe que la méthode retirerCartesDistribuees.
Pour bien refléter le jeu, j'ai implémenté une méthode `redonnerCartes` qui redistribue 10 cartes au joueur qui n'a plus de cartes (c'est une méthode optionnelle dans mon programme car je ne l'utilise pas, je l'ai seulement codé pour la simulation).

2.4 Stockage des joueurs

Le programme à un tableau dynamique pour stocker les joueurs, durant une partie le nombre de joueur peut varier de 2 à 10 donc j'ai opté pour un tableau dynamique. Initialement dans `initServ` le tableau est alloué à 0 joueurs mais lors de la connexion d'un client grâce à ma variable gloable `nbJoueurs` que j'utilise en guise de compteur je fais une réallocation de la mémoire à chaque nouvelle connexion de joueur (limité à 10 connexions) en incrémentant le compteur.

```
Joueur *joueurs; //Tableau des joueurs
int nbJoueurs; //Compteur de joueur
```

A chaque nouvelle connexion : la réallocation de mémoire que je fais dans **accepterJoueur**:

```
nbJoueurs++;
joueurs = realloc(joueurs, nbJoueurs * sizeof(Joueur));
```

3 Côté Serveur

Pour modéliser ce jeu, j'ai codé un serveur qui est en charge de tout. Il gère les connexions des clients (les joueurs), les règles, les communications, et les déconnexions.

3.1 Initialisation du serveur

Le serveur est la partie centrale de mon programme. Dans `GestionJeu` j'initialise le serveur via la méthode `initServ`. Le port et l'adresse IP sont des variables globales que j'ai définie dans le header de `GestionJeu`. Dans `initServ`, j'initialise le nombre de joueurs connectés à 0 et j'alloue le tableau dynamique nommé `joueurs` (je prépare le tableau pour la connexion des joueurs).
Important : Le serveur n'accepte pas tout de suite les connexions des clients. Pour pouvoir connecter un client au serveur, il faut passer par le menu du programme que j'expliquerai plus tard.

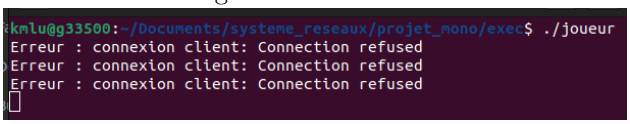
La méthode `initServ` est appelée dans le main de mon programme au tout début. C'est le point de départ du programme.

3.2 Gestionnaire de signaux

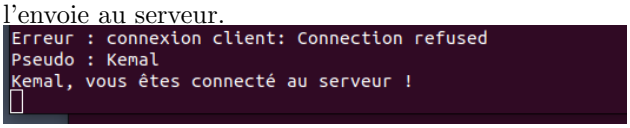
J'ai défini une méthode `handleSignal` qui est mon gestionnaire de signaux. Cette méthode gére les signaux associés à la terminaison du programme. Le code met en place des gestionnaires personnalisés pour les signaux SIGINT (généré par Ctrl+C) et SIGTERM (généralement utilisé pour demander une terminaison propre du programme). Lorsqu'un de ces signaux est reçu, la fonction handleSignal est appelée, effectuant différentes actions telles que la fermeture de la socket, l'affichage d'un message, l'appel d'une fonction pour quitter, et enfin la sortie du programme.

4 Côté Client

Le client qui va se connecter au serveur est enfaite un joueur. Dans `Joueur` j'initialise le client avec la méthode `initClient`, le client attend que le serveur accepte sa connexion (si il n'arrive pas à se connecter, il affiche un message d'erreur toutes les 2 secondes et réssaie la connexion).



Quand le serveur accepte la connexion, le client doit saisir le pseudo qu'il veut utiliser pour la partie et l'envoie au serveur.



Il y'a bien sûr une notification de connexion côté Serveur également que j'expliquerai plus tard.

Après la connexion réussie, le client se met dans une boucle while et attend des ordres de la part du serveur. Quand il reçoit un ordre (sous forme de chaîne de caractères), il compare à 3 conditions, si le mot reçu est afficher, il affiche le message reçu, si le mot reçu est saisir, le client demande à l'utilisateur de saisir, si le message reçu est quitter, le client quitte le serveur et se ferme proprement.

