Mancala Game

Final Report for the Distributed Systems Course 2024/2025

Kasra Sabertehrani
kasra.sabertehrani@studio.unibo.it
Parisa Sokuti
parisa.sokuti@studio.unibo.it

April 15, 2026

This report presents the design and implementation of a distributed Mancala game system built using Spring Boot and WebSockets. The system enables two players to play the classic Mancala board game over a network in real-time. The primary focus is on handling distributed challenges such as player disconnection, session management, inactivity detection, and state consistency across multiple clients. The implementation demonstrates core distributed systems concepts including asynchronous communication, fault tolerance, and timeout mechanisms. The game is implemented with a clean separation between domain logic (game rules), service layer (business logic), and presentation layer (WebSocket controllers). Key features include automatic room creation, player joining, real-time game state synchronization, graceful disconnection handling with reconnection support, and inactivity-based player forfeit mechanisms.

Disclaimer

During the preparation of this work, the author(s) used GitHub Copilot to assist with code generation and review. After using this tool, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the final report/artifact.

1 Concept

1.1 Product Type

This project is a **web-based, real-time multiplayer game application** combining:

- **Backend**: RESTful API with WebSocket support for real-time communication

- **Frontend**: HTML/CSS/JavaScript web interface for player interaction
- **Game Engine**: Java-based Mancala game logic implementation

1.2 Why is distribution needed?

Distribution is essential for this project for the following reasons:

- **Geographical separation**: Players must be able to play from different locations across a network
- **Real-time synchronization**: Game state must be consistent across multiple clients; any move by one player must be immediately visible to the other
- **Fault tolerance requirement**: Network failures are inevitable; the system must handle player disconnections gracefully and allow reconnection without losing game state
- **Concurrent interactions**: Multiple games must be able to run simultaneously on the same server, requiring proper concurrency control
- **Scalability consideration**: Although not a focus of this project, the distributed design allows multiple game instances to be load-balanced across servers

2 Requirements Elicitation and Analysis

2.1 Functional Requirements

1. **FR1: Game Room Management**
   - The system must allow a player to create a new game room
   - **Acceptance Criterion**: A player can create a room, receive a unique room ID, and join as Player 1
2. **FR2: Player Joining**
   - A second player must be able to join an existing room using its ID
   - **Acceptance Criterion**: Player 2 can join a room with Player 1, and the game automatically starts
3. **FR3: Game Rule Enforcement**
   - The system must enforce all Mancala game rules including stone sowing, capturing, and turn management
   - **Acceptance Criterion**: Moves are validated; invalid moves are rejected; game logic is correctly executed
4. **FR4: Turn-based Gameplay**

- The system must enforce strict turn-based gameplay where only the current player can make a move
- **Acceptance Criterion**: Non-active players cannot make moves; turns automatically switch after valid moves

5. **FR5: Winner Determination**
   - The system must determine the winner when the game ends naturally or by forfeit
   - **Acceptance Criterion**: Winner is correctly calculated based on scores or disconnect status
6. **FR6: Real-time State Synchronization**
   - All game state changes must be immediately broadcast to both players
   - **Acceptance Criterion**: Both players see the same game state at all times; updates arrive within 100ms
7. **FR7: Disconnection Handling**
   - When a player disconnects, the game should pause and allow reconnection within a grace period
   - **Acceptance Criterion**: Disconnected player can rejoin; game resumes from disconnection point
8. **FR8: Inactivity Detection**
   - If a player is inactive for a specified timeout, the opponent automatically wins
   - **Acceptance Criterion**: Inactive player is declared forfeit after timeout
9. **FR9: Room Cleanup**
   - When a game ends (naturally or by timeout), the room and its resources must be cleaned up
   - **Acceptance Criterion**: Room is removed from active rooms; no resource leaks occur
10. **FR10: Error Handling**
    - Invalid operations must return meaningful error messages to the client
    - **Acceptance Criterion**: Clients receive error messages describing why operations failed

2.2 Non-functional Requirements

1. **NFR1: Consistency**
   - Game state must be consistent across all clients at all times

- Target: Eventual consistency achieved within 100ms

2. **NFR2: Availability**
   - The system should remain available even if one player temporarily disconnects
   - Target: 99% availability during active games (excluding network failures)
3. **NFR3: Fault Tolerance**
   - The system must gracefully handle network failures and player disconnections
   - Target: Detect disconnection within 30 seconds; allow reconnection within 5 minutes

2.3 Implementation Requirements

1. **IR1: Programming Language**: Java 17
   - **Justification**: Requirement of Spring Boot framework; strong typing and mature ecosystem for distributed systems
2. **IR2: Backend Framework**: Spring Boot 4.0.3
   - **Justification**: Provides built-in WebSocket support, dependency injection, and scheduling capabilities
3. **IR3: Real-time Communication**: WebSocket (STOMP protocol)
   - **Justification**: Enables low-latency, bidirectional communication; essential for real-time game synchronization
4. **IR4: Frontend**: HTML5, CSS3, JavaScript (Vanilla)
   - **Justification**: No external framework required for this simple use case; reduces dependencies
5. **IR5: Build Tool**: Maven
   - **Justification**: Provided with Spring Boot; standard for Java projects

2.4 Glossary of Domain Terms

- **Pit**: A small cup on the Mancala board containing stones
- **Store**: A larger pit at the end of each player's side, used to collect captured stones
- **Sowing**: The action of distributing stones from a selected pit around the board
- **Capture**: When a stone lands in an empty pit on a player's side, allowing the opponent's stones in the opposite pit to be captured
- **Turn**: One player's opportunity to select and sow from a pit

- **Game Room**: A virtual space containing exactly two players and one active game
- **Room ID**: A unique identifier for a game room (e.g., "1", "2")
- **Player ID**: A unique identifier for a player (UUID generated server-side)
- **WebSocket Session**: A persistent bidirectional connection between client and server
- **Disconnection**: Loss of WebSocket connection due to network failure or client action
- **Reconnection**: Re-establishing a WebSocket connection after disconnection
- **Forfeit**: Automatic win granted to opponent due to inactivity or disconnection timeout

2.5 Relevant Distributed System Features

2.5.1 Transparency

Relevance: **PARTIAL**

The system does NOT attempt to hide distribution from the user:

- Users are aware they are playing over a network
- If a player disconnects, the system explicitly notifies both players
- Reconnection is a visible, user-initiated action

However, location transparency is achieved: users don't need to know the physical location of the server.

2.5.2 Fault Tolerance and Dependability

Relevance: **CRITICAL**

This is a primary focus of the system:

- **Availability**: The system remains available even if one player temporarily disconnects
- **Reliability**: Games do not lose state due to network failures
- **Integrity**: Game state cannot be corrupted by out-of-order messages or duplicate messages
- **Graceful degradation**: System continues to function with one player disconnected, waiting for reconnection

Implementation mechanisms:

- Timeout and recovery behavior are summarized in the Design section and detailed in Fault-Tolerance
- Synchronization and shared-state control are detailed in Design (Data and Consistency Issues)
- Concrete implementation examples are provided in Implementation (Timeout Mechanisms)

3 Design

This section describes how the requirements are met through architectural and design choices. The design is technology-agnostic and focuses on strategies rather than implementation details.

