Distributed Multiplayer Checkers Platform

MSc Artificial Intelligence and Data Engineering

Reza Almassi
Supervisor: Prof. Alessio Bechini
Distributed Systems and Middleware Technologies
University of Pisa

Academic Year 2024/2025

Abstract

This document details the design and implementation of a distributed online Checkers platform, built as a project for the Distributed Systems and Middleware Technologies course at the University of Pisa. The system demonstrates practical distributed systems concepts by allowing multiple users to play Checkers online in real time, with independent Java game servers coordinated by a robust Erlang/OTP backend for fault tolerance and synchronization. The project covers system architecture, coordination algorithms, fault recovery, and user interface, and is accompanied by complete code and usage instructions.

The full project source code and updates are available at: `https://github.com/Rezacs/DistributedSystems`

Contents

Chapter 1

Introduction

Distributed systems are the backbone of modern scalable web platforms, cloud services, and online gaming applications. This project presents the design and implementation of a robust, modular, and fault-tolerant distributed backend for the classic board game **Checkers**, built for the Distributed Systems and Middleware Technologies course at the University of Pisa.

1.1 Motivation

With the ever-increasing demand for real-time interactive platforms, especially in gaming, ensuring consistency, reliability, and high availability becomes paramount. Traditional monolithic architectures often struggle with scalability, resilience, and coordination when deployed across multiple machines. By building a distributed Checkers platform from scratch, this project offers hands-on exposure to the critical design decisions and trade-offs required in constructing dependable distributed applications.

1.2 Project Goals and Learning Objectives

This project sets out to:

- **Demonstrate distributed systems principles** through a concrete, user-facing application (multiplayer Checkers).
- **Design and implement** a platform where multiple independent nodes (servers) cooperate to provide a seamless gaming experience to many users simultaneously.
- **Explore synchronization, coordination, and communication** challenges and solutions, including leader election, turn-taking, state consistency, and session management.
- **Showcase fault tolerance and dynamic recovery** using Erlang/OTP and robust Java code for real-world distributed reliability.
- **Provide a user-friendly GUI** for both technical and non-technical users to experience the power of distributed computing firsthand.

1.3 Problem Statement

The central problem addressed by this project is: *How can we design a scalable, resilient, and user-friendly multiplayer game platform that maintains correctness and availability across multiple unreliable machines, while being simple for users to access and understand?*

Key sub-problems include:

- Maintaining consistent game state across distributed servers and clients.
- Handling user join/leave, session persistence, and game recovery.
- Detecting, reporting, and recovering from node (server) failures automatically.
- Providing easy server discovery and load distribution.
- Ensuring secure access so only valid players can join or resume games.

1.4 Scope of the Project

The project focuses on backend architecture, core distributed protocols, and client-server interaction for real-time gameplay. While the primary user interface is a JavaFX GUI, the design allows for web or CLI clients to be added in the future. The following are within scope:

- Design and implementation of the coordination service (Erlang/OTP).
- Development of game servers (Java), supporting multiple concurrent games and player sessions.
- User clients (JavaFX), including server and game discovery, game creation, joining, and move execution.
- Fault tolerance, heartbeat and registry protocols, and (optionally) session handover or failover.

Out of scope are large-scale persistent storage, advanced security (beyond player identity), and non-checkers games.

1.5 Overview of Chapters

This document is organized as follows:

Chapter 2 presents the overall architecture, including all system components and their interaction.

Chapter 3 describes the technology stack, libraries, and frameworks used.

Chapter 4 details the design and implementation of each component.

Chapter 5 explains how synchronization and coordination are enforced.

Chapter 6 discusses fault tolerance and recovery strategies.

Chapter 7 is a deployment and usage guide for running and testing the system.

Chapter 8 demonstrates the system in action, including distributed scenarios and fault injection.

Chapter 9 discusses limitations, lessons learned, and future improvements.

Chapter 10 provides references and external resources.

1.6 Project Context Diagram

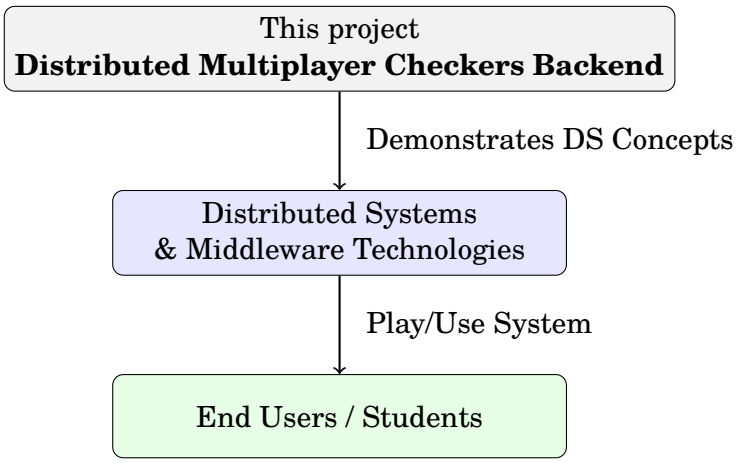

Figure 1.1: Project scope and educational context (vertical arrangement).

Chapter 2

System Architecture

This chapter provides a detailed overview of the system architecture for the Distributed Multiplayer Checkers Platform. The system is designed to be modular, scalable, and fault-tolerant, with each component running independently and communicating over well-defined interfaces.

2.1 High-Level Overview

The architecture is composed of three main layers:

1. **Erlang Coordination Service**: Acts as a registry and coordinator for all Java game servers. Responsible for server registration, heartbeat monitoring, and failure detection.

2. **Java Game Server Nodes**: Each node hosts multiple checkers game sessions, handles client requests, maintains local game state, and communicates with the coordinator.

3. **Game Clients**: JavaFX-based graphical clients (or optional web clients) that connect to any available server, allowing players to create/join games and play in real time.

2.2 Component Interaction Diagram

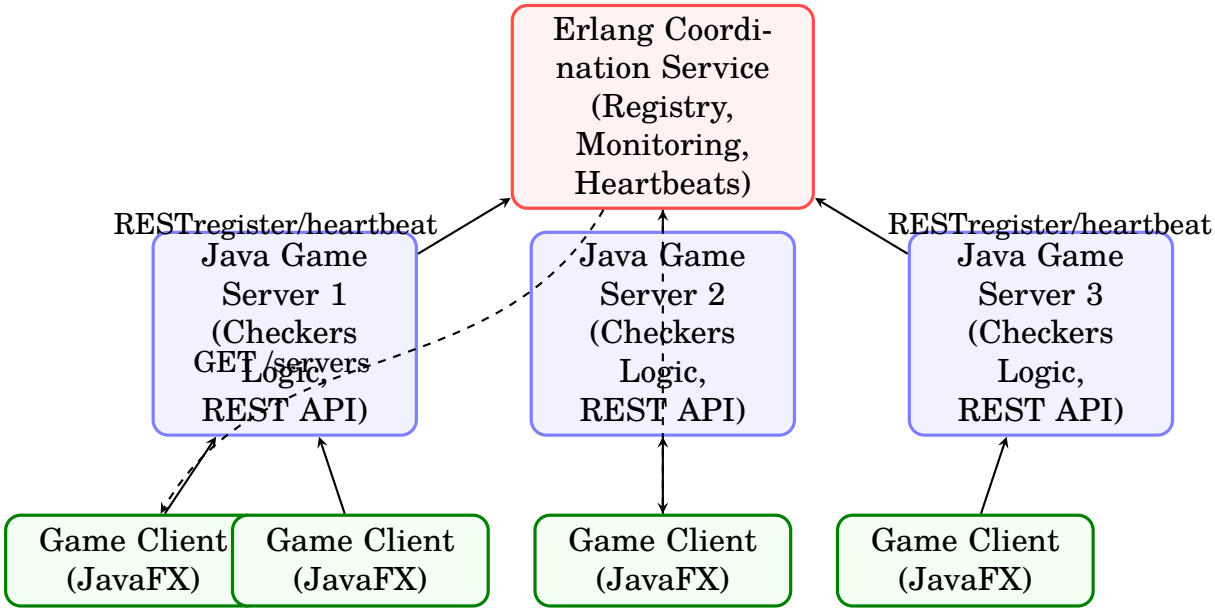

Figure 2.1: Distributed Multiplayer Checkers: System Architecture Diagram

2.3 Main Components Explained

2.3.1 Erlang Coordination Service

- Manages a registry of all active Java game servers.
- Handles server registration (`/register`), server list (`/servers`), and heartbeat messages.
- Detects and removes failed servers from the registry based on missed heartbeats.
- (Optional) May handle session handover and leader election for advanced recovery.

