Haochi - Cook&Pass

Final Report for the Distributed Systems Course 2025/2026

Elisa Yan
elisa.yan@studio.unibo.it

Anna Malagoli
anna.malagoli2@studio.unibo.it

July 3, 2026

This project aims to implement a distributed real-time multiplayer game inspired by *Family Sytle*, where 2 to 8 players cooperate in a shared virtual kitchen to prepare recipes under time constraints. Players join a game session through a room system: one player creates a room and receives a unique code that others use to enter the same match and participate in a synchronized environment.

Disclaimer

The images used in the project are AI-generated.

1 Concept

The type of product developed with the project is an application with a graphical user interface (GUI) implemented using the Pygame library. The application follows a client-server architecture in which the server supports multiple game sessions running simultaneously and manages interactions among clients through message exchange. The server also manages a document-oriented database (MongoDB) containing recipe data. Since the application does not provide user authentication or session persistence, no personal data is stored in the database. However, the server keeps track of user actions by storing room codes and information related to the score achieved by each player so that it can be displayed at the end of the game.

Figure 1: Client-server architecture with MongoDB persistence

- **Use case description:**

  Users (clients) must use a computer connected to the same network as the server in order to access the application. Users interact with the system whenever they create a new game room, join an existing room, pass a dish to the kitchen, or pass an ingredient to a nearby player during a match. These interactions occur through the exchange of messages between clients and the server using the WebSockets library, which allows messages to be transmitted in JSON format.

  The system supports two different user roles: a standard player and a host. The host is the player who creates a new room and waits for other users to join, while standard players join an existing room and participate in the game.

- **Why is distribution needed?**

  The distributed nature of the application allows users located on different hosts to connect to the game server and play together.

  Resource sharing is achieved through a single centralized game state (the kitchen) shared among all connected clients. In particular, each player must wait for all other players to complete their recipes before everyone can progress to the next level. Furthermore, players actively share game resources by exchanging ingredients with one another.

  The system does not implement fault-tolerance mechanisms capable of replacing the server in the event of failures. Therefore, the server represents a single point of failure. If the server becomes unavailable, users remain unable to continue until the server is available again.

2 Requirements Elicitation and Analysis

2.1 Functional Requirements

- **Room Creation**: the system must allow a user to create a game room with a unique 4-character code and assigning the user as Host
- **Room Joining**: the system must allow users to join an active room by entering its corresponding 4-character alphanumeric code
- **Dynamic Seating Topology**: the system must allow the host to arrange and set the physical orientation of the players before the match begins

- **Ingredient Passage**: the system must allow players to slide ingredients off the left or right edges of their screens to transfer them to adjacent players
- **Recipe Composition**: the system must allow players to combine the correct ingredients on a virtual plate to complete their assigned recipes

2.2 Non-functional Requirements

- **Performance (Latency and Rendering)**: the client application must maintain a stable local rendering rate of 60 FPS for smooth physics and dragging animations
- **Consistency of the Game State**: the server must guarantee that all connected clients maintain an identical view of the game metrics, ensuring that level transitions and final scoreboards are consistent for every player
- **Game Availability**: gracefully handling player disconnections to prevent gameplay interruption

2.3 Implementation Requirements

- Application developed entirely in Python
- Usage of the Pygame library for implementing the view interface
- Database management using MongoDB
- Networking is handled via WebSockets
- The application must be containerized via Docker
- Automated integration tests with pytest

2.4 Relevant Distributed System Features

- **Failure Transparency / Fault Tolerance**: The system is transparent because it hides network problems from the players. If a player disconnects, the server handles it automatically in the background and reorganizes the player order. This allows the game to continue smoothly for everyone else without crashing. If the host player (the room creator) disconnects, the system does not stop. The server automatically and transparently elects a new host from the remaining players. If the server crashes, the current match is interrupted. However, the clients do not crash. Instead, players are safely redirected back to the main menu, and the client automatically starts trying to reconnect to the server every 3 seconds in the background. As soon as the server is back online, the connection is restored, and players can look for a new match.

- **Scalability**: Scalability is not a main focus of our project. We implemented a single, centralized server that handles all matches and connections. Because of this architecture, the server represents a bottleneck. If the number of users or requests grows too large, the server will not be able to handle the traffic and might crash.

- **Security (Role-Based Authorization / Room Access Control)**: Players are divided into Host (creates the room, starts the game, and sets the turn order) and Standard Player. Room access is secured via a shared secret (a unique room code). Standard players can only join a game session by entering the correct code.

- **Resource Sharing (Shared Game State / Turn Synchronization)**: All players in a room share and access the same consistent game state. Coordination is maintained as players must actively cooperate by exchanging ingredients during the match.
  Strict synchronization is required at the end of each round. Players who complete their recipes early cannot advance until all other players finish. During this waiting period, they must remain active to continue exchanging ingredients with teammates.

- **Interoperability, Heterogeneity of Components**: The system support multiple platforms (Windows and macOS) and integrates a Python server with a MongoDB database to synchronize the game state.

- **Performance and Concurrency**: By using WebSockets, the system ensures low-latency communication and minimal bandwidth usage, keeping response times fast even with many concurrent operations.
  The server efficiently processes parallel messages sent by multiple players simultaneously, whether they are in the same game room or across different ongoing matches.

3 Design

Haochi-Cook&Pass adopts a classic client-server architecture to ensure scalability, synchronization and fairness throughout the gameplay. Players interact through a graphical user interface (GUI) built in Pygame. The client's application communicates with a centralized server that handles authorization, room coordination, game progression and result evaluation. The backend is logically divided into two primary modules: one that governs game logic, handling player actions is implemented in both the client and the the server, and another responsible for managing game data, such as match statistics and room configurations, directly within the server's memory. The server supports multiplayer modes, dynamically assigning recipes, ingredients to players and passing ingredients between players. Real-time communication is facilitated through WebSockets allowing clients and server to send and receive messages after they have established a connection. To improve reliability, fault tolerance is built into the system: players can

disconnect without disrupting the current game state because the server dynamically reorders the remaining players. If the server goes down, all clients will pause the game and continuously check if the server has come back online.
This modular design ensures clear separation of concerns—frontend focuses on rendering and user interaction, while the backend enforces core mechanics and ensures synchronized state across clients. This architecture delivers a consistent, responsive, and fair game experience. The structure of the system is illustrated in Figure 2 and 3.

