Rapport final du projet de Systèmes et réseaux « The Mind » expliquant l'architecture du projet, les fonctionnalités implémentées, les choix et les instructions de compilation.

Université de Bourgogne
UFR Sciences et techniques
Licence 3 – Informatique
Année universitaire 2021 – 2022

Rapport

Projet Systèmes et réseaux

Eddy DRUET – Clément GILI – Groupe TP 02

TABLE DES MATIERES

I. INTRODUCTION

Pour débuter, nous allons faire une introduction et rappeler le sujet, le contexte et les attendus du projet. Nous rappellerons brièvement ce que nous avons proposé lors du rapport préliminaire.

A. PRESENTATION DU SUJET

Ce projet de « Systèmes et Réseaux » nous demandait de reproduire une version simplifiée du jeu de cartes « The Mind ». Cependant, nous avons décidé de dévier légèrement du sujet, afin de rendre le jeu plus agréable : ce dernier, se joue en plusieurs manches, à chaque manche, chaque joueur reçoit aléatoirement autant de cartes que le numéro de la manche en cours, ces cartes sont numérotées d'un chiffre de 1 à 100. Le jeu étant coopératif, pour gagner, il faut que chaque joueur joue leur carte dans l'ordre croissant sans qu'aucun puisque communiquer entre eux ni oralement, ni physiquement. Au départ, trois vies sont données. Lorsqu'une manche est perdue, car une carte a été jouée dans le mauvais ordre, dans ce cas, la manche reprend du début, et une vie est perdue. Lorsque les trois vies sont consommées, la partie est terminée. Au contraire, lorsqu'une manche est gagnée, on passe à la manche supérieure. L'objectif ultime du jeu est de réaliser le plus grand de nombre de manches.

Le sujet nous demandait donc de porter ce jeu sur ordinateur, que l'on peut jouer à partir de terminaux en multijoueur avec ou non des robots.

B. ORGANISATION ET FONCTIONNEMENT DU SYSTEME

Le sujet nous imposait une organisation du système impliquant trois processus distincts, que nous avons implémentés comme ceci :

- « Gestionnaire de jeu » : ce processus correspond à notre serveur, processus central entre les joueurs et les robots, il gère les communications ainsi que la logique du jeu (accueil des joueurs, manches, distributions des cartes, etc.) ;
- « Joueur humain » : ce processus correspond à un terminal indépendant ouvert par un humain lui permettant d'afficher le jeu et de jouer ses cartes ;
- « Joueur robot » : ce processus est un fils du processus serveur puisqu'il est entièrement géré par ce dernier. Il tente de jouer sa carte la plus petite cartes lorsqu'aucun joueur humain ne se décide à jouer au bout d'un certain temps.

Nous avons donc une architecture client-serveur.

En bonus, parmi les deux options proposées par le sujet, nous avons décidé d'implémenter la génération de statistiques des parties dans un fichier PDF en fin de jeu.

C. STRUCTURE PRELIMINAIRE DU PROJET ENVISAGEE

Lors du rapport préliminaire, nous avions envisagé de faire communiquer les trois processus précédemment cités via une connexion socket sur le protocole TCP. Cela permettant de jouer en multijoueur sur Internet avec des machines distantes sur le réseau local ou non.

Nous avions également décidé de n'utiliser que le langage C et aucunement Shell ni Awk puisque nous pensions que ces deux derniers n'avaient pas lieu dans ce projet et dans ce type d'application. Finalement, Shell et Awk nous ont été utiles pour réaliser la génération du PDF des statistiques du jeu.

Enfin, nous avions établi trois étapes et phases de jeu : la phase de préparation où les joueurs sont dans une salle d'attente et peuvent se connecter au serveur de jeu, la phase de jeu où les joueurs et les robots jouent, et enfin la phase de fin de partie, où les joueurs peuvent recommencer une partie ou quitter le jeu.

Ceci était notre vision concernant l'architecture du projet au départ, que nous avons globalement réalisé finalement, en plus de l'ajout des langages Shell et Awk et le format d'un message différent que prévu initialement (nous le verrons plus tard). Vous pouvez retrouver tous les détails dans le rapport préliminaire.

D. POINTS SUR LES PROBLEMES POSES A PRIORI

Au départ, avant de débuter le projet, les difficultés nous semblaient être au niveau de la mise en place d'une connexion socket en C, puisque nous pensions que c'était bien plus compliqué qu'une connexion socket que nous avions fait dans un projet en Java.

Nous pensions, que l'envoi et la réception de messages allait être assez difficile. Et, en effet, nous avons eu certains problèmes lors de la transmission de messages, que nous verrons dans la partie sur les problèmes rencontrés et les solutions implémentées.

II. ENVIRONNEMENT DE DEVELOPPEMENT

Nous avons développé ce projet avec l'EDI[1] « Visual Studio 2022 » connecté à la « Windows Subsystem for Linux » (WSL), qui est une machine virtuelle exécutant une distribution Debian permettant d'exécuter et de programmer des programmes Linux sur Windows de façon « transparente ». Nous avons fait ceci pour des raisons de confort et de praticité ne possédant pas de machine sous Linux.

Nous avons donc été contraints d'utiliser « CMake » qui est un système de building permettant de vérifier les prérequis nécessaires à la compilation, de déterminer les dépendances entre les composants d'un projet, et de planifier la compilation ordonnée du projet sous la bonne plateforme. Nous avons dû apprendre à configurer ce dernier pour compiler notre projet.

[1] Environnement de développement intégré.

III. STRUCTURE LOGIQUE DU PROJET

Nous allons, dans cette partie, expliciter l'architecture du projet permettant la communication, l'organisation et les interactions entre les processus en jeu. Nous irons du plus global puis nous irons de plus en plus en détail.