3.1 Architecture

3.1.1 Architectural Style

The system follows a **Client-Server Architecture with Event-Driven Communication**:

- **Server-side**: Single authoritative game server maintains game state and enforces rules
- **Client-side**: Thin clients (browsers) for input and display only
- **Communication**: Asynchronous event-based (WebSocket) for real-time updates

This architecture was specifically chosen to address the core challenges of distributed multiplayer game state management. A centralized, authoritative server acts as the single source of truth, serializing concurrent requests to prevent race conditions and ensuring strong state consistency between both players. Furthermore, by relying on thin clients, the system inherently prevents client-side cheating and maintains a strict separation of concerns, cleanly decoupling the pure game domain from the frontend presentation.

3.2 Infrastructure

To support the real-time, stateful nature of the game, the infrastructure is designed around a minimal, centralized topology that manages all coordination between clients.

3.2.1 Infrastructural Components

1. **Clients (Web Browsers)**: The edge components where users interact. Exactly two clients connect per active game room.

2. **Spring Boot Server (Single Instance)**: The central authoritative node hosting several integrated sub-components:
    - **REST APIs**: Handles initial stateless HTTP requests for room creation and joining.
    - **WebSocket Broker**: Spring's built-in message broker that routes real-time game state broadcasts to subscribed clients.
    - **Session Tracker**: An in-memory component that maps active WebSocket session IDs to specific players and game rooms.
    - **WebSocket Event Listener**: An interceptor that monitors connection lifecycles to immediately detect ungraceful TCP drops.
    - **Timeout Scheduler**: A background worker process that periodically evaluates inactive rooms to enforce reconnection limits and forfeit rules.
    - **In-Memory Database**: Utilizes concurrent data structures ConcurrentHashMap acting as an embedded database to securely store active game rooms, board states, and session mappings

3.2.2 Network Distribution

For the scope of this project, the system utilizes a localized, monolithic distribution model:

- **Server Location**: The entire backend (including the broker, tracker, and scheduler) runs as a single process on a single host machine (deployed to `localhost`).
- **Client Location**: Clients act as remote nodes communicating with the central server over the local network loopback interface.

3.2.3 Component Discovery and Naming

Because the infrastructure relies on a single, self-contained server, complex service discovery mechanisms and load balancers are strictly avoided to reduce overhead:

- **Client-to-Server Discovery**: Clients locate the server using statically configured hostnames and ports mapped via local DNS (e.g., `http://localhost:8080` and `ws://localhost:8080/mancala-ws`).
- **Internal Discovery**: Server-side components (REST controllers, Event Listeners, Schedulers) reside entirely within the same Java Virtual Machine (JVM).

This component diagram maps the physical runtime architecture of the system. It delineates the strict boundary between the client-side web application and the central Spring Boot server.

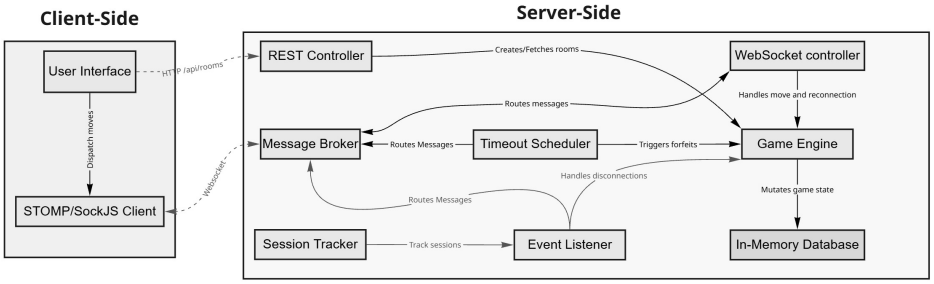

Figure 1: UML Component Diagram mapping client and server infrastructure

3.3 Modelling

3.3.1 Domain Entities

The system models four primary domain entities:

1. **Game**: Represents a single game instance
   - Encapsulates game rules and state transitions
   - Manages both players and the board
   - Determines winners and enforces turn order
2. **Board**: Represents the Mancala board
   - 14 pits (6 per player + 1 store each)
   - Handles stone sowing, capturing, and scoring
   - Completely decoupled from players or game status
3. **Player**: Represents a player in the game
   - Server-assigned UUID for identification
   - Player name provided by the user
   - No mutable state (purely identity)
4. **GameRoom**: Container for a game instance
   - Groups a game with its room ID
   - Minimal business logic; mostly a data holder

3.3.2 Mapping Domain Entities to Infrastructure

To bridge the gap between physical network communication and abstract game rules, the system employs a strict flow of data between infrastructural components and domain entities.

Interaction Flow (Mapping Domain to Infrastructure):

1. **Input Processing**: Client actions arrive at the server's infrastructure layer via REST API calls or WebSocket STOMP messages.
2. **Service Delegation**: The controllers map these network requests to the application services. The `RoomService` handles structural events (creating or joining a room), while the `GameService` orchestrates active gameplay (making a move) and network interruptions (disconnections and reconnections).
3. **Domain Mutation**: The services retrieve the target `GameRoom` from the in-memory database. The `GameRoom` passes the requested action down to the `Game` entity.
4. **Entity Delegation**: For physical game mechanics (e.g., sowing stones and capturing), the `Game` delegates the calculation to the `Board`. For meta-game state transitions (e.g., pausing the match due to a dropped WebSocket), the `Game` processes the logic internally.
5. **Output Broadcasting**: Once the domain entities have successfully mutated and stabilized the state, the infrastructure (WebSocket Broker or REST API) serializes the updated `GameRoom` and delivers it to the frontend clients to synchronize their local display states.

3.3.3 Domain Events

The system recognizes the following domain events:

1. **RoomCreated**: Player 1 creates a new game room
2. **PlayerJoined**: Player 2 joins an existing room, game starts
3. **MoveMade**: A player selects a pit and makes a valid move
4. **TurnSwitched**: The turn automatically changes to the other player
5. **PlayerDisconnected**: A player's WebSocket connection is lost
6. **PlayerReconnected**: A disconnected player re-establishes connection
7. **ReconnectionTimeoutExpired**: Grace period for reconnection expires; opponent wins
8. **PlayerInactive**: A player makes no moves for the inactivity timeout duration
9. **GameEnded**: Game ends naturally (all pits on one side empty)
10. **RoomCleaned**: Room is removed from active rooms

3.3.4 Message Types

The system exchanges two distinct categories of messages based on the communication protocol in use:

1. **HTTP REST Messages (Synchronous)**

- **Room Creation Command** (Client → Server):
  - `POST /api/rooms/create`
  - **Response**: Unique Room ID and generated Player ID (200 OK)
- **Room Join Command** (Client → Server):
  - `POST /api/rooms/join`
  - **Response**: Generated Player ID (200 OK) or Error if full/not found (400/404)

2. **WebSocket STOMP Messages (Asynchronous)**

1. **Commands** (Client → Server):
   - `/app/game.move`: Execute a move (pit index, room ID, player ID)
   - `/app/game.join`: Announce presence to start game (room ID, player ID)
   - `/app/game.reconnect`: Attempt to resume paused game (room ID, player ID)
2. **Events** (Server → Client, broadcast):
   - `Game state update`: Broadcasted to `/topic/room/{roomId}`
   - **Contains**: Player names, pit states, current turn, game status, scores, winner, disconnection info
3. **Errors** (Server → Client, point-to-point):
   - `/queue/errors`: Invalid moves, room not found, invalid state errors

This class diagram below illustrates the core domain model of the Mancala game engine. It highlights the primary entities—GameRoom, Game, Board, Player, and Pit—along with their structural relationships.