2.3.2 Java Game Server Nodes

- Each server node runs independently and can host multiple concurrent game sessions.
- Implements checkers rules, enforces turn-taking, and maintains per-session state.
- Registers itself with the Erlang coordinator and periodically sends heartbeats.
- Exposes RESTful endpoints for clients (e.g., `/newgame`, `/joingame`, `/move`, `/gamestate`).

2.3.3 Game Clients

- JavaFX-based GUI (and optionally, web client) lets players:
  - Discover available servers via the Erlang service.
  - Create or join active checkers sessions.
  - Play, view board state, and see live updates.
  - Rejoin games using their original username.
- Communicates with selected Java game server over HTTP.

2.4 Scalability and Modularity

The architecture supports:

- Easy scaling: Add more Java servers for increased capacity.
- Robustness: Failure of one server does not impact the overall system.
- Flexibility: Frontend clients can choose any available server at runtime.

Chapter 3

Technology Stack

The Distributed Multiplayer Checkers platform integrates a diverse set of programming languages, frameworks, and libraries to achieve high reliability, scalability, and maintainability. This chapter outlines the main technology choices and their roles in the system, including key protocols and design patterns.

3.1 Overview of Technologies Used

- **Erlang/OTP:** Core coordination and registry service. Offers powerful concurrency, supervision, and distributed fault tolerance features, ideal for system monitoring and leader election.
- **Cowboy:** Modern HTTP server for Erlang. Handles RESTful APIs for registration, server listing, heartbeat, and session handover endpoints.
- **Java:** Implements both the game server logic (session management, game state, validation) and the client interface. Chosen for its mature ecosystem, multithreading, and networking support.
- **SparkJava:** Lightweight microframework for building REST APIs in Java servers.
- **Unirest:** Simple HTTP client library for Java, used by both servers and clients to send RESTful requests and parse JSON.
- **JavaFX:** Rich, cross-platform GUI framework for desktop applications, providing a responsive and interactive player interface.
- **Gson (Google):** For efficient JSON parsing and serialization in Java.
- **Maven:** Project management and dependency resolution tool for Java components.
- **(Optional) Web/HTML:** The architecture is open to web clients, which can be added in the future.

3.2 Technology Stack Diagram

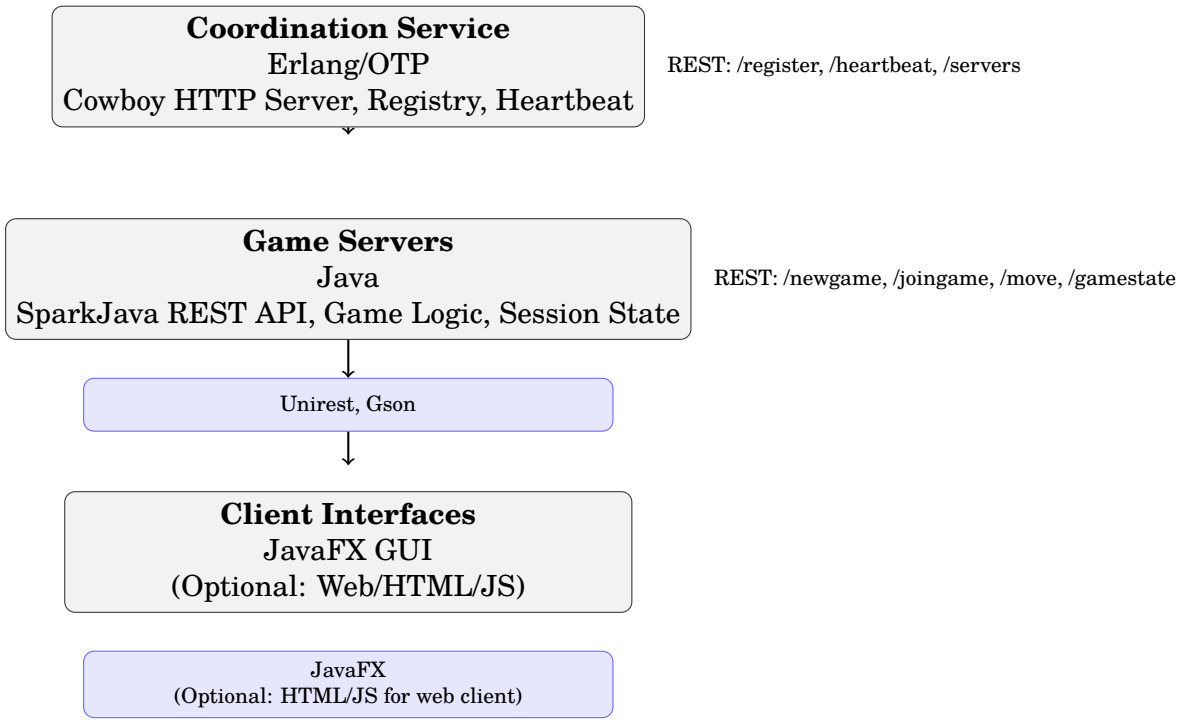

Figure 3.1: Technology stack: vertical arrangement of layers and libraries.

3.3 Communication Protocols

RESTful HTTP APIs

- **Java Servers ↔ Erlang Coordinator:**
  - `POST /register`: Register a new server.
  - `POST /heartbeat`: Periodic heartbeat to signal liveness.
  - `GET /servers`: Retrieve the list of available servers.
  - `(Optional) /savegame, /getgame`: Session handover support.
- **Clients ↔ Java Servers:**
  - `POST /newgame`: Create a new checkers session.
  - `POST /joingame`: Join an existing game.
  - `POST /move`: Submit a player's move.
  - `GET /gamestate/:gameId`: Get current state of a game.
  - `GET /games`: List active games on a server.

JSON Serialization

All requests and responses use JSON for cross-language compatibility (Java ↔ Erlang).

3.4 Why These Technologies?

- **Erlang/OTP:** Offers native support for distributed processes, supervision trees, and robust failure recovery. Cowboy is lightweight and reliable for REST endpoints.
- **Java:** Provides cross-platform deployment, strong ecosystem for networking and GUIs, and fits well with modern desktop and cloud environments.
- **REST and JSON:** Widely adopted for interoperability, simplicity, and language agnosticism.
- **JavaFX:** Delivers modern user experiences without the need for browser compatibility issues.

3.5 Extensibility

The system is designed to support the easy addition of:

- More game servers (for load balancing or geographic scaling).
- Alternative client frontends (e.g., browser-based clients).
- New coordination or monitoring tools, using open REST APIs.

Chapter 4

Component Design and Implementation

This chapter provides a comprehensive description of each major component in the distributed checkers platform. For clarity and best practices, each component is documented with its architecture, responsibilities, data structures, protocol endpoints, and, where helpful, annotated code excerpts. Vertical diagrams illustrate the internal structure and flow for each part.

4.1 Erlang Coordination Service

4.1.1 Overview and Responsibilities

The Erlang coordination service acts as the central registry and supervisor for the distributed system. Its core tasks include:

- Receiving and registering all active game servers.
- Periodically receiving heartbeat messages from servers to detect failures.
- Providing a live list of healthy servers to clients (for discovery).
- Optionally supporting session handover and leader election (advanced/future work).