A. ARCHITECTURE GLOBALE DU PROJET

Pour rappel, nous avons une architecture basée autour de trois processus distincts, ainsi, le projet a été découpé en trois sous-projets « ClientProject », « ServerProject », et « BotProject ». Ces trois sous-projets sont trois exécutables différents et ils partagent une base de code commune située dans le package « Shared ». Ce découpage permet de faciliter la maintenabilité du code.

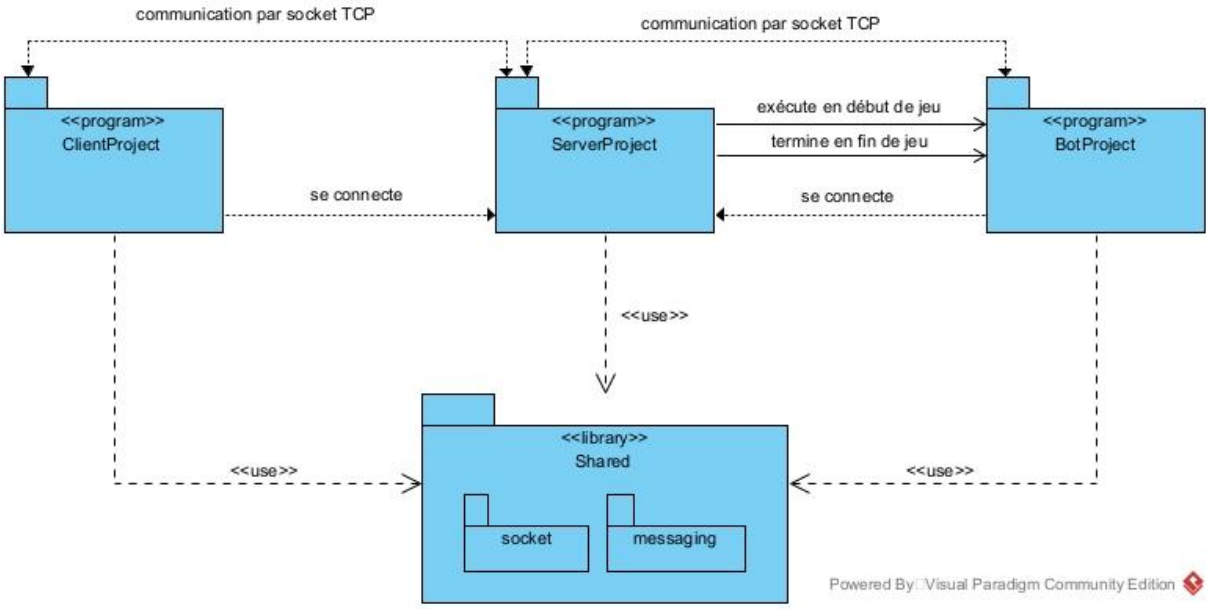

Diagramme 1 : Architecture globale du projet et des interactions principales entre processus

Le package « Shared » possède le code commun aux trois sous-projets, notamment le code régissant le système de socket, le système de message et des fonctions utilitaires notamment pour le débogage ou l'affichage à l'écran.

Comme nous l'avons vu, le client, et le/les robot(s) communiquent avec le serveur avec un socket TCP. Par ailleurs, les processus robot sont exécutés et terminés automatiquement par le serveur au moment opportun.

B. ORDRE D'EXECUTION ET DE TERMINAISON DES PROCESSUS

Ci-dessous, le diagramme de séquence montrant l'ordre dans lequel les processus en jeu sont exécutés et terminés au fil des phases du jeu.