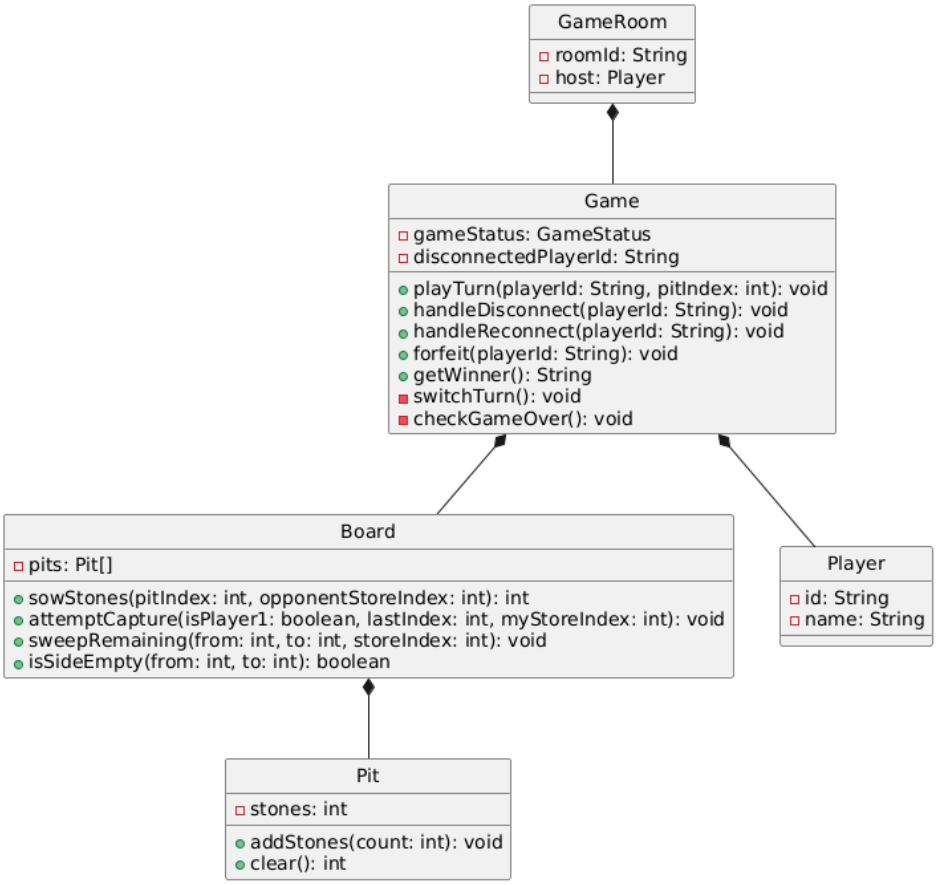

Figure 2: UML class Diagram

3.4 Interaction

3.4.1 Communication Patterns

The system employs two main communication patterns:

1. Request-Response (HTTP)

Used for the initial stateless game room creation and joining phases. The following sequence diagram illustrates how players obtain their unique IDs and room assignments before establishing a persistent WebSocket connection:

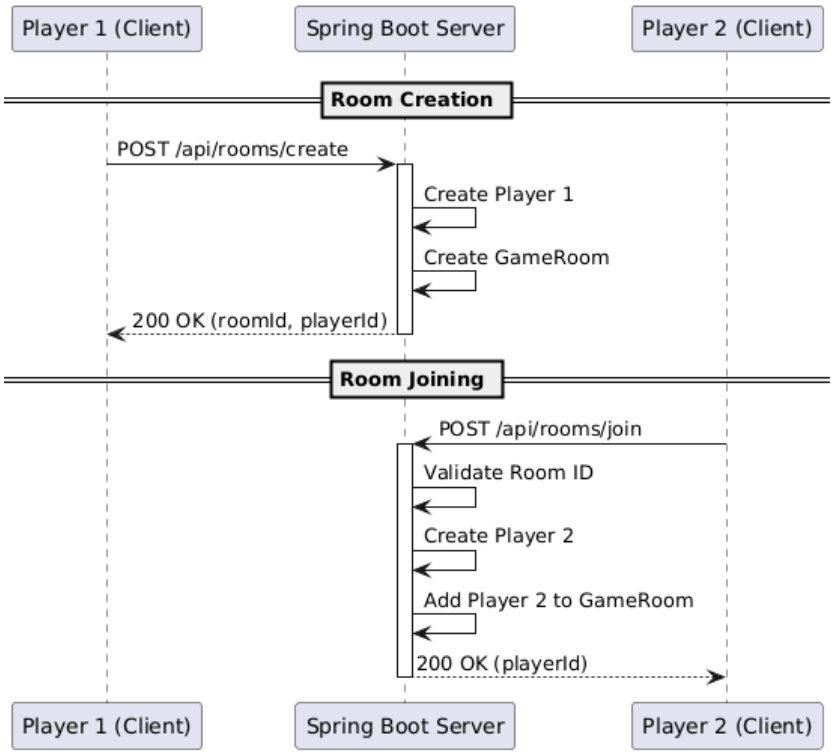

Figure 3: UML Sequence Diagram of HTTP REST Room Setup

2. Event-Driven (WebSocket)

This sequence diagram details the real-time, asynchronous event flow between the connected clients and the central authoritative server. It visualizes the initial WebSocket handshake followed by the primary gameplay loop, demonstrating how client move requests are routed, validated by the server-side engine, and subsequently broadcast back to all participants to ensure state consistency.

