**Title:** Multiplayer Hanabi Game: Project PPC

**Project Name:** Hannabis

**Author Information:**

* Name: Hanqi Lin
* Student ID: 4024801
* Course: TC3/1
* Institution: INSA LYON
* Report Date: 30/01/2024

**Table of Contents:**

1. Introduction
2. Game Design
3. Message queue
4. Multi-Processing Implementation
   * 4.1 Process Management
   * 4.2 Inter-Process Communication
5. Multi-Threading Approach
   * 5.1 Thread Synchronization
   * 5.2 Real-Time Data Handling
6. Network Communication
   * 6.1 Socket Programming
   * 6.2 Client-Server Model
7. Challenges and Solutions

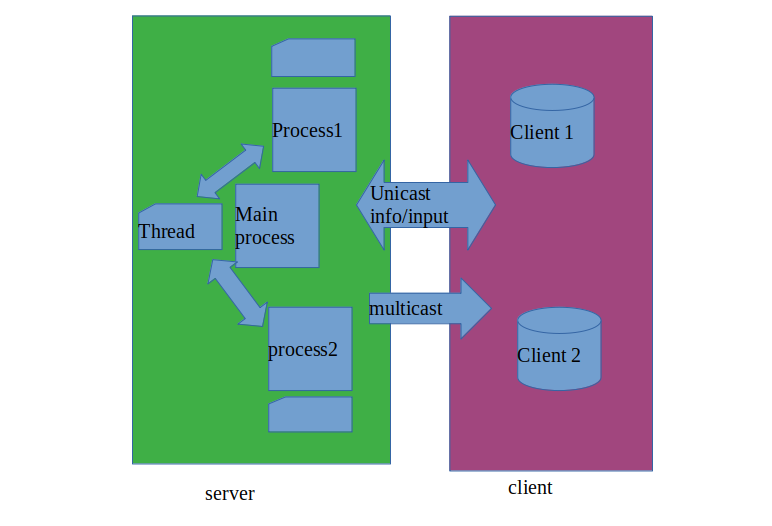
**I.Introduction**

This is a Hanabi game program designed by me, which integrates multi-processing, multi-threading, and Socket communication technologies to create a multi-player online gaming environment. The launch of the game involves a multi-process manager responsible for synchronizing shared variables between different processes. The number of players is determined by the input parameter, supporting 2 or 3 players. The randomly generated deck, players' hands, and the initial victory pool are all stored as shared variables for cross-process access.

The game server relies on Socket to listen for player connection requests on a designated IP address and port. Once all players have connected, the game server initiates the start of the game and generates an independent subprocess for each player to handle their game actions.

Direct interaction between the main process and subprocesses is achieved through message queues, while player clients establish connections with the server via Socket, thereby interacting with different processes within the game. The server utilizes unicast or multicast technology to send messages or input requests to clients, thereby enabling effective communication with the clients.

This design ensures the smooth operation of the game and real-time interaction among multiple players, providing a synchronized online multiplayer gaming experience.



**II.Game design**

In my Hanabi game program, players can perform the following actions each turn:

1. **Inform**: A player can choose to inform another player about specific information regarding their hand. This might involve the color or number of the cards in their hand. This action consumes an information token.
2. **Play a Card**: A player attempts to play a card to the victory pool. If the card matches the current sequence in the victory pool for that color (i.e., it is the next number in sequence), the card is successfully added. Otherwise, the card is discarded, and a fuse token is consumed.
3. **Discard a Card**: A player can choose to discard a card. This action earns the player an additional information token and allows them to draw a new card from the deck. This rule can make the game easier.

### How to Win the Game

To win the game, players need to successfully create a complete sequence of each color in the victory pool, from 1 to 5. This means that for each color, players must play the cards in order (1, 2, 3, 4, 5).

### How to Lose the Game

The game can be lost in several ways:

* **Exhausting All Fuse Tokens**: Each time a card is played incorrectly, a fuse token is consumed. When all fuse tokens are used up, the game ends.
* **Depleting the Deck**: When all the cards in the deck have been drawn, and players can no longer draw cards, the game ends.

Strategy and teamwork are key to winning the game. Players need to communicate effectively and plan their actions to ensure they use resources efficiently and play cards correctly.

**III.Message queue**

In my Hanabi game program, the message queue plays a crucial role in facilitating communication between the main process and the child processes, ensuring that game actions and information are accurately transmitted across different components of the application.

**Purpose and Functionality**

**Communication Mechanism**: The message queue serves as a communication channel in a multiprocessing environment. It allows for safe and reliable passage of messages between the main game server process and individual player processes.

**Synchronization**: It ensures that information remains consistent and synchronized across all processes, which is vital in a game where each player's actions can affect the game state.

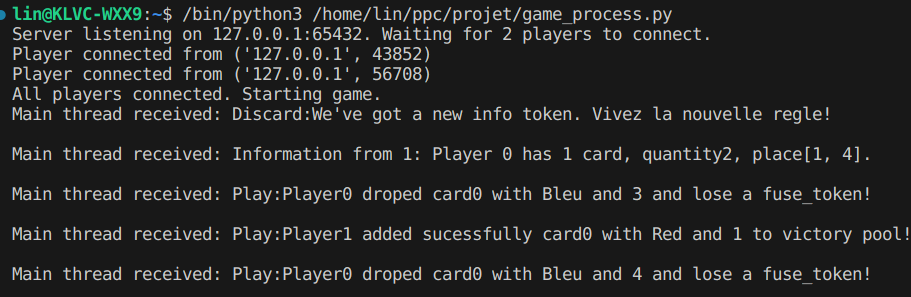
**How It Works**

**Sending Messages**: When a significant game event occurs, such as a player taking an action (informing, playing a card, or discarding a card), the corresponding process (either a child or the main process) puts a message into the queue.