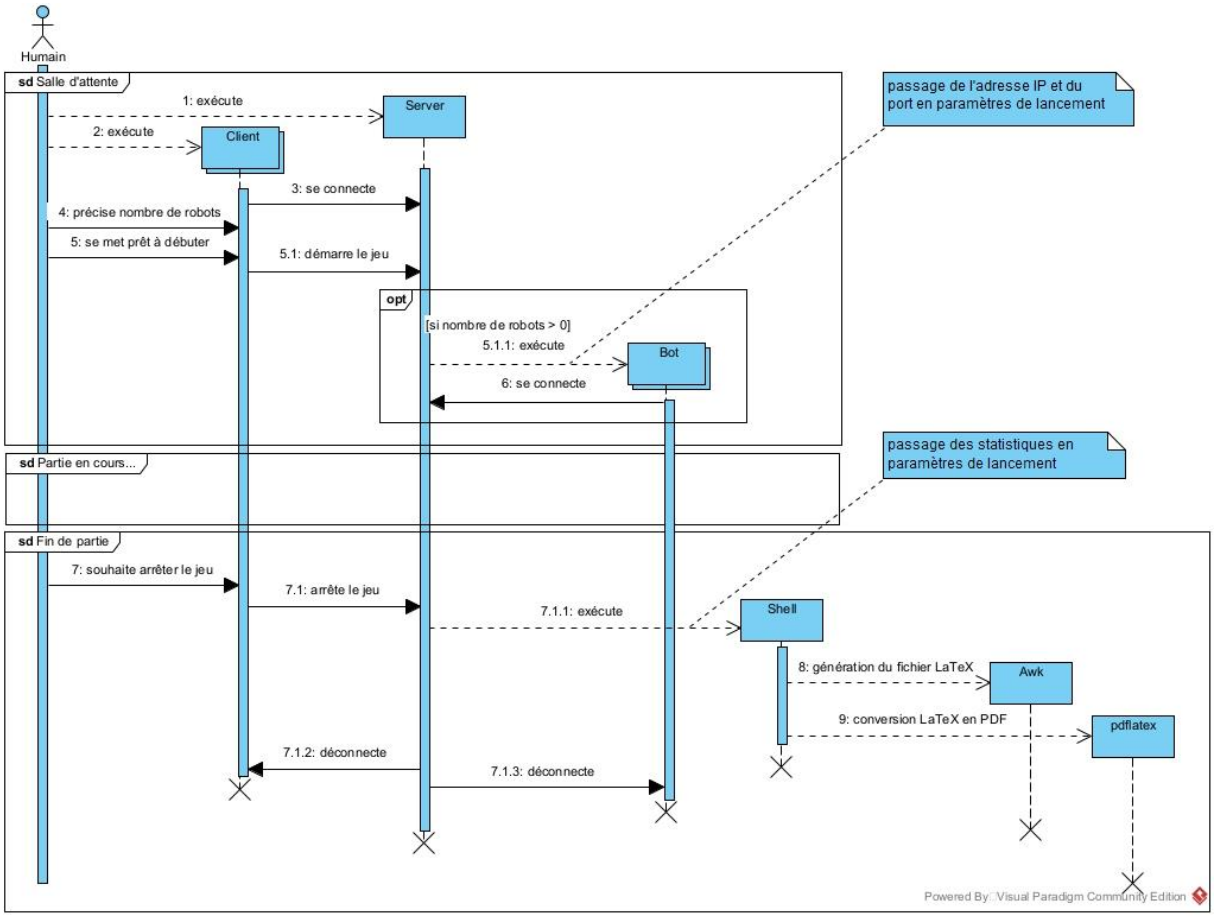

Diagramme 2 : Ordre d'exécution et de terminaison des processus en jeu

D'abord, le serveur est exécuté manuellement, puis des joueurs humains exécutent leur client et se connectent au serveur avec l'IP et le port de ce dernier. Les joueurs précisent le nombre de robots (facultatif) et lorsqu'ils se mettent prêt à débuter la partie, le serveur exécute les processus robots en leur fournissant l'IP et le port du serveur en paramètre de lancement. La partie débute et les joueurs jouent…

Lorsque la fin de partie arrive, et qu'un joueur arrête le jeu, le serveur exécute le script Shell permettant de générer le fichier PDF des statistiques du jeu en : générant le fichier LaTeX à partir d'un template grâce à Awk, puis la conversion en PDF avec l'outil « pdflatex ». Une fois, la génération terminée, les clients des joueurs et les clients robots sont déconnectés, et se terminent.

C. SYSTEME DE MESSAGES

Dans cette partie, nous allons expliquer le système de messages implémenté dans chacun des processus. L'implémentation principale de ce système se trouve dans le code commun du package « Shared » et est donc identique aux trois sous-projets.

1. STRUCTURE D'UN MESSAGE

Contrairement, à ce qu'on avait pensé dans le rapport préliminaire, nous n'avons pas utilisé des chaînes de caractère pour transmettre des messages entre les processus, puisque la manipulation des chaînes en C est fastidieuse et peu pratique. Nous avons plutôt décidé de transmettre directement des flux d'octets.

Le contenu d'un message possède donc une structure particulière dans cet ordre :

1) Une structure C constituant l'en-tête du message contenant :
   - La longueur en octets du corps du message contenant les données utiles ;
   - Le type du message permettant de l'identifier par rapport à un autre.
2) Une structure C spécifique au type de message contenant les données utiles.

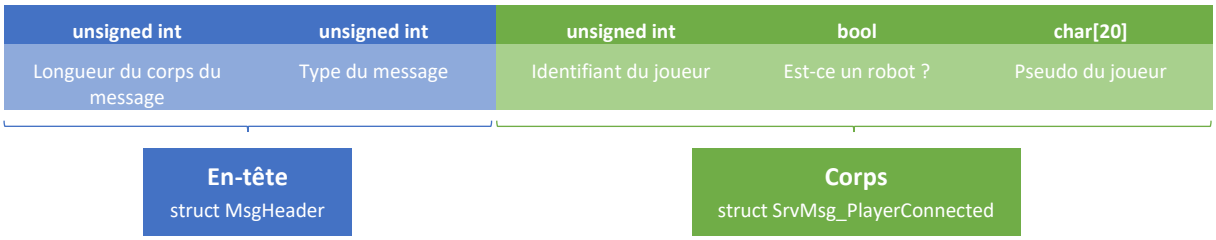

Schéma 1 : Structure d'un message informant la connexion d'un joueur à tous les autres clients

C'est donc la sérialisation de la concaténation de ces deux structures C qui est transmis sur le socket TCP et qui forme un message. À noter, que certains types de message ne possèdent pas de données utiles, et donc ne possède pas de 2ème structure C.

Ces structures sont définies dans le fichier « Shared/messaging/structs.h ».

2. TYPES DE MESSAGE

Les types de message sont divisés en deux catégories : les messages « client » envoyés par les clients, et les messages « serveur » envoyés par le serveur. Ces types permettent de différencier un message d'un autre pour y effectuer l'action adéquate. Vous aurez la liste des types de message quelque peu après.

Ces types sont définis dans une énumération dans le fichier « Shared/messaging/enums.h ».

3. GESTIONNAIRES DE MESSAGE

Pour chaque type de message, est défini un gestionnaire qui est une fonction permettant de gérer le message reçu en effectuant une action spécifique. Le gestionnaire est automatiquement appelé lors de la réception du type de message qui lui est spécifique.

Concernant les gestionnaires des messages « client » reçus par le serveur, ils reçoivent en arguments : l'identifiant du client émetteur, ainsi que la structure C contenant les données utiles du message.

Concernant les gestionnaires des messages « serveur » reçus par les clients, ils reçoivent qu'un seul argument : la structure C contenant les données utiles du message.

Ces gestionnaires sont définis dans le fichier « Shared/messaging/cli_handlers.h » pour les gestionnaires implémentés par le serveur, et « Shared/messaging/srv_handlers.h » pour les gestionnaires implémentés par les clients. L'implémentation de ces gestionnaires se trouve dans le fichier « handlers.c » de chaque sous-projet.

