

Escape Room – Final Report

Project in Distributed Systems

Daniel Meco

daniel.meco@studio.unibo.it

A.Y. 2024/25

Abstract

This project is a real-time, multiplayer escape room game designed to demonstrate the principles of distributed systems and real-time communication. Players join a lobby, coordinate via an integrated chat system, and collaboratively solve a series of mini-games and quizzes within a global time limit to “escape” the room. The front-end is built using React for dynamic user interface components and integrated with Phaser to manage the game’s 2D interactive canvas. The back-end is developed using Node.js with Express, and real-time communication is handled through socket.io. This architecture ensures that game states, such as puzzle completion and chat messages, are synchronized across all players in real time, providing a seamless cooperative gaming experience.

Objectives

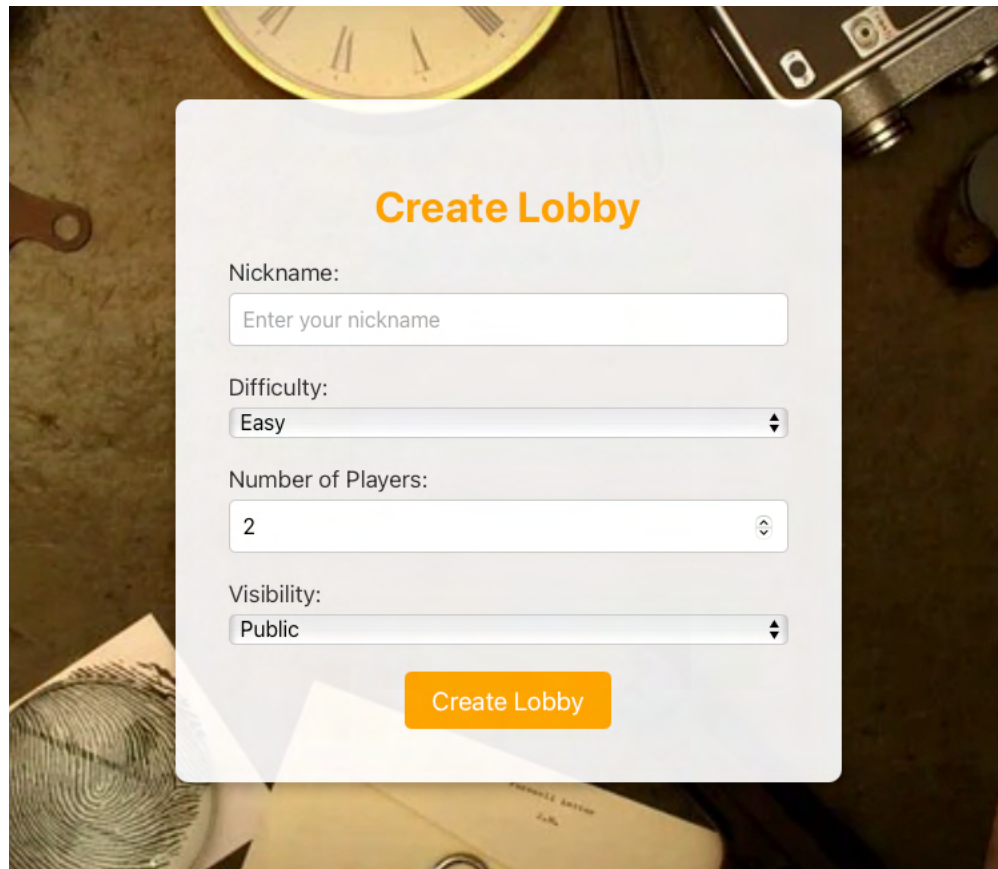
The primary objectives of this project are as follows:

1. Create an escape room game where multiple players can join a lobby, communicate via chat, and work together to solve puzzles within a limited time frame.
2. Ensure that all game events, such as puzzle interactions, player actions, and chat messages, are instantly synchronized across all connected clients using socket.io.
3. Integrate Modern Web Technologies:
 - **Front-end:** Utilize React for building the user interface and integrating game components. Leverage Phaser to handle the 2D graphics, animations, and interactive elements within the game canvas.
 - **Back-end:** Use Node.js and Express to set up a server that manages API requests, game logic, and real-time event broadcasting.
4. Showcase how distributed systems techniques—such as real-time data communication, client-server synchronization, and state management—can be applied to create an engaging and scalable multiplayer game.

Usage Scenarios

1. Creating a New Lobby and Inviting Friends

- **Scenario:**
A player launches the application and decides to host a new escape room session.
- **Steps:**
 - The player selects "Create Lobby" from the main menu.
 - A unique lobby ID is generated, and the host sets parameters such as the number of players and game rules (e.g., difficulty, global timer, visibility).
 - The host shares the lobby ID or link with friends via external communication (e.g., messaging apps or email).
- **Outcome:**
Friends join the lobby, and once the required number of players is reached, the host starts the game. This scenario demonstrates the lobby creation, user management, and initial setup process.



Create Lobby

Nickname:

Difficulty:

Number of Players:

Visibility:

Create Lobby

Figure 1 : Create Lobby page

2. Joining an Existing Lobby

- **Scenario:**

A player who wants to join a game opens the application and browses the list of available public lobbies.

- **Steps:**

- The player selects an available lobby from the list.
- Upon joining, the player is added to the lobby's participant list and can see the other connected players.
- The integrated chat system allows the player to introduce themselves and coordinate with others.

- **Outcome:**

The lobby updates in real time to reflect the new participant, and the player is now part of the group waiting for the game to begin.

Lobbies	
Lobby 1 Code: 9LI876	Public 1/3
Lobby 2 1/2	Private

Figure 2 : Join Lobby page

3. Coordinating Through In-Game Chat

- **Scenario:**
Once players are in the lobby or during gameplay, communication is essential for solving puzzles collaboratively.
- **Steps:**
 - Players use the in-game chat panel to share hints, assign tasks, and coordinate strategies.
 - The chat messages are broadcasted to all players in real time via socket.io.
 - Players might discuss which mini-game to tackle first or how to combine clues from different parts of the room.
- **Outcome:**
Enhanced collaboration leads to more efficient problem solving and a more engaging multiplayer experience.

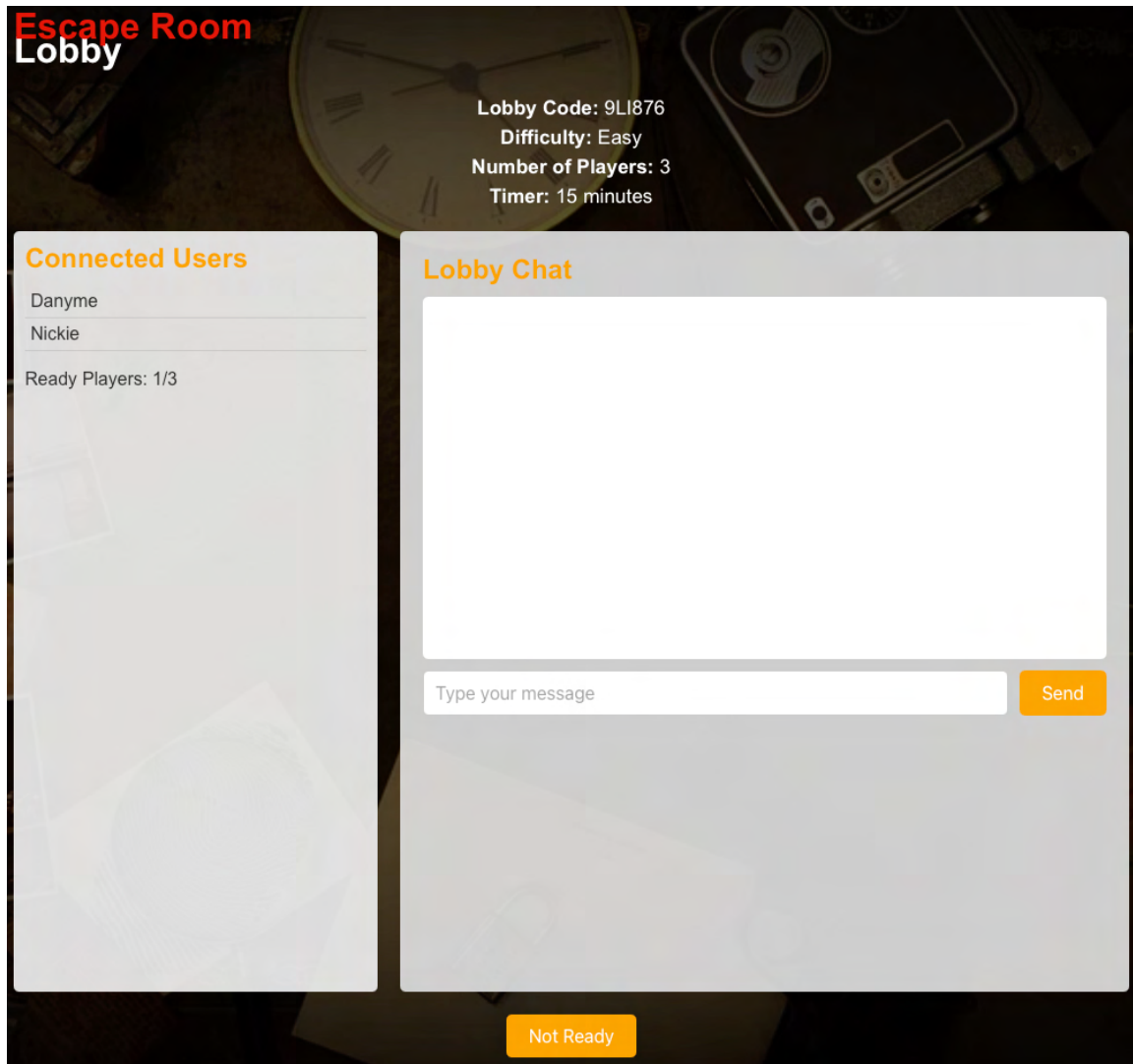


Figure 3 : The Lobby page with Chat

4. Solving Mini-Games/Quizzes in Real Time

- **Scenario:**
During the game, players must solve one or more mini-games or quizzes that appear as interactive overlays on the game canvas.
- **Steps:**
 - A player interacts with a hotspot (e.g., clicks on an object in the room), which triggers a mini-game (such as a multiple-choice quiz or a puzzle).
 - The mini-game opens in an overlay.
 - Once a correct answer is submitted, the client emits an event to the server indicating that the puzzle is solved.
 - The server validates the solution and broadcasts the update to all clients, causing the corresponding hotspot to display a "solved" state.
- **Outcome:**
All players see the immediate update across their devices, ensuring the game state remains consistent. This scenario demonstrates real-time event handling and synchronization between clients.

