# CS494 - Lab Report

## I. Notes

### 1. Demo video

Video link: https://youtu.be/DOMDaGIr23Q

### 2. Assets folders

The client executable file must be put in the same folder with the folder `client` in the `Release` folder.

This `client` folder contains the assets needed for the client to run properly.

Using PyInstaller to build the executable file for both client and server.

### 3. Default configurations

The default address of the server will be `localhost:54321`, the same for the client program, which automatically initiates a connection to this address when the user starts the client.

### 4. Build & run the code

- Build & run server with **default settings**:

  Run `server-binary.exe` in the `Release` folder.

- Build & run server with **custom settings**:

```
Usage: python server/server.py [-h] [-p PLAYERS] [-r
RACE_LENGTH] [-t TIME_ANSWER]

options:
  -h, --help            show this help message and exit
  -p PLAYERS, --players PLAYERS
                        Set the maximum number of players
Default to 10.
```

```
  -r RACE_LENGTH, --race-length RACE_LENGTH
                        Set the race length. Default to 10.
  -t TIME_ANSWER, --time-answer TIME_ANSWER
                        Set the answer time limit. Default to
30 (seconds).
```

# II. Game Story

In the country of Gubhit, every 4 years, they have to find out who is the best to lead the country. Traditionally, all candidates must participate in a racing game. *The special thing is that none of the cars have gasoline*. Each turn of the race, players must overcome an obstacle, specifically answering a calculation question correctly to get fuels for their car. On the other hand, if any player answers incorrectly, *their car will be pushed back by a mysterious force* (the future leader should not make any mistake!). The first to reach the finish line would be announced as leader of Gubhit. If no one reaches the finish line, then it is doom for the citizens all over Gubhit...

# III. General message format

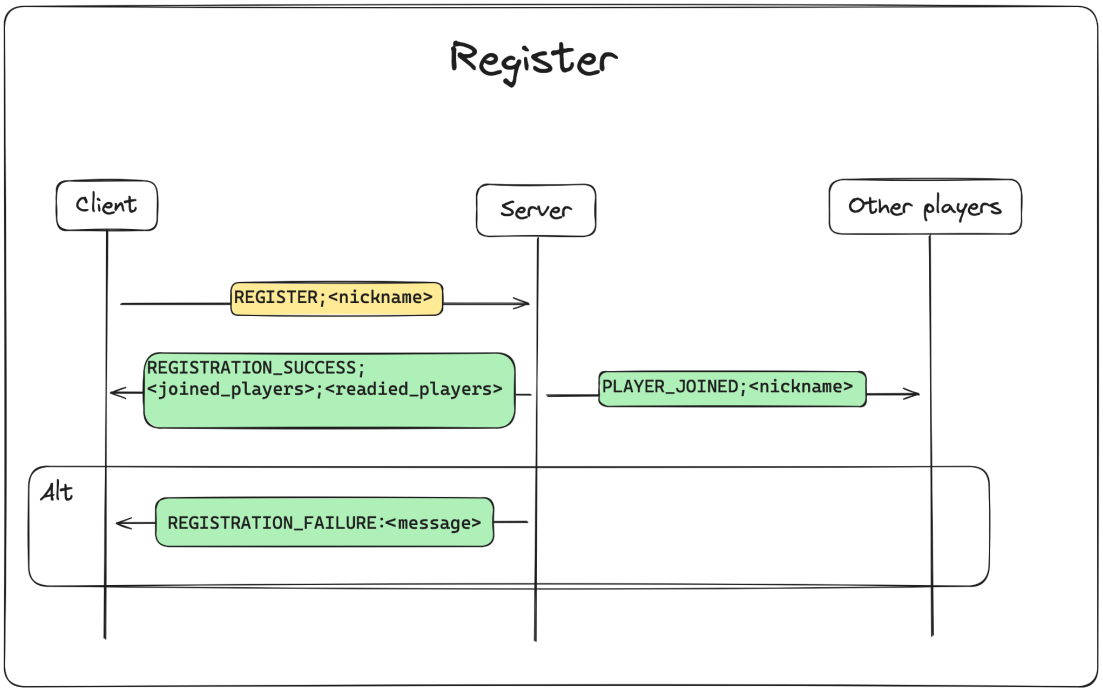
- We use **TCP** because of some benefits of this protocol: reliable, in-order delivery of messages, and connection setup out of the box. Other benefits that could be useful include congestion control, and flow control.

- Our design of messages is straightforward: **each component of the message is separated by a semicolon ";"** (components can be either a value or a tuple). The first component is like the "status code" of the message, allowing us to know what it is intended for. Depending on this code, the server or client will expect some or no components after it.

