

# Systemes et réseaux

---

RAPPORT PROJET

Mohamed BOUCHENGUOUR  
Mehdi ASNI  
TP GROUPE 6

ANNÉE UNIVERSITAIRE 2022-2023

## Table des matieres

Introduction.....	3
Objectifs .....	3
Structure .....	3
Fonctions utilisees.....	4
void clean_stdin().....	4
void fin(int) .....	4
Deroulement d'une partie.....	4
Strategie du joueur robot.....	13
Comparaisons statistiques : .....	13
Conclusion .....	14

## Introduction

Dans ce rapport, nous présenterons notre projet consistant à développer le jeu « 6 qui prend » en utilisant les langages shell, awk et c.

## Objectifs

L'objectif de ce projet est de manipuler les différents langages étudiés durant le semestre et de réaliser un programme qui utilise la communication inter-processus. En effet, notre programme aura 3 types de processus :

- Le gestionnaire du jeu : processus qui gère le déroulement d'une partie. C'est-à-dire un processus qui va accueillir les joueurs, distribuer les cartes, poser les cartes des joueurs sur le plateau...
- Le joueur Humain : processus qui affiche les cartes/plateau à un utilisateur, lui demande de choisir une carte/emplacement, afficher à l'utilisateur les informations reçues par le serveur.
- Le joueur robot : processus qui joue automatiquement au jeu comme un joueur Humain.

## Structure

Dans notre programme, nous avons 2 fichiers principales :

- Le fichier tcpServerPoll qui correspond au gestionnaire du jeu. Le lancement de ce fichier lance une partie.
- Le fichier tcpClientPoll qui correspond au joueur Humain. Le lancement de ce fichier connecte le joueur à la partie lancée par le gestionnaire du jeu.

Pour gérer la communication inter-processus, nous avons utilisé des sockets. Le gestionnaire du jeu sera le serveur de la partie et les joueurs Humains seront les clients. Dans le serveur, nous avons utilisé une structure poll. En effet, 6 qui prend est un jeu qui peut se jouer jusqu'à 10 personnes. Lorsque tous les joueurs choisissent leurs cartes, ils envoient leurs réponses au serveur. Le problème est que la fonction read est une fonction bloquante qui permet de lire la réponse d'un seul client à la fois. C'est-à-dire que lorsque le serveur attend une réponse de tous les clients, si nous n'utilisons pas de poll, le serveur va d'abord utiliser la fonction read sur le client numéro 1 puis 2 puis 3, etc. Si le client numéro 1 envoie sa réponse en dernier, le traitement des autres clients se feront après la réception du message du client numéro 1. Cependant, si nous utilisons un poll, le traitement se fera au fur et à mesure car la structure poll écoute tous les descripteurs clients en même temps ce qui permet d'avoir une application plus optimisée.

Dans notre application, toutes les fonctionnalités supplémentaires ont été implémentées.

Coté serveur et client, les messages reçus seront stockés dans une variable *buf*. Cette variable est une chaîne de caractères de taille 128. Les messages envoyés auront également une taille de 128 pour éviter tous problèmes.

Durant une partie, le client sera dans une boucle tant que la partie n'est pas finie (partie qui se termine lorsque le serveur ferme la connexion (envoi d'un message de taille 0)). Les messages reçus seront traités dans la fonction traitement qui réalise des actions spécifiques en fonction du message reçu.

```
while(partie){
    int bufSize = read(descClient,buf,TAILLE_BUF);
    if(bufSize < 0)
        panic("Erreur de lors de la lecture des données");
    if(bufSize == 0)
        partie = false;
    else
        traitement(buf);
}
```

Figure 1 - Boucle qui reçoit les messages du serveur tant que la connexion est ouverte

### Fonctions utilisées

void clean\_stdin()

Cette fonction permet de vider le buffer de saisie du clavier. Elle est nécessaire lorsque l'utilisateur saisit une mauvaise donnée (chaîne au lieu d'un entier par exemple) pour qu'il puisse recommencer sa saisie. Cette fonction est implémentée côté serveur et client.

void fin(int)

Cette fonction se déclenche lorsque l'utilisateur appuie sur Ctrl+C. Cette fonction ferme les descripteurs (client/serveur). Côté serveur, les fichiers générés sont supprimés (fichiers que nous allons aborder plus tard dans le rapport). Cette fonction est implémentée côté serveur et client.