4. LISTE DES TYPES DE MESSAGE

Pour vous donner une idée de la communication avec le serveur et les clients, ci-dessous la liste des types de message que nous avons créé ainsi que leurs données utiles.

Type du message	Données utiles	Utilité
CLI_MSG_SET_NAME	- Pseudo du joueur	Définir le pseudo du joueur.
CLI_MSG_SET_READY		Mettre prêt le joueur pour débuter la partie.
CLI_MSG_SET_NUM_BOT	- Nombre de robots	Définir le nombre de robots dans la partie.
CLI_MSG_PLAY_CARD	- Index de la carte	Jouer une carte.
CLI_MSG_REPLAY_GAME		Relancer une partie.
CLI_MSG_BOT_CONNECT		Confirmer la connexion d'un robot.
CLI_MSG_STOP_GAME		Arrêter le jeu.

Tableau 1 : Liste des types de message « client »

Type du message	Mode d'émission	Données utiles	Utilité
SRV_MSG_INFO_LOBBY	Broadcast	- Nombre de robots - Nombre de joueurs prêts	Le lobby a changé d'état (ex. nombre de joueurs prêts, etc.).
SRV_MSG_PLAYER_CONNECTED	Broadcast	- Identifiant joueur - Est-ce un robot ? - Pseudo du joueur	Un joueur humain s'est connecté au serveur.
SRV_MSG_CARD_PLAYED	Broadcast	- Identifiant joueur - Numéro de la carte	Une carte a été jouée.
SRV_MSG_NEXT_ROUND	Un par un, à chaque joueur	- Numéro de la nouvelle manche - Nombre de vies restantes - La manche précédente a été gagnée ? - Ensemble de cartes du joueur	La prochaine manche débute, envoi des cartes des joueurs.
SRV_MSG_GAME_END	Broadcast		La partie est terminée.
SRV_MSG_PLAYER_INFO	Au joueur qui vient se connecter	- Identifiant joueur	Envoi de l'identifiant du client au joueur qui vient de se connecter.
SRV_MSG_DISCONNECT_ALL	Broadcast		Déconnexion de tous les clients.

Tableau 2 : Liste des types de message « serveur »

D. SYSTEME DE COMMUNICATION PAR SOCKET

Nous avons vu dans la précédente partie, le système de message. Maintenant, nous allons voir le fonctionnement du système de communication par socket sur le protocole TCP permettant de transmettre et de recevoir ces messages.

1. SYSTEME ORIENTE EVENEMENTIEL

Nous avons opté pour un système de communication orienté événementiel plutôt que séquentiel. En effet, dans chacun des processus, nous avons une boucle s'occupant de réceptionner les messages reçus. À la réception d'un message, le gestionnaire correspondant est appelé automatiquement en fonction du type du message. Ce gestionnaire peut répondre en envoyant un ou plusieurs messages à son tour. Nous avons préféré pour cette approche, puisqu'elle est plus intéressante à réaliser et plus flexible.

2. IMPLEMENTATION DANS LE PROCESSUS « CLIENT »

Un socket TCP est créé et se connecte au serveur via l'adresse IP et le port saisis par l'utilisateur. Ensuite, l'écoute des messages reçus et donc l'appel du gestionnaire correspondant est déporté sur un thread dédié à cette tâche, tandis que le thread principal s'occupe de la gestion des entrées utilisateur dans le terminal (nous verrons ceci plus tard). Nous avons donc utilisé deux threads, car à la fois les entrées utilisateurs et l'écoute des messages est bloquant, ce qui empêchait d'effectuer les deux simultanément.

3. IMPLEMENTATION DANS LE PROCESSUS « ROBOT »

Un socket TCP est créé et se connecte au serveur via l'adresse IP et le port passés en paramètres de lancement par le serveur. Contrairement au processus « client », il n'y a aucun thread, et l'écoute des messages et de leur gestion se fait sur le thread principal puisqu'il n'y a aucune entrée utilisateur à gérer dans ce processus.

4. IMPLEMENTATION DANS LE PROCESSUS « SERVEUR »

Un socket assigné au port « 25565 » est créé, c'est celui-ci où les clients vont se connecter dessus. Le thread principal va écouter les demandes de connexions, et lorsqu'une connexion est acceptée, un thread est créé et est désigné pour gérer les communications avec le client connecté. Nous avons donc un thread par client connecté.

Le serveur peut envoyer un message à un client spécifique identifié par un entier unique, ou bien, il peut diffuser un message à tous les clients connectés.

5. RESSOURCES PARTAGEES ET GESTION DE LA CONCURRENCE

Puisque nous utilisons des threads, cela implique la gestion de la concurrence au niveau des éventuelles ressources partagées.

Dans le processus « robot », il n'y a aucun thread donc la question de la gestion de la concurrence ne se pose pas. Pareillement, dans le processus « client », nous n'avons pas identifié de ressources partagées bien que nous ayons un thread gérant la réception des messages et le thread principal gérant les entrées utilisateur. Ces deux threads n'utilisent aucune zone mémoire commune, puisque le thread principal n'envoie que des messages au serveur (ex. jouer la carte 52).

Cependant, dans le processus « serveur », il existe de nombreuses ressources partagées, notamment celles qui stockent l'état des joueurs, des manches et de la partie. En effet, chaque thread qui gère la communication avec leur client peut effectuer une action et modifier l'état du jeu suite à la réception d'un message. Si deux threads s'actionnent en même temps, car un message a été simultanément reçu de deux clients différents, par exemple si deux joueurs jouent une carte au même instant, l'état des ressources partagées est incertain.

La section critique se trouve donc au moment où le gestionnaire du type de message est appelé. Nous avons donc verrouillé un mutex[2] juste avant d'entrer en section critique, soit juste avant l'appel du gestionnaire de message, et déverrouillé le mutex à la fin de l'appel du gestionnaire de message. Ainsi, tant qu'un thread est en train d'exécuter un gestionnaire de message, les autres threads attendent la fin de son exécution pour passer à leur tour.