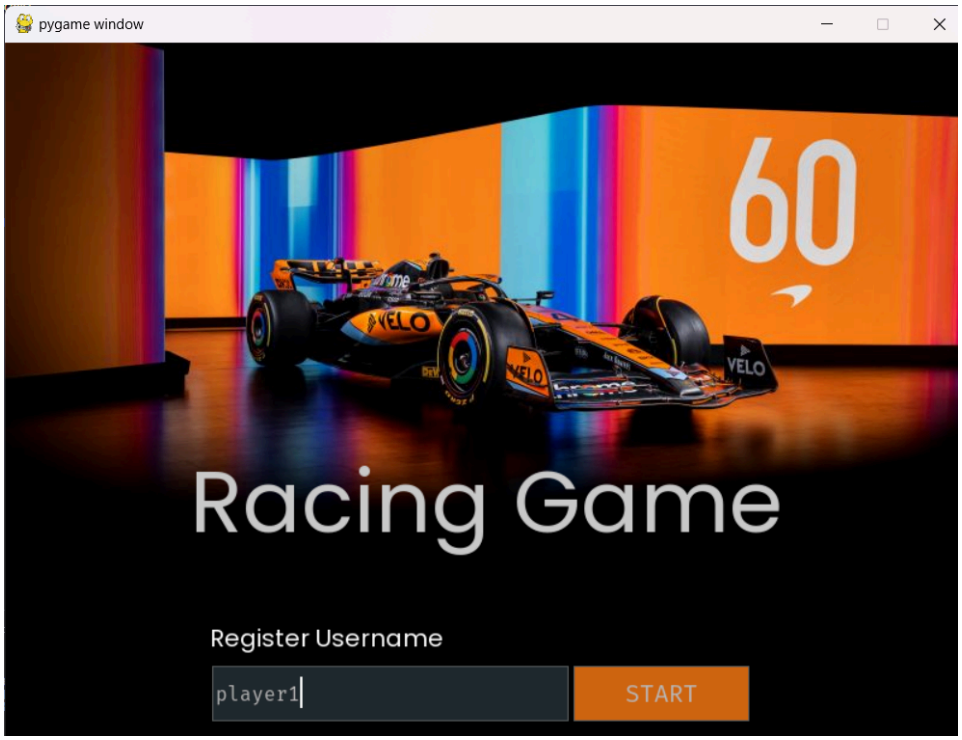
- See below for the detailed message formats.

# IV. Phase 1: Registration

## 1. Game flow




- The **Registration Scene**:




- This is the first scene that the client will see.

- In this scene, the client is required to register with a nickname.
- After this step, the client will join the Lobby as a registered player.

## 2. Code snippets

*(These code snippets are simplified from the source code. Use function names to search for the location in the source code)*

```
[server.py]
[ClientManager::handle_conversation()]
async def handle_conversation(
    self, reader: asyncio.StreamReader, writer: asyncio.StreamWriter
) -> None:
    while True:
        data: bytes = await reader.readline()
        message: str = data.decode().strip()

        # Parse message
        # Ex: ANSWER;123 --> command: "ANSWER", args: ["123"]
        command: str
        args: List[str]
        command, *args = message.split(";")

        if command == "REGISTER":
            if len(args) != 1:
                writer.write("REGISTRATION_FAILURE;Invalid
arguments.")
                continue

            # Parse message data
            nickname: str = args[0].strip()

            # Handle command
            # Note: If the game state not appropriate, this function
            # will raise an exception, which is already handled
            player: Player = game.handle_registration(nickname)
            # Return success status
            writer.write(
f"REGISTRATION_SUCCESS;{game.player_manager.pack_players_lobby_info()
```

```
}."
            )
            # Broadcast to other players that a new player has joined
            await client.broadcast(f"PLAYER_JOINED;{nickname}")

        elif command == "READY":
```

- In this project, we use the `asyncio` (a standard library in Python) to deal with network communication including receiving and sending messages. Which does automate a lot for us:
    - **Handles asynchronous communication**: It uses an event loop to efficiently manage multiple sockets simultaneously, instead of manually polling for data.
    - **Streamlined Data Reading and Writing**: Asyncio provides the `asyncio.StreamReader` and `asyncio.StreamWriter` classes that streamline data interaction. You simply use `await reader.readline()` to read a line or `writer.write(data)` to write data. Asyncio handles buffering and ensures complete data transfer.
    - **Coroutines**: `asyncio` lets us write your socket-handling logic as coroutines, making it more robust and structured.

- This part describes how the server parses a message from clients and handles the two cases in the registration phase.
    - Case 1: Registration Success. The client joins the Lobby.
        - The client is notified of the current lobby information, including who joined before them, and their READY statuses.
        - At the same time, other players are also notified that a new player has joined.
    - Case 2: Registration Failure. The server simply replies to that client what problem occurred.