Le programme serveur a une fonction envoyerMessage qui envoie un message à tous les clients.

L'insertion de données dans le fichier log se fait à l'aide de la commande **echo**, exécuté à l'aide de la fonction **system**.

### Déroulement d'une partie

Tout d'abord, nous lançons une partie en exécutant le programme tcpServerPoll. Le serveur demande tout d'abord le nombre de joueurs, le nombre de robots, le nombre de manches et le nombre de têtes de bœufs qui déclenche la fin de la partie. Lorsque l'utilisateur saisit le nombre de joueurs, le nombre de joueurs robots est calculé. En effet, 6 qui prend est un jeu qui se joue avec minimum 2 personnes et au maximum 10. Si l'utilisateur saisit 0 joueur, 2 robots sont au minimum requis, s'il y a un seul joueur, un robot au minimum est requis... Cela permet de gérer la saisie de l'utilisateur. Une fois que l'utilisateur a choisi ses paramètres de la partie, le serveur se lance et attend la connexion des joueurs.

```

Nombre de joueurs (min 0, max 10): 2
Nombre de robots (min 0, max 8): 2
Nombre de manche (0 si jusqu'à défaite) : 3
Nombre de tête de boeuf (minimum 1): 66
Serveur actif sur port 55555

```

Figure 3 - Terminal demandant les paramètres à l'utilisateur

```

//On attend la connexion des joueurs
while(attenteJoueur){
    printf("Attente de connexion\n");
    int pollResult = poll(pollfds, useClient + 1, 5000);
    if (pollResult > 0)
    {
        if (pollfds[0].revents & POLLIN)
        {
            struct sockaddr_in cliaddr;
            unsigned int addrlen = sizeof(cliaddr);
            int client_socket = accept(server_socket, (struct sockaddr *)&cliaddr, &addrlen);
            printf("Succes accept %s\n", inet_ntoa(cliaddr.sin_addr));
            for (i = 1; i <= joueurs; i++)
            {
                if (pollfds[i].fd == 0)
                {
                    pollfds[i].fd = client_socket;
                    pollfds[i].events = POLLIN | POLLPRI;
                    useClient++;
                    write(pollfds[i].fd, &i, sizeof(i)); //On envoie aux clients le numéro du joueur
                    break;
                }
            }
        }
    }
    if(useClient == joueurs){
        attenteJoueur = false;
    }
}

```

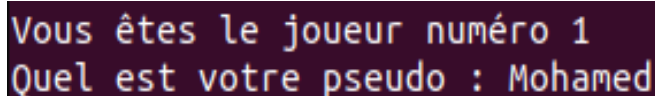
Figure 3 - Code qui permet d'attendre la connexion des clients

Notre application peut se jouer en réseau. Pour se connecter, le client doit mettre en argument l'adresse IP du serveur. Pour jouer en local, l'argument sera « localhost ».

```
./tcpClientPoll localhost
```

Figure 4 - Connexion au jeu en local

Lorsqu'un joueur se connecte, le serveur lui attribue un numéro. Le premier client à se connecter sera le joueur numéro un, le deuxième joueur aura le numéro deux, etc. Attribué un numéro à chaque client permet de les gérer plus facilement. Le serveur envoie ensuite au client qui vient de se connecter son numéro et le client l'affiche à l'utilisateur. Ensuite, le processus client demande à l'utilisateur de saisir un pseudo. Le client envoie ce pseudo au serveur. Nous avons décidé d'attribuer à chaque joueur un pseudo afin d'avoir une expérience de jeu plus agréable. De plus, cela permet d'avoir un fichier statistique plus fonctionnelle. C'est-à-dire qu'un joueur avec un pseudo sera toujours le même alors que si un joueur est identifié par un numéro, 2 joueurs différents peuvent avoir le même numéro d'une partie à une autre ce qui faussera les statistiques.