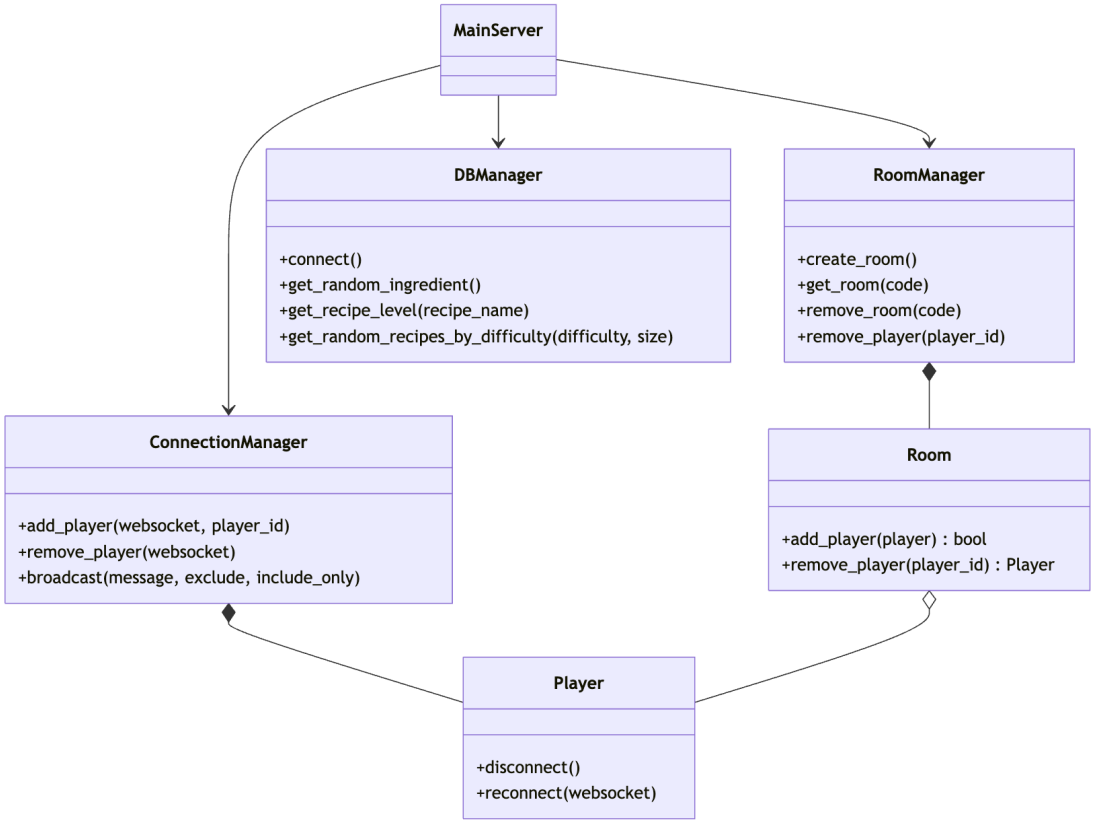

Figure 2: UML class diagram of the server architecture and its core components

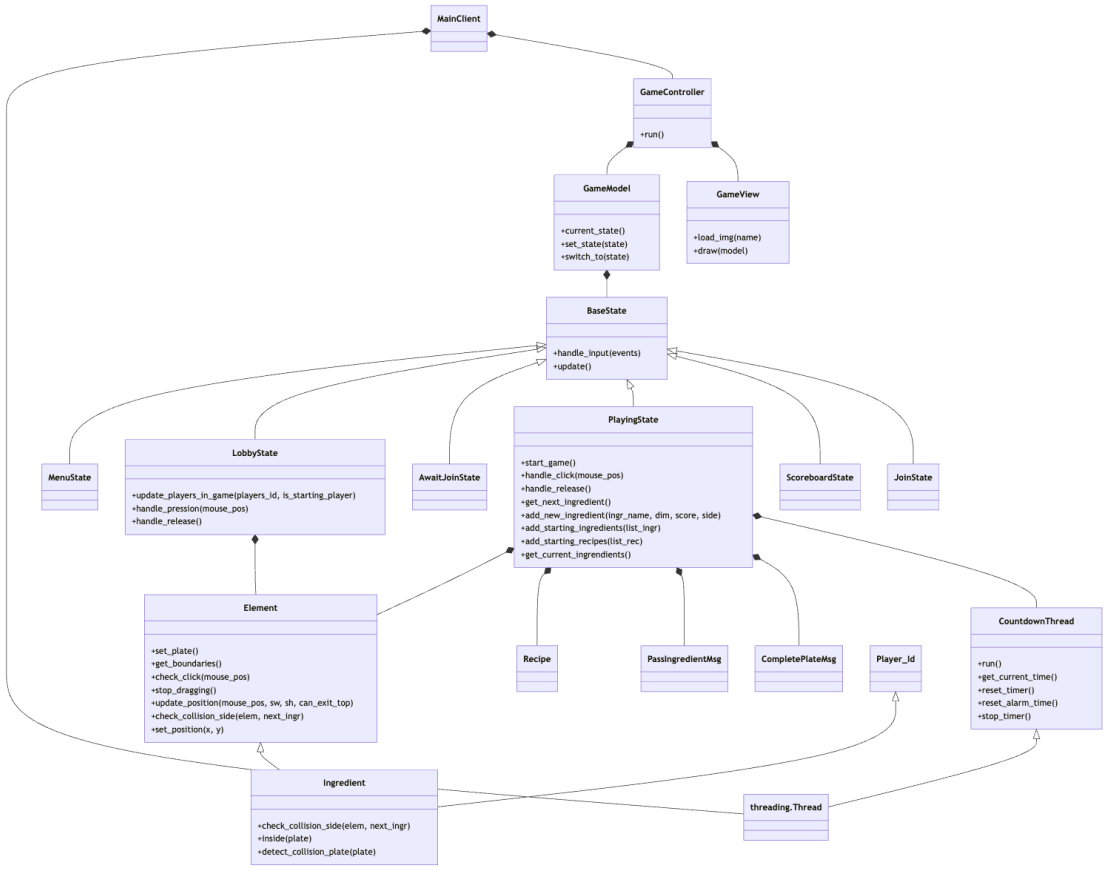

Figure 3: UML class diagram: MVC pattern and client state management

3.1 Architecture

Haochi-Cook&Pass adopts a centralized client-server architecture, which was selected to ensure strong consistency, controlled synchronization, and simplified game state management across multiple players. This architectural choice aligns with the core requirements of the game: real-time interaction, fairness in gameplay, and robustness in multiplayer coordination. By centralizing interaction between clients and data management on the server side, the system guaranties a single source of truth that all clients rely on that is however a single point of failure (if the server crashes, the whole game stops) and the bottleneck of the system (if too many players are connected, the server can slow down).

The architecture consists of two main entities: the client, built using Pygame as a graphical front-end, and the server, implemented in Python and exposed through WebSockets. Clients handle user interactions such as room creation or joining, and gameplay actions (e.g. pass ingredient and complete recipe). These actions are sent to the server, which validates and processes them. The server is logically divided into two parts: one

managing dynamic game state by getting messages sent from the clients and another maintaining persistent scores and other information about the users. This architecture supports multiplayer mode where disconnected players can be removed from the game without disrupting the match state. In short, we chose a centralized client-server model to ensure fairness and security. These are essential for an interactive multiplayer game like ours.

3.2 Infrastructure

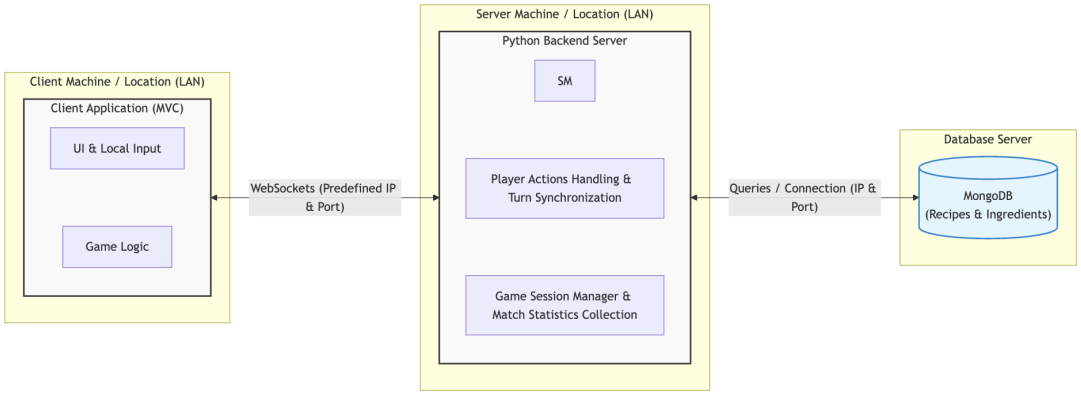

Figure 4: Component diagram showing the internal functionalities of the components

The infrastructure of system is structured around a centralized client-server model, where multiple clients interact with a single authoritative server using WebSockets. This architecture allows the server to centrally manage game sessions and gameplay synchronization while providing clients with real-time feedback and updates. The main infrastructural components include:

- The client applications are built following the MVC (Model-View-Controller) pattern.
- The MongoDB database that collects all the recipes and ingredients.
- The server application interacts with the document database and is composed only of the backend implemented in Python.