## 3. Message formats

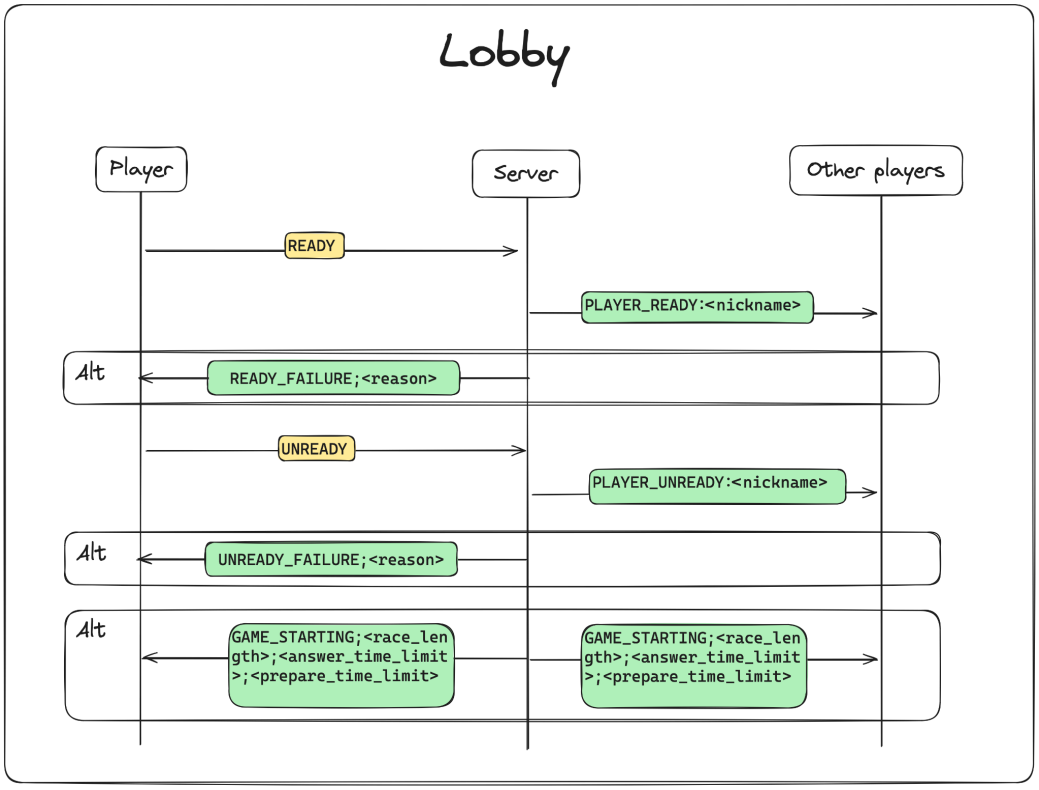
- When player sends register request to server:

```
REQUEST from client:
REGISTER;<nickname>
```

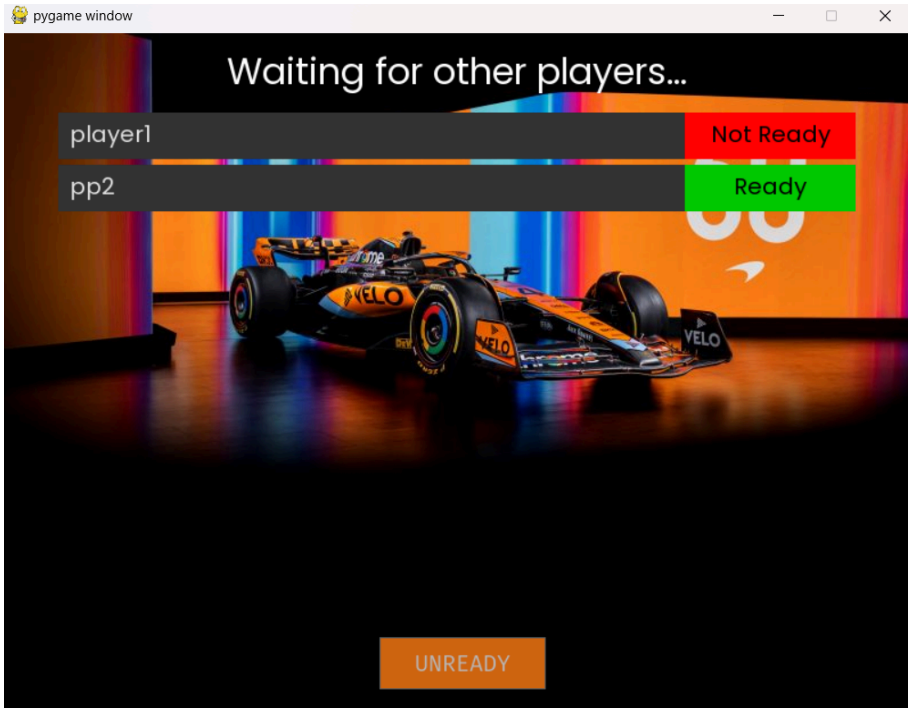
```
Server will send one of these RESPONSEs:
REGISTRATION_SUCCESS;<nickname 1>,<is_ready 1>;...;<nickname n>,<is_ready n>
REGISTRATION_FAILURE;<reason>

Server will BROADCAST to all other players:
PLAYER_JOINED;<nickname>
```

# V. Phase 2: Lobby

## 1. Game flow




- The **Lobby Scene**:




- In the Lobby phase, players can see other joined players and their current ready state.
- Players can press the “READY” button to send a ready message, then the server will broadcast the ready status to other players. The same process will happen in case the player presses the “UNREADY” button.
- When all players in the Lobby are ready, the server will start the *in-game phase* and **broadcast a game starting message** (with the code `GAME_STARTING`) to all players.

## 2. Code snippets

```
[server.py]
[ClientManager::handle_conversation()]
elif command == "READY":
    if len(args) != 0:
        writer.write("READY_FAILURE;Invalid arguments.")
        continue

    # Handle command
    await game.handle_ready(player_nickname)
```

```
    # Broadcast to all players to update the UI
    await client.broadcast(f"PLAYER_READY;{player_nickname}")

elif command == "UNREADY":
    if len(args) != 0:
        writer.write("UNREADY_FAILURE;Invalid arguments.")
        continue

    # Handle command
    game.handle_unready(player_nickname)
    # Broadcast to all players to update the UI
    await client.broadcast(f"PLAYER_UNREADY;{player_nickname}")

elif command == "ANSWER":
```

- The server handles 2 kinds of requests in this phase: READY and UNREADY.
- For details, it works exactly like the below description:
    - Receive READY: mark the player as ready and broadcast the information to other players.
    - Receive UNREADY: mark the player as unready and broadcast the information to other players.