[2] Verrou empêchant l'accès à une zone de ressources partagées dans un système concurrentiel.

E. EXEMPLE D'UNE COMMUNICATION SOCKET

Maintenant, que nous avons vu le système de message et de communication par socket, nous allons voir un exemple illustré par un diagramme de séquence. La situation décrite est : nous sommes à la manche n°1, et le client n°1 joue une carte. Nous allons voir les envois de messages dans cette situation.

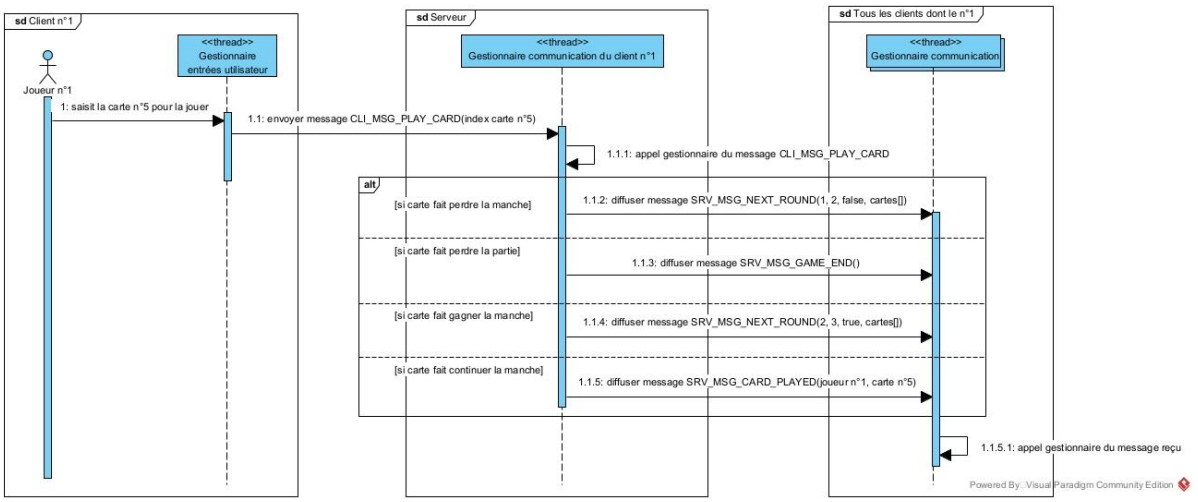

Diagramme 3 : Exemple d'une communication par socket

Le thread « gestionnaire des entrées utilisateurs » du client n°1 réceptionne le numéro de la carte que le joueur n°1 veut jouer. Cela déclenche l'envoi du message « CLI_MSG_PLAY_CARD » au serveur.

Le thread « gestionnaire communication du client n°1 » reçoit le message « CLI_MSG_PLAY_CARD » et appelle le gestionnaire spécifique. Ce gestionnaire effectue des vérifications (si la carte fait perdre, gagner, continuer la manche ; ou perdre la partie ?), et diffuse le message correspondant à tous les clients.

Le thread « gestionnaire communication » de chaque client reçoit le message du serveur et appelle le gestionnaire du message reçu.

F. PROBLEMES RENCONTRES ET LES SOLUTIONS IMPLEMENTEES

Nous avons rencontré plusieurs problèmes qui ne se manifestaient pas systématiquement, mais qui empêchaient le jeu de fonctionner correctement. Nous avons dû utiliser le débuggeur ainsi que des recherches sur Internet pour les résoudre.

1. TRANSMISSION D'UN MESSAGE SUR LE SOCKET

Le premier problème que nous avons rencontré est lors de la transmission d'un message sur le socket. En effet, en voulant lire le buffer du socket pour obtenir le contenu d'un message, parfois, nous nous retrouvions avec le contenu de deux messages au lieu d'un. Également, parfois, nous recevions le contenu de notre message en plusieurs fois.

Après recherche sur Internet, ceci est normal, car c'est le fonctionnement normal du protocole TCP qui ne possède pas de notion de « message » comme nous l'entendons. Il envoie simplement des données brutes sans signification, et il n'est pas garanti que les données envoyées soit envoyés en une seule fois et en même temps.

La solution a été donc de préfixer le contenu de nos messages par un en-tête avec la taille en octets du contenu du message (cf. partie sur la structure d'un message). Ainsi, nous avons créé notre propre fonction de lecture,

qui lit le buffer du socket tant que le message n'a pas été reçu en entier. Pareil, pour l'écriture d'un message dans le buffer du socket : tant que le message n'a pas été écrit entièrement, il continue de l'écrire.

2. PROBLEME DE CONCURRENCE

Lorsque le serveur recevait plusieurs messages au même moment, il y avait des bugs dans le jeu. Après réflexion, nous avons déduit qu'il y avait un problème de concurrence sur les ressources partagées du serveur. Nous avons résolu ce problème avec un mutex (cf. partie sur les ressources partagées et gestion de la concurrence).

IV. APPROCHE TECHNIQUE

Dans la partie précédente, nous avons vu la structure logique du projet, autrement dit, l'architecture globale et les principes utilisés pour la communication et les interactions entre les processus. Nous allons maintenant, voir plus en détail les approches techniques utilisés pour résoudre certains aspects importants du projet.

A. APPEL AUTOMATIQUE DES GESTIONNAIRES DE MESSAGE

Dès la réception d'un message, le gestionnaire est automatiquement appelé selon le type du message. Nous n'avons pas utilisé de switch, ni de long « if-else if-else if-else… ». Au contraire, nous avons fait un système semi-automatique, un peu plus flexible et intéressant à concevoir.