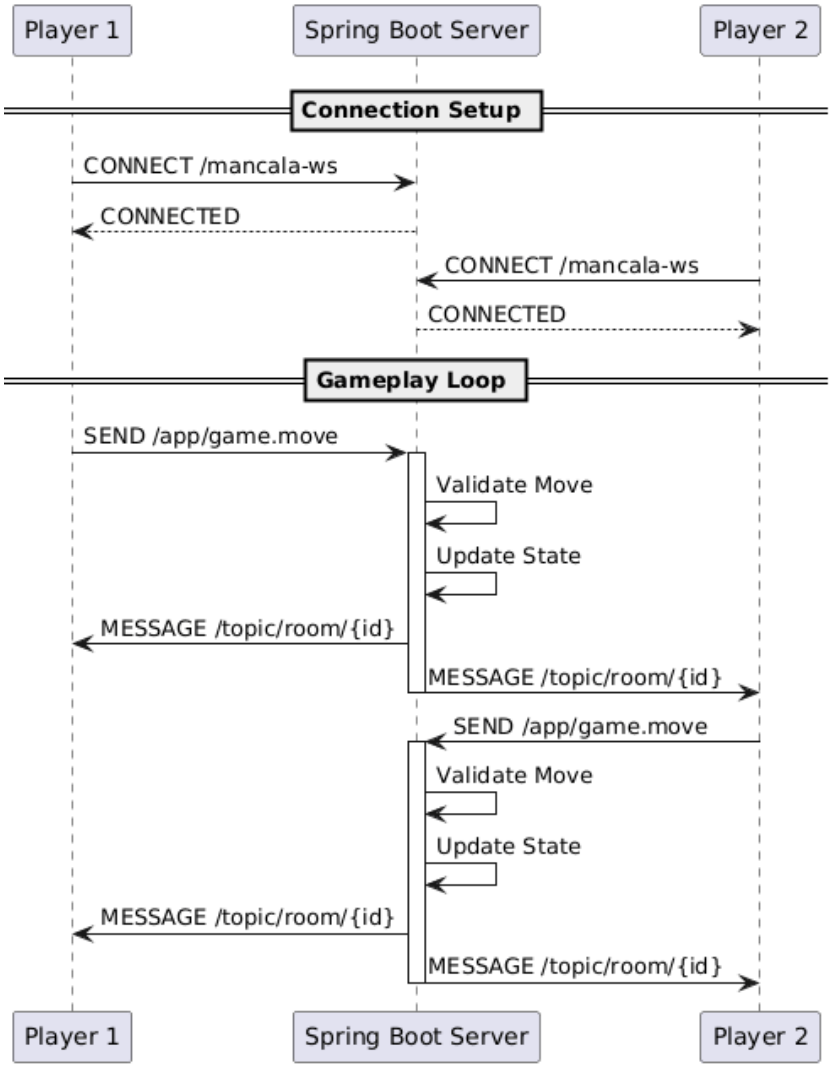

Figure 4: Game flow Sequence Diagram

3.4.2 Timeout Mechanism

```
Event: Player Disconnects
|
+- Detect disconnect and pause game
+- Start reconnection grace timer
+- If player reconnects in time - resume
+- Otherwise - forfeit, cleanup, broadcast final state
```

Detailed timeout values and scheduler behavior are specified in Fault-Tolerance and Implementation.

3.5 Behaviour

3.5.1 Game State Machine

The game transitions through discrete states as follows:

State: WAITING_FOR_PLAYER_2

- **Entry**: Player 1 creates a room
- **Actions**: Server waits for Player 2 to join
- **Transitions**:
  - - PLAYER_1_TURN when Player 2 joins
- **Constraints**: Only Player 1 exists; no moves allowed

State: PLAYER_1_TURN

- **Entry**: Game starts or after Player 2's move
- **Actions**: Player 1 can select a pit and move
- **Transitions**:
  - - PLAYER_1_TURN if Player 1 lands in their store (extra turn)
  - - PLAYER_2_TURN if Player 1's turn ends normally
  - - PAUSED_FOR_RECONNECT if Player 1 disconnects
  - - GAME_OVER if game ends

State: PLAYER_2_TURN

- **Entry**: After Player 1's turn ends (normal transition)
- **Actions**: Player 2 can select a pit and move
- **Transitions**: (Symmetric to PLAYER_1_TURN)

State: PAUSED_FOR_RECONNECT

- **Entry**: A player's WebSocket disconnects during an active turn
- **Actions**: Server remembers the previous state and waits for reconnection
- **Transitions**:
  - - PLAYER_1_TURN or PLAYER_2_TURN if disconnected player reconnects (resume)
  - - PLAYER_DISCONNECTED if reconnection timeout expires
- **Constraints**: No new moves allowed; waiting player cannot act

State: PLAYER_DISCONNECTED

- **Entry**: Reconnection timeout expires without player reconnecting
- **Actions**: Game is forfeit; opponent automatically wins
- **Transitions**:
  - - GAME_OVER (final state)

State: GAME_OVER

- **Entry**: Game ends naturally (board empty on one side) or by forfeit
- **Actions**: Calculate winner; return game to clients
- **Transitions**: (Terminal state)
- **Cleanup**: Room is removed after broadcast

This state machine diagram outlines the lifecycle of a single game session. It illustrates the discrete states the system transitions through—from the initial waiting phase to active turns—and explicitly models the fault-tolerance paths, showing how the system pauses for temporary disconnections and transitions to a forfeit state if the recovery timeout expires.