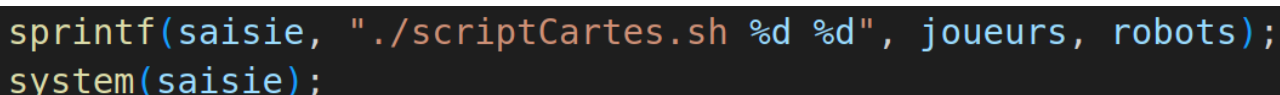


```
Vous êtes le joueur numéro 1  
Quel est votre pseudo : Mohamed
```

Figure 4 - Demande de saisie d'un pseudo côté client

Le pseudo de tous les joueurs sera stocké dans un tableau de chaîne de caractères « pseudo ». L'indice du tableau correspond au numéro du joueur (le pseudo du joueur numéro 1 sera stocké à l'indice 1 du tableau). Les robots ont également un pseudo qui sera initialisé par le serveur. Ces pseudos sont stockés après les pseudos des joueurs (le pseudo du robot numéro 1 (robot1) sera stocké à l'indice 3 s'il y a 2 joueurs humains par exemple). Les joueurs ayant un numéro inférieur au nombre de joueurs seront des joueurs humains et les numéros supérieurs seront des joueurs robots. Un tableau vie est également initialisé à 0. Ce tableau stocke le score des joueurs.

Une fois tous les pseudos récupérés, le serveur lance la première manche. Au début de chaque manche, le serveur lance le script shell **scriptCartes.sh**. Ce script prend 2 paramètres : le nombre de joueurs humains et le nombre de joueurs robots. Ce script ne retourne aucune valeur, on le lance donc à l'aide de la fonction **system**.



```
sprintf(saisie, "./scriptCartes.sh %d %d", joueurs, robots);  
system(saisie);
```

Figure 6 - Lancement du script scriptCartes.sh

Ce script se déroule de la façon suivante :

- Création d'un fichier vide cartes (fichier qui contiendra toutes les cartes du jeu), ordreJoueur (fichier qui contiendra les valeurs des cartes jouées par les joueurs) et log s'il n'existe pas (fichier log de la partie).
- Insertion des cartes (comprise entre 1 et 104) dans le fichier cartes.
- On trie aléatoirement ce fichier à l'aide de la commande **sort**.
- Pour chaque joueur humain, on récupère les 10 premières lignes du fichier cartes qu'on insère dans un fichier carteJoueur (le joueur numéro 1 aura ses cartes stockées dans le fichier carteJoueur1, le joueur numéro 2 aura ses cartes stockées dans le fichier carteJoueur2) puis on supprime les 10 premières lignes du fichier cartes pour éviter que 2 joueurs aient les mêmes cartes.
- Pour chaque joueur robot, on réalise la même procédure à la seule différence que les cartes d'un robot seront stockées dans un fichier carteRobot (le robot numéro 1 aura ses cartes stockées dans le fichier carteJoueur3 s'il y a 2 joueurs humains, le joueur numéro 2 aura ses cartes stockées dans le fichier carteJoueur5 s'il y a 3 joueurs humains, etc.).
- On récupère les 4 premières cartes du fichier cartes qu'on insère dans un fichier plateau (fichier qui stocke le plateau du jeu).
- Puis on supprime le fichier cartes.

On insère ensuite dans le fichier log « Début de la manche 1 ».

Une fois que nous avons les cartes pour tous les joueurs humains et robots, on envoie les cartes des joueurs humains aux processus clients. Pour ceci, on envoie d'abord le message « Cartes » aux clients. À la réception de ce message, les clients entrent dans une boucle for allant de 0 à 9. Dans cette boucle for, il y a un read dont le message reçu sera stocké dans un entier. C'est-à-dire que le client attend 10 entiers de la part du serveur. Pour envoyer les cartes aux joueurs, le serveur ouvre le fichier

carteJoueur d'un joueur à l'aide de la commande cat. Cette commande est exécutée à l'aide d'un popen. Puis le serveur envoie chaque ligne du fichier au joueur.

```
for (i = 1; i <= joueurs; i++){
    sprintf(cmd, "cat carteJoueur%d", i);

    if ((fp = popen(cmd, "r")) == NULL)
        panic("Erreur lors de l'ouverture du tube\n");

    while(fgets(buf, TAILLE_BUF, fp) != NULL){
        carte = atoi(buf);
        write(pollfds[i].fd, &carte, sizeof(carte));
    }

    if (pclose(fp))
        panic("Erreur lors de la fermeture du tube\n");
}
```