5. Global Time Constraint and Game Completion

- **Scenario:**
The escape room features a global timer that challenges the team to complete all puzzles within a fixed period (e.g., 5 minutes).
- **Steps:**
 - A countdown timer, managed by the server, is visible to all players throughout the game.
 - Players must work together to solve the puzzles before time runs out.
 - If the final puzzle is solved within the allotted time, the game state updates to indicate that the door is unlocked and the room has successfully escaped.
 - If the timer expires before all puzzles are solved, the game notifies the players that the session has ended unsuccessfully.
- **Outcome:**
This scenario emphasizes the need for quick, coordinated action and highlights the importance of real-time synchronization in a high-pressure, cooperative environment.

Theoretical Foundations

The development of this multiplayer escape room project is rooted in several key theoretical concepts and principles of distributed systems and real-time web applications:

- The project leverages a classic client-server model, where the server (built with Node.js and Express) acts as the central authority for managing game state, user sessions, and real-

time communication. This approach is grounded in distributed system theories that emphasize the challenges of state consistency, fault tolerance, and scalability across multiple networked clients.

- Real-time interactions are fundamental to the cooperative gameplay experience. We utilize socket.io to implement WebSocket-based communication, ensuring that events (such as puzzle completions, chat messages, and lobby updates) are synchronized across all connected clients.

It is important to note that WebSocket uses TCP as its underlying transport protocol. This ensures that connections are reliable and that messages are delivered in order. Unlike UDP—which does not guarantee reliability or ordered delivery—TCP provides the necessary assurances for maintaining consistent game state across clients, even in environments where data loss or reordering might occur.

- Handling multiple users interacting simultaneously with shared game elements introduces challenges such as race conditions and data inconsistencies. This system design takes inspiration from consistency models discussed in distributed computing literature (e.g., eventual consistency and strong consistency) to determine how state updates (like puzzle resolutions) are broadcast and applied across clients. The server acts as the single source of truth, mitigating the risks associated with concurrent modifications.
- Although the initial implementation targets a small-scale prototype, the architectural choices made—such as the separation of concerns between front-end and back-end, and the use of modular frameworks—are influenced by scalable system design principles. Future enhancements could involve load balancing and distributed databases to further improve the system's scalability and resilience to failures, aligning with theories related to distributed consensus.

Questions & Answers

Q1: What is the main goal of this project?

A: The project aims to create a real-time, multiplayer escape room game where players join a lobby, communicate via an integrated chat, and collaboratively solve a series of mini-games and puzzles within a limited time. This demonstrates key concepts of distributed systems, real-time communication, and client-server architecture.

Q2: Which technologies are used in the project?

A:

- **Front-end:**
 - *React* for building dynamic user interfaces and managing application state.
 - *Phaser* (integrated within React) for rendering the game's interactive 2D canvas and handling game logic.
 - *HTML/CSS* for basic layout and styling of pages outside the canvas (e.g., menus and chat panels).
 - **Back-end:**
 - *Node.js* with *Express* to build the server and manage API requests.
 - *socket.io* for enabling real-time, bidirectional communication between the server and clients.
-

Q3: How does real-time communication work in the project?

A: Real-time communication is handled via *socket.io*, which is built on *WebSocket* technology. *WebSockets* use *TCP* as their underlying transport protocol, ensuring that messages are delivered reliably and in order. This guarantees that updates—such as puzzle completions, chat messages, and lobby status—are synchronized across all connected clients.

Q4: How does the integration of Phaser with React work?

A: *Phaser* is integrated into *React* by creating a dedicated component (e.g., `PhaserGame.jsx`) that uses a *React* ref to designate a container element for the *Phaser* canvas. Using *React*'s `useEffect` hook, the *Phaser* game instance is created when the component mounts and destroyed when it unmounts. This approach allows *Phaser* to handle the interactive game scene while *React* manages the overall UI and state.

Q5: What are the main challenges addressed by this project?

A:

- **State Synchronization:** Ensuring that game state (e.g., puzzle progress, player interactions) is consistent and updated in real time across multiple clients.
 - **Real-Time Communication:** Using WebSockets (via socket.io) to handle low-latency communication and maintain a responsive multiplayer experience.
 - **User Coordination:** Facilitating effective communication and coordination among players through integrated chat and lobby features, which are essential for solving puzzles cooperatively under time constraints.
-

Q6: What happens if a player disconnects during a game?

A: If a player who is alone in a crucial room for the game's progression disconnects, then all the remaining players will be redirected back to the starting lobby. If the disconnected player was not alone in a crucial room for the game's progression, the remaining players will continue the game without interruption and the disconnected player will have 30 seconds to re-enter the game without losing the game state

Q7: Can a user join a lobby during a game?

A: No, because it means the lobby is full.

Q8: Is the username of the player unique?

A: Yes, it is inside a lobby.

Q9: When should a game start?

A: A game starts after all players in a lobby have marked themselves as ready.

Self-assessment policy

Q: How should the quality of the produced software be assessed?

A: The quality of the system will be evaluated mainly based on non-functional requirements, which will be defined and refined during the design phase. Key quality attributes to be considered include performance and scalability.

Q: How should the effectiveness of the project outcomes be assessed?

A: The effectiveness of the software will be evaluated primarily through its functional aspects. Clearly defined functional requirements will form the basis of this assessment, ensuring that all specified features are implemented as expected. Manual integration testing will be conducted to assess the overall system behavior.

Functional Requirements

1. *The player must be able to enter a nickname when launching the application. (This enables player identification during lobby creation or joining.)*
 2. *The player must be able to create a new lobby by entering the name, difficulty, visibility of the lobby and number of participants.*
 3. *The player must be able to view the list of active lobbies and join a desired one by entering the lobby code.*
 4. *The lobby must display a list of participants and allow the game to start when the group is complete.*
 5. *The player must be able to use an integrated chat to communicate with other participants, both in the lobby and during the game.*
 6. *The system generates a unique lobby code and notifies the creator.*
 7. *The game must display a static 3D room with interactive hotspots.*
 8. *Upon game start, the system shall assign players to two rooms: one randomly chosen player to Room1 and all others to Room2*
 9. *The system shall allow players to navigate between game rooms. In Room0, a door is displayed for the user to proceed to their assigned room.*
 10. *When the player clicks on a hotspot, a mini-game or quiz must open that needs to be solved.*
 11. *The system must implement a global timer for the entire escape room and, if necessary, specific timers for each mini-game or quiz.*
 12. *The game state (for example, the completion of a puzzle or the sending of a chat message) must be updated and synchronized in real time among all players via WebSocket communication.*
 13. *Once all puzzle solutions are correct, the system shall notify all users in the lobby that they have won the game.*
-

Non-Functional Requirements

1. *The system must ensure real-time communication with low latency to maintain immediate synchronization between clients.*
2. *The system must be designed to handle a high number of concurrent connections, with the potential for future scalability.*
3. *The system must be highly available and capable of handling errors or disconnections without compromising the gaming experience.*
4. *Communications between the client and server must be secured (for example, through encryption where necessary).*
5. *The system must implement validations and controls to prevent unauthorized access or injection attacks.*
6. *The code must be structured in a modular way to facilitate future updates and the integration of new features.*
7. *Coding standards and best practices must be followed, supported by the use of linting and testing tools.*
8. *The user interface must be intuitive, responsive, and accessible, allowing even non-expert users to navigate and interact with the system easily.*
9. *The system shall ensure that, when a player temporarily disconnects, the reconnection mechanism reliably restores the player's session, including game state and room assignments, with minimal latency.*

Implementation Requirements

IR-1: Technology Stack

The client shall be developed using React (with JSX), and the server shall be built using Node.js with Express and Socket.io. The software compiles and runs using the specified technology stack.

IR-2: Deployment

The server shall serve static assets from a public folder, and the client shall be developed using a tool like Create React App. Deployment documentation includes instructions for serving the static assets.

IR-3: Package Management

All dependencies must be managed via npm, with Socket.io used for real-time communication. The package.json lists all necessary dependencies and the application installs them using npm install.

R-4: Use of Phaser for Mini-Games

The mini-games (for example, dynamic interactions and puzzles within the Escape Room) must

be implemented using the Phaser HTML5 framework. The Phaser-based components integrate seamlessly with the rest of the application, ensuring smooth and responsive interactions.

IR-5: Version Control and Repository on GitHub

The complete source code shall be maintained in a Git repository hosted on GitHub. The project repository must be created on GitHub and contain all source code, configuration files, and documentation.

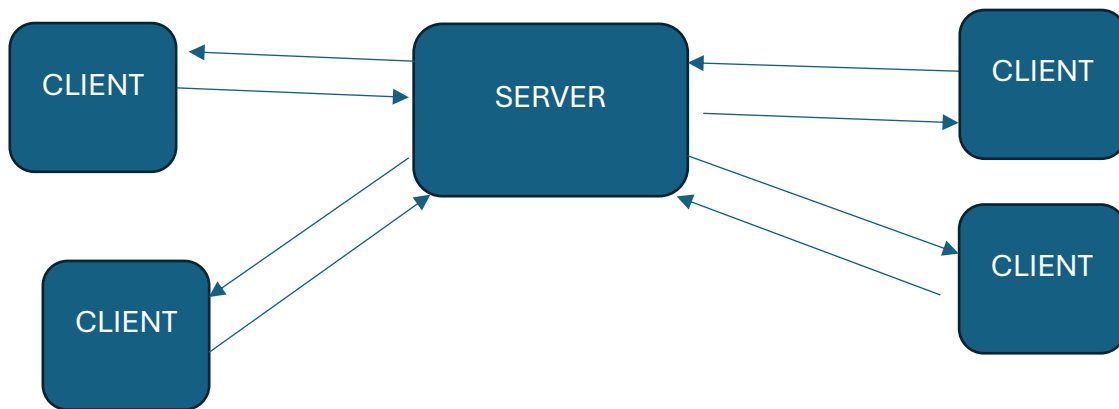
Architecture

The architecture of the Multiplayer Escape Room Game is built on a client-server model that emphasizes real-time interaction and a clear separation of concerns.