## 3. Message formats

- When player presses “Ready”:

```
REQUEST from client:
READY

Server may send RESPONSE:
READY_FAILURE;<reason>

Server will BROADCAST to all other players:
PLAYER_READY;<nickname>
```

- When player presses “Unready”:

```
REQUEST from client:
```

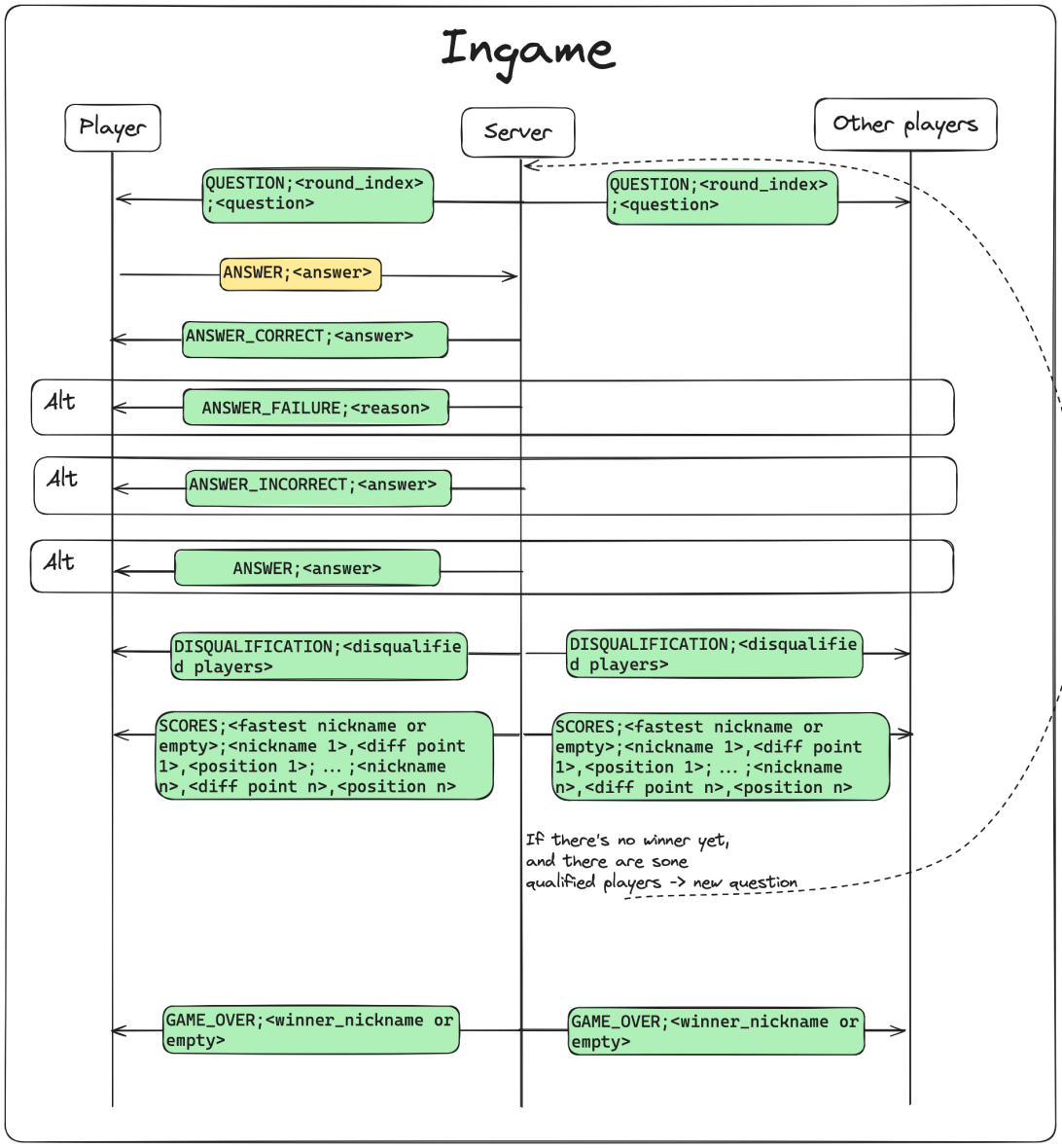
```
UNREADY

Server may send RESPONSE:
UNREADY_FAILURE;<reason>

Server will BROADCAST to all other players:
PLAYER_UNREADY;<nickname>
```

- When all conditions for the game to start is met:

```
Server will BROADCAST to all players:
GAME_STARTING;<race_length>;<answer time limit>
```

