**Projet Hanabi : Compte Rendu**

# Introduction :

Dans le cadre de ce projet de programmation parallèle et concurrente, nous devons concevoir et implémenter un programme multi-processus et multi-thread dans le but de recréer le jeu de carte collaboratif hanabi. Il utilise aussi des moyens de communications via des connexion TCP ou encore des messages queue. Nous détaillerons dans ce rapport les étapes de conception du code, en expliquant la mise en œuvre des principes de programmation parallèle et concurrente, tels que les outils de communication et de synchronisation inter-processus.

Dans Hanabi, les joueurs incarnent des artificiers cherchant à composer cinq feux d'artifice, chacun représenté par une couleur spécifique : blanc, rouge, bleu, jaune et vert. L'objectif est de constituer ces feux d'artifice en formant des suites croissantes de cartes de la même couleur, allant de la carte 1 à la carte 5. Les cartes multicolores ont été mises de côté dans cette version du jeu.

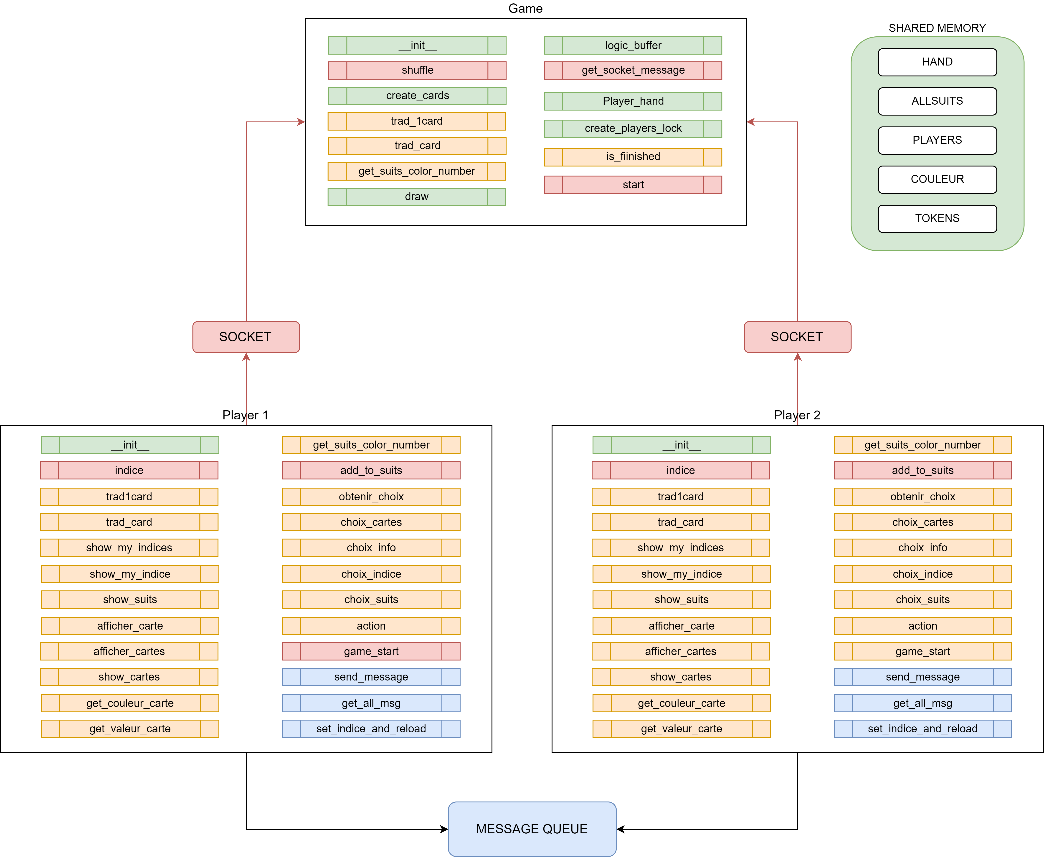
# Règles du Jeu :

Hanabi se déroule en tours, où chaque joueur peut effectuer différentes actions pour progresser vers la réalisation des feux d'artifice. Les joueurs peuvent piocher des cartes, en jouer une pour constituer un feu d'artifice, ou en défausser pour préparer une future action. L'aspect stratégique du jeu réside dans la gestion des cartes disponibles, la coopération entre les joueurs pour optimiser les actions et la prise de décisions pour maximiser les chances de succès.