5 Communication Serveur/Client

Le serveur communique avec les joueurs via les méthodes `envoyerMsg` et `recevoirMsg`. Comme je l'ai expliqué au niveau du client, quand un client se connecte il se met en boucle en attente d'ordre du serveur. Il y a 3 différents ordres que le serveur peut envoyer au client : afficher un message, saisir et quitter.
La méthode `envoyerMsg` envoie l'ordre d'afficher, dans le programme général, cette méthode est utilisé pour afficher des messages, afficher le plateau de jeu, les cartes du jeu et du joueur, le score.

```
1  void envoyerMsg(int numJoueur, char *message){
2      char msg[512];
3      //envoi de l'ordre d'affichage
4      snprintf(msg, sizeof(msg), "afficher");
5      send(joueurs[numJoueur].socket, msg, sizeof(msg), 0);
6
7      //Envoi du msg
8      snprintf(msg, sizeof(msg), "%s", message);
9      send(joueurs[numJoueur].socket, msg, sizeof(msg), 0);
10 }
```

Coté client la boucle pour attendre les ordres :

```
1  while (clientCo){
2      //printf("En attente d'ordre...\n");
3      memset(msg, 0, sizeof(msg));
4      recv(clientSocket, msg, sizeof(msg), 0);
5
6      if (strcmp(msg, "afficher") == 0){
7          memset(msg, 0, 512);
```

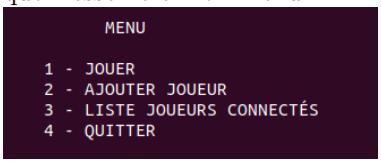
```
8           recv(clientSocket, msg, 512, 0);
9           printf("%s\n", msg);
10      }
11      else if (strcmp(msg, "saisir") == 0){
12          memset(msg, 0, 512);
13          scanf("%s", msg);
14          send(clientSocket, msg, 512 * sizeof(char), 0);
15      }
16      else if (strcmp(msg, "quitter") == 0){
17          clientCo = false;
18          close(clientSocket);
19      }
20  }
```

Pour la méthode `recevoirMsg` et `quitter` c'est le même principe que cette méthode `envoyerMsg`.

6 Le jeu

6.1 Menu du jeu

Après l'initialisation du serveur, le menu du jeu est affiché. Le menu est la partie centrale du jeu. Voici à quoi ressemble mon menu :



Mon menu comporte 4 choix possibles : lancer la partie, ajouter un nouveau joueur (humain ou robot), voir les joueurs connectés et quitter le programme.
Je vais expliquer dans l'ordre : l'ajout d'un nouveau joueur, voir la liste des joueurs connectés, lancer une partie puis quitter le programme.

6.2 Ajouter un joueur

Dans le jeu 6 qui prend il y'a des conditions pour jouer, il faut maximum 10 joueurs pour jouer une partie. J'ai implémenté une condition à vérifer avant l'ajout d'un nouveau joueur (humain comme robot), la condition est simplement un if qui vérifie si ma variable nbJoueurs est supérieure à 10 (cette variable est initalisée à 0 lors de l'initialisation du serveur au tout début et elle est incrémentée lors de la connexion d'un joueur). Si la condition n'est pas vérifiée, c'est à dire qu'il y'a déjà 10 joueurs de connectés, le message "Le salon est plein !" est affiché. Si la condition est vérifiée, le programme propose un deuxième "menu" pour choisir entre ajouter un humain ou un robot (ce menu sert à rien parce que l'ajout d'un humain ou d'un robot se fait avec la même méthode). Pour l'ajout d'un joueur, le serveur se met en attente en appelant la méthode `accepterJoueur` qui attend la connexion d'un nouveau client (joueur). Une fois le client connecté, la méthode incrémente le nombre de joueurs connectés, fait une réallocation dynamique sur le tableau de joueurs. Et pour finir, il demande au client de saisir un pseudo et créé un fichier via la méthode `creerFichier` avec un script Shell (je l'expliquerai plus tard dans le rapport). Après l'ajout du joueur, le menu est réaffiché.

6.3 Voir la liste des joueurs connectés

Pour voir la liste des joueurs connectés, c'est simple, une boucle for parcourt tous les joueurs via le tableau joueurs et affiche le pseudo de tous les joueurs. S'il n'y a pas de joueurs connectés, le message "Le salon est vide..." est affiché.