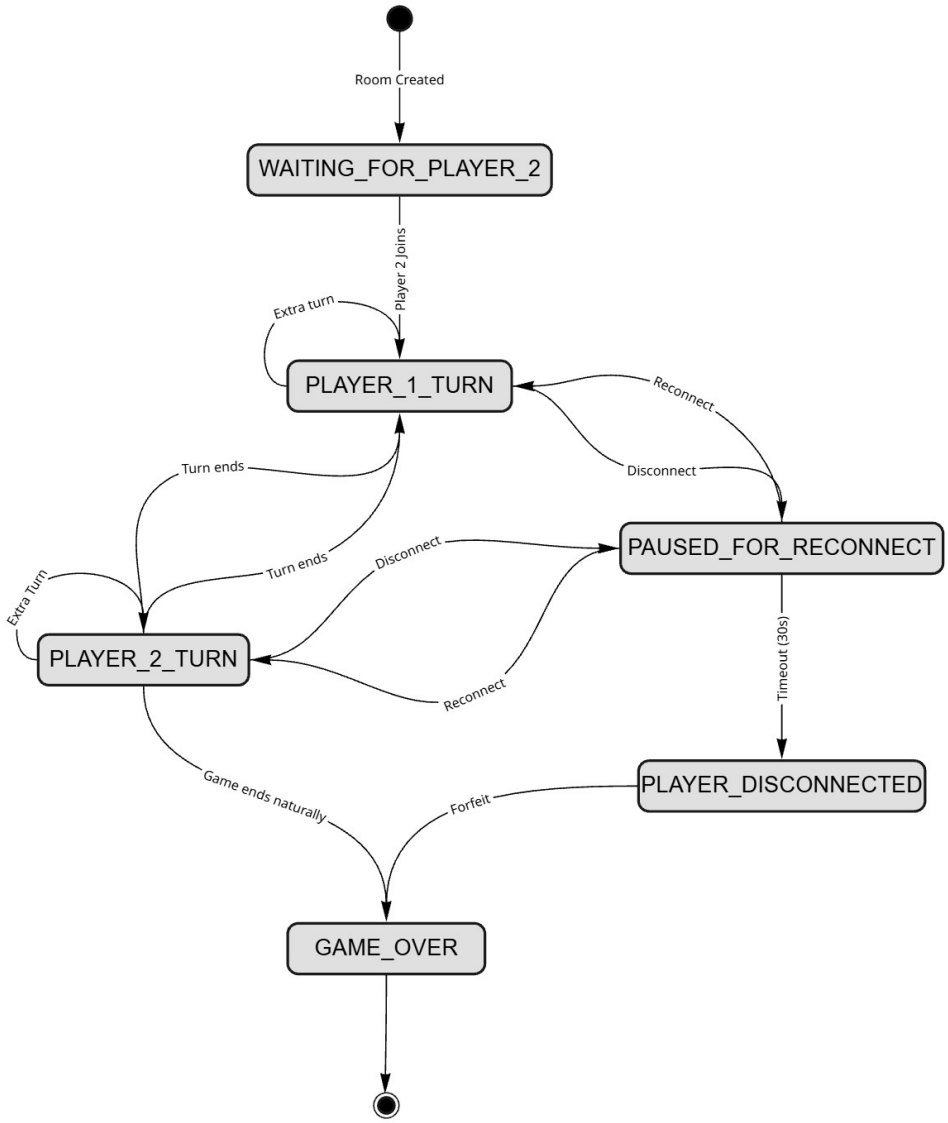

Figure 5: Game State Machine

3.6 Data and Consistency Issues

3.6.1 Data Storage

What data is stored?

- Game state: Pit values, turn information, game status
- Player information: Names, UUIDs
- Room information: Room IDs, room - game mappings

- Session information: WebSocket session IDs - player/room mappings
- Temporal information: Activity timestamps, disconnection timestamps

Where is it stored?

- All data in server's Java heap memory
- Two primary structures: `activeRooms` (ConcurrentHashMap) and `sessionSessions` (ConcurrentHashMap)
- No persistent storage (no database)

3.6.2 Consistency Model

The system implements **Strong Consistency** for game state:

- At any point in time, there is exactly one version of the game state
- All moves are applied sequentially (never in parallel)
- Before returning a move result, the new state is immediately broadcast to all clients
- Clients do not make independent decisions; they always display server-truth

Consistency guarantees:

- No lost updates: Each move is persisted in server memory immediately
- No dirty reads: Clients only read state after server confirms
- No race conditions: Synchronization prevents concurrent state modifications
- Ordered updates: Messages are delivered in the order they were sent

3.6.3 Query Operations

Who queries the database?
Since there is no persistent database, we consider in-memory queries:

- **RoomService**: Queries activeRooms map when retrieving/joining/removing rooms
- **GameService**: Queries rooms when processing moves and timeouts
- **SessionTracker**: Queries session mappings when tracking/removing sessions

What queries?

- Get room by ID: O(1) hash lookup

- Get all active rooms: O(n) iteration (rarely needed)
- Check session membership: O(1) hash lookup

When?

- On every move: 1ms processing time
- On disconnection event: Immediate
- On timeout check: Every 5 seconds (scheduled)

3.6.4 Shared State Management

The following data structures are shared between multiple components:

- **GameRoom**: Shared between RoomService (creator), GameService (updater), WebSocketEventListener (accessor)
- **Session mappings**: Shared between GameController (writer) and WebSocketEventListener (reader)
- **Activity timestamps**: Shared between GameService (writer) and GameTimeoutScheduler (reader)

Thread safety is ensured through:

- ConcurrentHashMap for all shared mutable collections
- Synchronized blocks around complex multi-step operations
- Immutable value objects (Player, Pit are largely immutable)

3.7 Fault-Tolerance

3.7.1 Heartbeat and Timeout Mechanisms

WebSocket-level heartbeat:

- SockJS (WebSocket fallback) provides automatic heartbeating
- Server detects client disconnection within a few seconds
- Triggered by network issues or client going offline

Application-level timeouts:

1. **Reconnection grace period**: 30 seconds (resume if player returns; otherwise forfeit)
2. **Inactivity timeout**: 60 seconds (current player forfeits when idle beyond threshold)
3. **Scheduler check interval**: 5 seconds (periodically evaluates both timeout rules)

3.7.2 Error Handling

Move validation errors:

```
Invalid pit index -- IllegalArgumentException
  |
GameController.@MessageExceptionHandler
  |
Send error to client via /queue/errors
  |
Client displays error message
```

Game state errors:

- Move on wrong turn: Throw IllegalStateException
- Move on game over: Throw IllegalStateException
- Invalid player ID: Throw IllegalArgumentException

Room not found errors:

- Room doesn't exist: Return HTTP 404
- Room is full: Return HTTP 400
- Proper error messages sent to client

3.7.3 Recovery Strategy

From temporary disconnection (¡ 30 seconds):

- **Server-Side:** The server pauses the game state and waits for a STOMP reconnect payload containing the original Player ID.
- **Client-Side Recovery:** The frontend utilizes both `localStorage` (to save the room and player IDs) and `sessionStorage` (to verify if the player was actively in a match within that specific tab). If a user accidentally refreshes the page, the JavaScript `window.onload` event intercepts the load, reads the `localStorage` payload, bypasses the lobby screen, and automatically fires an `/app/game.reconnect` message to the server to instantly resume the session.

From permanent disconnection (30 seconds):

- Opponent is declared winner
- Room is cleaned up
- Other player is notified
- Disconnected player sees game over state if they reconnect late

3.8 Availability

3.8.1 Caching

Client-side caching:

- Player ID and room ID stored in browser localStorage
- Used to attempt reconnection to the same game
- Reduces need to re-enter information after page refresh

Server-side caching:

- No caching; all state is in-memory anyway
- No database to cache query results

3.8.2 Load Balancing

Not implemented for this project:

- Single server instance handles all games
- If many games run simultaneously, server CPU/memory could saturate
- Production version would use load balancer to distribute across multiple servers

3.8.3 Network Partitioning

Scenario 1: Client-Server partition (client isolated)

- Client detects disconnection
- Game pauses on server (PAUSED_FOR_RECONNECT)
- Opponent continues to see paused game
- If partition persists 30 seconds, game forfeits to opponent
- Client cannot send moves (isolated)

Scenario 2: Client-Server partition (server isolated)

- Server detects both clients disconnected
- Server cannot send broadcasts (no recipients)
- All games timeout and are cleaned up
- Inbound moves are dropped (no clients)

Scenario 3: Slow network (high latency)

- Moves still processed correctly (all happens on server)
- Broadcasts are delayed reaching clients
- Game state will eventually converge (eventual consistency achieved)

The system favors **strong consistency** over availability: if there's any doubt about state, the server pauses the game rather than allowing conflicting moves.