On the client side, a **React** application handles all user interface elements and interactive experiences. The client presents various game screens including the lobby, multiple game rooms, and a real-time chat interface. It communicates with the server via Socket.IO, ensuring that events such as lobby creation, player joining, chat messaging, readiness, room transitions, and puzzle interactions are transmitted instantly. The client focuses exclusively on rendering the user interface, capturing user input, and providing immediate visual feedback.

On the server side, a **Node.js** application with **Express** and **Socket.IO** is responsible for managing the core game state and logic. The server maintains an in-memory data structure that tracks active lobbies, player lists, room assignments, and the quiz set used in puzzles. When a lobby is created, the server generates a unique lobby code and stores all related information, including game parameters. Once the game starts, the server assigns one player at random to Room2 while placing all other players in Room4. It then generates a set of three quiz puzzles—each with an image URL (served as static content from a public directory) and an associated solution—and stores this set within the lobby's state.

The server handles verification events by comparing solutions submitted by the player against the stored quiz answers. If all responses are correct, it emits a game win notification to all connected clients in that lobby. In this way, the server not only validates game progress but also orchestrates the overall flow by managing player assignments and ensuring that all static assets (like quiz images) are correctly served to the client. This architecture supports a robust, scalable, and maintainable system where the client is solely responsible for the user experience and presentation, while the server manages the critical game logic and state consistency.



Socket.IO is a library that enables real-time, bidirectional communication between clients (typically web browsers) and servers. It abstracts the underlying transport mechanisms and provides a simple event-based API, so developers can focus on sending and receiving messages without worrying about low-level details.

Socket.IO primarily uses **WebSockets**—a protocol that allows for persistent, full-duplex communication channels over a single TCP connection. However, it also supports fallback transports such as HTTP long polling in cases where WebSocket is not available or blocked by firewalls.

When a client connects, Socket.IO establishes a persistent connection to the server, typically on a designated port (for example, port 3001 in this project). Each connection is assigned a unique socket ID, which helps identify and manage communication with that specific client.

Both the client and server use a publish/subscribe model. Developers can define custom events and attach callback functions that execute when these events occur. This allows for clear, modular, and asynchronous handling of real-time communication. If a WebSocket connection cannot be established—due to browser limitations or network restrictions—Socket.IO automatically falls back to alternative methods (like HTTP long polling). This ensures robust connectivity under various network conditions.

Data is sent as JSON objects, which means complex data structures can be transmitted easily between client and server. This facilitates tasks like sending game state updates, user messages,

and puzzle solutions. Socket.IO connections are typically secured via the same mechanisms as HTTP/HTTPS, and you can configure settings such as allowed origins, transport methods, and heartbeat intervals. For example, this project's server is configured to listen on port 3001 and accept connections from any origin.

Node.js is a JavaScript runtime built on Google Chrome's V8 engine that allows developers to run JavaScript on the server side. At its core, Node.js is designed for building scalable network applications by employing a **non-blocking, event-driven I/O model**. This architecture enables Node.js to handle a large number of simultaneous connections with high efficiency.

In a Node.js server, all I/O operations—such as reading from a file system, accessing databases, or making network requests—are executed asynchronously. Instead of creating a new thread for each connection, Node.js operates on a **single-threaded** event loop that dispatches tasks and handles callbacks when operations complete. This event loop continuously monitors the call stack and the task queue, ensuring that incoming events and I/O operations are processed as soon as possible **without blocking other operations**.

The **non-blocking** nature of Node.js makes it **ideal** for **real-time** applications like multiplayer escape room game. With Socket.IO integrated into the server, Node.js can handle bidirectional, real-time communication between the client and server. Socket.IO abstracts away the complexity of managing persistent connections by falling back to alternative transport methods (such as HTTP long polling) if WebSockets are not available, ensuring reliable communication regardless of network conditions.

Another strength of Node.js is its ease of scalability. Although it runs on a single thread, **Node.js** can **scale horizontally** using **clustering** and **load balancing techniques**. Moreover, by using tools like the Redis adapter with Socket.IO, you can distribute state and events across multiple server instances. This distributed approach allows the application to support thousands of concurrent connections, which is essential for a game that expects many players to interact in **real time**.

Overall, the architecture of the Node.js server is designed to provide **high performance, low latency, and robust scalability**. The server is responsible for managing game state, lobby information, room assignments, and puzzle verification. It communicates with the client through Socket.IO, ensuring that every event—from player actions and chat messages to puzzle submissions—is handled promptly. This combination of asynchronous processing, real-time communication, and scalable design makes Node.js a powerful foundation for the multiplayer escape room game.

Architectural Considerations: MVC vs. Event-Driven Model

The **Model-View-Controller (MVC)** pattern is a widely adopted software architectural model, primarily used in web applications that rely on structured request-response cycles. However, this multiplayer escape room game does not strictly follow the MVC pattern, as it is built on **real-time, event-driven communication** facilitated by **Socket.IO and WebSockets**.

Why Event-Driven Model is the Best Fit for this Application?

Unlike traditional MVC-based applications that rely on HTTP request-response cycles, this game maintains **persistent bidirectional connections** using **WebSockets**. This allows real-time updates between clients and the server without requiring repeated HTTP requests.

In a typical MVC framework, the server is responsible for rendering HTML pages and managing user sessions. In contrast, this application utilizes **React.js for front-end rendering**, allowing dynamic UI updates without the need for server-rendered views. The server only transmits **game state updates** and does not control the visual presentation.

Traditional MVC applications have controllers that handle incoming requests and manage the interaction between models and views. This game follows a **real-time event-driven architecture**, where **Socket.IO handles event-based communication** between clients and the server. The server processes socket events, ensuring real-time updates to all connected clients.

Unlike an MVC structure, where a model layer typically interacts with a database, this application **manages game sessions and player states in memory**. The server maintains an in-memory data structure that tracks players, lobby assignments, quiz data, and game progress. This approach optimizes performance by reducing database overhead.

Alternative Architectural Model: Event-Driven Real-Time System

This game architecture is structured as follows:

- **Client (React.js) – Responsible for UI Rendering**
The client dynamically renders game rooms, interactive areas, and chat messages while handling user interactions. It listens to and emits **WebSocket events** to communicate with the server.
- **Server (Node.js + Socket.IO) – Acts as the Game State Manager**
The server manages **real-time game logic** by handling socket events. It tracks:
 - Player **assignments** to different rooms.

- **Quiz distribution** and validation.
 - **Game session data** stored in memory for performance optimization.
- **Real-Time Synchronization Between Players**
The server ensures **synchronized game progression** by broadcasting updates to all clients in a game session. This mechanism allows seamless communication between players in different rooms.

When Would MVC Be More Suitable?

If this game required a **persistent database** to store user profiles, progress, or game history, an **MVC-based backend** with a structured REST API might be appropriate.

However, given the nature of this **real-time multiplayer** gameplay, an **event-driven model** is significantly **more effective**.

Structure

Below is a detailed description of a conceptual class diagram for this application's server side.

1. Lobby

This class represents a lobby where users can join and interact. It holds the game configuration, such as difficulty, maximum number of players, timer, and visibility, and maintains a list of connected users (instances of **User**). Additionally, it manages the game state through an instance of the **Game** class. The **Lobby** class provides methods to add or remove users, broadcast messages to all connected clients.

- **Attributes:**
 - **code:** A unique string representing the lobby code.
 - **Name:** A unique string name.
 - **users:** An array of **User** objects representing players currently in the lobby.
 - **difficulty:** A string or enumerated type indicating the game difficulty.
 - **numPlayers:** The maximum number of players allowed in the lobby.
 - **timer:** A numeric value representing the game timer.
 - **Game[]:** Instances of a **Game** class.
- **Methods:**
 - **addUser(user: User):** Adds a new user to the lobby if there is capacity.
 - **removeUser(user: User):** Removes a user from the lobby, updating the state accordingly.
 - **broadcast(data):** Broadcast messages to all connected clients.

2. User

Represents an individual connected client. It stores a reference to the client's socket, the user's nickname, and the lobby code that the user is connected to. The class provides methods to join a lobby (which includes adding the user to the corresponding Socket.io room) and to send events back to the client.

- **Attributes:**
 - **nickname:** A string that uniquely identifies the player within a lobby.
 - **socketId:** A unique identifier corresponding to the player's Socket.IO connection.
 - **ready:** A boolean flag indicating whether the player is ready to start the game.
- **Methods:**
 - **Send():** Send messages to others.
 - **markReady():** Sets the user's ready status to true.
 - **connects():** Handles when the user connects, making sure they are added in the lobby.

3. Game

The Game class encapsulates all match-specific logic, including:

- **Attributes:**
 - **Id:** A unique identifier for the game
 - **users:** An array of **User** objects representing players currently in the lobby.
 - **quizSet:** An array of **Quiz** objects (each containing an image URL and the corresponding solution) that is generated and stored when the game starts.
 - **verifiedQuizSolutions:** A mapping (or dictionary) where keys are area numbers (1, 2, 3) and values are the submitted solutions that have been verified as correct.
 - **roomAssignments:** An object mapping room names (e.g., "room2", "room4") to arrays of **User** objects that have been assigned to those rooms.
- **Methods:**
 - **assignRooms():** Based on the current user list, randomly selects players to be assigned to a Room.
 - **setQuizSet(quizSet: Quiz[]):** Stores the quiz set generated for this lobby.
 - **verifySolution(area: number, solution: string):** Checks the submitted solution against the expected answer from the quiz set for the given area and updates the verifiedQuizSolutions mapping.

3. Quiz

Encapsulates the data for a single quiz item, including an identifier, the image URL, and the correct solution. It also provides a method, `checkSolution(input)`, to verify if a user's response matches the solution.