4.1.2 Architecture Diagram

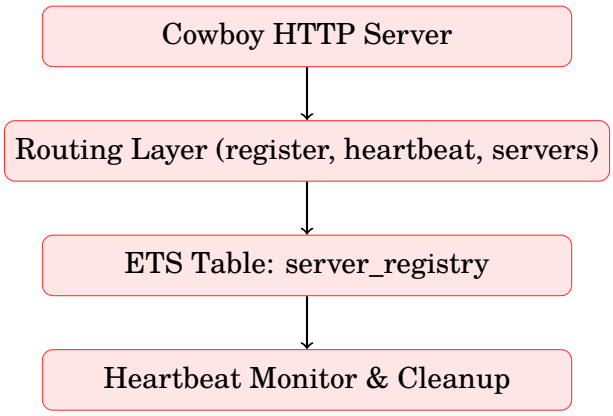

Figure 4.1: Internal vertical structure of the Erlang coordination service.

4.1.3 ETS Table Structure and Lifecycle

The coordination service uses an Erlang Term Storage (ETS) table called `server_registry`, which holds a tuple for each registered server:

```
{ServerId, Host, Port, LastSeen}
% Example: {"srv1", "127.0.0.1", 8081, 1752722000}
```

Listing 4.1: Sample ETS Table Entry in Erlang

- **Registration:** On POST to `/register`, a new tuple is inserted or updated.
- **Heartbeat:** On POST to `/heartbeat`, the LastSeen timestamp is updated.
- **Timeout:** A periodic process scans the table and removes servers whose last heartbeat is too old (dead/unreachable).
- **Listing:** GET on `/servers` returns all current entries.

4.1.4 Key Protocol Endpoints

- `POST /register` – Registers or updates a game server's entry.
- `POST /heartbeat` – Signals that a server is still alive.
- `GET /servers` – Returns all live servers for clients.
- `(Optional) /savegame, /getgame` – Advanced: session state backup/restore.

4.1.5 Sample Erlang Handler (Registration)

```
handle_register(Req, State) ->
    {ok, Body, Req1} = cowboy_req:read_body(Req),
    {ok, Map} = jsx:decode(Body, [return_maps]),
    ServerId = maps:get(<<"server_id">>, Map),
    Host = maps:get(<<"host">>, Map),
```

```
    Port = maps:get(<<"port">>, Map),
    Now = erlang:system_time(second),
    ets:insert(server_registry, {ServerId, Host, Port, Now}),
    reply_json(200, <<"{\"status\": \"registered\"}">>, Req1, State).
```

Listing 4.2: Erlang Cowboy handler for /register endpoint

This handler ensures atomic registration of servers and updates their information in the registry.

4.2 Java Game Server

4.2.1 Overview and Responsibilities

Each Java game server is a full-featured application that:

- Hosts multiple concurrent checkers game sessions.
- Implements full checkers logic, state, and move validation.
- Registers itself with the Erlang coordinator and sends regular heartbeats.
- Exposes REST API endpoints for clients: create/join game, move, gamestate, and list games.
- Maintains in-memory session state and, optionally, can save/restore games for session handover.

4.2.2 Server Internal Design (Vertical Diagram)

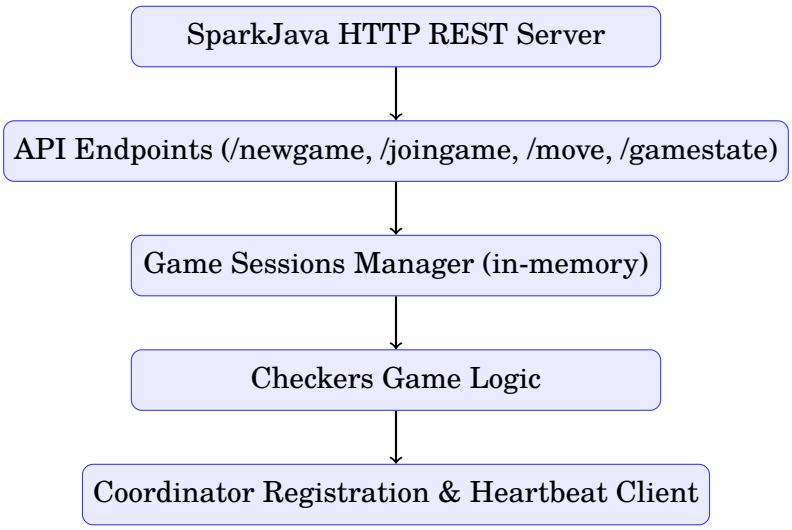

Figure 4.2: Internal vertical structure of a Java game server.

4.2.3 Session State and Game Logic

Each game session is represented by a `CheckersGame` class. A simplified excerpt:

```java
public class CheckersGame {
    public String gameId;
    public String player1, player2, currentTurn;
    public String[][] board = new String[8][8];
    // ... initialization and game rules ...
}
```

Listing 4.3: Java CheckersGame class: essential structure

4.2.4 REST API Endpoints: Full List and Explanation

The system uses a RESTful API between components. Each endpoint is numbered, colored by HTTP method, and described in detail below.

1. **POST** `/register`
   - **Purpose:** Register a game server with the Erlang coordinator.
   - **Input (JSON):**
   ```
   { "server_id": "...", "host": "...", "port": <int> }
   ```
   - **Output (JSON):**
   ```
   { "status": "registered" }
   ```

2. **POST** `/heartbeat`
   - **Purpose:** Signal that a game server is alive.
   - **Input (JSON):**
   ```
   { "server_id": "..." }
   ```
   - **Output (JSON):**
   ```
   { "status": "heartbeat ok" }
   ```

3. **POST** `/newgame`
   - **Purpose:** Create a new checkers game session.
   - **Input (JSON):**
   ```
   { "player": "<player_name>" }
   ```
   - **Output (JSON):**
   ```
   { "gameId": "<game_id>" }
   ```

4. **POST** `/joingame`
   - **Purpose:** Join an existing game session if your name matches player1/player2 or slot is open.
   - **Input (JSON):**
   ```
   { "gameId": "<game_id>", "player": "<player_name>" }
   ```

- **Output (JSON):**

```
{ "status": "joined" }
% or error:
{ "error": "Cannot join" }
```

5. **POST** `/move`

- **Purpose:** Submit a move in a game session (enforces turn and legality).
- **Input (JSON):**

```
{
  "gameId": "<game_id>",
  "player": "<player_name>",
  "fromRow": <int>,
  "fromCol": <int>,
  "toRow": <int>,
  "toCol": <int>
}
```

- **Output (JSON):**

```
{ "status": "move ok" }
% or error:
{ "status": "illegal move or not your turn" }
```

6. **GET** `/servers`

- **Purpose:** Retrieve the list of all currently registered game servers.
- **Input:** None
- **Output (JSON array):**

```
[
  { "server_id": "...", "host": "...", "port": <int>, "last_seen": <timestamp>
      },
  ...
]
```

7. **GET** `/games`

- **Purpose:** List all active game sessions on a given game server.
- **Input:** None
- **Output (JSON array):**

```
[
  {
    "gameId": "...",
    "player1": "...",
    "player2": "...",
    "isOpen": <true/false>
  },
  ...
]
```

8. **GET** `/gamestate/:gameId`

- **Purpose:** Retrieve the current state of a game session (must be player1 or player2).
- **Input:** URL parameter `:gameId`; query parameter `player=<player_name>`

- **Output (JSON):**

```
{
  "gameId": "...",
  "player1": "...",
  "player2": "...",
  "currentTurn": "...",
  "board": [[...],[...],...],
  "whiteScore": <int>,
  "blackScore": <int>,
  "winner": "..."
}
% or error:
{ "error": "You are not a player in this game!" }
```

9. **PUT** `/savegame` *(optional / advanced)*

   - **Purpose:** Save a game session's state for handover/failover (future/optional).
   - **Input (JSON):** Game state object.
   - **Output (JSON):** { "status": "saved" }

10. **DELETE** `/deregister`

    - **Purpose:** Gracefully remove a game server from the Erlang coordinator's registry, usually when the server is shutting down or restarting. This helps keep the registry accurate and prevents clients from trying to connect to unavailable servers.
    - **Input:**
      - Method: **HTTP DELETE**
      - URL: `http://localhost:8080/deregister`
      - Header: `Content-Type: application/json`
      - Body (JSON):

```
{ "server_id": "srv1" }
```

    - **Output:**
      - On success (server found and removed):

```
{ "status": "deregistered" }
```

      - On failure (server ID not found):

```
{ "error": "Server not found" }
```

    - **Example Usage (with `curl`):**

```
curl -X DELETE http://localhost:8080/deregister \
  -H "Content-Type: application/json" \
  -d '{"server_id": "srv1"}'
# Response:
# { "status": "deregistered" }
```

    - **Description:** When a Java game server intends to shut down, it should call this endpoint with its unique `server_id`. The Erlang coordinator will immediately remove this entry from its internal registry table (ETS). If the specified server does not exist in the registry, an error message is returned. This mechanism ensures that clients always see an up-to-date list of available servers.