Chaque feu d'artifice est composé de cartes de la même couleur, disposées dans l'ordre croissant. Les joueurs doivent travailler ensemble pour organiser les cartes de manière à ce que chaque feu d'artifice soit complet avant la fin de la partie. Les erreurs peuvent être coûteuses, car chaque carte jouée incorrectement ou chaque feu d'artifice inachevé diminue le score final. De plus, les joueurs ne disposent que d’un nombre de coup défini par les fuse tokens et les informations tokens. En effet si le nombre de fuse tokens passe à zéro, les joueurs perdent la partie et s’ils n’ont plus d’informations tokens, ils ne peuvent plus communiquer entre eux.

# Phase de Conception :

Avant de plonger dans le développement du code, nous avons pris le temps de conceptualiser le jeu dans son ensemble et de décomposer ses mécanismes essentiels. Cela nous a amenés à définir deux classes principales : la classe Game, chargée de la gestion globale du jeu, et la classe Player, représentant chaque joueur et ses interactions avec le jeu. Nous avons également identifié les principales fonctionnalités nécessaires, telles que la gestion des cartes, des ressources et des interactions entre les joueurs.

****Nous avons fait le choix de commencer par le code concernant le déroulement du jeu avant de passer à la mise en place des différents threads, des connexions TCP et des messages queue, dans l’optique de n’avoir après leur ajout au code, de n’avoir qu’à faire un minimum de modification au code pour qui soit terminer (notamment lors de l’affichage des cartes des autres joueurs pour un joueur qui joue, passant par les messages queue pour obtenir ces informations, mais aussi lors de l’ajout d’une carte à un feu d’artifice par un joueur, qui doit obligatoirement être fait par la classe Game et donc utiliser une connexion TCP). Voici le diagramme représentatif de notre programme :

# Classe Game :

Comme dit précédemment, la classe Game gère le jeu du début à la fin de la partie. C’est elle, et elle seulement, qui peut modifier les valeurs stockées dans le deck ou dans les piles en construction ainsi que les valeurs des tokens.

Elle prend en paramètres :

* Numplayers : qui correspond au nombre de joueurs
* port = qui est le numéro de port de la connexion tcp serveur

Il crée ensuite les valeurs et tableaux suivant :

* Players : qui est un tableau contenant tous les joueurs de la partie en cours
* locks : qui est un tableau contenant tous les semaphores des joueurs
* suits\_in\_construction et allsuits : qui sont toutes les piles stockés dans le tableau allsuits
* couleurs : qui est un tableau regroupant toutes les couleurs des cartes
* info\_tokens et fuse\_tokens : noms explicit
* discard\_deck = tableau du deck de défausse
* deck = tableau deck de pioche
* allHand = tableau contenant les mains de tous les joueurs
* game\_lock = Semaphore du jeu
* buffer\_lock et buffer\_locks : Semaphore sur le buffer pour tous les joueurs, stockés dans le tableau buffer\_lock
* buffer = nom explicite
* allplayers\_threads = nom explicit
* alltcp\_process= nom explicit
* tokens = qui est un tableau contenant les tokens info et fuse
* nb\_tour = nom explicit
* endpile = qui est un tableau où sont stockés des True et False, qui donne l’information si les piles sont complètes et bien rempli ou non.

Durant tout le programme, tous les tableaux sont liés car chaque case de chaque tableau de même rang correspond au joueur du même rang dans le tableau players. Ainsi allHand[0] renvoie la main de players[0] et cela pour tous les tableaux liés aux joueurs comme locks ou buffer\_lock.