- **Attributes:**
 - **id:** A unique identifier for the quiz.
 - **image:** A URL string pointing to the quiz image (served from the server's public assets folder).
 - **solution:** The correct answer for the quiz, which may be of variable length and content.
- **Methods:**
 - **checkSolution(proposedSolution: string):** Returns a boolean indicating whether the proposed solution matches the stored solution after appropriate normalization (e.g., lowercasing, trimming).

4. Puzzle

Manages the logic for a sliding puzzle game (a 3x3 grid). The class generates a solved board and then shuffles it by executing a series of valid moves. Its methods include a static function to generate a solved board and a method to shuffle the board state, thereby creating a new puzzle configuration.

- **Attributes:**
 - **boardState:** It is a 3x3 matrix that represents the current state of the puzzle.
- **Methods:**
 - **generateSolvedBoard():** This method creates and returns a 3x3 array in resolved configuration. It is used during initialization to have a reference of the correct configuration before shuffling the puzzle.
 - **shuffle():** Make a certain number of valid moves to shuffle the puzzle. It keeps track of the position of the empty space (represented by 0) and, with each move, determines the valid positions in which the space can move (up, down, left, right).

5. GameServer Manager

The central class that configures and starts the Express server and integrates Socket.io. It manages the active game lobbies (each represented by a Lobby instance) and stores arrays of quiz images and their corresponding solutions. The class includes methods for setting up routes, handling socket connections, generating random quiz sets, and creating unique lobby codes.

- **Attributes:**
 - **lobbies:** A collection (such as a JavaScript object or Map) mapping lobby codes to **Lobby** objects.
 - **server, io, PORT.**

- **Methods:**
 - **createLobby(nickname, difficulty, numPlayers, timer):** Creates a new **Lobby** instance and adds it to the lobbies collection.
 - **joinLobby(lobbyCode, user: User):** Adds a user to an existing lobby and updates the lobby's state.
 - **handlePlayerReady(nickname, lobbyCode, ready):** Updates the lobby's user readiness status and triggers the game start sequence if all players are ready.
 - **verifyQuizSubmission(lobbyCode, area, solution):** Retrieves the corresponding quiz from the lobby's quizSet using the area index, verifies the solution, and if all three are correct, emits a game win event to all clients.
 - **generateLobbyCode()**
Generates a unique alphanumeric code for a new lobby.
 - **getRandomQuizSet()**
Randomly selects 3 quizzes from the quiz arrays and returns an array of quiz objects. Each quiz object contains an `id`, an `image` URL, and the corresponding `solution`.
 - **getlobbiesList()**
Constructs and returns a summary list of active lobbies, including each lobby's name, code (if public), current number of users, and maximum capacity.
 - **handleUserLeaving():** This method is invoked when a user disconnects or leaves the lobby. It removes the user from the lobby's user list, makes the user leave the corresponding socket room, and if the lobby becomes empty afterward, deletes the lobby. Finally, it updates the list of lobbies for all clients.
 - **checkInsufficientPlayers:** This method verifies if the active players in each assigned are sufficient to continue the game.

In addition to helper functions for generating lobby codes, quiz sets, and puzzle boards, the server implements a comprehensive suite of Socket.IO event handlers that manage core functionalities. Handlers are essential because they serve as the bridge between the incoming real-time events from clients and the business logic encapsulated in the classes.

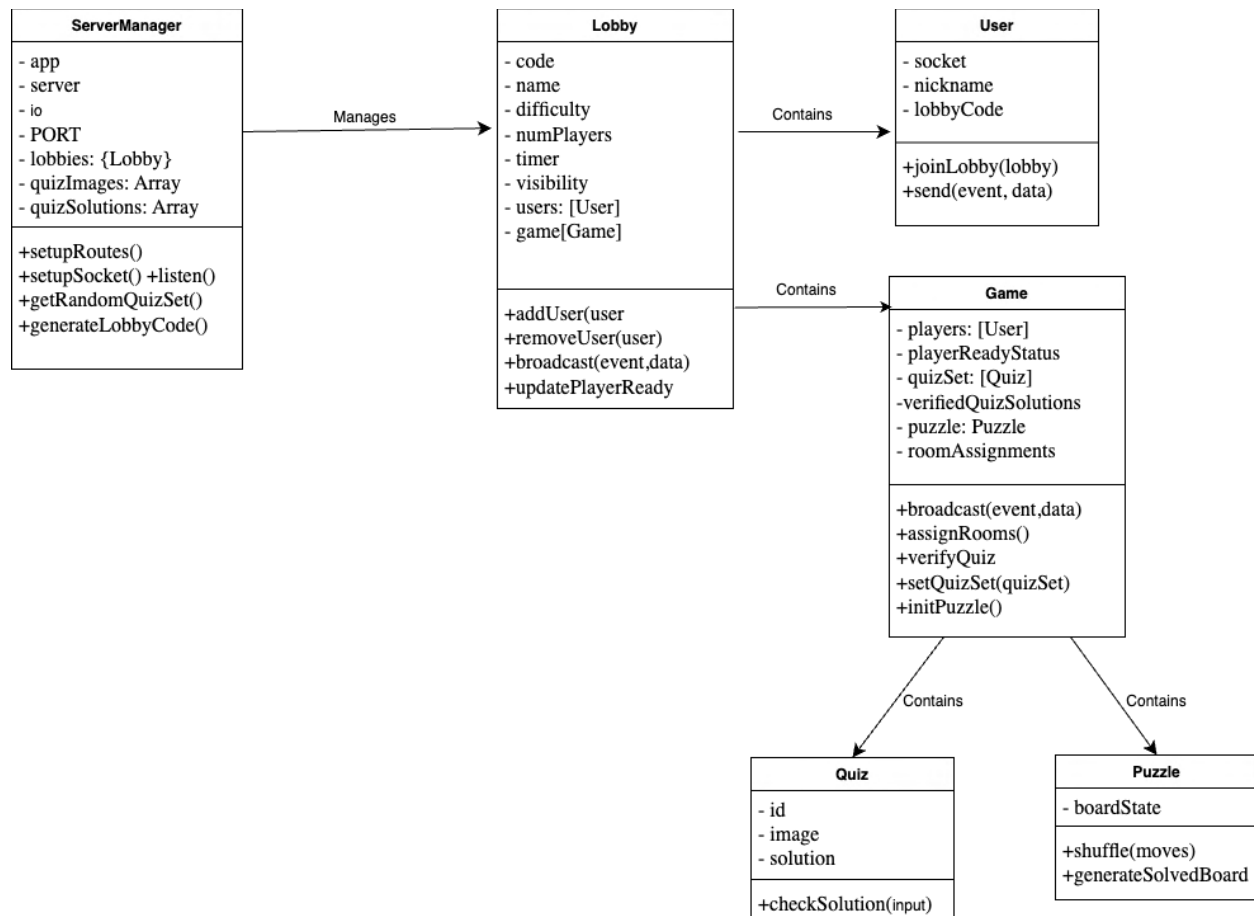


Figure 4: Conceptual class diagram server-side

Below is a conceptual class diagram for the client side of the Escape Room Game. Although the application is built using React (a functional and component-based paradigm rather than classical OOP), this diagram serves as a high-level model of the main components, their responsibilities, and their interactions.

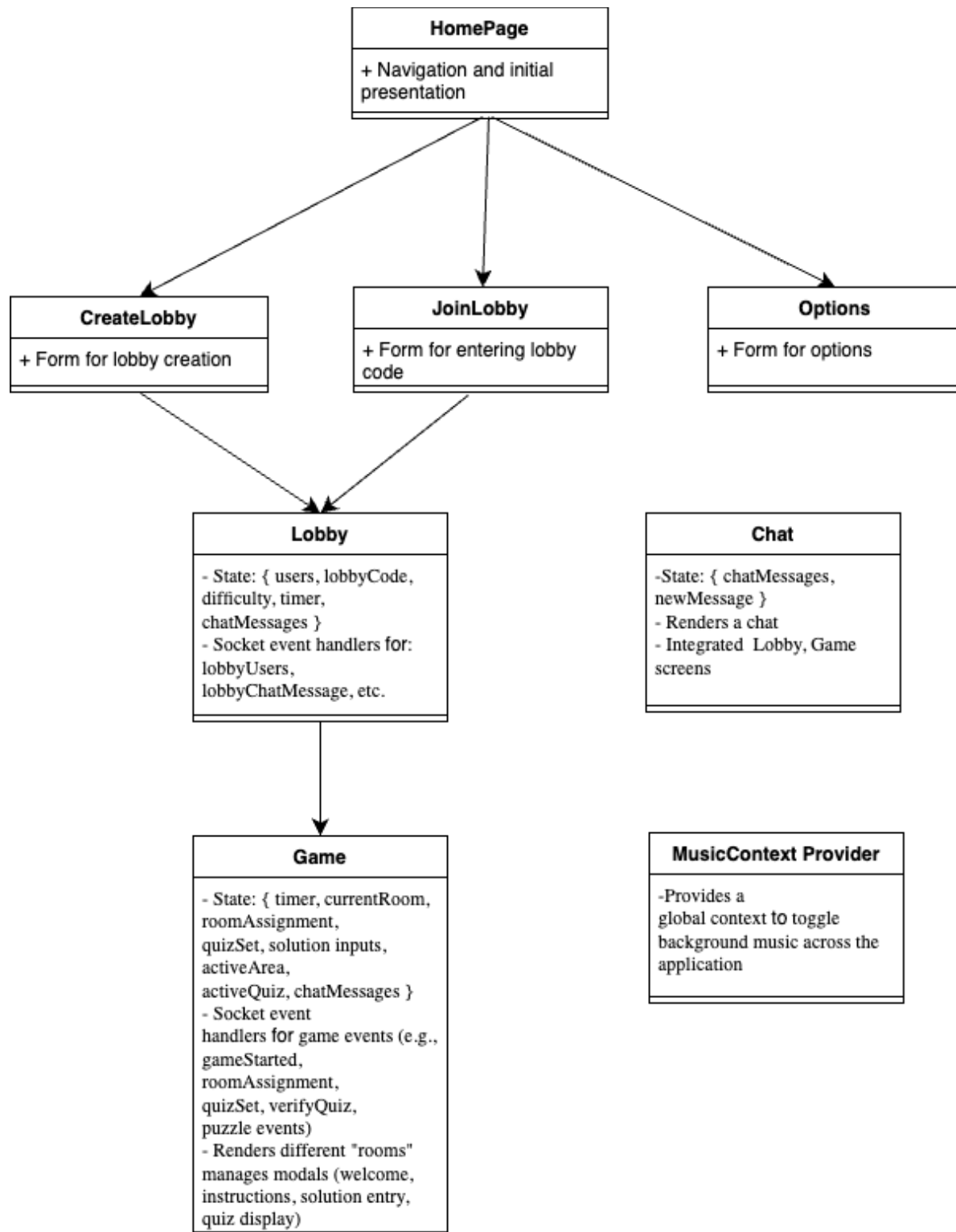


Figure 5: Conceptual class diagram client-side

Behavior and Interaction

The behavior of each node is summarized in the state diagram shown in the figure 4. This diagram outlines the main logic of the entire application, from initialization and setup to the conclusion of a game. It highlights two macro states: the behavior of the game during a lobby setup and during gameplay.