Figure 5 - Envoie des cartes à un joueur

Le client stock ensuite les cartes dans un tableau d'entier de taille 10 (cartes).

```
//Reception des cartes par le serveur
else if(strcmp(reponse, "Cartes") == 0){
    for(int i = 0; i < 10; i++){
        int bufSize = read(descClient, &carte, sizeof(carte));

        if(bufSize < 0)
            panic("Erreur lors de la lecture des données");
        else
            cartes[i] = carte;
    }
}
```

Figure 6 - : Réception des cartes

Une fois les cartes envoyées, le serveur commence le tour et envoie le message « Début du tour » aux processus clients. Ce message sera affiché aux utilisateurs.

Au début de chaque tour, le serveur envoie le message « Plateau du jeu : » aux clients. À la réception de ce message, les clients affichent le message « Plateau du jeu : » aux utilisateurs puis rentrent dans une boucle for allant de 0 à 3. Dans cette boucle, il y a un read. C'est-à-dire que le processus client attend 4 messages de la part du serveur. Ces 4 messages correspondent à chaque ligne du plateau. Pour les envoyer aux clients, le serveur exécute la commande **cat plateau** à l'aide d'un popen. Il envoie ensuite chaque ligne du plateau aux clients. A la réception d'une ligne, le processus client affiche les têtes de bœufs de chaque carte (calculé grâce à la fonction **nbTete(int carte)**) avec la valeur de la carte juste en dessous. Pour calculer le nombre de têtes de bœufs, nous avons utilisé les vraies règles du jeu :

- Les cartes qui finissent par 5 possèdent 2 têtes de bœufs.
- Les cartes qui finissent par 0 possèdent 3 têtes de bœufs.
- Les cartes formant un doublet (11, 22, etc.) possèdent 5 têtes de bœufs.

- La carte 55 est à la fois un doublet et un nombre 5, cette carte contient donc 7 têtes de bœufs.

```

Plateau du jeu :

(1)      (1)
3        14

(1)
94

(5)
22

(1)      (1)      (1)
26       68       78
  
```

Figure 7 - Exemple d'affichage du plateau

On insère dans le log le plateau du jeu.

Ensuite, le serveur envoie aux clients le message « Choisir cartes ». Ensuite, le serveur attend la réponse des clients.

```

while(attenteReponses){
    int pollResult = poll(pollfds, useClient + 1, 5000);
    if (pollResult > 0)
    {
        for (i = 1; i <= joueurs; i++)
        {
            if (pollfds[i].fd > 0 && pollfds[i].revents & POLLIN)
            {
                int bufSize = read(pollfds[i].fd, &carte, sizeof(carte));
                if (bufSize <= 0)
                {
                    printf("Au revoir\n");
                    pollfds[i].fd = 0;
                    pollfds[i].events = 0;
                    pollfds[i].revents = 0;
                    useClient--;
                }
            }
            else
            {
                buf[bufSize] = '\0';
                nbReponse++;
                if(nbReponse == joueurs)
                {
                    attenteReponses = false;
                    sprintf(saisie, "echo %d %d >> ordreJoueur", carte, i); // On stocke la saisie des joueurs
                    system(saisie);
                    wait(0);
                }
            }
        }
    }
}
  
```

Figure 8 - Code qui permet d'attendre la réception des messages



À la réception du message, les clients affichent les cartes du joueur avec le nombre de têtes de bœufs au-dessus puis demandent à l'utilisateur de saisir une valeur.

```
Voici vos cartes :

(3)    (1)    (1)    (1)    (1)    (1)    (1)    (3)    (1)    (1)
90      3     42     16     59     2     12     50      9     97

Choisissez une carte (mettre valeur) : 3
Saisie correct, en attente des autres joueurs
```