Color code:

- **POST** endpoints – Create, update, join, or act.
- **GET** endpoints – Query or fetch information.
- **PUT** endpoints – Save or update existing data (future work).
- **DELETE** endpoints – Remove a resource.

All data is in JSON format for cross-language compatibility. Errors are always returned as `{"error": "..."}` JSON objects.

4.2.5 Annotated Java Handler Example

```
post("/joingame", (req, res) -> {
    Map<String, String> body = gson.fromJson(req.body(), Map.class);
    String gameId = body.get("gameId");
    String player = body.get("player");
    CheckersGame game = games.get(gameId);
    if (game != null && game.join(player)) {
        return gson.toJson(Collections.singletonMap("status", "joined")
            );
    } else {
        return gson.toJson(Collections.singletonMap("error", "Cannot
            join"));
    }
});
```

Listing 4.4: Java: Join Game endpoint handler, strictly enforcing player identity

4.2.6 Heartbeat and Registration with Coordinator

The server registers itself and maintains liveness via heartbeats:

```
HttpResponse<String> regResp = Unirest.post(coordinatorUrl + "/register
    ")
    .header("Content-Type", "application/json")
    .body("{\"server_id\":\"" + serverId + "\",\"host\":\"" + host + "
        \",\"port\":" + port + "}")
    .asString();
new Thread(() -> {
    while (true) {
        Thread.sleep(5000);
        Unirest.post(coordinatorUrl + "/heartbeat")
            .header("Content-Type", "application/json")
            .body("{\"server_id\":\"" + serverId + "\"}")
            .asString();
    }
}).start();
```

Listing 4.5: Java server: registration and heartbeat to Erlang

4.3 Client Interface

4.3.1 JavaFX Graphical Client

The JavaFX client enables users to:

- Discover active servers (via the coordinator's `/servers` endpoint).
- Select a server, see available games, and create or join a session.
- Play interactively with a live-updating GUI, see turn, board, player names, and winner.
- Rejoin a session by entering the same name and game ID.

4.3.2 Client GUI Design (Vertical Diagram)

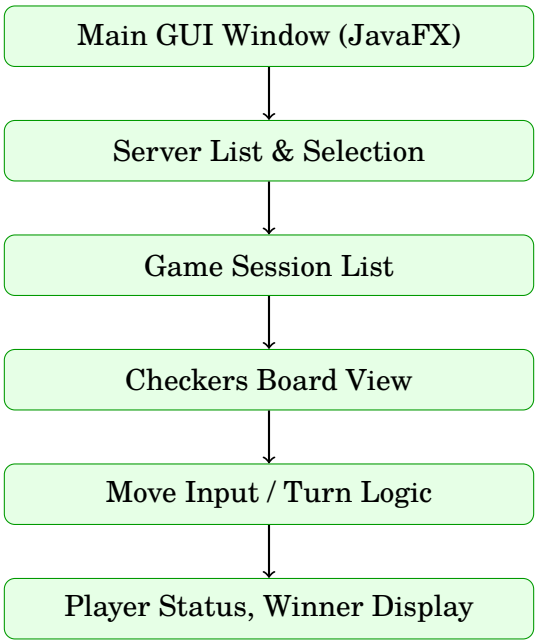

Figure 4.3: Vertical block diagram: JavaFX client user interface.

4.3.3 Example: Client Fetching and Joining a Game

```
var res = Unirest.get("http://localhost:8080/servers").asString();
JsonArray servers = JsonParser.parseString(res.getBody()).
    getAsJsonArray();
// User selects a server...
var joinRes = Unirest.post(selectedServerUrl + "/joingame")
    .header("Content-Type", "application/json")
    .body("{\"gameId\":\"" + gameId + "\",\"player\":\"" + player + "
        \"}")
    .asString();
```

Listing 4.6: JavaFX client: fetching server list and joining a game

4.3.4 Access Control and Session Recovery

- Only original players (player1 or player2) can rejoin a game; other names are denied.
- If a client is closed and reopened, the user can rejoin with the same name and gameId.
- The GUI gives clear feedback on join success/failure and game status.

4.3.5 (Optional) Web Client Design

The architecture allows for easy addition of a web-based client, reusing the same REST API endpoints.

Chapter 5

Synchronization and Coordination

This chapter explains how the distributed checkers system maintains a consistent, synchronized state across clients, servers, and the central coordinator. All interactions and protocols are carefully designed to ensure that player turns, moves, and session state are coordinated correctly, even in the presence of failures or network partitions.

5.1 Overview

Synchronization and coordination are at the heart of any distributed multiplayer application. In this platform, they are addressed at several levels:

- **Player Turn and Move Coordination:** Ensuring only the correct player can act at the correct time in each game session.
- **Game State Consistency:** Ensuring that the view of the board, scores, and winner are consistent for both players.
- **Server Registry Consistency:** Guaranteeing that all clients see an up-to-date list of available servers, even as servers join or leave.
- **Session Join/Rejoin Control:** Only original players can join or rejoin a game, preventing unauthorized access or conflicts.

5.2 Vertical Synchronization Flow Diagram

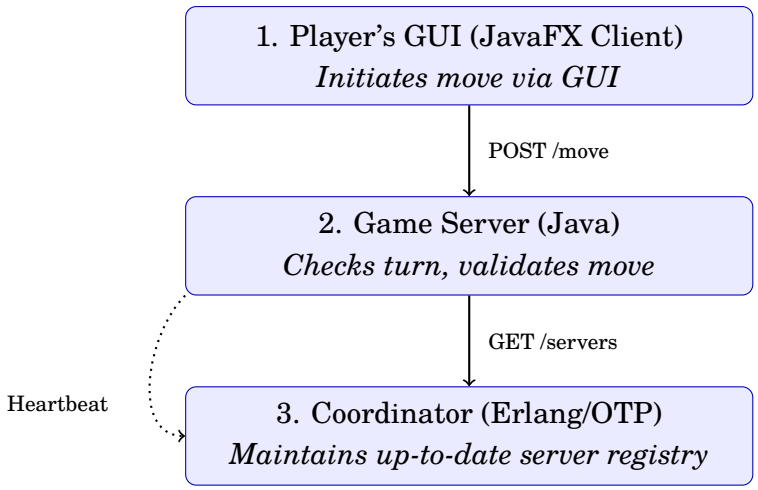

Figure 5.1: Strictly vertical synchronization and coordination flow in the distributed system. Each layer validates and maintains its own invariants.

5.3 Turn-Based Synchronization: How a Move Is Processed

The system strictly enforces player turns and move legality. The flow is as follows:

1. The client (GUI) submits a move via `POST /move`, including the player name, game ID, and source/destination coordinates.
2. The server receives the request and:
   - Checks if the player is either `player1` or `player2` for the game session.
   - Checks if it is currently that player's turn.
   - Validates the move for legality (per checkers rules).
3. If all checks pass:
   - The server updates the game state, switches turns, and may check for a win condition.
   - Returns a success status to the client.
4. If any check fails:
   - The server returns an error (e.g., "illegal move or not your turn").
   - The client updates its GUI accordingly.
5. The server optionally broadcasts the new game state (e.g., for real-time updates) or waits for the next client poll.