Pour cela, à chaque type de message, nous définissons une fonction qui devra être implémentée par un des processus qui souhaite traiter le message. Le pointeur de cette fonction est ajouté au tableau des gestionnaires. Ce tableau associe un index (numéro du type du message) au pointeur de la fonction qui gère ce type de message.

Ainsi, dans le thread qui réceptionne les messages, il lui suffit juste d'appeler la fonction qui gère le message reçu en accédant à son pointeur via l'index dans le tableau des gestionnaires, par exemple : « srvHandlers[4](&data) », si on a reçu un message « serveur » de type n°4. Notez, que les données du message sont passées au gestionnaire.

B. GESTION DES ENTREES UTILISATEUR

Dans le processus « client », l'utilisateur doit pouvoir saisir dans le terminal, par exemple, pour modifier le nombre de robots, jouer une carte, recommencer une partie, etc.

Le problème, c'est qu'en C, lorsque l'on demande de saisir quelque chose dans le terminal (ex. avec un « scanf » ou un « fgets »), il n'est plus possible d'annuler la demande de saisie dans le terminal, c'est-à-dire, d'arrêter le blocage induit par la saisie.

Or, il nous fallait justement pouvoir annuler la demande de saisie. Pour illustrer, voici un exemple typique que nous avons eu : lorsque nous sommes dans une manche, l'utilisateur peut saisir la carte qu'il veut jouer, mais si la partie est perdue avant que le joueur ne joue sa carte, et que l'écran de fin s'affiche en demandant si l'on veut recommencer une partie ou quitter le jeu, et qu'à ce moment l'utilisateur saisi quelque chose, cela lui fera jouer une carte et non recommencer la partie ou quitter le jeu selon sa saisie. En effet, nous n'avons pas pu annuler la demande de saisie de la carte à jouer, pour demander à place de saisir s'il veut recommencer ou quitter le jeu.

Pour résoudre ce problème, nous avons implémenté un système de « callback ». Pour résumer, maintenant, le terminal est toujours en mode saisie. Cependant, nous pouvons définir une fonction de rappel qui sera appelée à chaque saisie de l'utilisateur. Ainsi, si nous voulons ne rien faire lorsque l'utilisateur saisi quelque chose, nous ne mettons aucune fonction de rappel. Au contraire, par exemple, lorsque le client reçoit le message disant qu'une manche débute, la fonction de rappel est modifiée par celle qui permet de jouer une carte quand l'utilisateur saisi un nombre. Lorsque le client reçoit le message de fin de partie, la fonction de rappel est modifiée par celle qui permet de recommencer une partie ou de quitter le jeu en fonction de la saisie de l'utilisateur.

C. GESTION DES ERREURS

Afin de rendre plus facile de débogage, nous avons géré toutes les erreurs lors des appels de primitives systèmes, notamment en affichant le message de la fonction « perror[3] » dans la sortie standard d'erreur et en envoyant le

[3] Afficher le message d'erreur de la dernière primitive système appelée sur la sortie standard d'erreur.

signal d'extinction du processus, afin que le processus se ferme proprement en fermant les sockets ouverts et les threads en cours d'exécution (cf. partie sur la fermeture propre des processus).

D. FERMETURE PROPRE DES PROCESSUS

Les processus se doivent de fermer les sockets ouverts et les threads en cours d'exécution lorsqu'ils se terminent. Pour s'assurer de cela, nous avons créé une fonction qui s'occupe de cela, celle-ci est automatiquement appelée à la fermeture du processus grâce à la fonction « atexit[4] ».

Cependant, cela ne fonctionne que lorsque la fonction « exit[5] » est appelé dans le code, et non lorsque le processus est fermé depuis un signal extérieur comme le signal « TERM[6] » ou « INT[7] ». Nous avons donc intercepté ces signaux, et leur avons assigné une fonction de rappel, exécutant la fonction « exit ».

Ainsi, lorsque nous voulons terminer le processus à partir d'un thread, nous avons juste à envoyer le signal « TERM » au processus pour que l'arrêt se fasse dans le thread principal.

E. FONCTIONNEMENT DES PROCESSUS « ROBOT »

Nous allons voir le fonctionnement des processus « robot », de l'exécution, à la stratégie de jeu.

1. EXECUTION DES PROCESSUS

Comme dit au départ, les processus « robot » sont automatiquement exécutés par le serveur au commencement du jeu.

Pour cela, nous faisons autant de fork du serveur que de robots à exécuter. Dans chaque processus fils « robot » engendré, tous les descripteurs de fichier des sockets du serveur sont fermés, et l'image du processus fils est remplacée par l'image de l'exécutable du « robot » avec la fonction « execlp[8] ». En plus, l'adresse IP et le port du serveur sont passés en paramètre à la fonction « execlp ». Ainsi, chaque robot se connecte immédiatement au serveur grâce à l'adresse IP et le port extraits des arguments de lancement.

Ces paramètres de lancement sont ceux-ci :

```
Arguments available:
        --ip <ip-address>               Server IP address to connect to.
        --port <number>                 Server port to connect to.
        --help                          Print this list.

Note: server IP address and port are mandatory in order to run the program.
```

2. STRATEGIE DE JEU

Il est assez compliqué de réaliser un robot intelligent sur un jeu basé sur la devinette avec une grande part de hasard. C'est pourquoi, nous lui avons doté d'une stratégie simple basée sur le comportement des joueurs humain.

Le robot va jouer sa plus petite carte lorsqu'aucun autre joueur ou autre robot ne se décide à jouer au bout d'un certain temps. Ce temps est compris entre une dizaine et une vingtaine de secondes. Ainsi, le robot, va

[4] Permet de définir une fonction de rappel automatiquement appelée avant la fermeture du processus.
[5] Permet de terminer le processus.
[6] Signal engendrant la fin du processus.
[7] Signal déclenché par CTRL+C.
[8] Exécuter un programme en remplaçant l'image du processus appelant par l'image du processus appelé.

aléatoirement choisir un temps dans cet intervalle de seconde, et jouer dès que le temps est écoulé et que personne d'autres n'a jouer entre temps.