Figure 10 - Affichage des cartes du joueur

Seules les cartes dont la valeur est différente de 0 sont affichées.

```
for(int i = 0; i < 10; i++){
    if(cartes[i] != 0)
        printf("%d\t", cartes[i]);
}
```

Figure 9 - Affichage des cartes

En effet, lorsqu'un utilisateur joue une carte, celle-ci est mise à 0 afin de ne pas pouvoir la réutiliser lors du tour suivant.

```
while(!valide){
    printf("\n\nChoisissez une carte (mettre valeur) : ");
    scanf("%d", &carte);

    for(int i = 0; i < 10; i++){
        if((cartes[i] != 0) && (cartes[i] == carte)){
            valide = true;
            cartes[i] = 0;
        }
    }

    if(valide){
        printf("Saisie correct, en attente des autres joueurs\n");
        write(descClient, &carte, sizeof(carte)); //On envoie la réponse au serveur
    }
    else{
        clean_stdin();
        printf("Saisie incorrect, veuillez recommencer\n");
    }
}
```

Figure 10 - Code qui demande à l'utilisateur de saisir une carte

Lorsque le joueur saisit une bonne carte, le client envoie au serveur la carte jouée.

À la réception du message d'un client, le serveur met dans le fichier ordreJoueur le numéro de la carte ainsi que le numéro du joueur qui a posé la carte à l'aide de la commande **echo**.

```
sprintf(saisie, "echo %d %d >> ordreJoueur", carte, i);
system(saisie);
```

Figure 11 - Insertion des cartes dans le fichier ordreJoueur

Une fois les saisies des joueurs humains reçus, nous faisons jouer les joueurs robots. Pour ceci, le serveur crée un processus fils à l'aide d'un fork pour chaque joueur robot.

```
int p = pipe(tube);
pidRobot = fork();
```

Figure 12 - Création d'un processus (robot)

Le processus fils (robot) va exécuter le script shell **robotIntelligent.sh** qui prend en paramètre le numéro du joueur robot à l'aide d'un **popen**. Ce script retourne une carte du joueur robot en fonction d'une stratégie que nous aborderons plus tard dans le rapport. Le processus fils (robot) envoie ensuite la carte récupérée au processus père (gestionnaire du jeu) à l'aide d'un tube. Le serveur met dans le fichier `ordreJoueur` le numéro de la carte ainsi que le numéro du joueur robot qui a posé la carte à l'aide de la commande **echo**.

Une fois toutes les réponses reçues, le serveur trie par ordre croissant le fichier `ordreJoueur` à l'aide de la commande **sort -n -o ordreJoueur ordreJoueur** afin d'avoir l'ordre de placement des cartes des joueurs. La première ligne correspondra donc à la première carte à poser. On récupère chaque ligne de ce fichier puis on envoie à tous les clients les cartes jouées par les autres utilisateurs.

```
Robot2 a joué la carte 31
Mohamed a joué la carte 38
Mehdi a joué la carte 67
Robot1 a joué la carte 69
```

Figure 13 - Affichage à l'utilisateur des cartes jouées par les autres joueurs

On insère ensuite dans le log les cartes jouées par les joueurs.

Ensuite, le serveur récupère la première ligne du fichier `ordreJoueur` (valeur de la carte et numéro du joueur). Il informe ensuite aux clients la carte qui est en train d'être posée. Le client l'affiche à l'utilisateur.

```
Placement de la carte de Mehdi
```

Figure 14 - Message lors du placement d'une carte

Ensuite, il lance le script shell **cartePlacement.sh** avec en paramètre la valeur de la carte.

Ce script permet de placer une carte sur le plateau. Il marche de la façon suivante :

- On applique sur chaque ligne du plateau le script awk **ranger.awk** avec en paramètre la carte que l'on souhaite placer sur le plateau. Ce script retourne 0 si la carte est inférieure à la dernière carte de la ligne du plateau. Sinon, il envoie la valeur de la dernière carte de la ligne. Sur les 4 lignes, on récupère la ligne du plateau qui possède la dernière carte la plus élevée.
- Si on peut poser la carte sur une ligne, on récupère le nombre de carte sur cette ligne avant de poser la carte à l'aide du script awk **'NR=='\$emplacement' {print NF}' plateau**. On récupère le nombre de têtes de bœufs de la ligne du plateau avant de poser la carte à l'aide du script awk **nbTete.awk** avec en paramètre la ligne du plateau. Ce script calcule pour la ligne donnée le nombre de têtes de bœufs

de chaque ligne à l'aide des règles du jeu. On place ensuite la carte sur le plateau à l'aide du script awk **placementCarte.awk** appliqué sur le fichier plateau. Ce script prend en paramètre une carte et un emplacement puis place la carte à l'emplacement souhaité. Si la ligne du plateau en question a déjà 5 cartes, celles-ci sont supprimées puis la carte est posée à la place.