Nous avons d’abord commencé par créer le deck et l’affichage des cartes. Pour ce faire il a fallu trouver un moyen d’exprimer à partir d’une unique valeur numérique, le chiffre et la couleur d’une carte. Nous avons fait le choix de traduire ces 2 informations par le reste+1 et le quotient d’une division par 5 de cette valeur (% et //). Pour les cases vides de ce tableau et pour tous les autres tableaux de cartes tel que les mains des joueurs, nous avons fait le choix arbitraire de prendre -1 comme valeur. On obtient donc la fonction create\_cards(), qui créer le deck et le mélange via la fonction shuffle().

Pour obtenir la couleur et la carte en question, nous avons écrit 2 nouvelle fonction trad\_card() et trad\_1card() qui, respectivement, traduisent les valeurs ou la valeur donnée pour en sortir une couleur et un chiffre.

Nous nous sommes ensuite penchés sur un affichage esthétique de ces cartes par les fonctions affiche\_carte() et affiche\_cartes(), permettant d’afficher les mains des joueurs ainsi que les suits en construction. Pour ce faire, on regarde d’abord, si la carte existe en passant par la fonction 1 carte avant de l’afficher en prenant en compte en résultat obtenu. On obtient des cartes de la forme suivant :

+-------+

| 4 |

| y |

| 4 |

+-------+

Pour la couleur, pour faciliter l’affichage de la carte, nous avons fait le choix de n’afficher que la première lettre. On obtient donc « r » pour « red », « b » pour « blue », « y » pour « yellow » etc…

Il fallut ensuite créer les mains des joueurs à partir du deck. Nous avons donc créer la fonction Player\_hand() qui créer les mains de tous les joueurs et les stockent dans le tableau allhand au rang approprié pour qu’elles soient attribuées aux joueurs lors de leurs création. Pour cela, on prend la carte du rang 0 du deck, et on décale toutes les autres valeurs de i vers i-1 et l’on attribue à la dernière case du tableau la valeur -1.

Enfin, avant de passer à la message queue, au thread et aux connexions TCP, il fallut implémenter la logique de modification du deck des piles en construction ou sur la valeur des tokens, lorsque le joueur prendra une décision, que ce soit donner une information ou poser une carte dans une pile. Pour cela, nous avons écrit 3 fonctions : get\_suits\_color\_number(), prenant en paramètre un string d’une couleur et qui fait simplement le lien entre le nom d’une couleur et sont positionnement dans le tableau allsuits, draw(), prenant en paramètre le rang de la carte dans la main du joueur, et le joueur en question, qui permet de remplacer la carte posée ou non par le joueur par une nouvelle carte du deck et enfin suivre la même logique que dans la fonction logic\_buffer(), qui avant l’implémentation de la connexion TCP, prenait en paramètre un message, mais après son implémentation, elle pourra directement avoir accès au buffer de cette connexion. En fonction des informations reçu à travers le buffer, elle pourra faire les modifications nécessaires au bon déroulement de jeu.

Passons maintenant à l’ajout des connexions TCP et des threads, implémentées dans la classe Game (message queue est implémentée dans la classe Player). Pour l’implémenter, il a fallu importer plusieurs librairies tel que *threading* et *socket* et créer plusieurs fonctions :

* logic\_buffer()
* get\_socket\_message(client,lock,\_my\_lock)
* create\_player\_lock()
* is\_finished()
* start()
* create\_player\_lock()

Commençons par get\_socket\_message() qui prend en argument le client, c’est-à-dire la connexion TCP vers ce client, et 2 locks, un pour lock le client en question, pour qu’il ne modifie plus le terminal à la fin de son tour et faire un my\_lock.acquire(), et un lock pour le game quand c’est à son tour de faire les modifications, avec un lock.release(). Cette fonction, qui sera ensuite utilisée dans un thread créé par la fonction start(), permet d’obtenir les informations envoyées par un client vers le serveur et de les stock dans la variable buffer pour qu’elles soient accessibles à toutes les fonctions de la classe Game.

La fonction creat\_player\_lock() crée les joueurs en fonction de la valeur de num\_players et crée leurs locks.

La fonction is\_finished() vérifie les conditions de fin du jeu et l’arrête si elles sont remplies en renvoyant une valeur True. Il vérifie aussi les conditions de victoire et renvoie vrai ou faux en fonction.