In this application, all interactions between the client and the server are carried via structured JSON payloads. Every event transmitted over Socket.IO uses a clearly defined JSON format that encapsulates the necessary data for that particular interaction. For example, when a player sends a chat message, the payload includes fields such as *lobbyCode*, *nickname*, and *message*, ensuring that the message is routed to the correct lobby and identified by the sender. Similarly, events like *verifyQuiz* include keys such as *lobbyCode*, *area*, and *code* (the submitted solution), which the server uses to verify the answer against the expected solution stored in the quiz set. This standardized approach not only simplifies the parsing and validation of data on both ends but also enhances debugging and maintenance, as every payload is predictable and consistent. Overall, using JSON for all interactions ensures clear, extensible, and reliable communication, which is essential for a real-time multiplayer game environment.

```
Received lobbyUsers on client: ▼ {users: Array(2)} ⓘ Lobby.jsx:40
    ► users: (2) ['daniel', 'diego']
    ► [[Prototype]]: Object

Received lobbyChatMessage on client: Lobby.jsx:46
▼ {nickname: 'diego', message: 'ciao'} ⓘ
  message: "ciao"
  nickname: "diego"
  ► [[Prototype]]: Object

>
```

Figure 6 : JSON objects

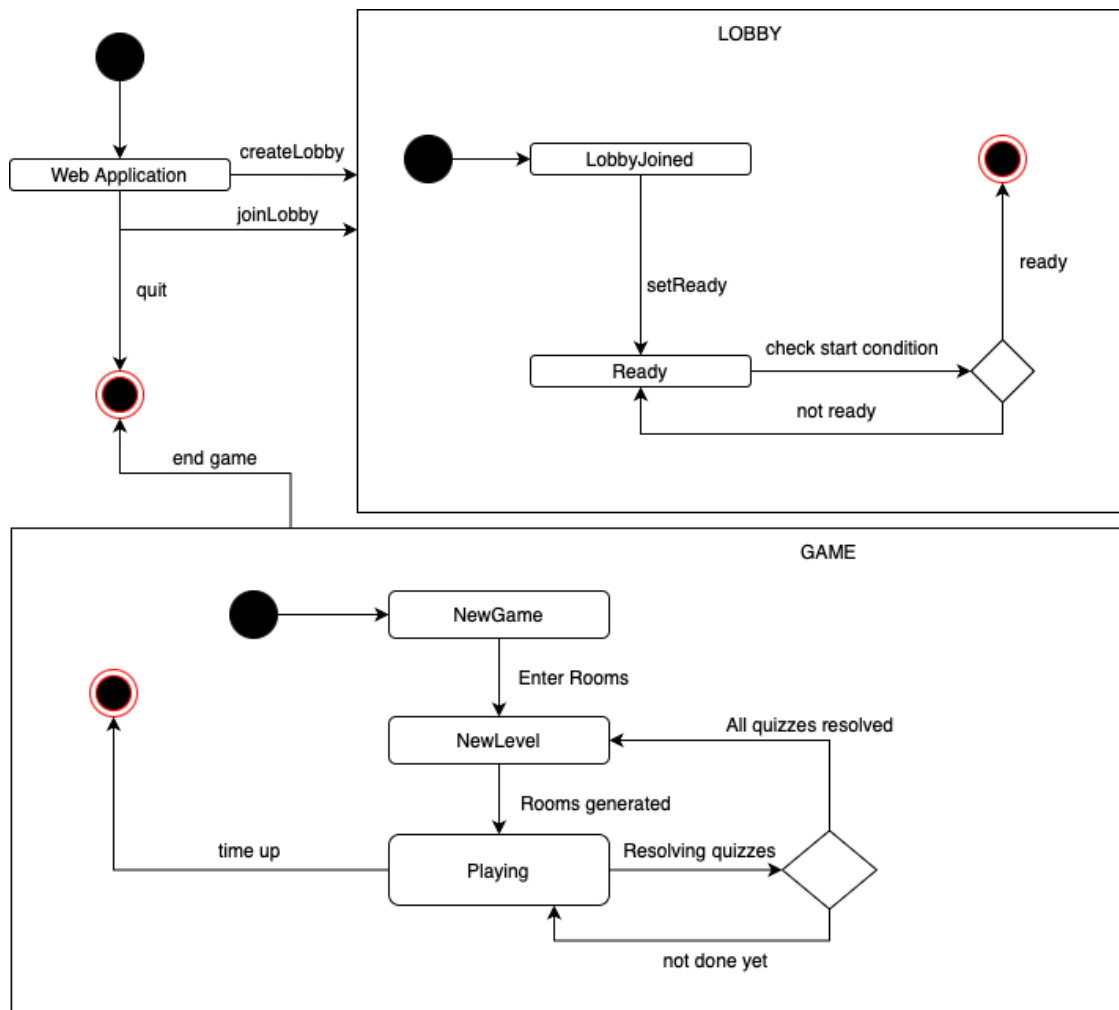


Figure 7: State diagram of the game application

Below, the schema describes the lobby workflow where a client can create or join a lobby by sending events that include necessary configurations such as nickname, number of players, timer settings, and visibility. The lobby creates a user instance when someone joins and immediately broadcasts the updated list of users to everyone in the lobby. It also handles real-time chat messages sent to the lobby and manages user disconnections by removing users and updating all connected clients accordingly. The entire process is coordinated by the ServerManager which creates and maintains lobby instances, ensuring that all participants receive timely information about the lobby state without any manual refresh requirements.

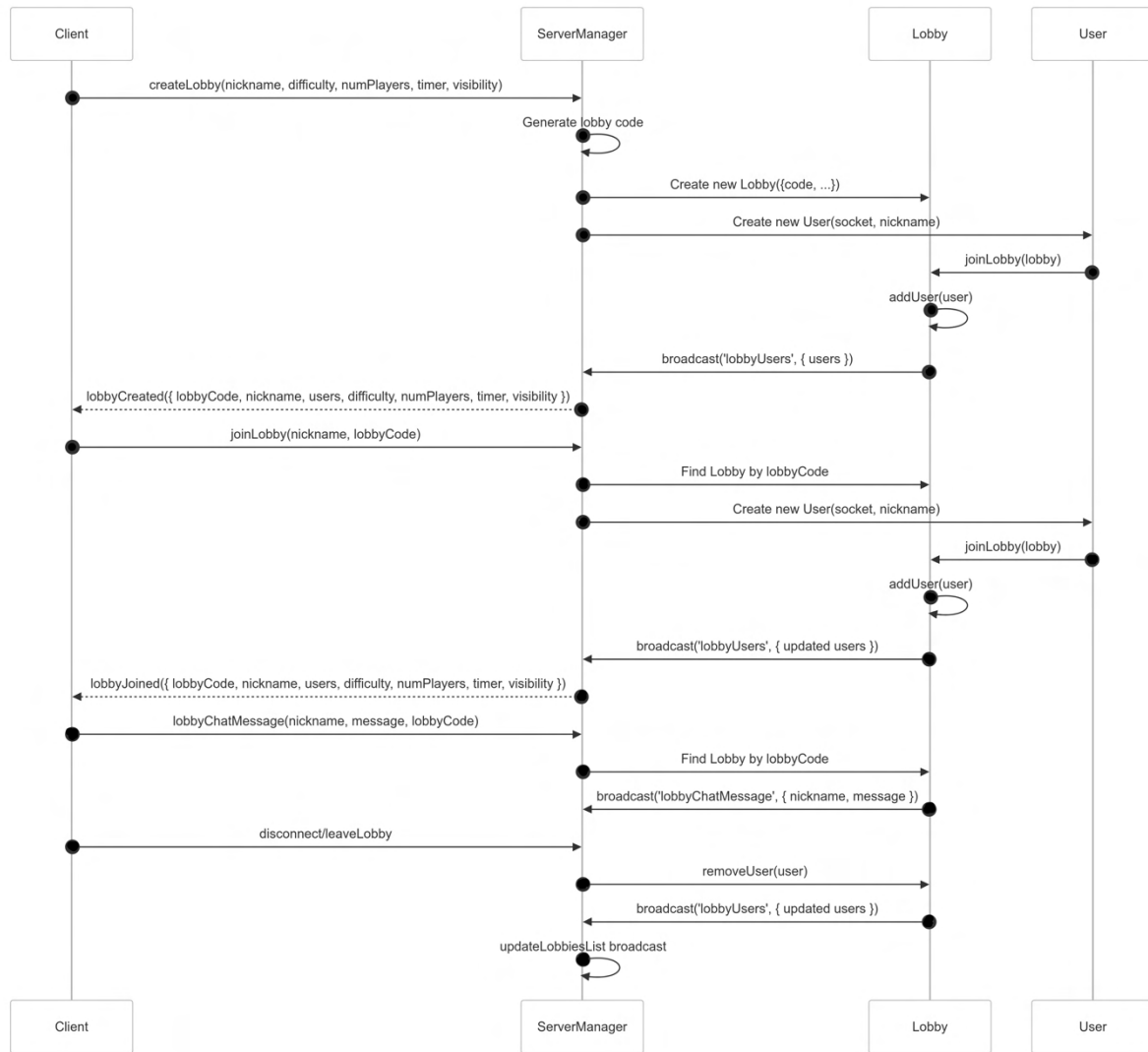


Figure 8: Lobby sequence diagram

Below, the sequence diagram details the flow of an Escape Room game (level 1) starting from when a client joins the game until the game ends either by winning through a correct quiz or puzzle solution or by the timer running out. The client sends a join request and receives the game configuration such as timer and maximum players, then signals readiness to start. Once all players are ready, the server starts the game, assigns roles for either the quiz or puzzle modules, and begins the timer. For clients assigned to the quiz role, a random quiz set is selected and sent; the client verifies solutions which update the game state until all quizzes are solved, resulting in a game win announcement. For clients assigned to the puzzle role, a puzzle instance is created with a solved board that is then shuffled, and the client interacts with the puzzle via tile movements until the puzzle is solved, which triggers a victory message. Meanwhile, the timer is decremented each second and upon reaching zero, a time up event is broadcast, ending the game if no win condition is met.