- Enfin, le script envoie le numéro de la ligne, le nombre de cartes sur la ligne avec le nombre de têtes de bœufs (qui seront à 0 si la carte ne peut pas être posée).

On exécute le script shell **cartePlacement.sh** à l'aide d'un popen afin de récupérer les différentes données retournées par le script. Le serveur vérifie ensuite si la carte a été posée (en vérifiant si le numéro de la ligne est différent de 0). Si la carte ne peut pas être posée, le joueur regarde si la carte a été jouée par un humain ou un robot (à l'aide du numéro du joueur).

Si c'est un joueur humain, le serveur envoie à son processus le message « Placement impossible ». À la réception de ce message, le processus client demande à l'utilisateur de saisir un emplacement entre 1 et 4 puis envoie la réponse au serveur.

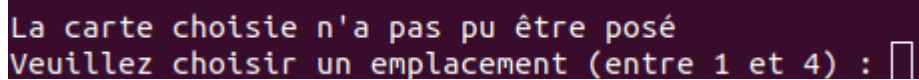


Figure 15 - Demande de saisie d'un emplacement

Si c'est un joueur robot, un processus fils est créé à l'aide d'un fork. Le processus robot lance le script awk **robotEmplacement.awk** sur le fichier plateau. Ce script calcul le nombre de têtes de bœufs de chaque ligne du plateau et retourne la ligne du plateau ayant le moins de têtes de bœufs. Le processus fils (robot) envoie au processus père (gestionnaire du jeu) l'emplacement souhaité.

Lors de la réception de l'emplacement, le serveur exécute le script awk **nbTete.awk** sur le fichier plateau avec l'emplacement reçu en paramètre. On rappelle que ce script calcul le nombre de tête de bœufs sur une ligne du plateau.

Ensuite, il exécute le script awk **placementCarte.awk** sur le fichier plateau avec la carte et l'emplacement en paramètre. On rappelle que ce script place une carte sur le plateau à l'emplacement souhaité.

Une fois la carte posée, on ajoute au score du joueur le nombre de têtes de bœufs qu'il a récupéré (s'il a posé la sixième carte sur une ligne, si sa carte n'a pas pu être posée et qu'il a dû choisir un emplacement). On envoie aux clients la carte qui a été posée et le nombre de têtes de bœufs que le joueur a récupéré s'il en a récupéré.

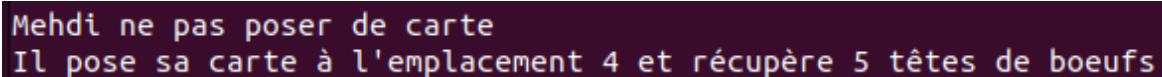


Figure 17 - Message lorsqu'un joueur doit choisir un emplacement

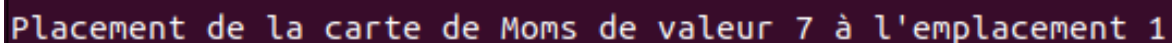


Figure 16 - Placement d'une carte sur un plateau

```
Placement de la carte de Robot2 de valeur 100 à l'emplacement 1  
Sixième carte posé, Robot2 prend 13 têtes de boeufs
```

Figure 18 - Placement d'une carte au sixième emplacement

On insère ensuite dans le log le message.

Le serveur envoie aux clients le plateau du jeu à chaque fois qu'une carte est posée, puis le client affiche à l'utilisateur le plateau reçu.