Ensuite, la fonction logic\_buffer(). Elle permet, à partir d’une information reçue par le buffer, qui correspond donc à l’action du joueur, de modifier les valeurs du jeu en fonction. Que se soit modifier la valeur des tokens, ou ajouter une carte à une pile ou à la défausse, c’est à cette fonction de faire toutes ces modifications.

Enfin, la fonction start(). C’est elle qui lance le jeu en créant les threads nécessaires à l’échange TCP mais aussi pour le bon déroulement du tour d’un jour. Elle crée aussi les semaphores des buffers des players et du jeu. Il lance enfin le jeu via une boucle while qui ne s’arrêtera qu’à travers la condition du is\_finished()

# Class Player :

La classe Player est appelée en tant que thread par la class Game. Elle permet de gérer toutes les interactions possibles pour chaque joueur. Nous avons commencé par initialiser toutes les variables nécessaires à son fonctionnement :

* ID = l’identifiant du joueur
* Queue = la message queue permettant aux threads des joueurs de communiquer entre eux
* Lock = Son semaphore
* Port = le port de la connexion TCP
* Hand = tableau contenant les mains de tous les joueurs
* Allsuits : tableau contenant toutes les piles
* Players : tableau contenant tous les joueurs
* Couleur : tableau contenant toutes les couleurs
* Tokens = tableau contenant tous les tokens

Nous avons ensuite implémenté toutes les fonctions utiles aux joueurs en commençant par la création des indices, par la fonction def\_indice. En effet, la liste hand\_indice est une copie de la main réel du joueur mais lié à une liste indice qui contient des valeurs False,False à chaque case. Permet de cacher les valeurs et les couleurs des cartes, ou de les afficher en mettant la valeur à True. L’affichage passe par les fonctions show\_my\_indice() et show\_my\_indices(), qui sont quasiment des copies de la fonction afficher\_carte() et affiche\_cartes() de la classe Game (aussi dans la classe Player), mais elle prend en compte les valeurs stockées dans la liste indice.

Respectivement, l’affichage des piles et des cartes des autres joueurs que le joueur actuel passe aussi par la classe Player à travers la fonction show\_suits() et show\_cartes(), utilisant afficher\_carte.

Passons maintenant au déroulement d’un tour. Il utilise un bon nombre de fonctions telles que choix\_suits(), choix\_info(), choix\_cartes, et choix\_indices, qui respectivement, affiche les piles, envoie les informations sur la message queue pour être lu par le prochaine joueur, pose une carte dans une pile en utilisant add\_to\_suits() qui traite la carte et envoie par connexion TCP des valeurs en fonction de si la carte peut être posée ou non, et enfin affiche les indices reçus par la message queue. Toutes ces fonctions utilisent la fonction get\_choix pour permettre au joueur de faire des input de ses choix. La fonction action() est ensuite une série de choix permettant au joueur de jouer son tour et de prendre les informations du jeu.

Pour la message queue, permettant aux joueurs de communiquer entre eux, ils nous faut ajouter à cela 3 autres fonctions :

* send\_message() : qui met un message dans la queue
* get\_all\_msg() : qui renvoie un liste du contenu de la message queue
* set\_indice\_and\_reload() : qui traite la message queue en ne prenant que les infos qui intéresse le joueur actuel pour modifier sa hand\_indice et remet le reste dans la queue.

Enfin, la fonction game\_start() crée les sockets clientse, se connecte au serveur, et crée une boucle while, qui lock son semaphore en attente d’être libéré quand ça sera son tour.

# Conclusion :

En conclusion, le projet Hanabi a été une expérience enrichissante qui nous a permis d'explorer les aspects techniques et conceptuels du développement de jeux de cartes en Python, mais aussi et surtout, de la manipulation de threads et la communication entre eux. Malgré les défis rencontrés lors de l'implémentation des règles spécifiques du jeu et de la coordination des actions des joueurs, nous sommes parvenus à créer un jeu fonctionnel et esthétique.