During deployment, the client and server components can either reside on the same physical machine or be distributed across different locations. In both scenarios, they communicate over a local network (LAN) using predefined IP addresses and ports.
Each component is responsible for a clearly defined set of tasks. The client handles the user interface, local input, and game logic, while the server manages player state transitions (e.g., lobby, joining, and playing), turn synchronization, and final score calculation. The system relies on asynchronous, event-driven communication, which maintains

a smooth and responsive user experience even under high network latency. All critical events, such as a round termination, joining and ingredient passing, are propagated to all relevant clients to maintain state consistency.

3.3 Modelling

The Haochi-Cook&Pass system models several domain entities, each representing fundamental components of the game play and of user interaction. The primary entities includes:

- Player: each player is modeled as a `Player` entity. Upon connection, the server assigns a unique ID to each player. This entity tracks the player's WebSocket connection (used for transmitting and receiving messages), its current connection state (connected or disconnected), and the room code if the player is currently in a game. Additionally, it maintains the player's specific position in the turn rotation and the real-time data required to calculate final match statistics.
- Room: each game match is represented as a `Room` entity. This room keeps track of the unique room code created at the beginning, the ID of the host player who created the room, a collection of all players who joined the room, the current level of the game, the room status, which can be: INIT, READY, IN_GAME, or OVER.
- RoomManager: manages all game rooms. It includes functionalities to create and remove a room, assign a unique code to it, and look up a specific room using its code. Additionally, it handles removing a player from a room.
- ConnectionManager: keeps track of all the players that are connected to the application.
- Element: each game item is modeled as an `Element`, such as a plate or an ingredient. It includes the rendering data, the current position on the user interface, the movement speed, and a flag indicating whether it is currently being moved by the player.
- Ingredient: Unlike a simple `Element` (such as a plate), an ingredient has also a name and must track whether it has been placed on the plate or not.
- Recipe: the `Recipe` entity models the recipes assigned by the server. It contains a list of ingredients, the score the player will receive upon delivering the recipe to the kitchen, and the time limit for its completion.
- CountdownThread: this entity represents the recipe timer. If the player does not complete the recipe within the time limit, the timer expires, and the unfinished recipe is automatically removed.

These domain entities are mapped to infrastuctural components as follows:

- The server maintains an instance of both the `ConnectionManager` and `RoomManager` to track the complete state of all players and rooms.
- Clients maintain temporary views of different game states (such as the menu, lobby, and active gameplay), which are rendered via the GUI, and communicate game events to the server through the established WebSocket connection.
- Game-related events (such as joining a room or passing ingredients) are handled server-side and broadcasted only to the specific players affected by the change.

The system tracks several important domain events:

- A player creates or joins a game room.
- A list of recipes is assigned to all players in the room for every round after the game starts.
- A player can pass ingredients to neighboring players or complete a recipe.
- A player disconnects during the match.

The messages exchanged includes:

- Commands like change model state, update current players, start game, join game, pass ingredient and plate complete.
- Notifications that are broadcasted to clients, like room created, room closed, starting recipes, starting ingredients and new ingredient.

The system state includes active game rooms, players' match participation, and live metrics used to compute final match statistics. All shared state is maintained server-side for consistency and, when needed, synchronized with clients to reflect the current game status.

3.4 Interaction

The interaction flow in the system is based on a centralized client-server model where all communications and state transitions are mediated through the authoritative server. The interaction begins with the room creation phase, where a player (Client 1) sends a request to create a new game room. Once the room is initialized, other players (such as Client 2) can send a request to join the existing room, by providing the correct unique code of the room they are willing to join to the server. The server dynamically manages the lobby state and broadcasts updates to the game's host (Client 1) to ensure that he has a synchronized view of the current players that are waiting for the match.
The core game play phase is initiated when Client 1 sends a "start game" command to the server. The execution loop then shifts into a round-based structure. At the beginning of each round, the server distributes the list of required recipes and ingredients to all clients in the room. During game play, clients interact asynchronously through a

neighbor-based passing mechanism: a player can pass an ingredient to their immediate left or right neighbor by specifying the intended direction (left or right) in the request sent to the server. The server then processes this event and immediately delivers the ingredient to the designated recipient, explicitly notifying them whether the ingredient is arriving from their left or right neighbor.
When a player successfully fulfills the requirements, they send a "complete recipe" request to the server. The server then validates the action, updates that specific player's score locally, and transitions the client into a waiting state for the termination of the turn. Players don't have to stop during the waiting state. They can stay active and keep passing ingredients to their teammates until the round ends.
Once all players have completed their recipes and the round sequence ends, the server triggers the end-game routine. It calculates the final standings, aggregates the performance statistics, and broadcasts the definitive scores to all clients simultaneously, concluding the session lifecycle.
The interaction sequence is visually summarized in Figure 6, which illustrates the step-by-step exchange of messages, lifecycle activations, and asynchronous events between the distributed clients and the central server.

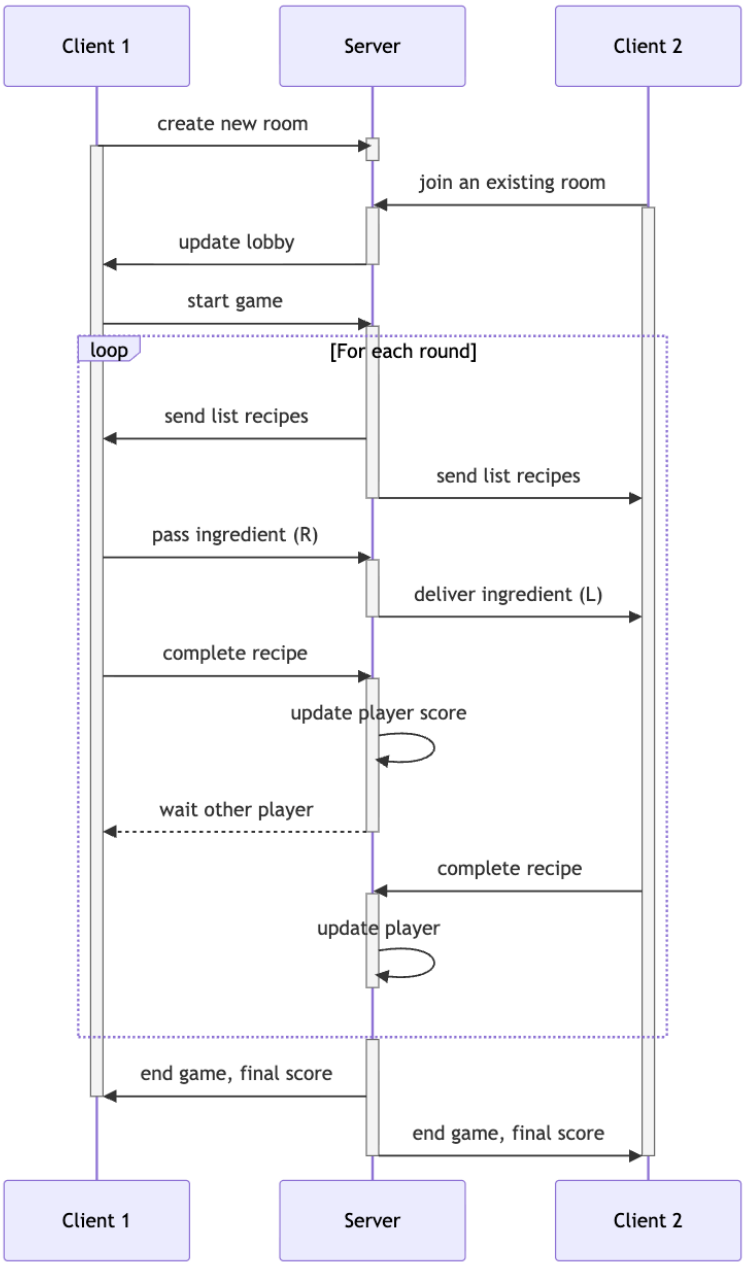

Figure 5: Sequence diagram that shows a simple example of Client-Server interaction