Ensuite, on vérifie si le joueur a atteint le nombre de têtes de bœufs. Si c'est le cas, on met fin à la partie. Sinon, on affiche la vie de tous les joueurs si un joueur a récupéré des têtes de bœufs.

```
Mehdi a 5 têtes de boeufs  
Moms a 0 têtes de boeufs  
Robot1 a 0 têtes de boeufs  
Robot2 a 13 têtes de boeufs
```

Figure 19 - Affichage des têtes de bœufs des joueurs

On insère ensuite dans le log la vie de chaque joueur.

Le serveur supprime du fichier ordreJoueur la carte qu'il vient de joueur (première ligne) puis effectue la même procédure jusqu'à placer toutes les cartes. Une fois toutes les cartes posées, le tour suivant commence.

Une fois 10 tours joués, on commence une nouvelle manche.

À la fin de la partie, le serveur cherche dans le tableau vie le joueur ayant le moins de têtes de bœufs (gagnant) et le joueur ayant le plus de têtes de bœufs (perdant) puis on envoie aux clients (et on insère dans le log) le message de fin de partie (nombre de manches fixé jouée ou si un joueur a obtenu le nombre de têtes de bœufs limite).

Enfin, on envoie aux clients et on insère dans le log le gagnant avec le nombre de têtes de bœufs obtenues.

```
Fin de la partie, Moms a eu 23 têtes de boeufs  
Le gagnant est Mehdi avec 11 tête de boeufs
```

Figure 20 - Message de fin de partie

À la fin de la partie, le serveur exécute le script shell **suppression.sh**. Ce script prend en paramètre le nombre de joueurs robots et de joueurs humains. Ce script supprime le fichier plateau, les fichiers contenant les cartes des joueurs, le fichier ordreJoueur et convertit le fichier log en fichier pdf à l'aide de la commande : **vim log -c "hardcopy > log.ps | q"; ps2pdf log.ps**

Pour chaque joueur, on appelle le script shell **ajoutStat.sh** qui prend en paramètre le pseudo du joueur, s'il a gagné ou perdu (1 ou 0) puis le nombre de têtes de bœufs qu'il a récupéré pendant la partie.

Ce script vérifie si le pseudo du joueur est dans le fichier stats. Si le joueur n'existe pas dans le fichier, on ajoute le pseudo ainsi que ses statistiques à la fin du fichier. Sinon, le script appelle le script awk **majStats.awk** qui prend en paramètre un pseudo, si le joueur a gagné ou perdu (1 ou 0) ainsi que le nombre de têtes de bœufs qu'il a récupéré pendant la partie. Ce script cherche la ligne qui stocke les statistiques du joueur puis les met à jour avec les données mises en paramètres.

Utilisateurs	Parties jouées	Parties gagnées	Parties perdus	Points totales	Points moyens/partie
Moms	2	0	2	37	18,5
Mehdi	2	1	1	28	14
Robot1	2	0	2	18	9
Robot2	2	1	1	16	8

Figure 21 - Fichier statistiques

## Stratégie du joueur robot

Tout d'abord, on cherche dans les cartes du robot les cartes qui peuvent être posées. Si aucune carte ne peut être posée, on récupère la plus petite carte car ce sont les cartes qui sont les plus difficiles à poser. Sinon, on filtre les cartes qui se posent au sixième emplacement d'une rangée. Si toutes les cartes du robot se posent au sixième emplacement, on récupère la carte qui nous fera prendre le moins de têtes de bœufs. Sinon, on récupère la carte la plus sûre à poser. Pour ceci, on regarde l'écart entre la carte qui se pose à un emplacement avec toutes les dernières cartes de chaque rangé du plateau.

Par exemple, si le robot a les cartes 48, 96, 27, 85 et que le plateau est comme ceci :

```
25 32 45
12 15 19 20
48 56 62
98 100 101
```

Le robot choisit la carte 96 qui sera placée à l'emplacement 3 car l'écart entre la carte 96 et 101 est de seulement 5 alors que la carte 48 a un écart de 14 avec la carte 62, la carte 85 a un écart de 16 avec la carte 101 et la carte 27 a un écart de 18 avec la carte 45. Nous avons opté pour cette stratégie car si autre joueur pose une carte sur l'emplacement souhaité avant le robot, le robot aura moins de chance de récupérer des têtes de bœufs. Par exemple, si un joueur pose la carte 84, la carte 96 peut toujours être posée. Cependant, si le robot choisit la carte 27 et qu'un joueur pose la carte 35, le robot sera bloqué et devra récupérer les têtes de bœufs d'une rangée. Si toutes les cartes du robot sont supérieures à toutes les dernières cartes du plateau, le robot choisit la carte la plus grande pour avoir plus de chance de pouvoir la poser.