3.9 Security

3.9.1 Input Validation

Server-side validation (strong):

- Player ID is validated on every move
- Pit index is validated (must be 0-5 for P1, 7-12 for P2)
- Move is validated against game rules
- Game state is validated (only allow moves in playable states)

Client-side validation (weak):

- The Vanilla JS actively disables click events and applies CSS `.disabled` and `.opponent-pit` classes to prevent players from clicking empty pits or opponent pits.
- While not a strict security measure (as it can be bypassed via the browser console), this prevents accidental malformed payloads from reaching the server and provides immediate user experience constraints.

4 Implementation

This section describes the technology-dependent choices made to realize the design described above.

4.1 Network Protocols

Protocol Choice: STOMP over WebSocket (with SockJS fallback)

- **HTTP REST**: Session setup (room create/join) via `/api/rooms/create` and `/api/rooms/join`
- **WebSocket + STOMP**: Real-time gameplay events and state broadcasts with low latency
- **SockJS**: Transport fallback for environments where native WebSocket is unavailable

4.2 Data Representation

In-transit data format: JSON

Example game state broadcast:

```
{
  "player1": {
    "id": "550e8400-e29b-41d4-a716-446655440000",
    "name": "Alice"
  },
  "player2": {
    "id": "6ba7b810-9dad-11d1-80b4-00c04fd430c8",
    "name": "Bob"
  },
  "board": {
    "pits": [4, 4, 4, 4, 4, 4, 0, 4, 4, 4, 4, 4, 4, 0]
  },
  "gameStatus": "PLAYER_1_TURN",
  "player1Score": 0,
  "player2Score": 0,
  "disconnectedPlayerId": null,
  "winner": null
}
```

4.3 In-Memory Data Structures

```
// Room storage
ConcurrentHashMap<String, GameRoom activeRooms

// Session tracking
ConcurrentHashMap<String, SessionInfo sessions
  where SessionInfo = (playerId, roomId)

// Activity tracking
ConcurrentHashMap<String, Instant lastActivityTimestamps
ConcurrentHashMap<String, Instant disconnectTimestamps

// Synchronization
synchronized (room) { ... }
```

Why ConcurrentHashMap?

- Thread-safe without locking entire map
- Multiple readers can access simultaneously
- Writers use fine-grained locks (lock striping)
- Perfect for multiple concurrent games

Why synchronized blocks?

- Ensure complex multi-step operations are atomic
- Prevent inconsistent state during game transitions
- Example: Move validation + board update + turn switch must be indivisible

4.4 Timeout Mechanisms

This section contains implementation-level examples only. Policy definitions (values and rationale) are provided earlier in Design and Fault-Tolerance.

Reconnection timeout implementation example

```
private static final Duration RECONNECT_GRACE = Duration.ofSeconds
    (30);

// When player disconnects:
disconnectTimestamps.put(roomId, Instant.now());

// In scheduler (every 5 seconds):
Duration elapsed = Duration.between(disconnectedAt, Instant.now())
    ;
if (elapsed.compareTo(RECONNECT_GRACE)  0) {
  // Forfeit game
  room.getGame().forfeit(disconnectedPlayerId);
}
```

Inactivity timeout implementation example

```
private static final Duration INACTIVITY_TIMEOUT = Duration.
    ofMinutes(1);

// When move is made:
lastActivityTimestamps.put(roomId, Instant.now());

// In scheduler (every 5 seconds):
Duration inactive = Duration.between(lastMove, Instant.now());
if (inactive.compareTo(INACTIVITY_TIMEOUT)  0) {
  // Current player is idle; they lose
  String idlePlayer = (gameStatus == PLAYER_1_TURN)
    ? player1.getId()
    : player2.getId();
  room.getGame().forfeit(idlePlayer);
}
```

Scheduler invocation implementation example

```
@Scheduled(fixedRate = 5000)  // 5000 milliseconds
public void checkTimeouts() {
  List<GameRoom timedOutRooms = gameService.processTimeouts();
  for (GameRoom room : timedOutRooms) {
```

```
        messagingTemplate.convertAndSend("/topic/room/" + room.
            getRoomId(), room.getGame());
    }
}
```

4.5 Technological Details

4.5.1 Spring Boot Framework

Version: 4.0.3

Key dependencies:

- **spring-boot-starter-webmvc**: Provides HTTP server and REST controllers
- **spring-boot-starter-websocket**: Enables WebSocket + STOMP support
- **spring-boot-starter-webmvc-test**: Testing framework
- **lombok**: Reduces boilerplate (not heavily used in this project)

4.6 Frontend Implementation (Client-Side)

The client-side interface was developed exclusively using HTML5, CSS3, and Vanilla JavaScript, completely avoiding heavy frameworks to ensure a lightweight, instantly loading application.

User Interface and Styling:

- **DOM Structure:** The HTML is divided into two primary views: the `lobby-screen` for room creation and the `game-screen` for the active board. The transition between these screens is handled dynamically via CSS classes.
- **CSS Animations:** Visual cues are strictly enforced using CSS. For example, when it is a player's turn, their valid pits receive a `.active` class that triggers a custom `@keyframes pulse` animation, drawing the user's attention. Opponent pits are visually locked with a `.disabled` class and a restricted cursor.

WebSocket Integration (app.js):

- **STOMP & SockJS:** The client utilizes the SockJS library to establish the connection and the STOMP protocol to format the messages.
- **Event Listeners:** The JavaScript attaches click event listeners to all `.pit` elements. When clicked, it extracts the pit index (e.g., parsing "4" from `pit-4`), verifies if it is locally the player's turn to prevent spam clicking, and sends a JSON payload to the server's `/app/game.move` endpoint.
- **State Rendering:** Upon receiving a broadcast from `/topic/room/{roomId}`, the `updateBoard()` function iterates through the JSON state, updating the inner text of all 14 pits, checking the `gameStatus` to update the UI banners (e.g., switching to a "Paused" state), and enabling/disabling pit interactivity.

5 Validation

5.1 Automated Unit Testing

To ensure the reliability and correctness of the core game engine, a comprehensive automated testing suite was implemented using JUnit 5. The test suite specifically targets the domain logic and state transitions, achieving **100% test coverage** across the fundamental model classes: `Player`, `Pit`, `Game`, `GameRoom`, and `Board`.

By isolating the domain model from the infrastructure and presentation layers, the tests utilize the Arrange-Act-Assert pattern for strict state verification. Key automated test scenarios include:

- **Board Mechanics:** Validating stone sowing, boundary enforcement, and complex capture scenarios (ensuring stones are correctly swept to the store only when landing in an empty pit on the player's side).
- **Game Lifecycle:** Testing turn switching, extra turns granted by landing in the store, and the correct determination of game-over states and winners.
- **Exception Handling:** Asserting that illegal moves (e.g., playing out of turn, selecting an empty pit, or attempting to sow from an opponent's pit) correctly throw the expected exceptions and prevent state corruption.
- **Network State Simulations:** Verifying the logical transitions for player disconnections, reconnections, and forfeit determinations within the `Game` entity.