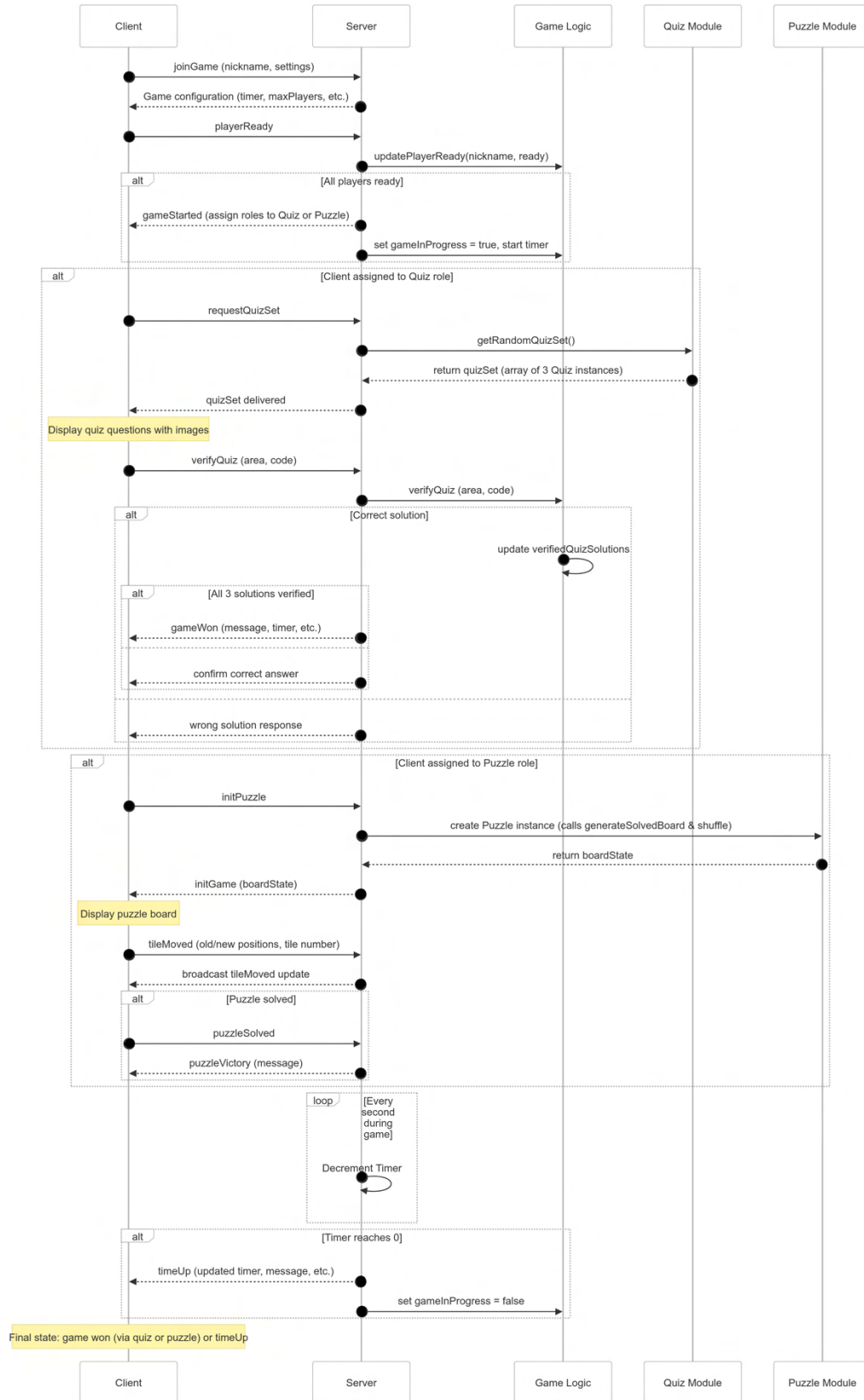


Figure 9: Game sequence diagram

Implementation details

The core of the project resides in the **ServerManager** class, which:

1. Creates the Express application.
2. Creates the HTTP server on the chosen port.
3. Integrates Socket.io for bidirectional communication with the clients.

```
const express = require('express');
const http = require('http');
const socketIo = require('socket.io');

/**
 * Manager class that handles the server and various Socket.io event handlers
 */
class ServerManager {
  constructor() {
    // Create the Express app
    this.app = express();
    // Create the HTTP server
    this.server = http.createServer(this.app);
    // Integrate Socket.io with the HTTP server
    this.io = socketIo(this.server, {
      cors: { origin: "*", methods: ["GET", "POST"] }
    });

    // Configure the listening port
    this.PORT = process.env.PORT || 3001;

    // Map of active lobbies: { lobbyCode: instance of Lobby }
    this.lobbies = {};

    // Set up Express routes and Socket.io handlers
    this.setupRoutes();
    this.setupSocket();
  }

  /**
   * Definition of basic routes with Express
   */
  setupRoutes() {
    // Serve static files (HTML, CSS, images, etc.) from the "public" folder
    this.app.use(express.static('public'));

    // A GET route for testing that simply returns a "Server is running" message
    this.app.get('/', (req, res) => {
      res.send('Server is running');
    });
  }
}
```

```

/**
 * Configuration of Socket.io listeners to handle various events
 */
setupSocket() {
  // Triggered whenever a new client connects, creating a unique socket
  this.io.on('connection', (socket) => {
    console.log('New client connected');

    // Example: send a welcome message
    socket.emit('welcomeMessage', { message: "Welcome to the Escape Room Server!" });

    // Event handlers for createLobby, joinLobby, etc.
    // ... (other event handlers that manage game logic)
  });
}

/**
 * Start the HTTP server
 */
listen() {
  this.server.listen(this.PORT, () => {
    console.log(`Server listening on port ${this.PORT}`);
  });
}

// Create and start the ServerManager instance
const manager = new ServerManager();
manager.listen();

```

Figure 10: ServerManager

An `app` object is created via `express()`. This app handles the usual HTTP routes and serves static files (HTML, CSS, images, etc.) from the `public` folder.

With `http.createServer(this.app)`, we build an HTTP server that wraps our Express app. This allows **Socket.io** to use the same HTTP server to manage WebSocket connections.

The Socket.io object `io` is initialized by passing the HTTP server as an argument, so any client that connects can use a real-time channel (WebSocket or fallback) to exchange messages with the server.

`process.env.PORT || 3001` sets the port on which the server listens. If no environment variable is provided, the default port 3001 is used.

On the **setupRoutes()** method:

- `this.app.use(express.static('public'))` serves the `public` folder so clients can load HTML pages, scripts, and assets directly from the server.
- The `GET /` route simply returns “Server is running,” confirming that the server is active.

On the **setupSocket()** method:

- `this.io.on('connection', ...)` sets up a listener invoked every time a new client establishes a Socket.io connection.
- Inside this listener, you can define various events (e.g., `createLobby`, `joinLobby`, `playerReady`, etc.), each with its own logic (like creating a new lobby, joining an existing lobby, updating game state, etc.).

Finally, the `listen()` method actually starts the HTTP server on the configured port and logs a confirmation message to the console.

Each lobby can be associated with an instance of **Game**, where puzzle and quiz logic resides. Thus, each group of users can operate independently in their own “virtual room,” while the server architecture remains scalable and capable of handling multiple simultaneous matches.

Joining a Lobby and Creating a Room

When a client creates or joins a lobby, the server assigns a unique lobby code to the lobby and sets this code (along with the client’s nickname) on the socket. For example, within the [User.joinLobby](#) method:

```
joinLobby(lobby) {
  this.lobbyCode = lobby.code;
  // Associate the lobby code and nickname with the socket
  this.socket.lobbyCode = lobby.code;
  this.socket.nickname = this.nickname;
  lobby.addUser(this);           // Add the user to the lobby's list
  this.socket.join(lobby.code); // Join a Socket.IO room named by the lobby code
}
```

Here, the call to `this.socket.join(lobby.code)` is key. It tells Socket.IO to place the client's socket into a room identified by the lobby code. This room acts as a subgroup of connections that are part of that specific lobby.

Once sockets are grouped into a room, sending messages to all users within that lobby is straightforward. For example, when an event such as a game update occurs, the server can broadcast the event only to those sockets in the specified lobby:

```
socket.broadcast.to(data.lobbyCode).emit("tileMoved", data);
```

This ensures that only the clients whose sockets belong to the room with the corresponding lobby code receive the message.

When a user leaves the lobby—either by clicking a logout button, navigating back, or disconnecting unexpectedly—the socket is removed from its room. The server then updates the lobby's user list and may remove the lobby if it becomes empty.

```
socket.on('leaveLobby', () => {
  const lobbyCode = socket.lobbyCode;
  if (lobbyCode && this.lobbies[lobbyCode]) {
    const lobby = this.lobbies[lobbyCode];
    lobby.removeUser({ nickname: socket.nickname });
    socket.leave(lobbyCode);
    // Remove the lobby if no users remain
    if (lobby.users.length === 0) {
      delete this.lobbies[lobbyCode];
    }
  }
});
```

Similarly, when a socket disconnects unexpectedly (for example, when a user closes their browser window), the disconnect event handler handles removal in a similar fashion.