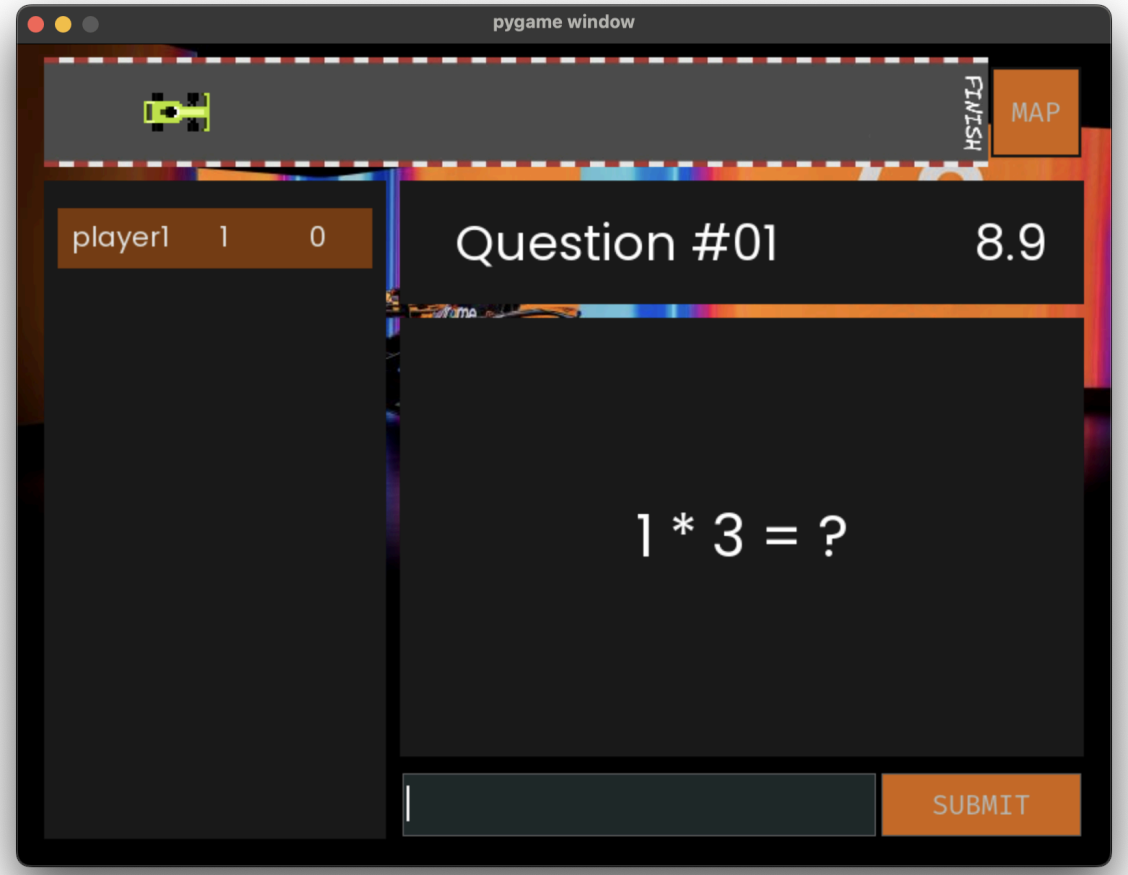
# VI. Phase 3: Play the Game!