Par exemple, la manche n°1 débute. Le robot décide aléatoirement qu'il va jouer dans 12 secondes. Un joueur joue au bout de 5 secondes. Alors, le robot re-tire aléatoirement un nouveau temps, et décide qu'il va jouer dans 18 secondes. Aucun joueur ne se décide et ne joue, le robot joue donc sa carte la plus basse au bout de 18 secondes.

Pour planifier le moment où le robot va jouer, nous utilisons la fonction « alarm[9] » qui va permettre d'envoyer un signal « ALRM[10] » au processus « robot » au bout d'un temps. Ainsi, nous interceptons ce signal, et nous y appelons la fonction qui permet de jouer la carte la plus basse du robot dès la réception du signal. La planification est réinitialisée dès qu'un autre joueur joue une carte avant le robot.

Il peut y avoir plusieurs robots, dans ce cas, chaque robot décide aléatoirement le moment a lequel ils vont jouer.

[9] Permet de planifier l'émission du signal « ALRM » dans le temps.
[10] Signal déclenché par la fonction « alarm » lorsque la planification arrive à sa fin.

V. OPTION DE PROGRAMMATION

Nous avons décidé de prendre l'option demandant de générer un fichier PDF des statistiques des parties jouées en fin de jeu. Ce fichier PDF inclut les statistiques suivantes :

- Statistiques globales :
  - Nombre de joueurs ;
  - Temps de réaction minimum (en secondes) ;
  - Temps de réaction maximum (en secondes) ;
  - Temps de réaction moyen (en secondes) ;
  - Nombre moyen de manches gagnées par partie ;
  - Joueur qui a fait perdre le plus de manches ;
  - Graphique du nombre de manches gagnées par partie
- Tableau des statistiques des joueurs :
  - Nom du joueur ;
  - Temps de réaction minimum (en secondes) ;
  - Temps de réaction maximum (en secondes) ;
  - Temps de réaction moyen (en secondes) ;
  - Nombre d'échecs.

Ce fichier PDF est généré au moment où un des joueurs arrête le jeu.

A. LANGAGES ET OUTILS UTILISES

Pour réaliser cette fonctionnalité, nous avons utilisé le langage C, un script Shell, un fichier Awk ainsi qu'un fichier LaTeX servant de template. La conversion du fichier LaTeX en PDF a été réalisée avec l'utilitaire « pdflatex ». Juste après, nous verrons, le rôle de chacun de ces langages et outils, ainsi que le passage des informations d'un langage et outil à l'autre.

B. INTERACTIONS ET ECHANGES D'INFORMATIONS ENTRE CES LANGAGES ET OUTILS

Nous avons dû trouver un moyen de faire passer les informations entre tous ces outils, pour cela, nous nous sommes principalement aidé des paramètres de lancement de ces outils en cherchant dans leur documentation.

1. PROCESSUS « SERVEUR »

Le processus « serveur » en langage C, exécute le script Shell au moment d'arrêter le jeu. Pour cela, il effectue un fork, et exécute dans le processus fils, le script Shell avec la fonction « execlp ». Les statistiques sont passées en paramètres de lancement au script Shell. À ce stade, le processus « serveur » est en attente de la fin de l'exécution du script Shell (attente de la fin du processus fils).

2. TEMPLATE LATEX

Il constitue la base du document PDF, et il possède des emplacements à remplacer par des statistiques. Ces emplacements sont des commentaires LaTeX avec au début un nom de variable, et un ou plusieurs spécificateurs de format « printf », par exemple :

```
\section{Exemples d'emplacements de statistiques}
%NOM_VARIABLE Texte : %d
%PLAYER_COUNT Nombre de joueurs : %d
%REACTION_AVG Temps de réaction moy. : %.2f secondes
```

Les variables seront utiles pour le passage des statistiques du processus « serveur », jusqu'au script Shell, jusqu'au fichier Awk qui traitera le fichier template.

3. SCRIPT SHELL

Il est exécuté par le processus « serveur », auquel il a reçu les arguments contenants les statistiques devant remplacer les emplacements dans le fichier template. Ces arguments sont de la forme :

```
-v NOM_VARIABLE=VALEUR
-v PLAYER_COUNT=2
-v REACTION_AVG=6.24
```

Il y a la présence d'un « -v » devant, car c'est un paramètre de lancement disponible dans Awk permettant de passer une variable externe dans l'environnement d'exécution de Awk. Cela nous permettra d'utiliser ces variables dans notre fichier Awk.

D'abord, le script Shell, exécute le fichier Awk en lui passant les variables reçues permettant de générer le fichier LaTeX rempli à partir du template et des statistiques reçues. Ensuite, il exécute la conversion du fichier LaTeX rempli en PDF avec l'outil « pdflatex ».

Après, le fichier PDF généré est renommé par la date et l'heure du jour et est déplacé dans le répertoire « ServerProject/game_stats ».

Enfin, s'il n'y a eu aucune erreur, les fichiers temporaires sont supprimés. Sinon, le script s'arrête et les erreurs sont écrites dans un fichier log dans le répertoire « ServerProject/template ».

4. FICHIER AWK

Il analyse ligne par ligne le fichier template, et dès qu'il trouve en premier mot un commentaire avec un nom de variable qu'on lui a fourni en arguments, il supprime le premier mot, et effectue un « printf » sur toute la ligne, afin de remplacer le ou les spécificateurs de format par la valeur de la variable correspondante.