Reconnection Implementation

The reconnection mechanism has been designed to allow players to temporarily disconnect and later rejoin an active game without losing their game state. The system leverages both client-side storage (using `localStorage`) and server-side tracking of disconnected users.

When a player disconnects (either by closing the browser or navigating away), the server's disconnect event is triggered. Instead of immediately removing the player from the lobby, the system invokes the *checkInsufficientPlayers* function. In this function, if the active players are still sufficient, the server sets a 30-second timer before permanently removing the user.

When the client comes back, it retrieves the saved old socket id from *localStorage* and emits the *checkActiveGame* event with that id. The server then checks if a record exists. If the game is in progress, the server clears the removal timer, removes the old record from *disconnectedUsers*, and updates the user's record in the lobby with the new socket. Finally, the server emits a *reconnectAllowed* event containing game details.

Self-assessment / Validation

The self-assessment process verified each aspect of the server's functionality to ensure compliance with the intended requirements in a distributed environment. First, the ability to create and join lobbies was tested, confirming that unique lobby codes were generated and that real-time updates were issued to all connected users. Chat functionality within each lobby was validated by observing that participants promptly received each other's messages through Socket.io. The "player ready" feature was inspected, ensuring that changing a user's state triggered server-wide updates and initiated the game only after every connected client was marked as ready. Additionally, core game logic was examined by assigning players to distinct rooms, verifying quiz answers against stored solutions, and emitting a "gameWon" event only once the necessary quizzes had been solved. The sliding puzzle mechanic was scrutinized to confirm accurate synchronization of tile movements among all clients and the broadcasting of a victory announcement when the puzzle was completed. Error handling was tested by attempting to join full lobbies, using duplicate nicknames, and referencing invalid lobby codes, and the server was confirmed to send appropriate error messages in each scenario. Scalability and concurrency were assessed by running multiple lobbies concurrently with multiple test clients, ensuring each lobby remained isolated and responsive. Console messages were used for logging and monitoring, capturing significant events such as new connections or user disconnections, and users who disconnected unexpectedly were correctly removed from their lobbies, leading to lobby deletion if they were the last member. This comprehensive testing demonstrated stable performance, accurate synchronization, and reliable

communication features, indicating that the system is suitably prepared for a real-time, distributed Escape Room experience.

Deployment Instructions

In the development of the project, it was ensured that the deployment process is flexible enough to work seamlessly both in a local development environment and on a remote production server.

Dependencies and Versions

Below are the main dependencies and recommended versions used in this project:

- **Node.js:** v18.x
- **Express:** ^4.18.2
- **Socket.io:** ^4.6.1
- **React:** ^18.2.0
- **Phaser:** ^3.60.0

Local Deployment

The project is configured to run locally using Node.js (version 18.x), with the backend powered by Express (version 4.18.2) and real-time communication managed through Socket.io (version 4.6.1). On the frontend, React along with Phaser is utilized. To run the application locally, the developer first installs dependencies for both server and client by executing *npm install* in the root directory and inside the *client/* folder. The Node.js server, located at *server/server.js*, listens by default on port *3001*, as defined in the server code with *const PORT = process.env.PORT || 3001*. The React frontend, typically initiated via the command “*npm start*” from within the *client/* directory, will run on its default port (*3000*) and connect to the backend server via Socket.io, automatically resolving to *localhost:3001*. This configuration allows seamless communication between client and server in the local development environment without further modifications.

Remote Deployment

For remote deployment, such as deploying to Heroku, it was configured the application to dynamically handle server port and client connections. The Node.js server retrieves the port number automatically from the environment variable *process.env.PORT* provided by Heroku at runtime, rather than using a fixed port. The server listens on this dynamically assigned port, ensuring compatibility with Heroku's hosting environment.

When the deployment is completed, Heroku automatically sets the *NODE_ENV* environment variable to "production" and dynamically assigns a port to the Node.js server through *process.env.PORT*.

```
const PORT = process.env.PORT || 3001;
server.listen(PORT, () => {
  console.log(`Server listening on port ${PORT}`);
});
```

On the client side, Socket.io connections are handled using relative URLs to avoid hardcoded references:

```
import { io } from 'socket.io-client';

const URL =
  process.env.NODE_ENV === 'production' || window.location.hostname !==
  'localhost'
    ? '/' // It use relative address in production
    : 'http://localhost:3001'; // It use localhost during development

const socket = io(URL);
```

In this setup, the client application and assets are served by Express from the client/build directory. Static assets such as images are correctly served by configuring static routes within Express:

```
app.use('/assets', express.static(path.join(__dirname, 'public/assets')));
```

This configuration ensures the client automatically connects to the correct server IP and port, as the address is dynamically resolved based on the URL provided by Heroku (e.g., <https://your-app-name.herokuapp.com>). Thus, the deployment environment remains flexible, scalable, and easily maintainable.

Usage Example

Players can exchange messages in the lobby until the game starts. When everyone is ready, then the game starts.

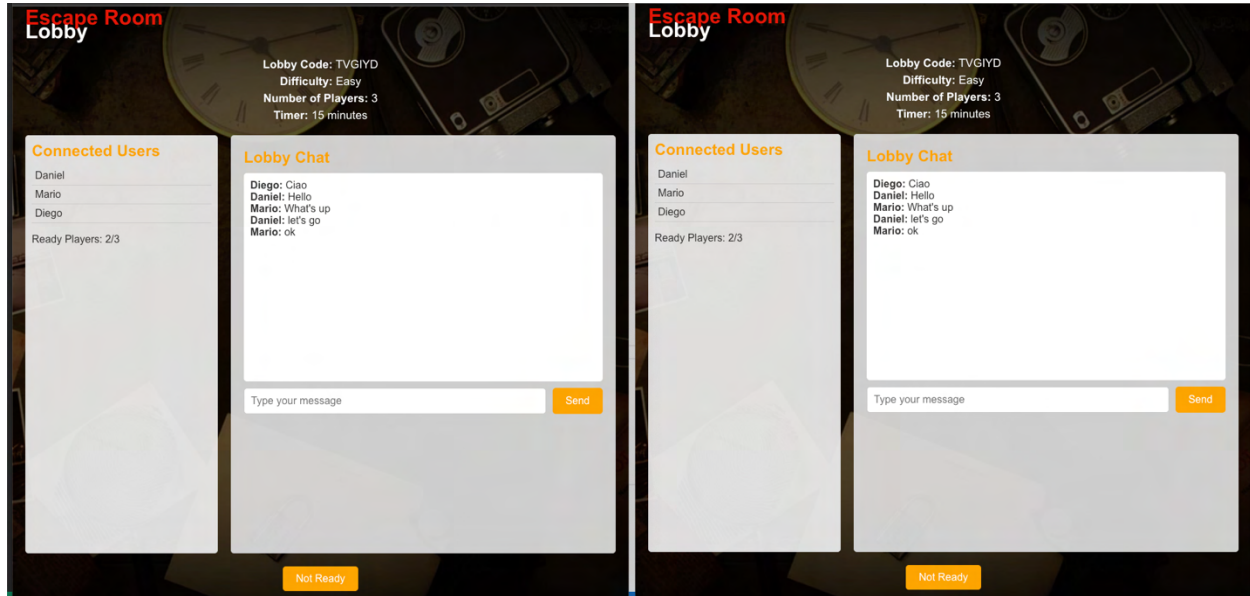


Figure 11: Lobby

In the first level they are divided into 2 rooms, where one player takes care of sending the solutions, and the others remain to complete the puzzles and quizzes.