## 1. Game flow



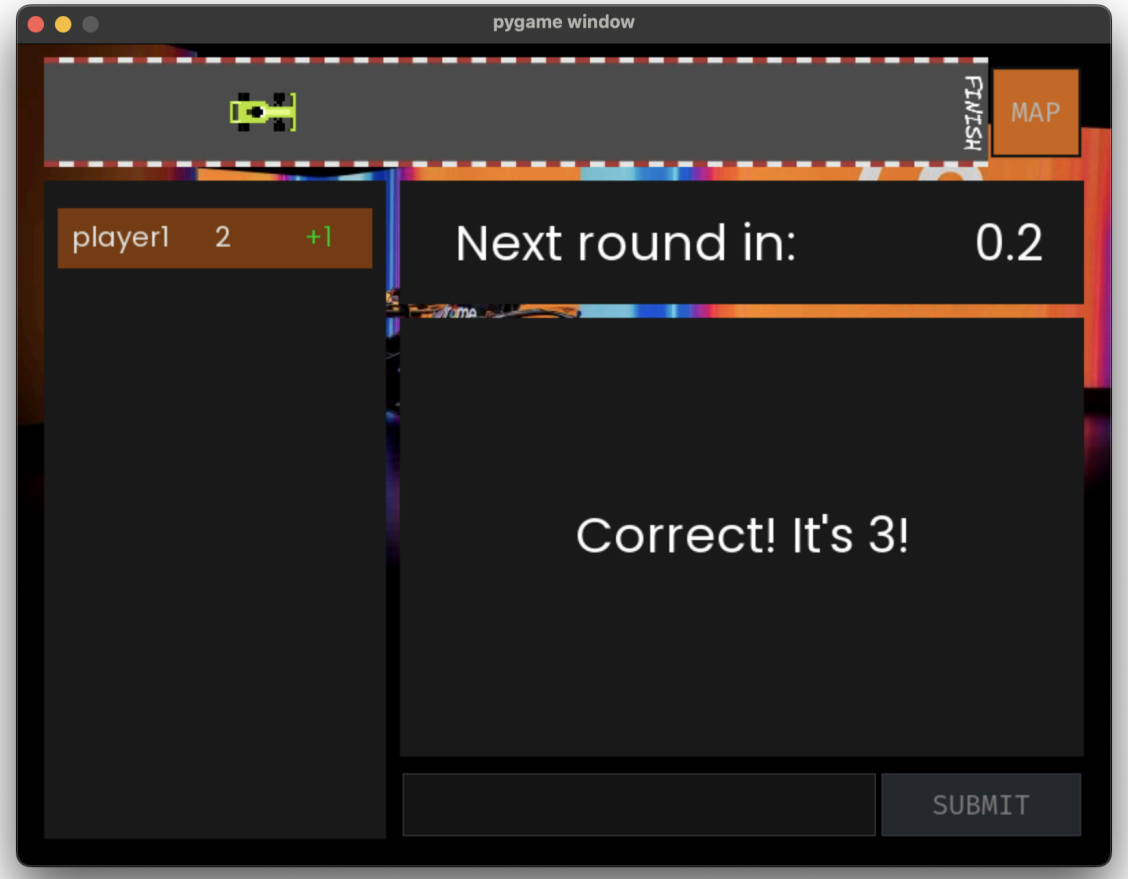

- While in-game, the players will have to answer a series of questions. The game will end in 2 cases:
    - One player has reached the winning points (called `race_length`)
    - All players have been disqualified or disconnected.

- Each time the server asks a question and players respond is called a round. The diagram shows the messages sent and received during a round **in chronological order**. In details:
    - First, the server will **broadcast the question** to all players (disqualified players included). The server will accept answers from players for some time (set by `answer_time_limit` in `GAME_STARTING` message last section).



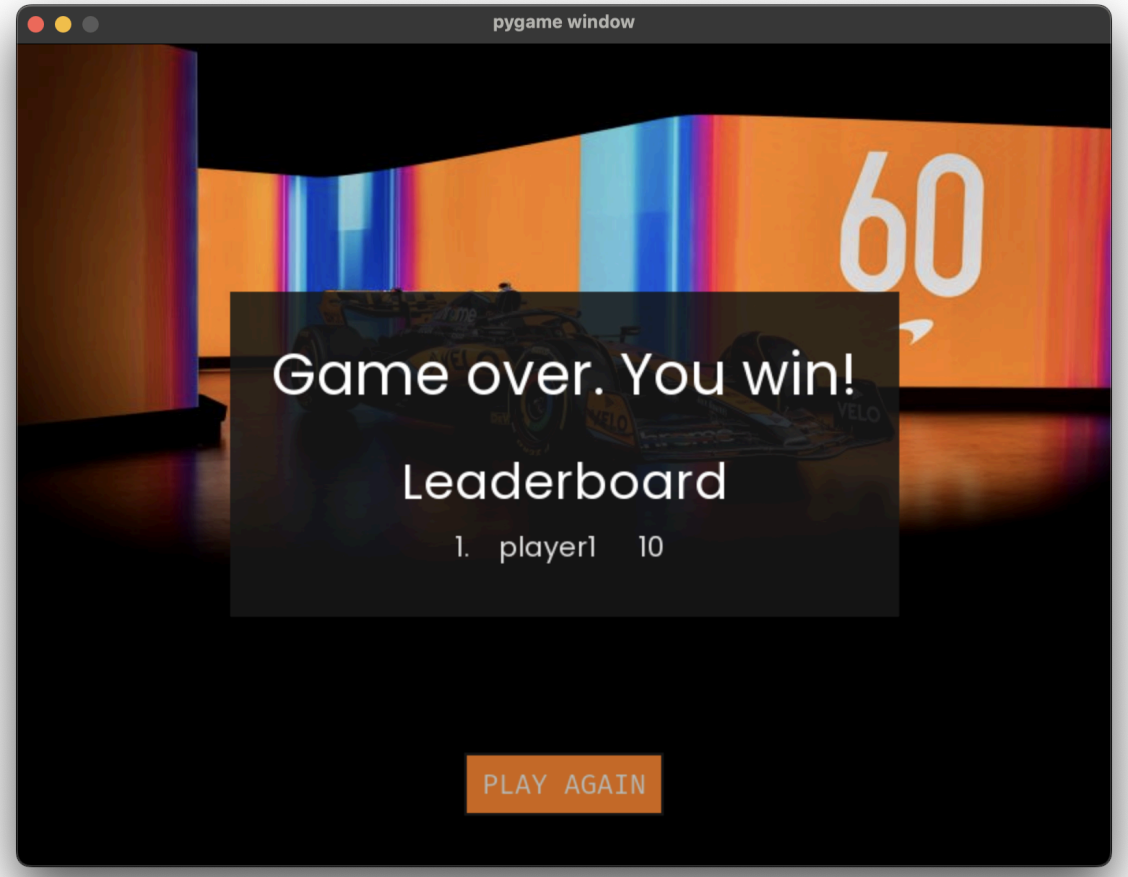

    - Each qualified player will **send their answers back** to the server.
    - After the time is up, the server will send the result to qualified players (correct or incorrect), and the correct answers to all players. Note that the server will only consider its own timer, and **ignore the client's timer**.

- Next, the server will **broadcast the list of players who have been disqualified**.
- Then, the server will **broadcast the scores of qualified players** to all players. The score will be calculated as followed:
  - Those who don't give an answer or answer incorrectly receive the same penalty: -1 score. Players with 3 consecutive wrong answers/not giving answers will be disqualified.
  - Those who give the correct answer will receive +1 score.
  - The correct and fastest player will also receive bonus score, which is the total score others lose (if everyone is correct, then bonus score is just 0).