6.4 Jouer : lancer une partie

Pour lancer une partie, il y'a plein de condition. Le nombre minimum de joueurs requis est 2 et le maximum est 10. Si cette première condition n'est pas vérifiée, la partie ne se lance pas et le message "La partie ne

peut pas se lancer" est affiché. Si la condition est vérifiée, la méthode `jouerPartie` est appelée.
Dans cette méthode il y'a une variable booléenne pour marquer la fin de la partie (finPartie), elle est initialisée à false au début. Il y'a un compteur de manches cptManches et cptTours, le compteur de manches peut compter sans limites, il n'y a pas de consignes dans l'énoncé du rapport à propos du nombre de manche. Parcontre, le compteur de tours compte de 1 à 10 et repart à 1. Chaque manche à 10 tours (consigne de l'énoncé). Pour pouvoir suivre à qui est le tour de jouer durant la partie, j'ai déclaré une variable `tourJoueur` de type int qui s'incrémentera au nombre de joueurs durant la partie.

J'initialise les 104 cartes dans la méthode `initManche` qui est appelée après la déclaration des variables que j'ai expliquée juste avant. Cette méthode initialise les cartes du jeu, elle initialise également le plateau de jeu et distribue 10 cartes du paquet de 104 cartes qu'elle vient d'initaliser à chaque joueurs qui joue. Je vais expliquer les initialisations qu'elle fait :
Initialisation des cartes : Elle alloue de la mémoire pour le tableau de carte (nommé cartes) pour pouvoir stocker les 104 cartes du jeu. Pour chaque carte ajouté, une variable globale nbCartesTotal est incrémentée (elle nous servira à gérer la taille du tableau dynamiquement). Une boucle for allant de 1 jusqu'à 104 initialise chaque carte avec la valeur i de la boucle (pour la valeur numérique) et aléatoirement la valeur tête de boeuf (valeur aléatoire entre 1 et 7).
Initialisation du plateau Le plateau du jeu est un tableau de 4 lignes et 6 colonnes. Dans une boucle for qui parcourt chaque ligne du tableau, le programme tire aléatoirement une valeur entre 1 et le nombre total de cartes dans le tableau cartes et place cette carte dans la première case de la ligne, et elle met à 0 toutes les autres cases du tableau. Les cartes tirées du paquet de carte sont mit à 0 pour marquer qu'elles sont déjà distribuées. J'ai implémenté une méthode `retirerCartesDistribuees` pour retirer du paquet les cartes distribuées (pour ne pas les réutiliser). J'ai expliqué cette méthode au début de mon rapport.
Initialisation des cartes des joueurs : Une boucle for parcourt tous les joueurs, pour chaque joueur parcourut, elle alloue de la mémoire pour stocker 10 cartes. Il y'a une boucle for à l'intérieur de cette boucle qui parcourt chaque carte du joueur parcourut et lui attribue aléatoirement une carte du paquet de carte. Je remet les valeurs des cartes distribuées à 0 pour ensuite les enlever du paquet avec la mme méthode `retirerCartesDistribuees`. Après avoir distribuer les cartes, je fais une mise à jour des scores, chaque joueur a au total le nombre de têtes de boeufs qu'il a dans son paquet de carte comme score (cette règle n'est pas dans les règles du jeu).

6.5 Déroulement de la partie

Après avoir terminé l'initialisation de la manche, le programme rentre dans une boucle while qui tourne tant que ce n'est pas la fin de la partie. Dans cette boucle, la variable tourJoueur est mis à 0 et le compteur de tours à 1. Cette boucle représente une manche complète, il y'aura autant de manches que possibles tant que ce n'est pas la fin de la partie (avec la variable booléenne finPartie). Dans chaque manche, il y'a 10 tours donc j'ai une deuxième boucle while qui tourne tant que le compteur de tours est inférieur à 10 et ce n'est pas la fin de la partie. Pour chaque tour, le programme affiche le plateau de jeu à tous les joueurs, il envoie au joueur à qui c'est le tour de jouer son paquet de carte ainsi que son score. Il lui envoie un message informatif pour lui dire que c'est son tour de jouer. La méthode `poserCarte` est appelée sur le joueur à qui c'est le tour de jouer. Cette méthode est expliquée dans la partie suivante. Une fois que le joueur pose une carte, il y'a une condition if qui vérifie si le score du joueur a atteint ou est supérieur à 66. Si ce n'est pas le cas, la partie continue et la variable tourJoueur est incrémenté pour passer au joueur suivant (le compteur de tours est également incrémenté). Si la condition if n'est pas respectée donc un joueur a un score supérieur ou égale à 66 la partie s'arrête, tous les joueurs quittent la partie. On enregistre le résultat dans les fichiers respectifs de chaque joueur.