5.4 Sample Move Validation Code

The following Java code ensures that only the right player can move, and only when it is their turn:

```
public boolean move(String player, int fromRow, int fromCol, int toRow,
    int toCol) {
    if (!player.equals(currentTurn)) return false; // Not your turn
    String color = getCurrentPlayerColor();
    String piece = board[fromRow][fromCol];
    if (!piece.equals(color)) return false; // Can't move other's piece

    // ... (move legality checks and update logic) ...
}
```

Listing 5.1: Enforcing player turn and move validation in Java

5.5 Coordinator and Registry Consistency

The Erlang coordinator maintains the live registry of servers:

- Game servers register and send heartbeats regularly.
- If a server stops heartbeating, it is removed from the registry.
- Clients always see only healthy servers when fetching the server list.

5.6 Session and Identity Control

To prevent conflicts and maintain security:

- Only `player1` and `player2` (by name) may join, rejoin, or play in a session.
- All join and gamestate API calls check the player's identity.
- Unauthorized users (wrong name or third party) are denied access.

5.7 Summary

Through a combination of careful protocol design, input validation, and robust registry maintenance, the distributed checkers system ensures strong consistency, correctness, and player experience, even across failures and restarts.

Chapter 6

Fault Tolerance and Recovery

Fault tolerance is a fundamental requirement in distributed systems, ensuring the platform continues to operate correctly even if some nodes (servers) crash or become unreachable. This chapter explains how our system detects failures, recovers from faults, and keeps the registry and clients up-to-date.

6.1 Overview

The system uses a combination of heartbeats, timeouts, and registry updates to achieve robust fault tolerance:

- Game servers periodically send heartbeat messages to the Erlang coordinator.
- The coordinator maintains a timestamped registry of all active servers.
- If a server fails to send a heartbeat within a given timeout, it is marked as failed and removed.
- Clients and users always see an up-to-date list of healthy servers.
- Stopped or failed servers do not affect ongoing games on other servers.
- (Optionally) Game session handover or backup is possible for advanced recovery.

6.2 Vertical Failure Detection and Recovery Diagram

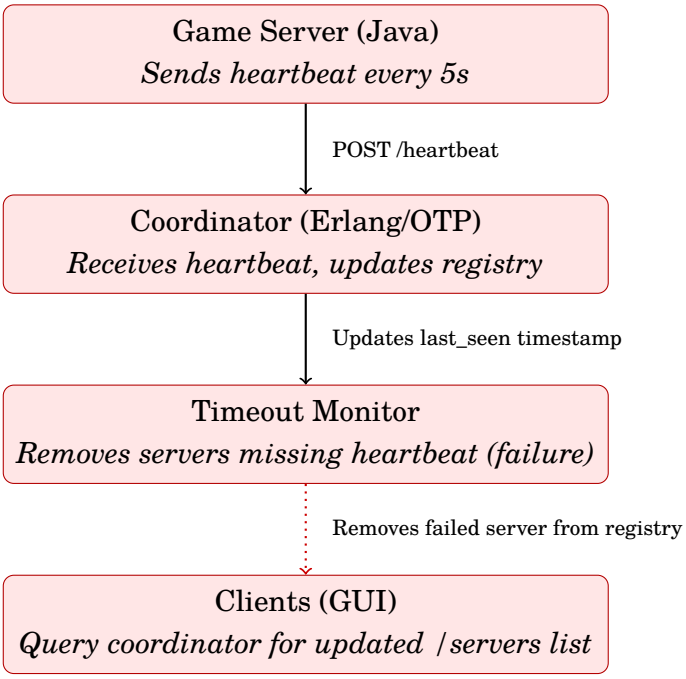

Figure 6.1: Vertical flow of fault detection and recovery: Game server sends heartbeats, coordinator monitors, removes failed entries, clients always get an up-to-date server list.

6.3 Heartbeat and Failure Detection Logic

1. Heartbeat Sending (Java Game Server)

Every game server starts a background thread to POST a heartbeat every 5 seconds:

```
new Thread(() -> {
    while (true) {
        Thread.sleep(5000);
        Unirest.post(coordinatorUrl + "/heartbeat")
            .header("Content-Type", "application/json")
            .body("{\"server_id\":\"" + serverId + "\"}")
            .asString();
    }
}).start();
```

Listing 6.1: Java: heartbeat thread

2. Heartbeat Handling (Erlang Coordinator)

The Erlang handler for `/heartbeat` updates the server's `last_seen` timestamp in the registry:

```
handle_heartbeat(Req, State) ->
    {ok, Body, Req1} = cowboy_req:read_body(Req),
    {ok, Map} = jsx:decode(Body, [return_maps]),
    ServerId = maps:get(<<"server_id">>, Map),
    Now = erlang:system_time(second),
    case ets:lookup(server_registry, ServerId) of
        [{ServerId, Host, Port, _}] ->
            ets:insert(server_registry, {ServerId, Host, Port, Now
               }),
            reply_json(200, <<"{\"status\": \"heartbeat ok\"}">>,
               Req1, State);
        [] ->
            reply_json(404, <<"{\"error\": \"Server not registered
               \"}">>, Req1, State)
    end.
```

Listing 6.2: Erlang: /heartbeat handler

3. Failure/Timeout Detection (Erlang)

A periodic Erlang process scans the registry, removing servers whose `last_seen` is too old:

```
remove_dead_servers() ->
    Now = erlang:system_time(second),
    Dead = [S || {S, _, _, Last} <- ets:tab2list(server_registry),
                  Now - Last > 10], % 10s timeout
    lists:foreach(fun(ServerId) -> ets:delete(server_registry,
       ServerId) end, Dead).
```

Listing 6.3: Erlang: timeout and removal

6.4 Client-Side Recovery and User Experience

When a server fails:

- The failed server is quickly removed from the `/servers` list.
- Clients can immediately see the updated server list.
- Users simply choose another server, or reconnect later.
- Ongoing games on other servers are unaffected.

6.5 Advanced: Session Handover (Optional)

For future work, session state can be periodically backed up and transferred so that in-progress games survive even a server crash, allowing seamless failover to another server.

6.6 Summary

The distributed checkers platform provides strong fault tolerance using simple but robust mechanisms: heartbeats, timeouts, and a live registry. This ensures high availability and a smooth user experience, even in the face of failures.

Chapter 7

Deployment and Usage Guide

This chapter describes how to set up, deploy, and use each component of the distributed checkers platform. Every step is explained in detail, with commands, expected outputs, and a vertical deployment flow chart.

7.1 System Deployment: Vertical Overview Diagram

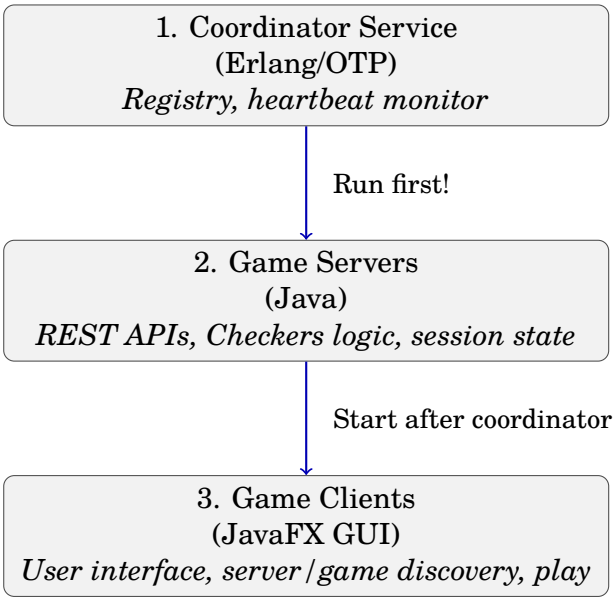

Figure 7.1: Vertical deployment and usage flow: Start coordinator, then servers, then clients.