5.1.1 Manual Testing (Performed)

The following manual testing was performed during development. Visual evidence of these tests can be seen in figs. 6 to 10.

1. **Normal game flow**
   - Created room
   - Joined room from second browser
   - Played complete game
   - Verified winner determination
   - **Result**: PASS
2. **Disconnection and reconnection**
   - Simulated disconnection by blocking network (DevTools)
   - Waited less than 30 seconds
   - Reconnected
   - Game resumed correctly

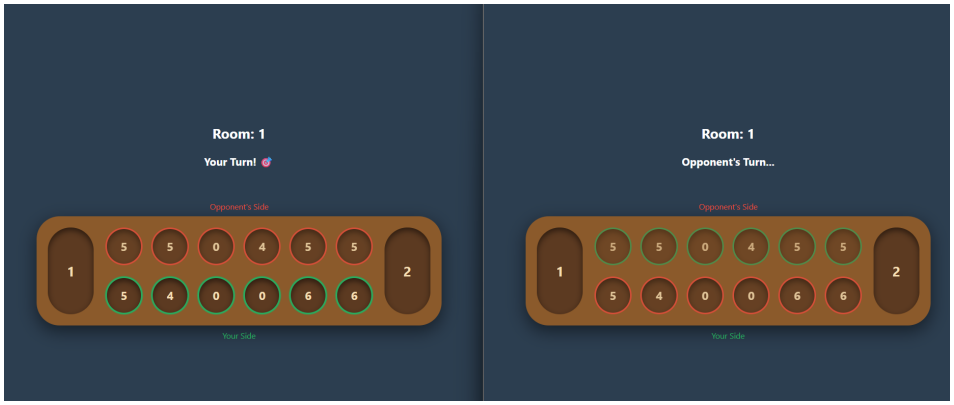

Figure 6: Manual test evidence: normal in-game state after both players join.

- **Result**: PASS

3. **Reconnection timeout**
   - Simulated disconnection
   - Waited 30 seconds
   - Opponent was declared winner
   - **Result**: PASS
4. **Inactivity timeout**
   - Set inactivity timeout to 60 seconds (for testing)
   - Played moves until active player stopped moving
   - Waited 60 seconds
   - Inactive player was forfeit
   - **Result**: PASS

Manual testing screenshots:

6 Deployment

6.1 Installation Instructions

6.1.1 Prerequisites

- Java 17 or later installed
- Maven 3.6+ installed
- Git (to clone repository)
- Modern web browser (Chrome, Firefox, Safari, Edge)

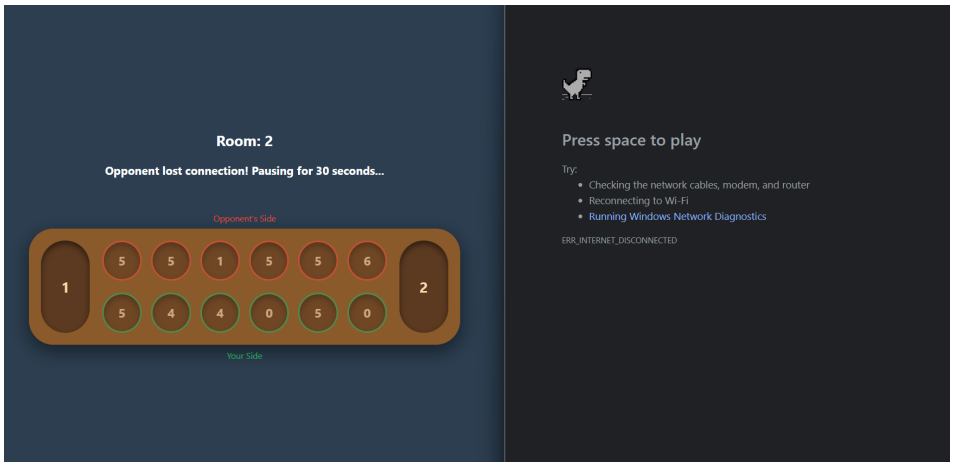

Figure 7: Manual test evidence: disconnection detected and reconnect wait state shown.

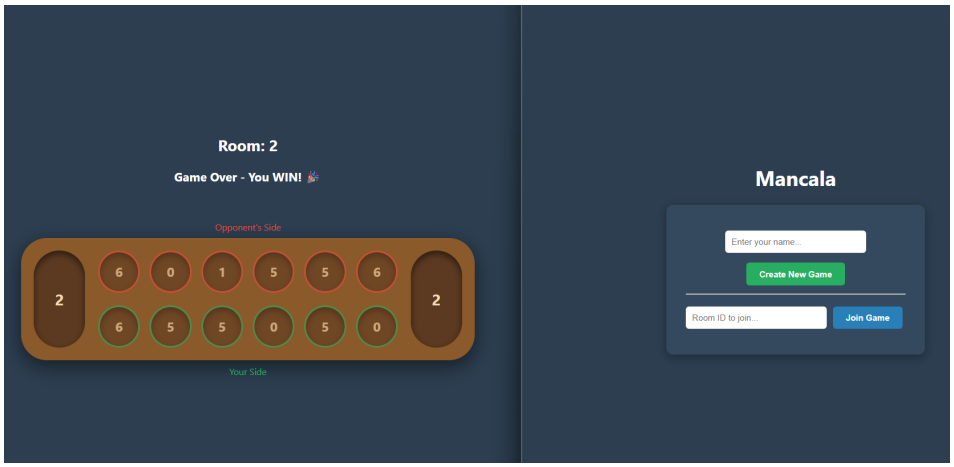

Figure 8: Manual test evidence: reconnection timeout progression.

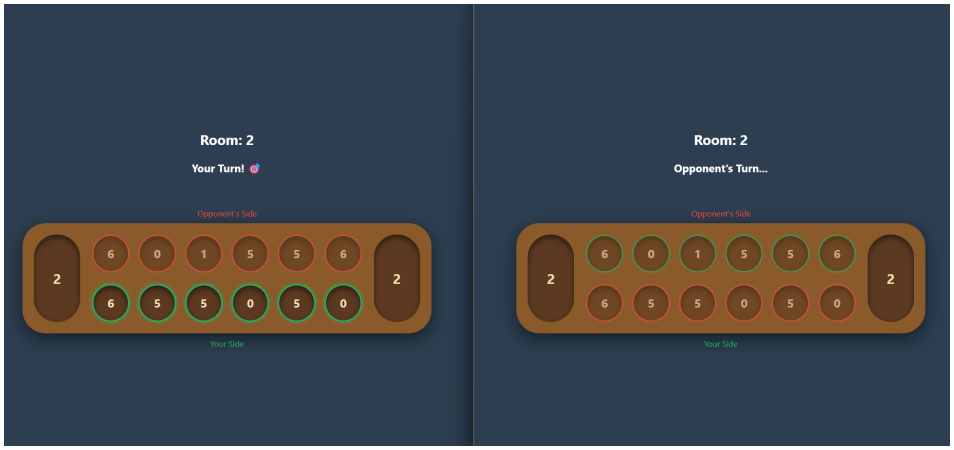

Figure 9: Manual test evidence: successful recovery after reconnection.