6.6 Poser une carte

La méthode `poserCarte` permet au joueur à qui c'est le tour de jouer de poser une carte dans le plateau de jeu. Lors de son tour, le joueur est invité à choisir une carte de sa main qu'il souhaite jouer. Cette sélection est soumise à des vérifications, garantissant que le numéro de la carte choisie est valide, c'est-à-dire compris entre 1 et le nombre total de cartes dans la main du joueur.

En parallèle, le joueur doit également indiquer la rangée dans laquelle il souhaite poser la carte sélectionnée. Encore une fois, une validation est effectuée pour s'assurer que le numéro de la rangée est dans la plage autorisée de 1 à 4. Le rôle principal de cette méthode est le placement de la carte dans la rangée choisie. Si le compteur de cartes dans cette rangée atteint la limite de 5 cartes, conformément aux règles du jeu, le joueur doit récupérer l'ensemble des cartes de cette rangée. En conséquence la taille de la main du joueur est augmentée pour pouvoir récupérer les cartes de la rangée. Les cartes récupérées sont transférées à la fin de la main, la carte qu'il voulait poser est enlevé de la main et placée à la première place de la rangée.
Si la rangée n'est pas pleine, le joueur place simplement sa carte à la première position disponible de cette rangée. En cas de succès, la carte est retirée de la main du joueur, ajustant ainsi la taille de celle-ci. Le score du joueur est également mis à jour pour refléter les modifications apportées à sa main. Avant d'effectuer le placement de la carte dans la rangée choisie, le programme compare la valeur de la carte sélectionnée avec celle de la dernière carte posée dans la rangée correspondante.
Si la valeur de la carte choisie est supérieure à celle de la dernière carte posée dans la rangée, l'action est autorisée, et la carte est correctement positionnée. Cela reflète la règle fondamentale du jeu, où les cartes doivent être placées dans une rangée dans un ordre croissant.
Cependant, si la valeur de la carte choisie est inférieure à celle de la dernière carte posée dans la rangée, le joueur récupère toutes les cartes de la rangée et pose sa carte à la première place.

6.7 Quitter

Quand la méthode `quitter` est appelée au cours du programme j'effectue plusieurs opérations pour quitter proprement le programme. Tout d'abord une boucle for parcourt tous les joueurs et leur envoi l'ordre de quitter pour les déconnecter, en même temps je ferme leur socket et je libére la mémoire utilisée pour stocker leur pseudo. Ensuite, je libére la mémoire pour les cartes du jeu, pour chaque paquet de carte des joueurs, pour chaque joueur. Pour terminer, j'affiche un message pour notifier que le programme se ferme.

7 Le robot

Dans mon programme, il n'y a pas trop de différences entre le joueur humain et le joueur robot. Le joueur robot se met en boucle après sa connexion et attend des ordres de la part du serveur. Cette fois l'affichage des messages est du même principe que le joueur humain la seule différence est la saisie (par exemple pour le choix de cartes ou de rangée). Pour que le robot saisisse lui même j'ai implenté une réponse aléatoire tout simplement. Quand il reçoit un ordre de saisie le robot répond en générant un chiffre entre 1 et 10. Le robot n'est pas intelligent. C'est une amélioration que je peux apporter au projet ultérieurement.
A la connexion du robot puisqu'on veut tout automatiser, le robot choisit aléatoirement un pseudo entre les pseudos que j'ai déjà définis (ronaldo_bot, mbappe_bot, messi_bot, etc...).

8 La génération de statistiques

8.1 Fichier Joueur

Quand un joueur se connecte à la partie, le serveur appelle la méthode `creerFichier` qui appelle à son tour un script Shell pour créer un fichier au pseudo du joueur. Si le fichier existe déja, le script écrit à la suite du fichier. Lors de la connexion d'un joueur pour avoir une trace de sa connexion, le script écrit dans le fichier la date et heure de connexion. À la fin de la partie, il ajoute dans ce fichier le résultat de sa partie si il a gagné ou perdu.

8.2 Fichier scores

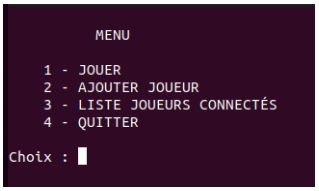
A la fin de la partie, un fichier qui contient le score de tous les joueurs de la partie est créé. Ce fichier se trouve dans le répertoire historique (logs/historique/scores.txt). Ce fichier est juste le récapitulatif de la dernière partie jouée. Un script Shell utilise ce fichier pour ensuite trier les scores et générer un fichier PDF (cf prochain paragraphe).