7.2 Step-by-Step Deployment Instructions

1. Start Erlang Coordinator Service

1. Ensure Erlang/OTP and `rebar3` are installed.
2. Navigate to the coordinator directory:

```
cd CoordinationService
```

3. Launch the service:

```
rebar3 shell
```

4. The coordinator listens on `localhost:8080` and manages the server registry.

2. Start One or More Java Game Servers

1. Open a new terminal for each server you wish to start.
2. Build the Java project (if not done):

```
cd GameServer
mvn package
```

3. Start a server instance with unique port and ID (run each in a new terminal):

```
java -jar target/gameserver-1.0-SNAPSHOT.jar 8081 srv1
java -jar target/gameserver-1.0-SNAPSHOT.jar 8082 srv2
```

4. Each server registers with the coordinator and sends heartbeats.
5. You should see confirmation in both the Java and Erlang terminals.

3. Start the JavaFX Client

1. Make sure JavaFX is installed (set `-module-path` as needed).
2. Launch the client:

```
java --module-path ~/javafx-sdk-17.0.15/lib --add-modules javafx.
    controls,javafx.fxml \
     -cp target/gameserver-1.0-SNAPSHOT.jar com.example.
        CheckersClient
```

3. The GUI will open, allowing you to:
   - Refresh and browse available servers (fetched from coordinator).
   - Select a server, create or join games, and play.
   - See all game state, moves, player names, and winner live.

7.3 Sample Usage Scenarios

A. Playing Distributed Multiplayer Checkers

1. Start coordinator, then two or more servers, then clients.
2. Players use the GUI to join the same or different servers.
3. Game sessions run independently, with full checkers rules and move synchronization.

B. Demonstrating Fault Tolerance

1. While playing, kill (Ctrl+C) one server process.
2. Within seconds, the coordinator removes it from the registry.
3. Clients see an updated server list, and ongoing games on other servers are unaffected.

7.4 Expected Command-Line and GUI Outputs

- **Erlang terminal:** Logs registration, heartbeat, and server removal messages.
- **Java server terminal:** Shows registration confirmation, heartbeats, and game events.
- **GUI client:** Shows live server/game lists, in-game board, turn, and winner information.

7.5 Troubleshooting Tips

- Always start the coordinator before any servers.
- Ensure all ports used (e.g., 8080 for Erlang, 8081/8082 for Java servers) are free.
- If a server fails to register, check coordinator logs and network connectivity.
- GUI errors often stem from missing JavaFX modules; check `-module-path`.

7.6 Deployment Flexibility

The platform can be run on a single machine (multiple terminals), across several physical or virtual machines, or even in Docker containers for full distributed deployment. For advanced setups, adjust IPs and firewall settings to allow cross-machine HTTP communication.

7.7 Summary

This deployment guide enables anyone to install, configure, and run a complete distributed checkers system, illustrating real distributed operation, synchronization, and resilience in action.

Chapter 8

Demonstration & Results

This chapter demonstrates the distributed checkers system in action, highlighting its real-time distributed behavior, robust fault tolerance, and ability to handle a range of operational scenarios. The purpose of this demonstration is to provide tangible evidence that the design, synchronization, and recovery mechanisms work as intended in realistic conditions.

Throughout this chapter, we document a series of carefully selected test cases designed to showcase the most important features and properties of the platform:

- **System initialization:** Bringing up the Erlang coordinator, launching multiple independent Java game servers, and starting client GUIs on different machines or terminals.
- **Server discovery and registration:** How servers appear in the global registry and how clients dynamically discover available servers.
- **User experience:** The process of game creation, joining, and real-time play from the player's perspective, including GUI interactions, move validation, and turn-taking enforcement.
- **Game state synchronization:** How all clients see up-to-date board states, turns, and scores, even as they switch servers or recover sessions.
- **Resilience to faults:** What happens when a server is stopped or crashes, including immediate removal from the active server list, uninterrupted play on remaining servers, and error handling in the GUI.
- **Session persistence and identity control:** How a player can rejoin a game after closing the client, provided they use the same name, and how unauthorized access is prevented.
- **Logging and diagnostics:** Example logs from both the coordinator (Erlang) and game servers (Java), confirming registration, heartbeat, removal, and error events.

In addition to step-by-step explanations, this chapter includes annotated screenshots from the JavaFX client GUI and representative command-line output from the backend processes. These artifacts not only validate the system's design but also serve as a practical reference for users and future developers.

Together, these demonstrations provide a comprehensive picture of the platform's distributed capabilities, including how it achieves seamless gameplay, dynamic server management, and reliable recovery, all while offering an intuitive and engaging user experience.

8.1 Scenario Flow: Distributed Play and Fault Tolerance

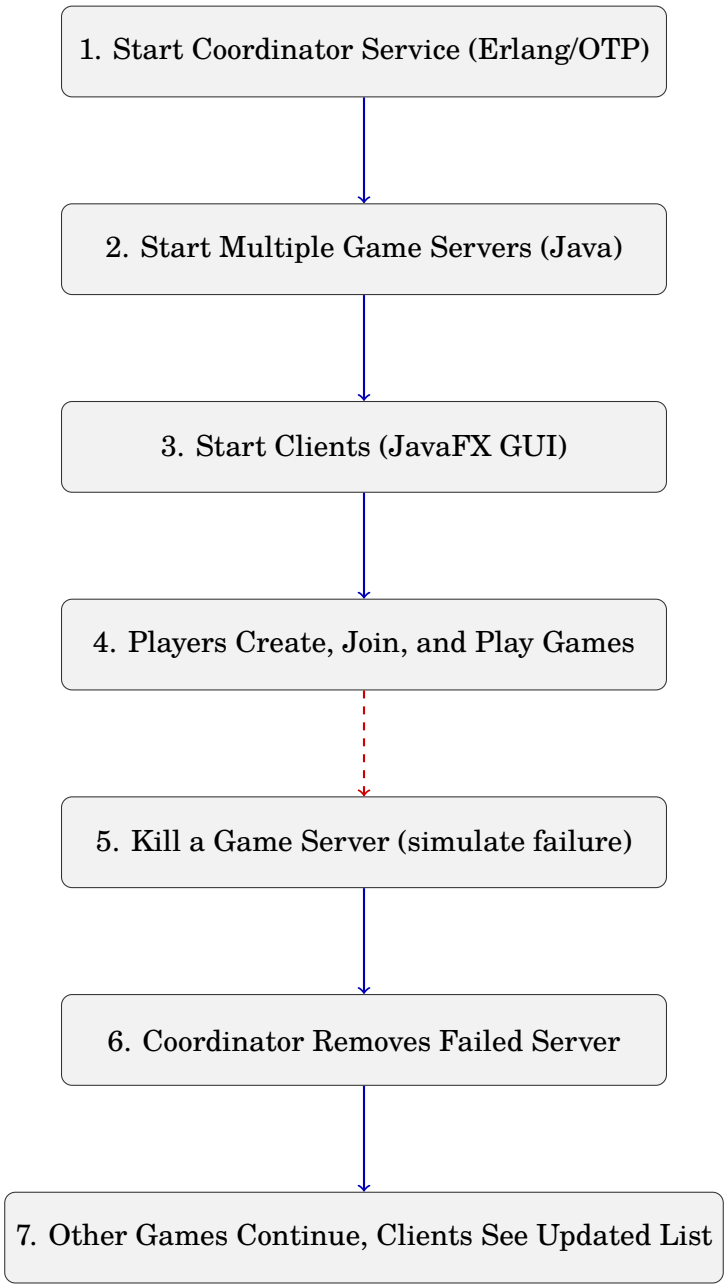

Figure 8.1: Vertical demonstration scenario: distributed play, server failure, and recovery.