3.5 Behaviour

Component Behaviour

The system is divided into components that react dynamically to network events and user inputs, split between stateful and stateless behaviors:

- **The Server (Stateful)**: The server-side architecture is strictly stateful. It maintains a persistent and global view of the entire game ecosystem. Through the ConnectionManager and RoomManager, the server tracks active player connections, session states, and room lifecycles. It continuously processes incoming client commands and evaluates game logic (such as scoring and players reordering after a disconnection).
- **The Clients (View-Driven with Local Game Logic)**: The clients act as view-driven interfaces that render the game states via the GUI. However, they also embed and manage specific game rules locally. For example, the client-side interface tracks the current plate and the next required ingredient, restricting the player from placing any incorrect items. The client handles this verification entirely on its own, ensuring that only valid actions are captured, translated into command messages, and forwarded to the server over the WebSocket connection.

State Updates
The server is the sole authority in charge of updating the system state. Clients never mutate the shared state directly; they only request modifications.

- **Who updates the state**: The server-side asynchronous functions. Every time a new message is received, it triggers the specific asynchronous function associated with the action defined inside that message.
- **When it updates**: Updates occur deterministically in response to two types of triggers:
  - **Client-Driven Events (Commands)**: When a player triggers an action, such as joining a room, passing an ingredient, or submitting a completed dish.
  - **Time-Driven Events (Internal Timers)**: When a recipe's countdown timer expires before the player can complete it.
- **How it updates and synchronizes**: When an event is received, the server validates the action, updates its internal state, and synchronizes the change to clients. State updates are not blindly broadcasted to everyone. In fact, the server targets only the relevant clients affected by the modification (e.g. notifying the room host via update_current_players when a new user joins, or sending change_model_state command specifically to players inside that active match).

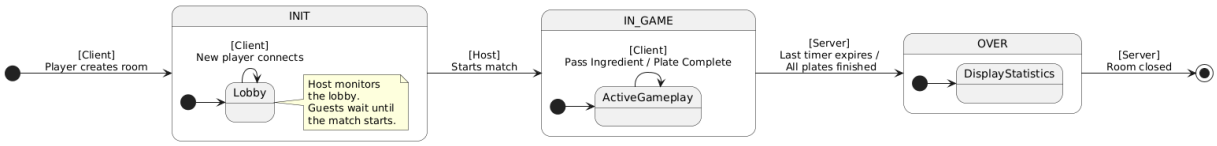

Figure 6: State diagram showing the different client states that can be updated in response to client or server actions.

3.6 Data and Consistency Issues

The data that needs to be persisted concerns the game content, not the players' state. Specifically, we store ingredients (the basic building blocks) and recipes, where each recipe is defined by the list of ingredients that compose it, its difficulty level, and the preparation time associated with it.

This data is stored because it represents the static catalog of content used by the server to generate game levels: it must be available before a match starts and must persist across server restarts.

Conversely, player data does not need to be persisted: the game is designed as a "pick-up-and-play" experience, where a player can join and start playing as soon as the match begins, with no account, profile, or progress to save between sessions. Player-related state (score, current ingredients, position in the level, etc.) exists only transiently in memory for the duration of the match.

- Persistent data (ingredients and recipes) is stored using **MongoDB**, a document-oriented database.
- This choice is motivated by the nature of the data: a recipe is naturally represented as a document containing a variable-length list of ingredients plus a few scalar attributes (difficulty, preparation time). A document store avoids the overhead of relational joins for what is essentially static, read-mostly catalog data, while offering schema flexibility (e.g. recipes with different numbers of ingredients) and fast retrieval by attributes such as difficulty level.

The server is the only component that interacts directly with the database. Queries are perfomed when a new level needs to be generated, i.e. at the start match or when a level transition occurs.

The data are shared between components once the server retrieves a recipe from the database, this information must be shared with all the clients participating in the same match. This information is needed by clients to render the current level correctly, to know which ingredients the players must collect and combine, and to synchronize the countdown timer with the server.

3.7 Fault-Tolerance

- **Heart-Beating and Timeout Mechanism**

  The connection between the server and the clients is monitored using the heart-beating mechanism provided by the `websockets` library. The connection established between the client and the server is continuously monitored to detect possible disconnections. The mechanism is based on three main parameters:

  - **Ping interval**: Every 20 seconds, the library automatically sends a Ping message to the other endpoint.

- **Ping timeout**: After sending the Ping message, the library starts a timer. If the other endpoint does not respond with a Pong message within 20 seconds, the connection is considered lost.
- **Pong transmission**: The Pong message is sent immediately upon receiving a Ping message and is handled automatically by the library in the background.

- **Error Handling**

  If a client disconnects during a game, the server detects the failure through the heart-beating mechanism provided by the `websockets` library. If at least two players remain in the game, the server removes the disconnected player and updates the player order accordingly. Specifically, the departure of a player causes the participant on their left (A) and the participant on their right (B) to exchange ingredients directly with each other, bypassing the disconnected player.

3.8 Availability

In the event of a network partition or the disconnection of a system component, the system behaves as follows:

- **Standard Player Disconnection**: The server detects the connection loss (e.g., through the heartbeat timeout) and removes the player from the room. The game continues normally with the remaining players.
- **Host Disconnection**: The server applies a fault-tolerance mechanism by automatically electing a new Host from the remaining players in the room. This guarantees the continuity of the game without any loss of game state.
- **Server Unreachable**: The clients detect that communication with the central server is no longer possible, preventing players from sending completed dishes to the kitchen or exchanging ingredients. In this case, the game is immediately stopped and all players are redirected to the game menu. This behavior follows the constraints of the CAP theorem by prioritizing consistency of the game state over availability, stopping the game rather than displaying or processing unsynchronized data. Once the clients have closed the WebSocket connection with the server, they attempt every 3 seconds to verify whether the server is reachable again and establish a new connection.

3.9 Security

- **Authentication and Authorization:** No authentication mechanism has been implemented, as players are not required to provide any credentials to access the game. Instead, the server relies on a lightweight *session identification* mechanism: when a client connects via WebSocket, the server internally generates and assigns a `player_id`, used only to distinguish one WebSocket connection from another for the duration of the match. Once a player creates or joins a room, an `ingr_id` is

additionally assigned at random from the database, acting as the player's in-game identity (displayed as an ingredient name). Neither identifier is provided or proven by the client, so neither serves an authentication purpose; they are only used to identify an active session, which is consistent with the fact that no player account or persistent profile exists across matches.

But players can have two distinct **roles** within the game server, depending on the phase of the game.

- **Host**: as soon as a player creates a new room, they are automatically elected as the host of that room. The host is the only player who can start the match, and only once at least two players have joined the room. If the host leaves the room before the match starts, the room is closed and all the other players who had joined are notified.
- **Standard player**: once the match has started, all players share the same role and have the same set of available actions, with no distinction between host and the other participants. If any player leaves during the match, the ingredients that were assigned to them are redistributed evenly among the remaining players in the room, in order to preserve the continuity of the game. If, after one or more players leave, only one player remains in the room, the match ends early.

The server enforces **access control** when entering game rooms, based on a shared secret, namely the unique code associated with each room. A standard player can join a given match only by entering the correct identification code for that room.

4 Implementation

- Which **network protocols** to use?
  The communication protocol relies on asynchronous, bidirectional message passing over **WebSockets**. Utilizing Python's asyncio and websockets libraries, the client manages communication in the background via dedicated input (`msg_queue`) and output (`send_queue`) queues.