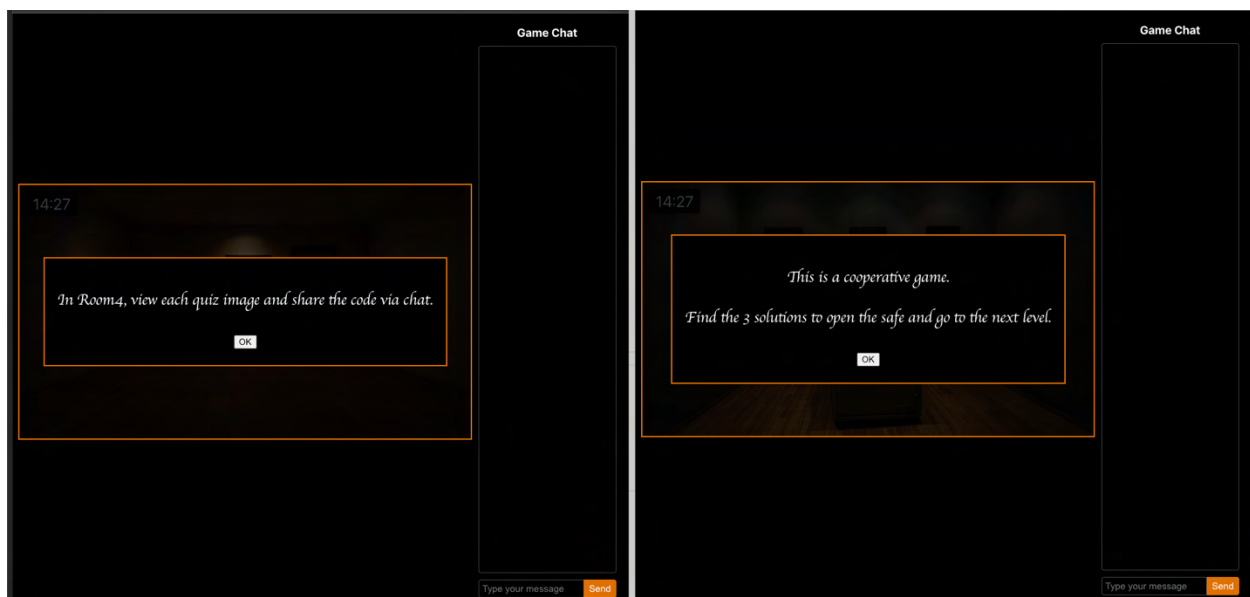


Figure 12: Divided into rooms

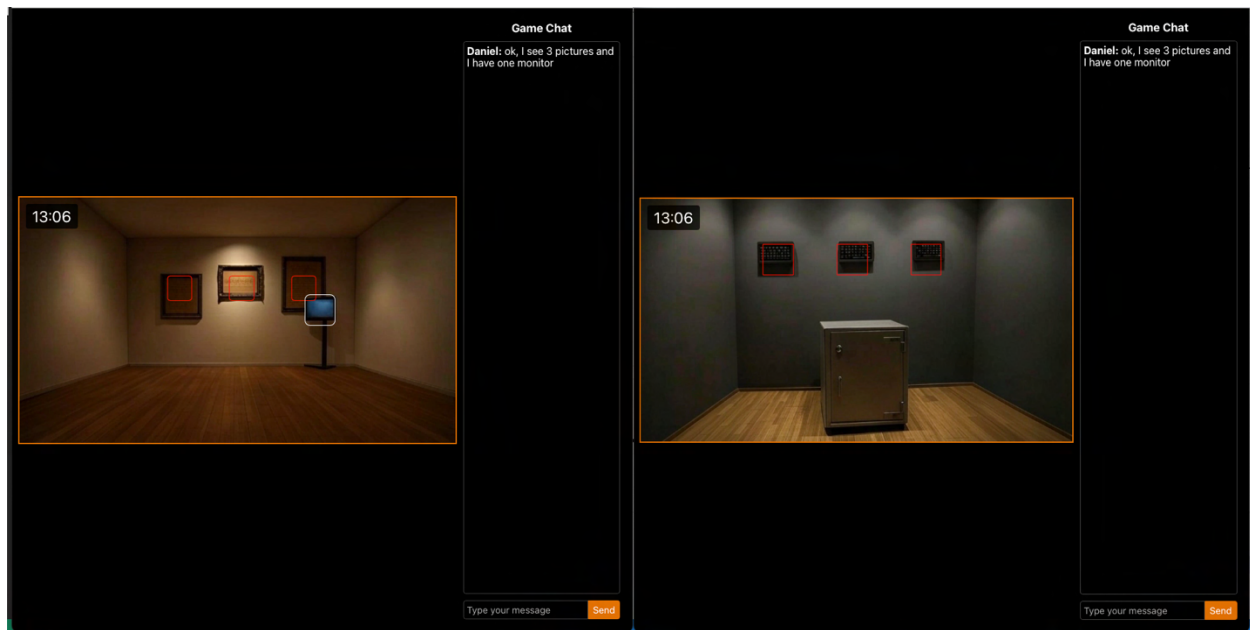


Figure 13: Game

Puzzles can be played synchronously by all the players in the room and the effects of the movements can be seen live, but players can divide the tasks to be quicker in completing everything in the time limit.

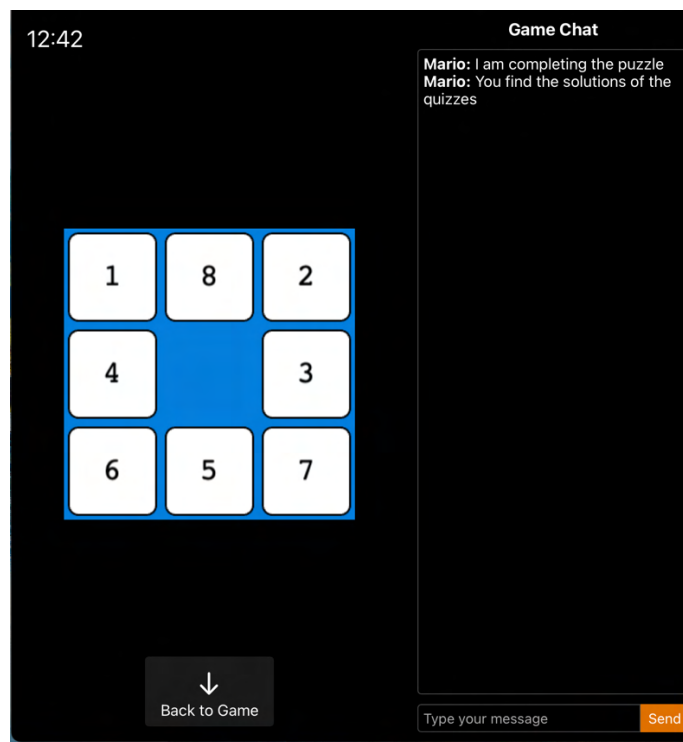


Figure 14: Puzzle

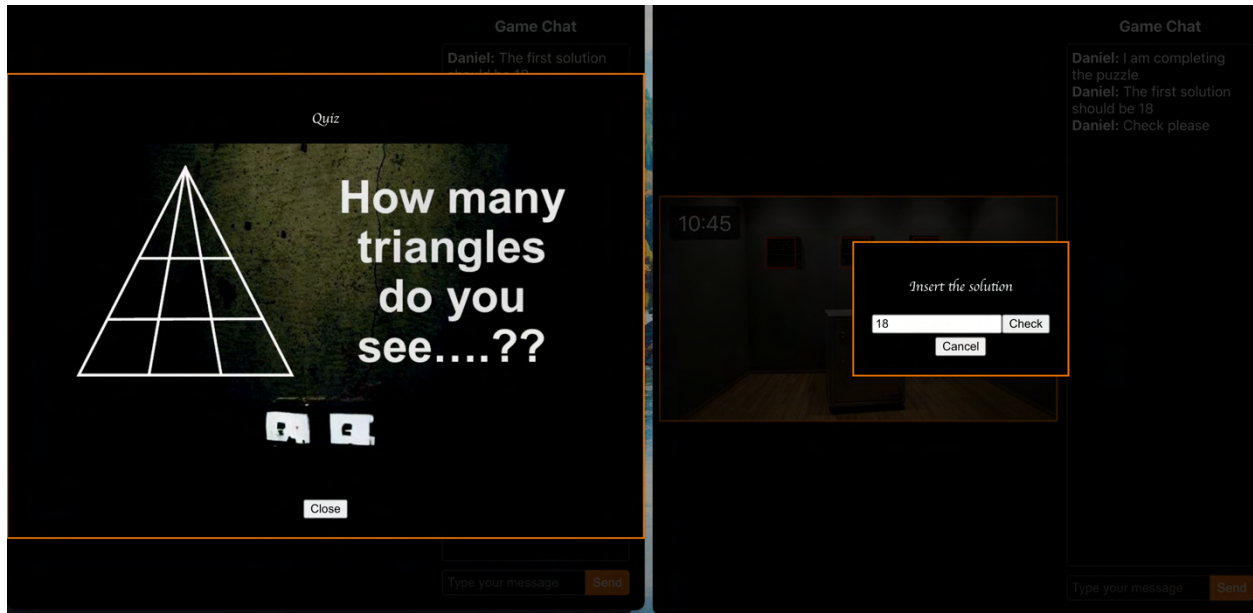


Figure 15: Check solution to the quiz

Once all the quizzes and puzzles have been solved, then the level is passed, and you move on to the next one.

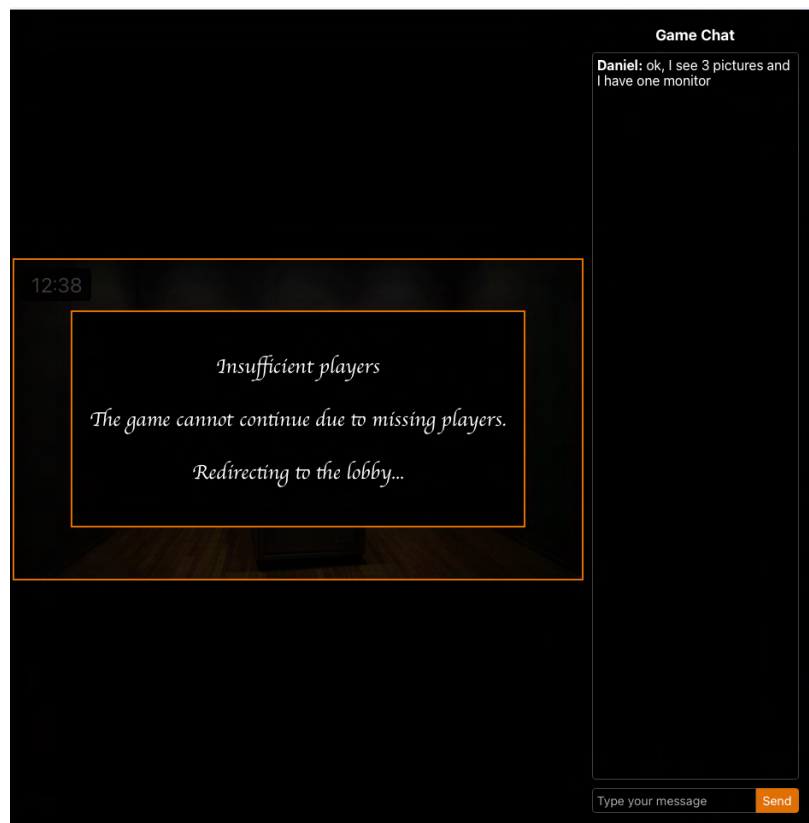


Figure 16: There are insufficient players

Conclusions

The Multiplayer Escape Room Game project has demonstrated the viability of building a real-time, event-driven application using web technologies such as Node.js, Express, Socket.IO, and React. Throughout the development process, it was built a system that manages lobbies, real-time chat, dynamic room assignments, and interactive puzzles. This project not only provides a fun and engaging user experience but also serves as a case study in managing state and communication in distributed systems.

Looking forward, the game can be expanded in several ways. Future enhancements might include the integration of a user registration system, allowing players to create accounts and save their progress. Additional levels with more complex puzzles and diversified game mechanics could further enrich the gameplay experience. Moreover, the incorporation of persistent storage—such as a database for player statistics and game history—would enable long-term data retention and personalization.

From a distributed systems perspective, this project has taught the importance of handling real-time communication, fault tolerance, and scalability. In addition, it taught that technologies such as Socket.IO, when paired with a well-designed back-end architecture, can effectively support many simultaneous connections through features such as heartbeat, automatic reconnection, and horizontal scaling using tools such as Redis adapters. Additionally, challenges encountered in managing a shared status among distributed nodes provided a valuable insight into the complexities of distributed systems.

Overall, this project represents both a technical achievement and a foundation for further innovation in interactive, real-time gaming experiences.