8.2 Example: Real Distributed Play

A. Server Startup and Registration

```
Eshell V12.0  (abort with ^G)
1> Received registration from srv1 at 127.0.0.1:8081
1> Received registration from srv2 at 127.0.0.1:8082
1> Heartbeat from srv1
1> Heartbeat from srv2
```

Listing 8.1: Erlang shell output

B. Client GUI: Server and Game Discovery

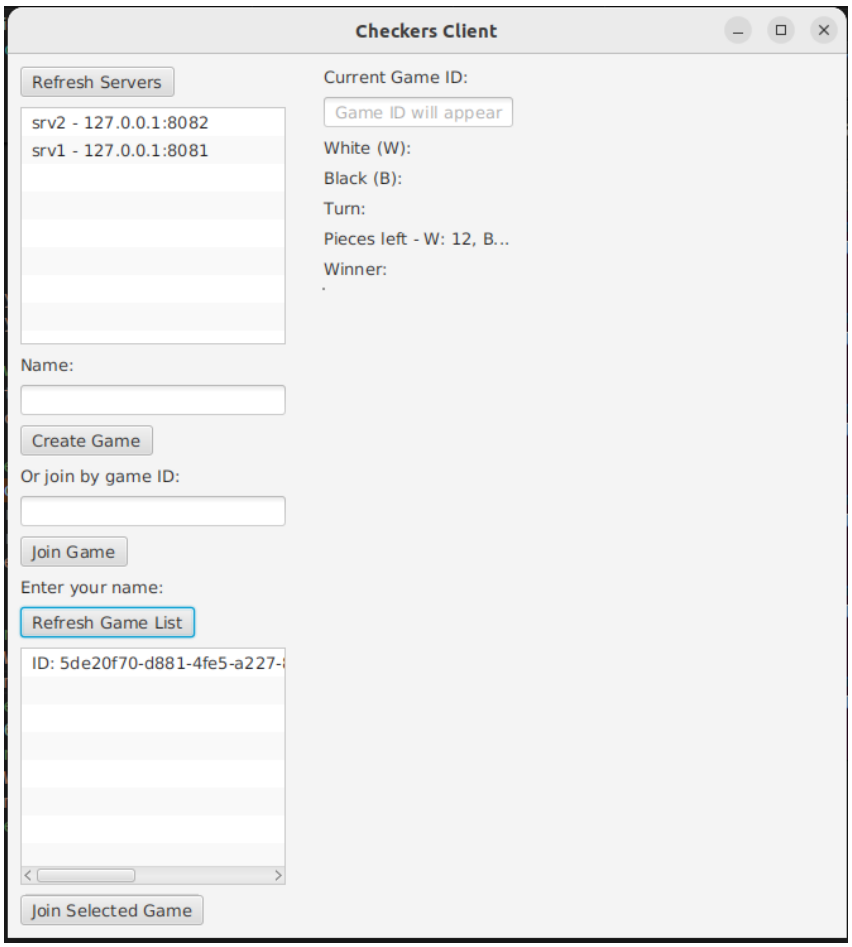

Figure 8.2: GUI: Server discovery and selection.

C. Game Creation, Join, and Play

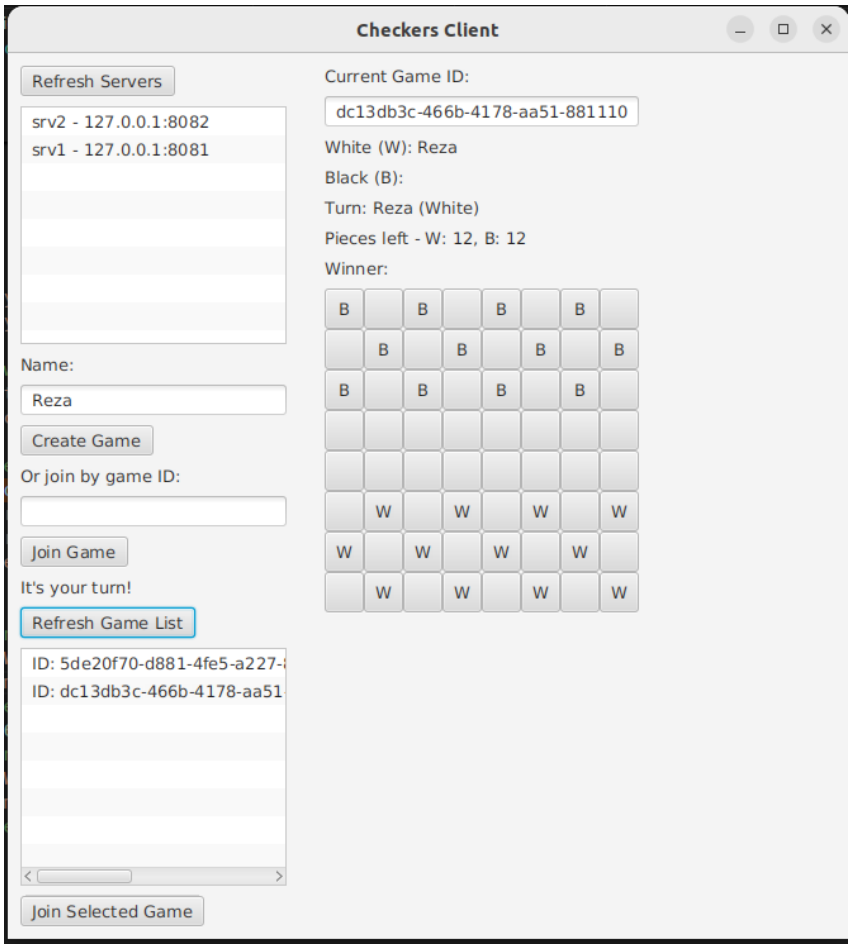

Figure 8.3: GUI: Players join a game, board is updated live.

D. Demonstrating Fault Tolerance

1. While playing, a server process is stopped or killed (Ctrl+C).
2. The coordinator logs the missed heartbeat and removes the server from its registry.
3. The client GUI, after a refresh, displays an updated server list (failed server removed).
4. Games running on other servers continue unaffected.

```
% After srv2 is killed:
Coordinator: Missed heartbeat from srv2
Coordinator: Removing srv2 from registry
```

Listing 8.2: Erlang shell output: server removal

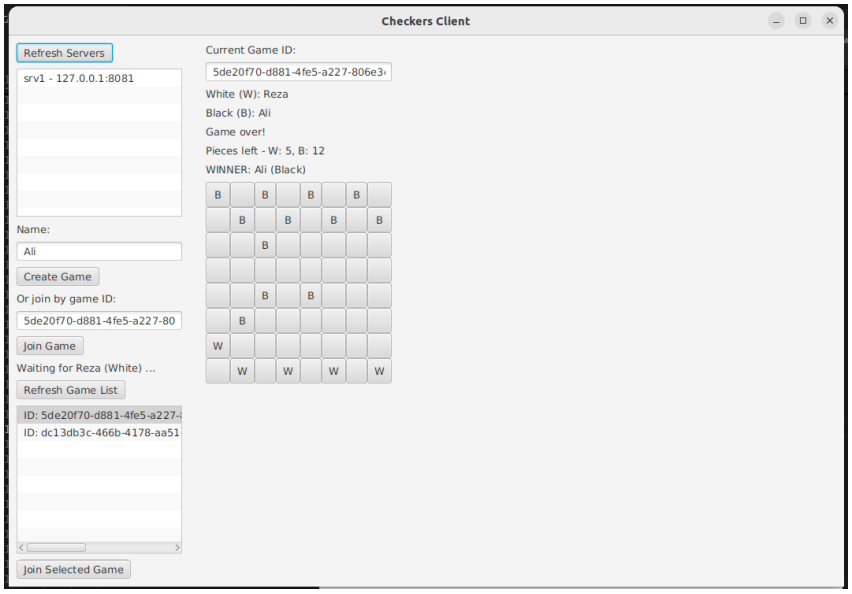

Figure 8.4: GUI: Updated server list after failure, ongoing games continue.

8.3 Discussion of Results

- The system demonstrates distributed operation: multiple servers, concurrent games, and client-server interactions.
- Failure detection and recovery are automatic; no user intervention is needed.
- Game state and logic remain consistent for all connected players.
- The platform can be easily scaled or extended with more servers or clients.