- How should *in-transit data* be **represented**?
  - In-transit data is represented using the **JSON (JavaScript Object Notation)** format, which provides a lightweight, human-readable, and structured way to exchange information.
  - **Serialization Process**: Regardless of whether a message originates from the client or the server, all outgoing data is initially structured as a standard Python dictionary (a collection of key-value pairs representing application attributes, commands, or states). Immediately before transmission over the WebSocket connection, these dictionaries are serialized into raw text strings using Python's native `json.dumps()` function.

  - **Deserialization Process**: Upon receiving a raw text payload, the receiver (either the client or the server) decodes the JSON string back into a native Python dictionary using `json.loads()`. This structure is protected by explicit exception handling (`json.JSONDecodeError`) to isolate malformed packets.

- How should databases be queried?
  - The system implements a NoSQL database approach using MongoDB (hosted in cloud via MongoDB Atlas) and queried via the pymongo Python library. A NoSQL, document-oriented paradigm was selected over traditional SQL due to the highly flexible and hierarchical nature of the game data (ingredients and recipes).
  - **Data Organization**: Data is stored inside lightweight BSON/JSON-like documents within two main collections: `ingredients` and `recipes`. Recipes natively embed lists of strings (e.g., `"ingredients": ["bread", "avocado"]`), mapping perfectly to Python lists without requiring heavy relational JOIN operations or junction tables.
  - **Querying Mechanisms**: Database interactions are encapsulated within a dedicated `DBManager` class, which abstracts the database layer from the core server logic:
    * **Basic Queries**: Standard lookups use structured document filtering, such as `find_one({"name": recipe_name})` to quickly extract a document's attributes.
    * **Aggregation Pipelines**: Advanced queries leverage MongoDB's aggregation framework. For example, selecting random gameplay elements dynamically filters matching documents and extracts a statistical sample directly on the cluster side using stages like `$match` and `$sample`:

```
pipeline = [
    {"$match": {"difficulty": difficulty}},
    {"$sample": {"size": size}},
    {"$project": {"_id": 0}}
]
```

  - **Data Seeding and Lifecycle**: Database initialization is handled programmatically via a standalone script using `insert_many()` operations, allowing bulk insertion of entire arrays of raw Python dictionaries during development or initial system deployment.

- How should components be *authenticated*?
  - **Session-Based Identification**: The system uses a simple, session-based identification system instead of complex authentication protocols like OAuth

or JWT. Because the game does not need permanent player accounts or persistent profiles, players do not need to register or log in. Instead, they can connect and start playing immediately.

  - **Dynamic ID Generation**: Upon establishing a new WebSocket connection, the server's `register_client` routine automatically generates a unique identifier for the player using a truncated Version 4 Universally Unique Identifier (`str(uuid.uuid4())[:8]`).
  - **Connection Tracking**: This dynamic `player_id` is immediately bound to the active network socket and tracked in memory by a centralized `ConnectionManager`. This ensures that every subsequent in-transit JSON payload is automatically tied to a verified network identity without forcing the client to continuously re-authenticate or attach tokens to every message.

- How should components be *authorized*?
  Game session management is based on an Authoritative Server architecture. In fact, only the server validates and synchronizes the state among the clients:
  - **Action Validation**: The server verifies the legitimacy of each request (e.g., checking the maximum limit of 8 players before accepting a join request).
  - **Host Management**: When a player creates a game, the server registers it internally as the `host_id` and sends a command to the client to render the main lobby view. Furthermore, the host is the only player who receives real-time updates whenever a new participant connects, and it is the only one authorized to start the match (which is permitted only after at least one other player has joined).
  - **Standard Player Management (Join)**: For players who join a room, the server simply updates the `players` dictionary. On the client side, this triggers an automatic state transition to the waiting view (*join view*).

4.1 Technological details

The project exploits several key technologies and frameworks to achieve its functionality:

- **Pygame**:
  * Used for rendering the game's graphical user interface (GUI) and hadling user input.
  * In the `templates`' files Pygame is used to create the application windows, display text and buttons, and render ingredients images.
  * The `GameView` class in `view.py` defines the common properties of the graphical user interface, such as the window size, fonts, and background images. It contains a `draw` method that is called by the client controller at each iteration of the game loop. Inside this method, the system decides which specific window template to render based on the current state of the model. The available states managed by this view are:

- **MENU**: Displays the main homepage and sub-menus (Figure 7).
- **LOBBY**: Shows the waiting room for the host before the game starts (Figures 11-12).
- **JOIN_INPUT**: The screen where a player types the room code to join.
- **AWAIT_JOIN**: A waiting screen for guest players while connecting to a game room (Figure 10).
- **PLAYING**: The core gameplay screen where ingredients and recipes are rendered (Figures 13-14).
- **SCORE**: The final scoreboard displaying the player's and team scores and statistics (Figure 15).

- **WebSocket Communication**:
  * Enables full-duplex, bi-directional, and real-time communication between the client and the server using the WebSocket protocol over TCP.
  * In the client-side component, the `websocket_client` function within `connection.py` leverages Python's `websockets` and `asyncio` modules to establish persistent network connections, handling concurrent message transmission through distinct send and receive queues while managing automatic reconnection logic.
  * On the server side, the `main_server.py` script initializes and hosts the WebSocket server on port `8765`, registering newly connected clients and continuously listening for incoming messages. When a JSON message is received, the server extracts the `"action"` field from the data payload; this key is then used to look up the corresponding function within the `ACTION_HANDLERS` routing dictionary defined in `controller_server.py` (such as `handle_start_game` or `handle_join_room`), automatically executing the appropriate server-side logic required to process that specific request.
  * Connection lifecycle and messaging routines are managed by the `ConnectionManager` class in `connection_manager.py`. This component tracks active connections and coordinates asynchronous message broadcasting, allowing the system to transmit updates to all connected players, restrict distribution to specific target sockets, or explicitly exclude certain players.

- **Concurrency and Multithreading:**
  * On the client side, multithreading is used to isolate the main controller's game loop from the full-duplex network connection. By executing the network routine in a separate background thread, the client can smoothly process incoming WebSocket messages and transmit data in real-time without blocking Pygame's rendering cycles or causing frame drops.
  * Additionally, the client spawns a dedicated daemon thread at each game level to act as a timer. This thread independently monitors the elapsed time,

ensuring accurate tracking of the duration spent by players preparing recipes without interfering with user inputs or graphical updates.

  * On the server side, explicit OS-level threads are avoided in favor of an asynchronous, event-driven architecture. Using Python's `asyncio` loop within `main_server.py`, the server handles hundreds of concurrent player connections inside a single process, leveraging lightweight coroutines to achieve cooperative concurrency instead of traditional multithreading.

- **State Management**:
  * The `GameModel` class in `model.py` represents the global state of the client application, acting as the centralized data container for critical context details such as the `game_code` and `player_id`.
  * This class manages the application's current behavior by implementing a State Pattern. It uses the `self.states` dictionary to map text keys to specific state classes (such as `MenuState` or `PlayingState`) that extend a common `BaseState`. Each class independently manages its own user inputs via `handle_input` and data updates via the `update` method.
  * Through the `current_state` property and dynamic state-switching methods like `switch_to`, the model safely encapsulates state-specific behaviors, exposing only the active game context to the controller and the rendering views at any given time.

5 Validation

5.1 Automatic Testing

The project includes some automated tests implemented with `pytest` and the `unittest.mock` library. Since the project was developed in a limited amount of time, the tests do not cover the whole application but focus on the most important parts of the system, especially the server logic, room management, and some client-side utilities.

5.1.1 Unit testing

Unit tests were implemented for both server-side and client-side components.

Client side The tests mainly cover the utility classes used during the game:

- `Element`: checks that graphical boundaries are computed correctly, that mouse clicks are detected correctly whether they happen inside or outside the element, and that dragging can be stopped correctly.
- `Ingredient`: verifies that the ingredient name is correctly obtained from the image filename, and that the program can detect whether an ingredient is placed inside a plate.