## Comparaisons statistiques :

Pour comparer notre robot intelligent et un robot qui choisit une carte aléatoirement, nous avons lancé 1004 parties avec 2 robots intelligents (robot1 et robot2) et 2 robots qui choisissent une carte aléatoirement (robot3 et robot4). Nous avons décidé de faire jouer que 4 robots car lorsqu'il y a un trop grand nombre de joueurs, il y a une part d'aléatoire trop élevée.

Utilisateurs	Parties jouées	Parties gagnées	Parties perdus	Points totales	Points moyens/partie
Robot1	1004	340	664	42704	42,5339
Robot2	1004	384	620	41909	41,742
Robot3	1004	135	869	53984	53,7689
Robot4	1004	145	859	52910	52,6992

Figure 22 - Statistiques entre les robots intelligents et les robots aléatoires

Sur 1004 parties, 721 parties ont été remportées par un robot intelligent avec une moyenne de 42,13 points par partie tandis que les robots qui choisissent une carte aléatoirement ont eu en moyenne 53,23 points par partie. Notre robot intelligent est donc plus efficace avec un taux de partie gagné de 71,8 %.

## Conclusion

Bien que notre projet avec toutes les implémentations supplémentaires fonctionne, il y a plusieurs points qu'on aurait pu améliorer/modifier. En effet, il y a d'abord l'utilisation des sockets lorsque le jeu est joué en local. En effet, notre but était de pouvoir jouer en réseau et nous avons donc directement implémenté les interactions inter processus à l'aide des sockets. Cependant, on aurait pu faire cette partie-là en utilisant des tubes nommés (mkfifo). Il y a ensuite la structure des joueurs dans le programme du serveur. En effet, nous avons stocké le score et les pseudos dans 2 tableaux différents. Utiliser un tableau de structure contenant une variable pseudo et score aurait été plus propre. Toujours côté serveur, lorsque le programme est fermé avec un ctrl+c, les descripteurs clients ne sont pas fermés. Nous n'avons pas réussi à régler ce problème, car la taille du tableau de la structure poll (contenant les descripteurs des clients) est initialisé dans la main à partir du nombre de joueurs et de robots ce qui est impossible de déclarer en variable globale. Côté client, le stockage des cartes auraient pu être plus optimisés. En effet, utiliser une liste chaînée (ou doublement chaînée) aurait permis d'éviter d'avoir un tableau contenant des valeurs inutiles (les cartes jouées sont mises à 0). Lorsqu'un client se déconnecte, la partie prend fin (et nous ne savons pas pourquoi) et le serveur ne prévient pas les autres joueurs. Cependant, il aurait été plus efficace de pouvoir remplacer ce joueur par un joueur robot ou bien d'envoyer un message aux clients prévenant qu'un client s'est déconnecté puis désigner un vainqueur qui ne soit pas le joueur qui a quitté. Lorsqu'un joueur perd, nous n'avons pas géré le cas où il y a plusieurs gagnants. Si à la fin de la partie plusieurs joueurs ont le même nombre de têtes minimum, un seul gagnant est désigné. Il aurait fallu qu'on les fasse tous gagner ou bien laisser la partie continuée entre les vainqueurs. Le joueur qui aurait pris des têtes de bœufs en derniers aurait gagné. Concernant notre robot intelligent, nous n'avons testé qu'une seule stratégie. On aurait pu avoir un robot plus performant en testant d'autres stratégies. L'interface de jeu n'est pas très agréable et on aurait pu l'améliorer en affichant des messages type ASCII art. De plus, nous avons tout codé dans le main. En effet, il aurait fallu créer plus de fonctions et répartir notre code dans plusieurs fichiers et utiliser un makefile.