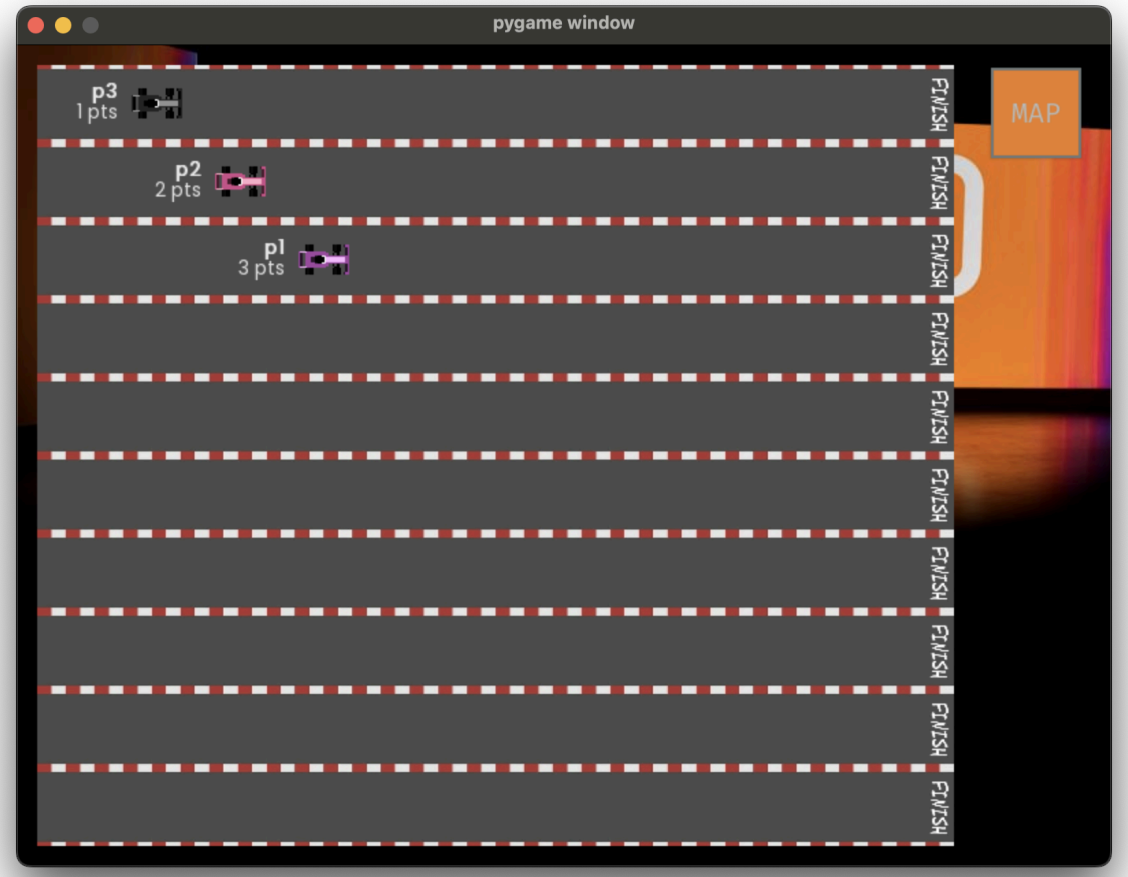

- The server will check if there is a winner. A winner is someone who has a score >= `race_length`. If such person exists:

        - The server broadcasts a `GAME_OVER` message, along with **the winner's name**.
    - Else, if there is no winner, but rather, all players have been disqualified or disconnect:
        - The server broadcasts a `GAME_OVER` message **with an empty winner name**.




    - Otherwise if the game is not over, clients will be given some time to display the results, and the server will **start another round**.

- At any time in-game, players can view the racing lanes by pressing “Map” button:




- **Disqualified players can still observe the game**: they can see the questions and the answers, but they cannot send their answer in any way.

## 2. Code snippets

```
[server.py]
[Game::game_loop()]
async def game_loop(self):
    while self.state != GameState.LOBBY:
        # LOBBY state: If in this state, then not in-game.
        # PROCESSING state: Preparing the question, processing the
answers. Server does not accept any message from clients.
        # WAITING_FOR_ANSWERS state: Waiting for the clients to
```

```
answer the question. Server accepts only answers from clients.

        self.state = GameState.PROCESSING
        # Time for players to prepare
        await asyncio.sleep(PREPARE_TIME_LIMIT)

        question: Question = self.generate_question()
        # Send the question to all clients
        await client.broadcast(f"QUESTION;{round_index};{question}")

        self.state = GameState.WAITING_FOR_ANSWERS
        # Wait for the clients to answer
        await asyncio.sleep(ANSWER_TIME_LIMIT)

        self.state = GameState.PROCESSING

        for player in self.players:
            # Check player's answer and calculate score
```

- In this game, everybody is given time to prepare for the next question. This code snippet is in charge of putting itself in the `PROCESSING` state to make sure it does not receive any more requests when it is waiting (via sleeping) to send the next question. After that, it will switch to the "waiting for answers" state (`GameState.WAITING_FOR_ANSWERS`) and "processing answers/generating questions" state (`GameState.PROCESSING`).
- This is done in a loop, which loops until the game is over.

```
[server.py]
[ClientManager::handle_conversation()]
elif command == "ANSWER":
    if len(args) != 1:
        writer.write("ANSWER_FAILURE;Invalid arguments.")
        continue

    player_answer: int = int(args[0].strip())
    # Handle command
```

```
    game.handle_answer(player_nickname, player_answer)
```