- **Client-created messages**: `PassIngredientMsg` is tested to verify that the message contains the correct action, ingredient name, direction and score, while `CompletePlateMsg` is tested to check that the completed ingredients, the score and the `finished_all_plates` flag are correctly included in the message sent to the server.

Server side The tests focus on the `Room` and `RoomManager` classes:

- `Room`: checks that the first player added to a room becomes the host, that a room cannot contain more than eight players, and that the room state changes from `INIT` to `READY` when there are enough players. Other tests verify that the room state is not changed incorrectly after the game has already started, that player positions are updated when a player leaves, and that the correct neighbour is selected when an ingredient is passed to the left or to the right.
- `RoomManager`: checks that new rooms can be created correctly, that room codes are unique, and that rooms are removed when needed. It also verifies that when a room is removed, the players inside it are reset, and that the room is closed if the host leaves.

These tests are automated by checking the internal state of the objects with assertions. When external components are needed, such as WebSocket connections, they are replaced with mocks. In this way, the tests can be executed without starting the full application.

5.1.2 Integration and communication testing

Besides testing the individual classes, we also wrote tests to check that the different server components work well together, in particular `Room`, `Player` and `RoomManager`, through the message handlers (`test_controller_server.py`). The real WebSocket connections are replaced with `AsyncMock` objects, so we can check exactly which JSON messages get sent, to whom, and in what order, without actually having to start the server.

Room setup and level start

- `handle_join_room`: tested both for an invalid room code, where the server should respond with an `ERROR` message, and for a successful join, where the new player receives a `CHANGE_MODEL_STATE` message while the host is separately notified with an `UPDATE_CURRENT_PLAYERS` message containing the updated list of players.
- `handle_start_level`: verifies that each player receives both the assigned recipes and the corresponding ingredients, and that the total number of distributed ingredients matches exactly what's needed to complete the recipes.

Core gameplay loop

- `handle_pass_ingredient`: checks that when a player passes an ingredient in a given direction, the correct neighbour (computed with `Room.get_near_player`) receives it through a `NEW_INGREDIENT` message with the direction flipped.

- `handle_plate_complete`: verifies that the player's score and completed-plate count are correctly updated on the server side.

Even though these tests don't use real network connections, they still give us good confidence that the communication protocol between components works as intended, since they cover the whole chain: from the incoming message of the client, through the room and player state changes, all the way to the outgoing messages sent to the right recipients.

5.1.3 End-to-end testing

No automated end-to-end tests were implemented. The complete gameplay flow was tested manually by running the server and multiple clients. During these manual tests, we checked that a player could create a room, that other players could join it, that the host could start the game, and that the players could receive recipes and ingredients. We also tested the main gameplay actions, such as passing ingredients, completing plates, updating the score and continuing the game flow.

End-to-end tests were not automated because the game depends a lot on real-time multiplayer interaction and on graphical actions such as drag-and-drop. These aspects are harder to test automatically in a meaningful way, so we decided to validate them manually during development.

5.1.4 Test automation and execution

All automated tests are implemented using pytest and can be executed from the project root with:

```
pytest
```

The tests are executed automatically without requiring a running server because all external dependencies are replaced by mocks.

5.2 Acceptance test

Manual testing was performed throughout development, in addition to the automated unit test suites. After implementing each new feature, the game was manually played end-to-end by the team in order to verify that the feature behaved as expected within the actual gameplay flow. A feature was considered accepted once its observed behavior during manual play matched the corresponding functional requirement.

This manual playtesting was not automated because it relies on direct human interaction with the game's real-time, multiplayer flow (player coordination, timing of ingredient passing, UI feedback, and overall game feel), aspects that are difficult to meaningfully verify through scripted tests and are better assessed through direct play.

6 Deployment

To deploy the Haochi-Cook&Pass game system from scratch, the following steps are required. The system is composed of a server implemented in Python and a Pygame-based client that not only renders the user interface but also implements core game logic and real-time physics locally. The communication between client and server relies on WebSockets.

Instructions

1. Clone the repository:

   ```
   git clone https://github.com/elisayan/haochi-cook-and-pass.git
   cd haochi-cook-and-pass
   ```

2. Create and activate a virtual environment (optional but recommended):

   ```
   python -m venv .venv
   source .venv/bin/activate # Linux/Mac
   .venv\Scripts\activate # Windows
   ```

3. Install dependencies:

   ```
   pip install -r requirements.txt
   ```

   (This should include packages like pygame and others used in the codebase.)

4. Run the server:

   ```
   # To run the server directly on the operating system:
   python -m haochi-cook-and-pass.server.src.main.main_server

   # To run the server inside a Docker container:
   docker compose up --build
   ```

   This will start the backend server on localhost:8765, and initialize the MongoDB database.

5. Run the client application. By default, a connection to a local server (`localhost`) is established. If the server is running on a different machine within the same LAN, the server's IP address can optionally be appended as a command-line argument:

   ```
   # To connect to a local server (default):
   python -m haochi-cook-and-pass.client.src.main.main_client

   # To connect to a remote server:
   python -m haochi-cook-and-pass.client.src.main.main_client <server_ip>
   ```

Executing this command launches the Pygame-based application, allowing players to create or join rooms and interact with the server to play the game.

Expected Outcomes
After successful deployment:

- The server will be running and waiting for incoming client connections.
- Players will be able to create or join rooms and to play through the Pygame GUI.
- The UI will dynamically update based on server responses and player's actions.
- Game logic including turn-based rules and passing of ingredients will function as intended.

6.1 Software Dependencies

The required dependencies and their specific versions are listed in the `requirements.txt` file that include:

- `websockets==12.0`
- `pygame==2.6.1`
- `pymongo[srv]==3.12`
- `numpy==2.2.3`
- `pytest==8.3.5`

6.2 Containerization

For the deployment of this project, containerization is used exclusively for the server side. As shown in the `docker-compose.yml` file, the configuration builds and runs only the server component, exposing port 8765. The client is intentionally excluded from the Docker container because it relies on `pygame` for its graphical user interface (GUI). Since Docker containers do not natively support GUI rendering required by `pygame`, the client must be executed directly on the host machine's operating system rather than inside a container.

7 User Guide

The Haochi-Cook&Pass game provides a simple and intuitive user interface that allows players to create or join rooms, and participate in the gameplay. The following guide explains how to use the application.

Instructions

1. **Launch the client interface**: After running the client, the user enters a main menu where they can choose to start playing, view the tutorial, or exit the application. (interface image 7).
2. **Tutorial of the game**: Before starting the game, players can view the tutorial page to learn how to play (interface image 8).
3. **Mode selection**: Players can choose between (inteface image 9):
   - **Create a Room** Initializes a new game session.
   - **Join a Room**: Joins an existing game session by entering the room unique code.
4. **Lobby Interface**: Once a new game session is created, the UI displays:
   - The player's unique ingredient ID within the game session (interface images 10).
   - **Host-Specific Setup**: Only the host can view and configure the player turn order. The host's screen displays a list of ingredients (each representing a player's ID) on the left, and a circular area with plates on the right. The host must place one ingredient onto each plate to define the playing order (interfcae image 11-12).
5. **Game Interface**: Once in-game, the UI displays (interface image 13):
   - A plate.
   - A timer that shows the remaining time to complete the recipe.
   - The next ingredient that needs to be added to the current recipe.
   - The player's available ingredients.
   - The player's score.
6. **Gameplay**:
   - **Complete Recipes and Placement Rules**: Players must place all the required ingredients into their plate to complete the recipe (interface image 13). If a player tries to place an ingredient that is not next in the recipe order, a red "X" appears over the plate, and the action is blocked (interface image 14).
   - **Share Ingredients**: Players can pass ingredients to neighbors on their left or right (interface image 14). Clicking on an ingredient that is not currently needed for their own plate will display left and right arrows to indicate the directions to pass it.
   - **Submit Dishes**: Once a recipe is fully completed, an upward arrow is rendered on the screen to show where to pass the finished plate to the kitchen.