**Receiving Messages**: The receiving process (whether the main process or a child process) continuously monitors its respective message queue. When a new message arrives, it retrieves it for processing.

**Content of Messages**: Messages may contain various types of information, such as notifications of actions taken by players, updates to the game state, requests for player input, or even control messages to synchronize game progress.

Currently, message\_queue is used to deliver key information. Specifically, each player's turn actions are passed here, and then processed and distributed in the main program. This is why after playing a game, you can see concise game records in the server.



**IV.Socket**

In my Hanabi game program, sockets are employed to establish a network communication interface between the game server and the clients (players).

**Server-Client Communication**

**Connection Establishment**: The server opens a socket on a specific IP address and port, listening for incoming connections from clients. Each client, through its own socket, connects to this server socket.

**Data Transfer**: Once the connection is established, the server and clients communicate by sending and receiving data through their respective sockets.

**Client-Side Socket Usage**

**Connecting to the Server**: The client socket initiates a connection request to the server's IP address and port. Upon successful connection, the client receives confirmation, and a dedicated communication channel is established.

**Message Handling**: The client continuously listens for messages from the server. These messages are received as a stream of bytes, which the client decodes and processes.

**Buffer Management**: The client maintains a buffer to store incoming data. Since messages can be fragmented or combined during transmission, the buffer helps in assembling complete messages based on a delimiter (newline \n in your case).

**Interactive Responses**: When the server requests input (indicated by messages starting with "Input: "), the client captures user input and sends it back to the server.

**Multiple methods**: There are three basic communication methods in the game, unicast information, unicast input request, and broadcast information. Each type of communication plays a vital role and fulfills different needs.

**Role in the Game**

**Real-Time Interaction**: Sockets enable real-time interaction between the server and multiple clients. Players receive immediate updates about game events, and their actions are promptly relayed to the server.

**Multiplayer Coordination**: They facilitate coordination among players in this multiplayer setting, as each client communicates with the central server.

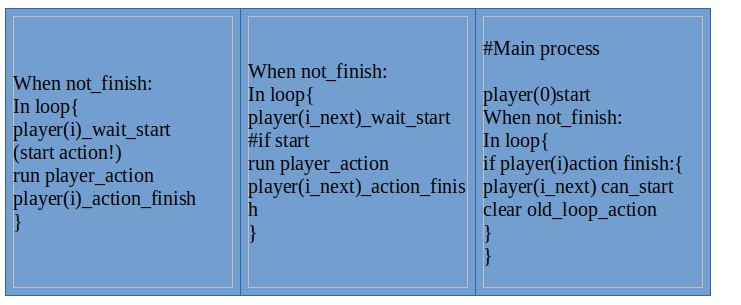
**Flexibility and Scalability**: The use of sockets allows the game to be played over a network, supporting remote multiplayer gameplay and enhancing the game's scalability.

V.Loop and End game

Ensuring that multiple processes run in alternating cycles is a complex issue. But we can solve it easily:

1. We ensure that all processes in the game will wait for each other before initialization is completed to avoid errors at the beginning of the cycle.

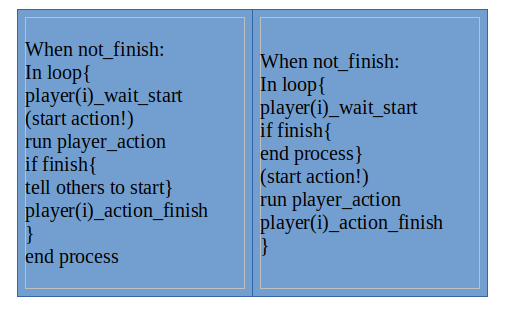
2. If running processes concurrently cannot meet the requirements of a turn-based game, we can achieve this by letting another process (such as the main process) set the state of some global variables:



In actual application, the player's position is replaced by (No.player+1) mod (number of player)

It can be seen that whenever the cycle of a process ends, the next process will be notified to start running, thus satisfying the cycle requirement.

3.If a player's player\_action ends running, the conditions for the game to end are met. But the other process is still waiting for the start notification and gets stuck there. That's really bad!

To do this, we set up a "lure" code that forces the other program out of the loop. :

Now all processes can be stopped in a right way!

VI.What can be improved

I have to admit, this project was very challenging for me. The basic game logic is not difficult, but when it comes to adding various codes to meet the requirements, I encounter a lot of difficulties!

So first of all I would say, I should plan my time better! Avoid rushing through tasks!

Then, my current client can print all the information, and since this is a turn-based game, the information is all timely. However, the order of printing information is relatively random, and some of the displayed contents are stacked together without line breaks.

If I want to improve, I will change the printing logic of the program, and also use tools (such as locks, signals) to improve the output sequence.

Finally, I will make the code more concise and readable. Of course, now I just add some basic annotations. If only I had enough time to finish it!