- While in-game, the only type of message players can send is `ANSWER`. When receiving an answer from a player, the server saves that answer and its timestamp. The server then checks and calculates the corresponding scores later. This is demonstrated in the code section below.

```
[server.py]
[Game::game_loop()]
# Calculating bonus score for fastest player is trivial and so is
   omitted
for player in self.players:
       # Check player's answer and calculate score
       if not player.is_disqualified:
           if is_correct(player.answer):
               # Correct answer
               player.score += 1
               player.wa_streak = 0

               await client.write_to_player(
                   player, f"ANSWER_CORRECT;{question.answer}"
               )
           else:
               # Incorrect answer
               player.score = min(0, player.score - 1)
               player.wa_streak += 1

               await client.write_to_player(
                   nickname, f"ANSWER_INCORRECT;{question.answer}"
               )
       else:
        # Send answer to disqualified players
           await client.write_to_player(player,
  f"ANSWER;{question.answer}")
```

- This section calculates scores, sends results to qualified players, as well as the answer to disqualified players to observe.

## 3. Message formats

- When server sends new question to all players:

```
Server will BROADCAST to all players:
   QUESTION;<round index>;<first number>;<operator>;<second
   number>
```

- When player sends answer to server:

```
REQUEST from a qualified player:
   ANSWER;<answer>

   Server may send RESPONSE:
   ANSWER_FAILURE;<reason>
```

- When server announces player's answer is correct:

```
Server will send to players with correct answer:
   ANSWER_CORRECT;<correct answer>
```

- When server announces player's answer is incorrect:

```
Server will send to players with incorrect answers:
   ANSWER_INCORRECT;<correct answer>
```

- When server sends answer to disqualified players (disqualified players can still observe the game):

```
Server will send to disqualified players who are observing:
   ANSWER;<correct answer>
```

- When server announces score and disqualified status of all players after a round:

```
Server will BROADCAST to all players:
   DISQUALIFICATION;<nickname 1>;...;<nickname n>
   SCORES;<fastest nickname or empty>;<nickname 1>,<diff point
   1>,<position 1>;...;<nickname n>,<diff point n>,<position
   n>
```

- When the game is over:

```
Server will BROADCAST to all players
   GAME_OVER;<winner nickname or empty>
```

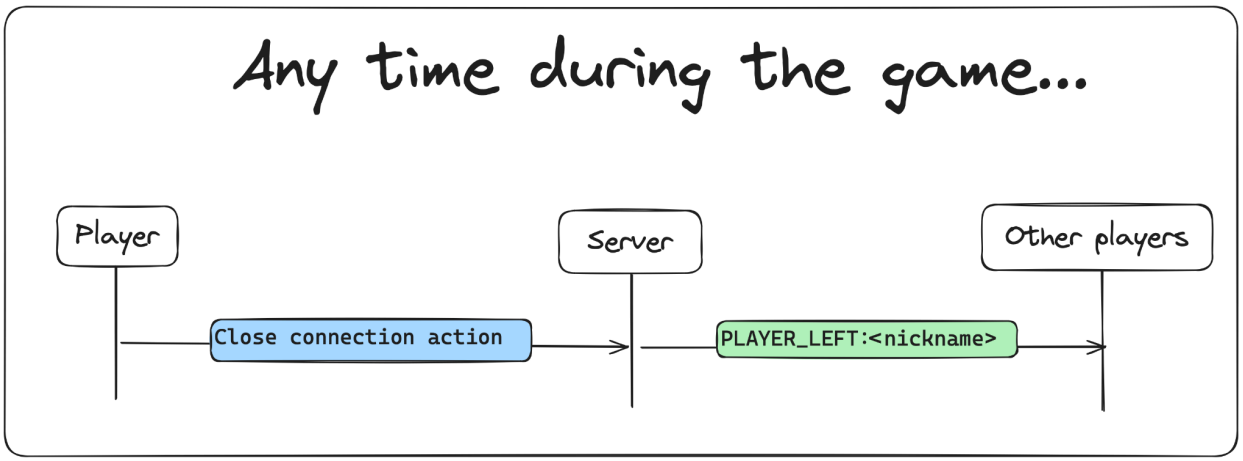
# VII. Phase 4: Game Over

## 1. Game flow

- On game over, the client will show **(1.) the top 5 leaderboard, and (2.) the winner's name.**
- Meanwhile, **the server will reset**. Essentially, this means that all players are kicked out of the game: **their connection to the server is closed**. To rejoin the game, they can click the "Play again" button, which re-register themselves.

# VIII. Common cases across phases

## 1. Game flow




- Any time in the game, if a player quits the game or loses connection, the server will **broadcast a** `PLAYER_LEFT` **message to inform other players**.

## 2. Message formats

- When a player disconnects from server at any point:

```
Server will BROADCAST to all players:
   PLAYER_LEFT;<nickname>
```

# IX. References


Pygame GUI: https://pygame-gui.readthedocs.io/en/v_069/
Asyncio server example: https://github.com/brandon-rhodes/fopnp/blob/m/py3/chapter07/srv_asyncio2.py