7. **Final Score Interface**: Once the match ends, the UI displays (interface image 15):
    - The individual player's final score.
    - The total number of ingredients passed and dishes completed by the single player.
    - The collective score achieved by the entire group.
    - The total number of completed dishes during the single game session.

Expected Outcomes

Figure 7: Main interface displayed when the game starts

Figure 8: Tutorial page explaining the game rules and mechanics

Figure 9: Choice interface

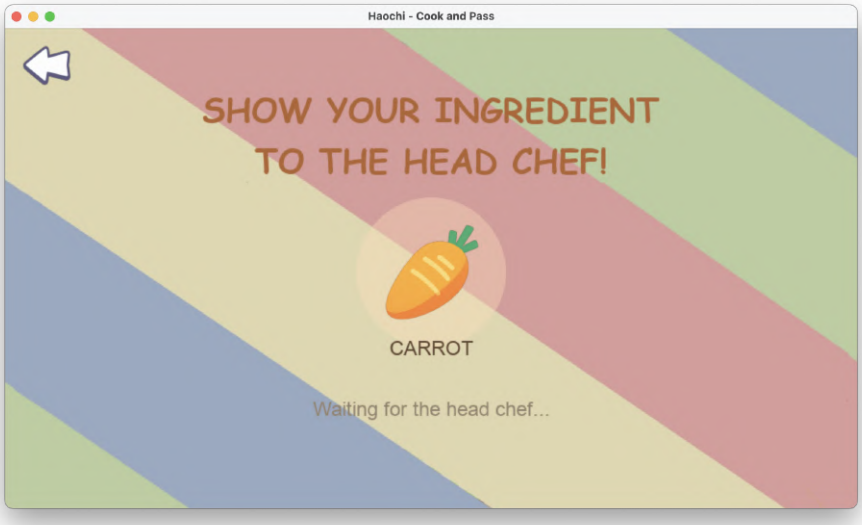

Figure 10: Join interface

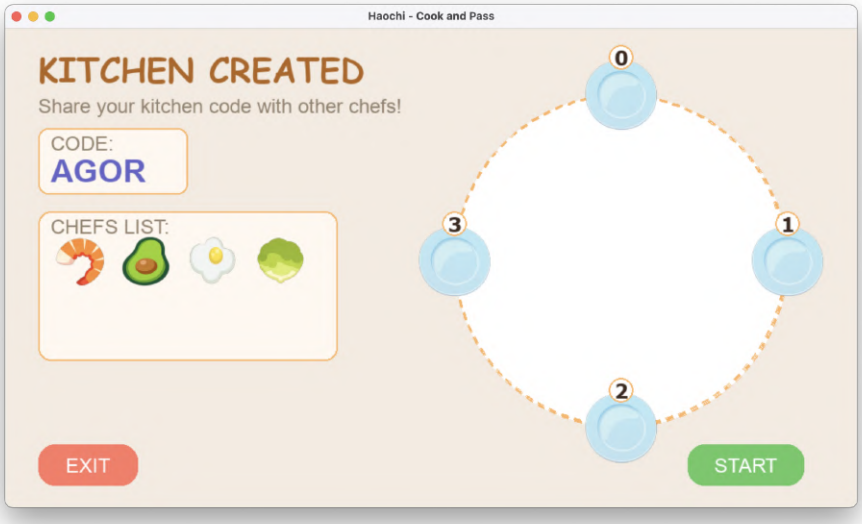

Figure 11: Lobby interface with player list

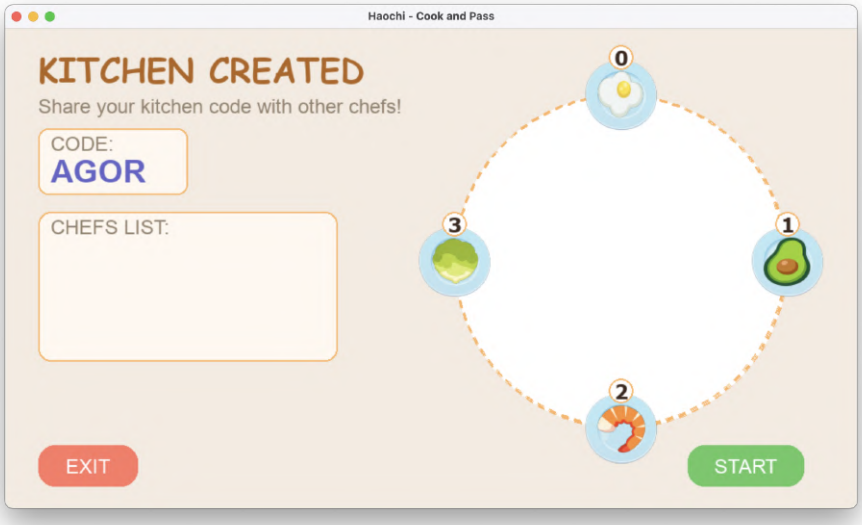

Figure 12: Lobby page after all players are ready

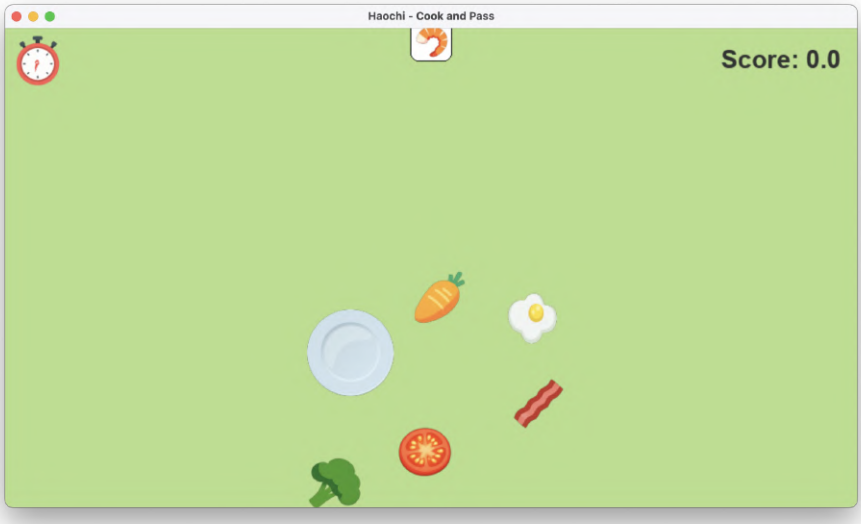

Figure 13: In-game interface with plate, timer, next ingredient of the current recipe, player's available ingredients and score