8.4 Lessons Learned

Building a real distributed system reveals the importance of:

- Careful protocol design for synchronization and fault detection.
- Defensive coding to prevent race conditions and unauthorized access.
- Clear user feedback and logs for troubleshooting.

8.5 Summary

The demonstration confirms that the distributed multiplayer checkers platform achieves its goals of real-time distributed gameplay, strong fault tolerance, and smooth user experience across server failures and recovery.

Chapter 9

Discussion and Future Improvements

This chapter discusses the main limitations and challenges encountered during the project, the lessons learned, and several concrete directions for future enhancement of the distributed checkers platform. It concludes with a vertical roadmap diagram illustrating next steps for the system.

9.1 Current Limitations

Despite achieving its core objectives, the current implementation has some boundaries and areas where further development would provide significant benefits:

- **Session handover and persistence:** At present, game sessions are only stored in memory. If a game server crashes, any ongoing games on that server are lost. Implementing persistent session storage and automated handover to another server would greatly improve resilience.
- **Scalability:** While the platform supports multiple servers, scaling to dozens or hundreds would require performance testing, optimization, and possibly sharding or load-balancing mechanisms.
- **Security:** User identity is enforced by player name only. Full authentication (e.g., login, tokens) and encryption (HTTPS) are not yet implemented.
- **Web client interface:** The system is primarily demonstrated using a JavaFX desktop GUI. While the architecture supports web clients, a production-ready web or mobile frontend would increase accessibility.
- **Automated failover:** If a server fails, users must manually choose another server and start a new session. Automatic redirection or session migration is not yet implemented.
- **Persistent audit logs and monitoring:** Detailed logging exists, but integrating with external monitoring tools or dashboards (e.g., Prometheus, Grafana) would support real-time system health checks and long-term auditing.

9.2 Lessons Learned

This project provided valuable insights into the complexities of real-world distributed system engineering:

- **Protocol design:** Clear and robust APIs are vital for synchronization, extensibility, and recovery.
- **Error handling:** Defensive programming and clear error messages significantly aid troubleshooting and improve user experience.
- **Coordination strategies:** Heartbeat and registry protocols are effective for fault detection but require careful tuning to avoid false positives/negatives.
- **User experience:** Real-time feedback, clarity on join/fail conditions, and visual cues in the GUI are essential for non-technical users.
- **Testing:** Distributed systems are inherently difficult to test; automation and repeated scenario testing are crucial for reliability.

9.3 Future Improvements

A number of concrete enhancements are planned or proposed:

- **Persistent session storage:** Use a database or distributed key-value store to save all game states, enabling recovery after server failure.
- **Automated session handover:** On server failure, another server can claim the session and resume play for connected clients.
- **Full web client:** Develop a browser-based frontend using HTML/JavaScript for universal accessibility.
- **Secure authentication:** Integrate login, access tokens, and HTTPS for improved security and privacy.
- **Elastic scaling:** Support for Kubernetes, Docker Swarm, or cloud-native orchestration to launch and manage servers on demand.
- **Advanced monitoring and analytics:** Add system dashboards, alerts, and in-depth performance metrics.
- **Spectator and tournament modes:** Allow more users to observe games, or coordinate distributed tournaments across servers.

9.4 Vertical Roadmap Diagram: Future Directions

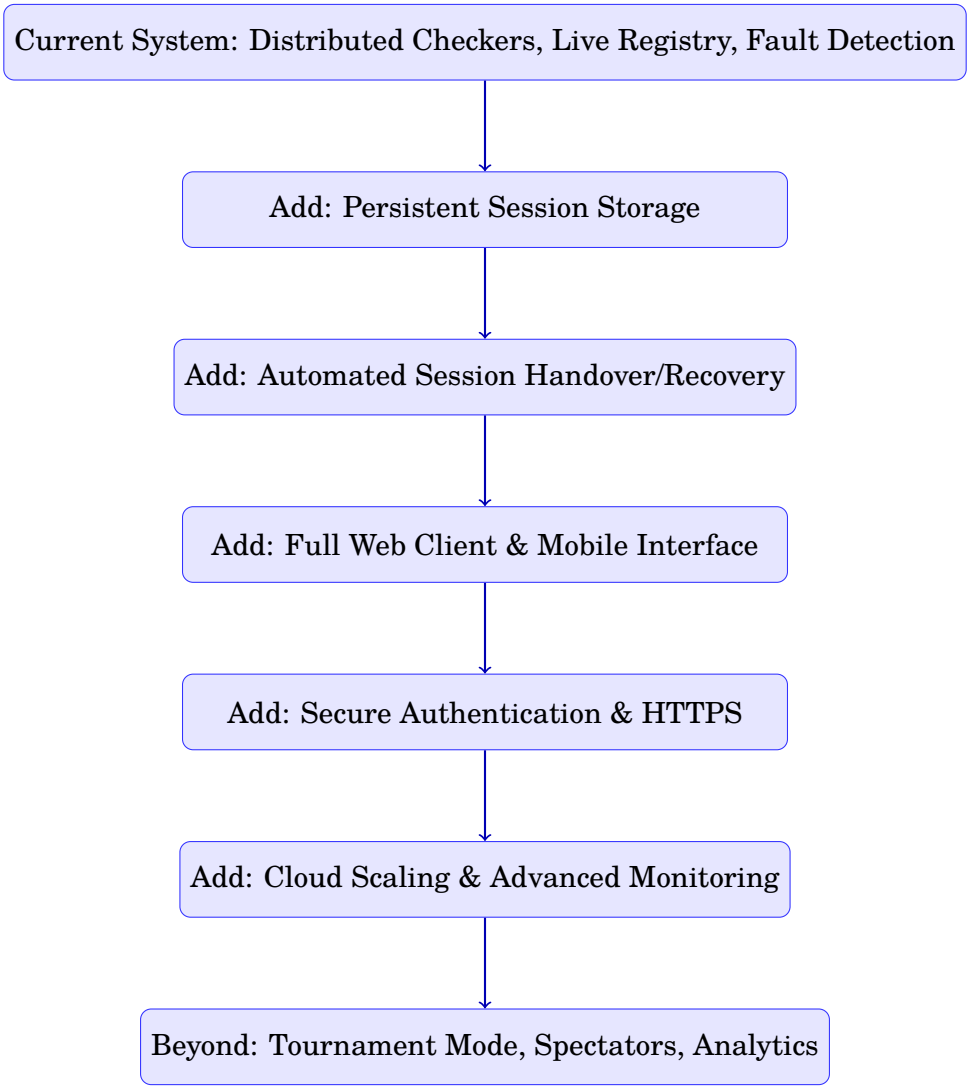

Figure 9.1: Vertical roadmap for future development of the distributed checkers platform.

9.5 Summary

The distributed checkers project achieves a strong baseline for real-time, multi-node gameplay and robust coordination. With further investment in persistence, web access, and scalability, the platform can evolve into a production-grade, cloud-native distributed game system.

Chapter 10

References

- **Erlang/OTP documentation:** `https://www.erlang.org/doc/`
- **Cowboy HTTP Server documentation:** `https://ninenines.eu/docs/en/cowboy/2.9/guide/`
- **Java documentation:** `https://docs.oracle.com/en/java/`
- **SparkJava microframework:** `http://sparkjava.com/documentation`
- **Unirest for Java:** `https://kong.github.io/unirest-java/`
- **Gson (Google):** `https://github.com/google/gson`
- **JavaFX documentation:** `https://openjfx.io/`
- **Maven build system:** `https://maven.apache.org/`
- **LaTeX TikZ package:** `https://ctan.org/pkg/pgf`
- **Distributed Systems Textbook:**
  Andrew S. Tanenbaum and Maarten van Steen, *Distributed Systems: Principles and Paradigms*, 2nd Edition, Pearson, 2006.
- **Distributed Systems and Middleware Technologies (Course Material):**
  University of Pisa, Prof. Alessio Bechini, 2024/2025.
- **Project Source Code:**
  `https://github.com/Rezacs/DistributedSystems`