8.3 Classement en PDF

Un fichier PDF est créé à la fin de la partie dans le répertoire statistiques. Il récolte les lignes de textes du fichier scores dans le répertoire historique puis fait un classement des joueurs. Par la suite je pourrai ajouter des fonctionnalités comme la moyenne de chaque joueur (en allant récupérer les données dans les fichiers des joueurs), la moyenne des parties gagnées etc...

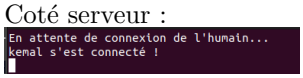
9 Démonstration du menu

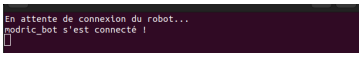
Lancement du serveur + affichage du menu :



Ajout de joueurs (1 humain et 1 robot) :

- Coté serveur :

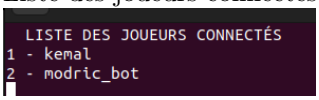




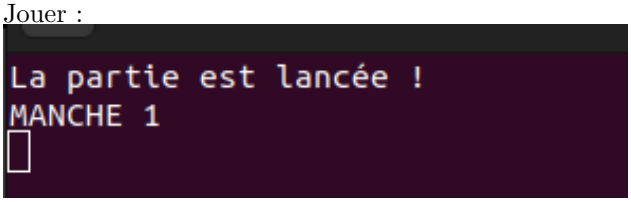
- Coté humain :

- Coté robot :

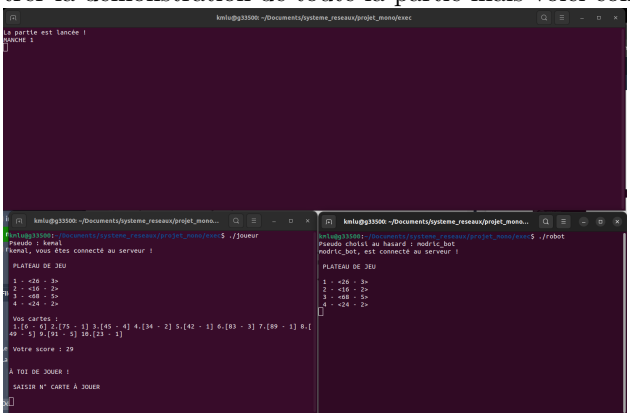
Liste des joueurs connectés :



Jouer :



Le serveur lance la partie ensuite, impossible de montrer la démonstration de toute la partie mais voici comment cela se passe :



10 Mono machine vs multi machine

Le programme en mono machine marche parfaitement sans soucis de communication. Le programme en multi machines a quelques problèmes que je n'ai pas eu le temps de corriger. Le serveur se lance, il accepte des connexions, les clients arrivent à se connecter, le serveur fonctionne parfaitement, les clients également mais lors du lancement de la partie il y'a un problème de communication le message d'envoi du plateau est incomplet. Je pense que si le problème de communication est résolut le programme fonctionnera sans aucun soucis en multi-machines.

11 Les améliorations à ajouter

Classement PDF :
Actuellement, la génération du classement en PDF est absente à la fin de chaque partie, il y'a seulement un fichier pdf vide qui est généré. Intégrer cette fonctionnalité serait utile pour documenter et partager les résultats de manière triée.
Robot intelligent :
Le robot actuel effectue des choix aléatoires de chiffres de 1 à 10 pour la sélection de cartes ou de rangées. Pour une expérience plus immersive, il serait avantageux d'intégrer une logique intelligente au robot, lui permettant de prendre des décisions plus stratégiques.
Redistribution des Cartes : Dans le cas où le joueur n'a plus de cartes, une redistribution automatique de 10 cartes devrait être mise en place, surtout dans le contexte où d'autres robots ne possèdent pas d'intelligence. Cette fonctionnalité est cruciale pour garantir la continuité du jeu.
Statistiques plus précis :
Pour rendre le jeu encore plus intéressant, on pourrait ajouter des stats sympas, combien de points en moyenne chaque joueur fait et combien de temps dure une partie. On pourrait extraire toutes ces infos des fichiers texte des joueurs et avoir une super analyse pour voir comment tout évolue dans le jeu. Ça rendrait les choses plus fun et on pourrait vraiment voir qui est le pro du jeu !
Un makefile pour pouvoir compiler sans problèmes. Pour l'instant pour compiler (depuis le répertoire Code):

- gcc GestionJeu.c -o ../exec/jeu
- gcc Joueur.c -o ../exec/joueur
- gcc Robot.c -o ../exec/robot