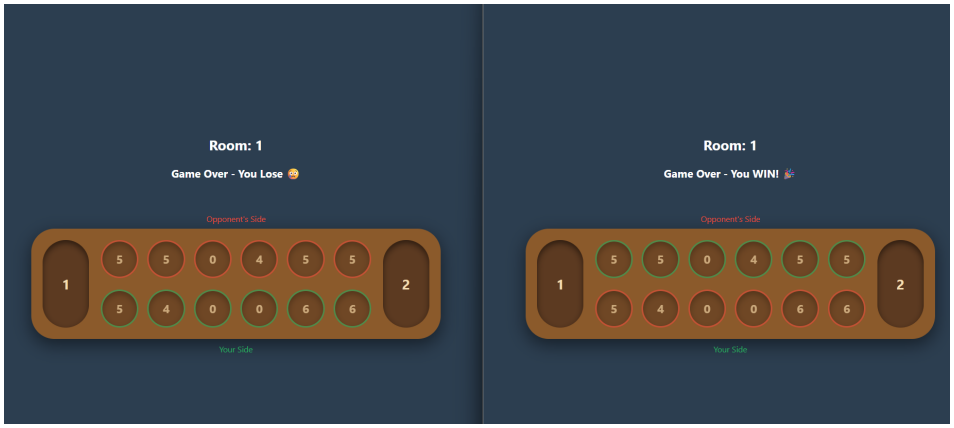

Figure 10: Manual test evidence: inactivity timeout and automatic forfeit outcome.

6.1.2 Build and Run

```
# 1. Clone the repository
git clone <repository-url
cd mancala-game-springboot

# 2. Build the project
mvn clean package

# 3. Run the application
mvn spring-boot:run


# 4. Access in browser
# Navigate to http://localhost:8080
```

7 User Guide

7.1 Starting the Application

1. Open terminal/command prompt
2. Navigate to project directory
3. Run: `mvn spring-boot:run`
4. Wait for startup message: "Started MancalaGameSpringbootApplication in X seconds"
5. Open browser to `http://localhost:8080`

7.2 Creating a Game

1. On the home page, enter your player name
2. Click "Create Room"
3. A room ID will be displayed (e.g., "1")
4. Share this room ID with your opponent (via chat, email, etc.)
5. Game waits with message "Waiting for Player 2"

7.3 Joining a Game

1. Obtain room ID from Player 1
2. Open browser to `http://localhost:8080`

3. Enter your player name
4. Enter the room ID
5. Click "Join Room"
6. Game board appears; Player 1 makes first move

7.4 Playing the Game

- Click any pit on your side to select and sow stones
- The system enforces all Mancala rules:
  - Stones are distributed around the board
  - Captures occur when landing in empty pits
  - Extra turns granted when landing in your store
  - Game ends when one side is empty
- Game status is displayed at top (whose turn)
- Scores update in real-time
- Winner is announced when game ends

7.5 Handling Disconnection

- If your connection is lost, game pauses
- Message appears: "Opponent disconnected"
- You have 30 seconds to reconnect before opponent wins automatically
- To reconnect: Refresh page; system detects and resumes game
- If reconnection fails within 30 seconds, game is forfeited

7.6 Rules Reminder

How Mancala is played:

1. Players alternate turns
2. On your turn, select a pit on your side
3. Stones are distributed counter-clockwise
4. Stones are skipped in opponent's store
5. If last stone lands in your store, you get another turn

6. If last stone lands in an empty pit on your side, capture opposite pit
7. Game ends when one side is completely empty
8. Winner is player with most stones in their store

8 Self-evaluation

8.1 Kasra Sabertehrani

Role within the group: As the backend developer, my primary responsibility was engineering the authoritative server and managing the distributed state. I designed the Spring Boot architecture, implemented the STOMP WebSocket message broker, and wrote the core Mancala game engine. I specifically focused on solving concurrency challenges by designing thread-safe operations using `ConcurrentHashMap` and synchronized blocks to ensure state consistency during simultaneous network requests. Additionally, I implemented the `SessionTracker` and the scheduled timeout mechanisms to handle silent TCP connection drops.

Product Strengths (Backend Perspective):

- **Reliable Real-Time Sync**: The WebSocket broadcasting infrastructure successfully keeps both players aligned with sub-second latency.
- **Safe Concurrency**: The explicit thread locking prevents race conditions, successfully supporting multiple simultaneous game rooms without cross-room interference.
- **Robust Session Handling**: Disconnection/reconnection and timeout paths are explicitly modeled, preventing the server from holding onto dead connections indefinitely.

Product Weaknesses (Backend Perspective):

- **Ephemeral State**: No persistent database is utilized; all active games and player sessions are lost if the Spring Boot server restarts.
- **Single Server Deployment**: The architecture lacks horizontal scaling. If the single node crashes, there is no failover mechanism.
- **Security**: Communication is unencrypted (no HTTPS/WSS) and lacks a formal authentication protocol.

8.2 Parisa Sokuti

Role within the group: As the frontend developer, my role focused on building the client-side architecture and user interface using HTML, CSS, and Vanilla JavaScript. I was responsible for integrating the SockJS and STOMP client libraries to establish the

persistent connection with the server. I engineered the logic to parse incoming JSON state broadcasts and dynamically manipulate the DOM to render the Mancala board, update scores, and provide real-time visual feedback for game events, including network state changes (e.g., displaying "Paused" screens when the opponent disconnects).

Product Strengths (Frontend Perspective):

- **Lightweight Client**: By utilizing Vanilla JS rather than a heavy framework like React or Angular, the client application loads instantly and minimizes unnecessary overhead.
- **Responsive Feedback**: The UI reacts strictly to server-driven events, ensuring the player's view is always perfectly synchronized with the authoritative state.
- **Graceful Degradation**: The interface clearly communicates network issues to the user, masking the complexity of the distributed system behind user-friendly countdowns and pause screens.

Product Weaknesses (Frontend Perspective):

- **DOM Manipulation Complexity**: Direct DOM manipulation in Vanilla JS becomes increasingly difficult to maintain as the application's complexity grows compared to a component-based framework.
- **Client-Side Caching**: If a player refreshes the page manually, the UI completely resets until the server pushes the next full state broadcast, causing a momentary flash of unstyled or empty content.
- **Basic Styling**: The interface is functional but lacks advanced graphical polish, animations, or responsive mobile optimizations.

9 Future works

1. **Spectator Mode**: Allow additional players to watch games without participating
2. **Undo/Replay**: Allow players to undo recent moves or replay games
3. **AI Opponent**: Implement an AI player for single-player mode
4. **Game Variants**: Support different Mancala variants (different board sizes, rules)
5. **Chat Feature**: Allow players to communicate during games
6. **Rating System**: Track player ratings, history, statistics
7. **Tournament Mode**: Support bracketed tournaments with multiple games