Par exemple, pour un fichier template :

```
\section{Exemple d'un fichier template rempli par Awk}
%PLAYER_COUNT Nombre de joueurs : %d
%REACTION_AVG Temps de réaction moy. : %.2f secondes
```

Et pour ce fichier Awk :

```
substr($1, 0, 1) != "%" { print($0) }
$1 == "%PLAYER_COUNT" { $1 = ""; printf($0, PLAYER_COUNT) }
$1 == "%REACTION_AVG" { $1 = ""; printf($0, REACTION_AVG) }
```

Avec les variables « PLAYER_COUNT=2 » et « REACTION_AVG=6.24 » reçues en arguments par le script Shell.

Cela nous donne le fichier LaTeX :

```
\section{Exemple d'un fichier template rempli par Awk}
Nombre de joueurs : 2
Temps de réaction moy. : 6,24 secondes
```

C. RECAPITULATIF DE PROCESSUS DE GENERATION DU PDF

Ci-dessous, le diagramme de séquence correspondant au processus de génération du PDF. Il met en évidence les interactions et le passage d'informations entre les différents outils en jeu.

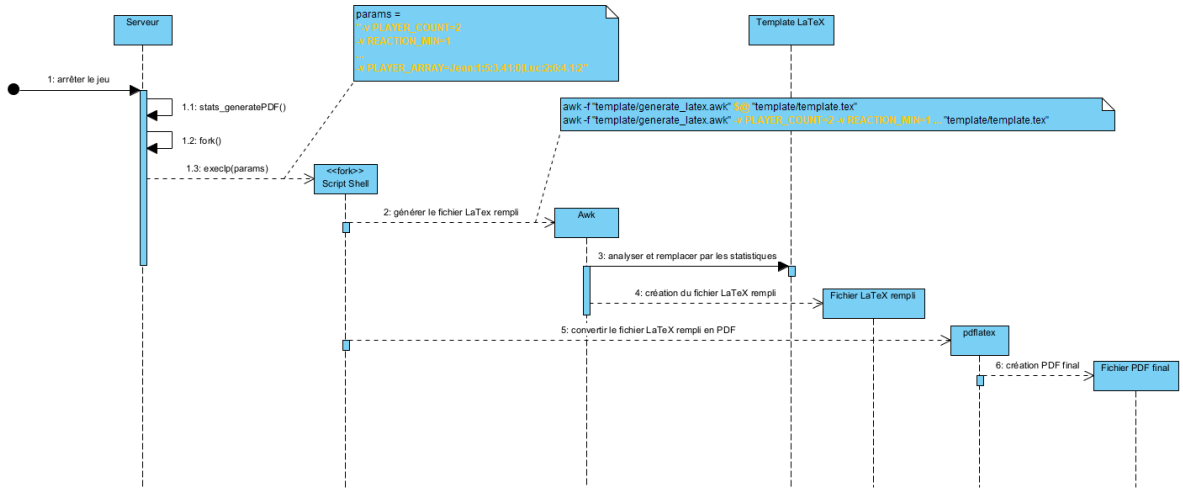

Diagramme 4 : Processus de génération du PDF

VI. CONCLUSION

Nous arrivons à la fin de ce rapport, et nous pensons que nous avons réalisé correctement toutes les fonctionnalités qui étaient demandé par le sujet. Nous avons finalement su utiliser tous les langages et outils tels que C, Shell, Awk et LaTeX, ainsi que toutes les notions principales étudiées dans le module comme : signaux, fork, threads, sections critiques et mutex, sockets, et paramètres de lancement. Par ailleurs, nous nous sommes amusés à faire un système de communication événementielle dans laquelle les messages ne sont pas de simples chaînes de caractère, mais des structures C sérialisées.

Concernant, les choses que l'on aurait pu faire autrement, c'est la gestion de la communication avec les clients au niveau du serveur. Au lieu d'utiliser un thread par client, nous aurions pu utiliser un multiplexeur d'entrées-sorties synchrones sur nos sockets avec la fonction « select » par exemple.

Nous pensons qu'il n'y a rien d'autres à ajouter pour ce projet, et que nous avons fait tout ce qui était nécessaire, si ce n'est, l'amélioration du jeu en lui-même, c'est-à-dire implémenter le vrai jeu en entier et pas une version simplifiée.

VII. INSTRUCTIONS DE COMPILATION ET D'EXECUTION

Dans cette partie, nous allons expliquer comment compiler le projet. Notez, que les fichiers sources se trouvent dans le répertoire « TheMind ».

A. DEPENDANCES NECESSAIRES

Ci-dessous, la liste des dépendances nécessaires pour compiler le projet, et l'exécuter.

- Dépendances nécessaires pour compiler :
  - sudo apt-get install make
  - sudo apt-get install cmake

- Dépendances nécessaires pour exécuter :
  - sudo apt-get install texlive-latex-base
  - sudo apt-get install texlive-latex-extra
  - sudo apt-get install texlive-lang-french

- Dépendances optionnelles :
  - sudo apt-get install doxygen

B. COMPILER LE PROJET

Pour compiler le projet, vous devez dans le répertoire « TheMind », taper les commandes :

- cmake .
- make

C. DOCUMENTATION DOXYGEN

Le projet a été entièrement documenté au format Doxygen. La documentation a déjà été générée au format HTML que vous pouvez ouvrir dans le répertoire du sous-projet correspondant : « TheMind/Docs/<Sous-projet>/html/index.html ».

Si vous souhaitez la générer vous-même, exécutez le script Shell « TheMind/Docs/run_doxygen.sh ».

D. EXECUTER LE PROJET

Voici les chemins vers les exécutables :

- Pour lancer le serveur : « TheMind/ServerProject/ServerProject »
- Pour lancer un client : « TheMind/ClientProject/ClientProject »

E. INFORMATIONS DE CONNEXION AU SERVEUR

Voici les informations de connexions pour se connecter au serveur dans l'écran de connexion d'un client :

- Adresse IP : « 127.0.0.1 »
- Port : « 25565 »

TABLE DES DIAGRAMMES

TABLE DES SCHEMAS

TABLE DES TABLEAUX