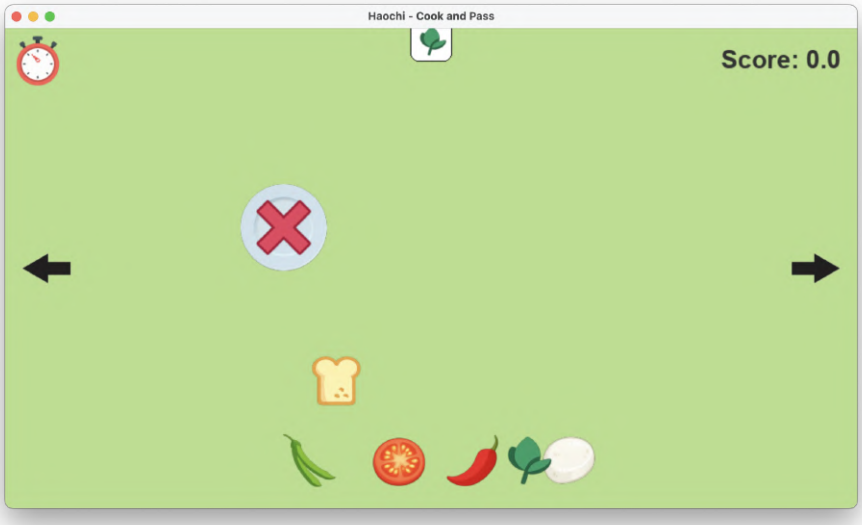

Figure 14: In-game interface when selecting an incorrect recipe ingredient

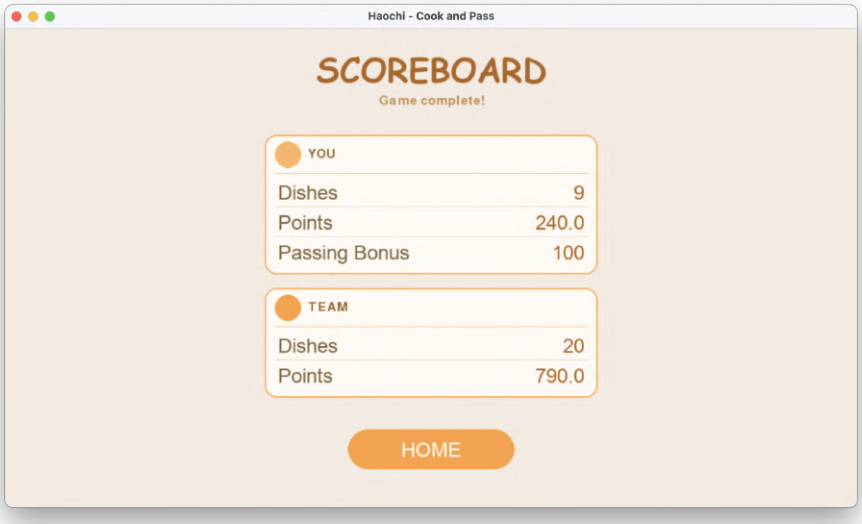

Figure 15: Score interface showing the final score of the player and the team

8 Self-evaluation

8.1 Elisa Yan

During the project, I mainly worked on the server-side logic and on improving parts of the UI. In particular, I focused on the main game states, such as join, start, lobby and scoreboard, trying to make the flow between the different phases as clear and smooth as possible. I think one of the best results of the project is that the game actually feels like a multiplayer game: players can enter a room, start together, interact during the match and finally see the result on the scoreboard. It was satisfying to see the different parts, which at the beginning looked quite separate, finally working together. The hardest part for me was the beginning of the project. Before writing code, we had to understand how to structure the system and how to split the problem into smaller parts. Once this "initial puzzle" became clearer, the development became much more fluid. Another challenging part was testing the game, since some bugs and missing edge cases only became clear once I actually sat down and played the game myself, rather than just reading through the code. Looking back, I would improve the time management of the project. With more time, I would have liked to polish the code structure, handle more edge cases and test the gameplay more deeply.

Overall, this project helped me understand better how a multiplayer application works behind the scenes. I also learned that good structure at the beginning can save a lot of confusion later, and that testing a game is not only about checking if the code runs, but also if the experience works well for the players.

8.2 Anna Malagoli

My primary responsibilities in the project were to design and implement the core game logic and the join room. I developed the game elements, such as ingredients and recipes, and defined the movement and interaction logic within the user interface. Additionally, I created a thread that acts as a timer to ensure users complete recipes within the time limit. I developed the code for managing the join room and the game state on both the client and server sides. This included managing message exchanges to update players' state (e.g., from LOBBY to PLAYING), updating the GUI in real time based on server messages (such as exchanging ingredients or receiving new recipes), and collecting data to calculate end-game statistics. I also enabled the clients to handle server crashes by redirecting players back to the main menu, and I implemented an automatic reconnection mechanism that waits for the server to become available again. One of the most successful outcomes of the project is its modular design, which makes it easy to extend. This allows us to add new gameplay elements, like a cutting board or a deep fryer, to make the game more complex and fun. The hardest part for me was the beginning of the project, when the server connection was not yet implemented. I had to create an initial client-side version using predefined variables and simulate server actions using timers. As a result, the most challenging task was integrating this standalone code into the distributed system once the server was built and the connection was established.

In conclusion, this project taught me how important it is to build a resilient architecture. Managing server-client synchronization and handling unexpected crashes showed me that writing good code is not just about creating features, but also about preventing errors and keeping the network stable.

9 Future works

Even though Haochi-Cook&Pass's distributed architecture and soft real-time synchronization already work quite well, there are several things that could be improved or added in the future.

9.1 Content expansion

On the functional side, one possible improvement would be to expand the game content. At the moment, recipes and ingredients are already managed through MongoDB, so the database could be extended with more recipes, more ingredients and possibly more complex preparation mechanics. For example, at the moment a recipe is completed simply by collecting the correct set of ingredients on a plate. A possible extension would be to introduce **multi-step recipes**, where ingredients need to be combined or transformed in a specific order before the plate can be considered complete. This would require changes on multiple levels:

- **Database**: extend the recipe documents in MongoDB with an ordered list of steps, instead of a flat list of ingredients, distinguishing between raw ingredients and intermediate "prepared" ingredients obtained by combining others.
- **Client side**: extend the drag-and-drop interaction so that combining two ingredients produces a new one.
- **Server side**: update the validation logic in the plate-completion handler to check the correct sequence of steps, instead of only the final set of ingredients.

This kind of extension would make the gameplay more varied and would increase replayability, since players would need to coordinate not only on which ingredients to pass, but also on the order in which preparation steps are carried out.

9.2 Unimplemented feature: player authentication

One important feature we did not implement is player authentication. In the current version, players are temporary users identified only during the match, for example by their connection, room code and ingredient identifier. Once the match ends or a player disconnects, no information about them is preserved.

In a future version, each player could have a real account with a username and password. This would make it possible to store personal scores, match history, statistics, preferences and possibly unlocked content.

9.2.1 Client-side changes

New states could be introduced in the `GameModel`, for example `LOGIN` and `REGISTER`, in the same way as the existing states such as `MENU`, `LOBBY`, `JOIN_INPUT`, `PLAYING` and `SCORE`. The `GameView` could then choose the correct template to draw depending on the current state, as it already does for the menu, lobby, join screen, gameplay and scoreboard. Separate templates such as `login_view` and `register_view` could be added to keep the UI consistent with the current organization.

9.2.2 Server-side changes

New controller functions could be added, for example `handle_register` and `handle_login`, which would receive the credentials from the client, validate them and communicate with MongoDB. A new `users` collection could store information such as username, email, password hash, statistics and preferences. The existing `Player` class could also be extended with a `user_id` field, so that a temporary player connected through WebSocket could be linked to a persistent account.

9.2.3 Security considerations

Security would be an important part of this feature:

- Passwords should never be stored in plain text; instead, they should be hashed using a secure algorithm such as bcrypt or Argon2, together with a salt.
- During login, the server would hash the inserted password and compare it with the stored hash.
- The system should also validate user input, avoid duplicate usernames or emails, and prevent sensitive data from being sent back to the client.

9.2.4 Sessions and reconnection

After a successful login, the server could generate an authentication token or use a session-based mechanism. The client would then include this token in future messages or when opening the WebSocket connection so that the server could recognize the player not only as a temporary connection but as a real authenticated user. This would also improve other parts of the game:

- The scoreboard could become persistent, showing not only the result of the current match but also previous scores or global rankings.
- If a player disconnects during a match, authentication could make reconnection easier, since the server could identify the returning user and restore their previous room, position, score and current ingredients.

10 Demo link

The following link contains a short video showing a full playthrough of the game, including room creation, gameplay and the final scoreboard: here.