Distributed Connect Four: A Fault-Tolerant Networked Multiplayer Game Architecture

Final Report for the Distributed Systems Course 2025/2026

Arvin Lajani
arvin.lajani@studio.unibo.it

May 28, 2026

This report details the architectural design, protocol specifications, and validation framework of a highly available, networked multiplayer implementation of the classic strategy matrix board game Connect Four. Developed using Python, FastAPI, WebSockets, and PyQt6, the system addresses core concerns of modern distributed environments, specifically fault isolation, dynamic session state routing, and state machine replication. Adhering to the principles of the CAP Theorem, the architecture favors an AP (Availability and Partition Tolerance) model to preserve match state permanence across abrupt software failures. By implementing a strict master-slave warm standby synchronization model governed by real-time heartbeat monitors, the system demonstrates an automated, transparent 10-second failover recovery routine. This design ensures seamless user connection migration and prevents data corruption or game session resets when an active cluster coordinator unexpectedly terminates.

Disclaimer

During the preparation of this work, the author utilized the Gemini large language model to refine grammatical phrasing, format block code syntax, and structure structural outline definitions within the LaTeX typesetting environment. After utilizing this service, the author thoroughly reviewed, modified, and edited the text as needed and takes full responsibility for the technical contents of the final report.

1 Introduction

This project implements a networked, fault-tolerant multiplayer version of the popular strategy board game "Connect Four." The client application is engineered using the `PyQt6` framework to deliver a highly responsive desktop Graphical User Interface (GUI)

where remote users can authenticate safely against an administrative directory, enter a synchronized matchmaking queue lobby, pair up into active session instances, and participate in real-time turn-based gameplay. The backend infrastructure layer operates as a decoupled microservice cluster, asynchronously managing multi-user connections, user queues, state persistence, turn-enforcement rules, and network fault events.

The core gameplay centers on a structured grid board geometry where two opposing players drop colored tokens into a vertical 6×7 matrix array. The backend server environment acts as the absolute authoritative source of truth, executing matrix validation algorithms, coordinating turn synchronization boundaries, and evaluating horizontal, vertical, or diagonal four-in-a-row winning alignments. To ensure absolute data integrity, thin clients are restricted from calculating game states locally; instead, user interactions are gathered as raw coordinates packets and dispatched over full-duplex network socket pipelines, updating client viewports only when a verified server configuration is broadcast back.

Crucially, the architecture's focus extends beyond standard game mechanics into foundational distributed systems concerns, specifically high availability, state machine replication (SMR), and automated network partition recovery. Operating under the constraints of the CAP Theorem, the platform establishes an AP (Availability and Partition Tolerance) topology. A primary server daemon continuously replicates its active match variables via an append-only JSON transaction log stream directly to a hot-standby backup cluster node. Regulated by a real-time 5-second heartbeat loop, the standby machine automatically detects primary process crashes and completes an unassisted backup-to-primary promotion within a strict 10-second failover timeout. This ensures complete transparency for active players, allowing game connections to migrate dynamically and continue uninterrupted without resetting volatile board states.

2 Concept

The users of the system are remotely located players who connect to the distributed backend components over a network infrastructure.

- **The type of product developed with the project:**
  - **Application**: The project is a desktop application with a Graphical User Interface (GUI), implemented using the `PyQt6` library. The client side provides a highly responsive graphical interface for players to interact dynamically with the game board.
  - **Command-line application**: The server components (`primary.py` and `backup.py`) can be considered command-line applications as they run entirely without a GUI in terminal environments and handle heavy back-end logic. However, the overall final product is classified as a graphical application, as it requires the GUI client to function from a user perspective.
  - **Library**: The project utilizes various native and third-party Python modules, but the project itself is built as a standalone runnable application, not a

library meant to be imported into other external software packages.

  - **Web-Service(s)**: The backend layer acts as a suite of decoupled web services. The matchmaking component leverages `FastAPI` to expose REST endpoints over HTTP, while the game engine services host full-duplex networking services over a network connection.

- **Use case collection:** The remote users interact with the system when they desire to play a Connect Four match against an online opponent, and the frequency of interaction depends entirely on their choice to join the lobby pool. The system handles and tracks temporary user data such as player names, assigned matching tokens, and operational game-specific variables including the 6×7 grid board matrix, current active turn tracking, and append-only move logs. All of these operational states are stored dynamically in volatile memory on the backend servers. The application accommodates multiple roles, transitioning an unauthenticated connection into an authenticated queue participant, before finally resolving into active opposing players (Player 1 and Player 2) within a specific match container.

- **Why is distribution needed:** Distribution is fundamentally required in this architecture to enforce two critical non-functional properties that a single localized machine cannot provide:
  - **Fault Tolerance and High Availability**: In a standard single-server architecture, a socket error or process termination causes an instantaneous failure of all ongoing matches. By distributing the backend across separate Primary and Standby Backup nodes, the application achieves resilience, executing an automatic server promotion loop to prevent data loss or game resets.
  - **State Authority and Resource Sharing**: To completely eliminate client-side manipulation or cheating, the authoritative game rules engine must be separated from the users. Distributing the state matrix onto remote backend services ensures that a shared source of truth is securely maintained and replicated across the network in real time.

2.1 Use Case Description

The core use case centers on arbitrary users authenticating against a central network pool, entering a synchronized pairing lobby, and executing turn-based matches of Connect Four.

- **What the software does**: It coordinates player pairing, enforces strict matrix move validation rules, replicates transient state updates, handles network failures, and determines victory configurations.

- **Where the users are**: Users are assumed to be geographically isolated nodes communicating over public or private TCP/IP network infrastructures.

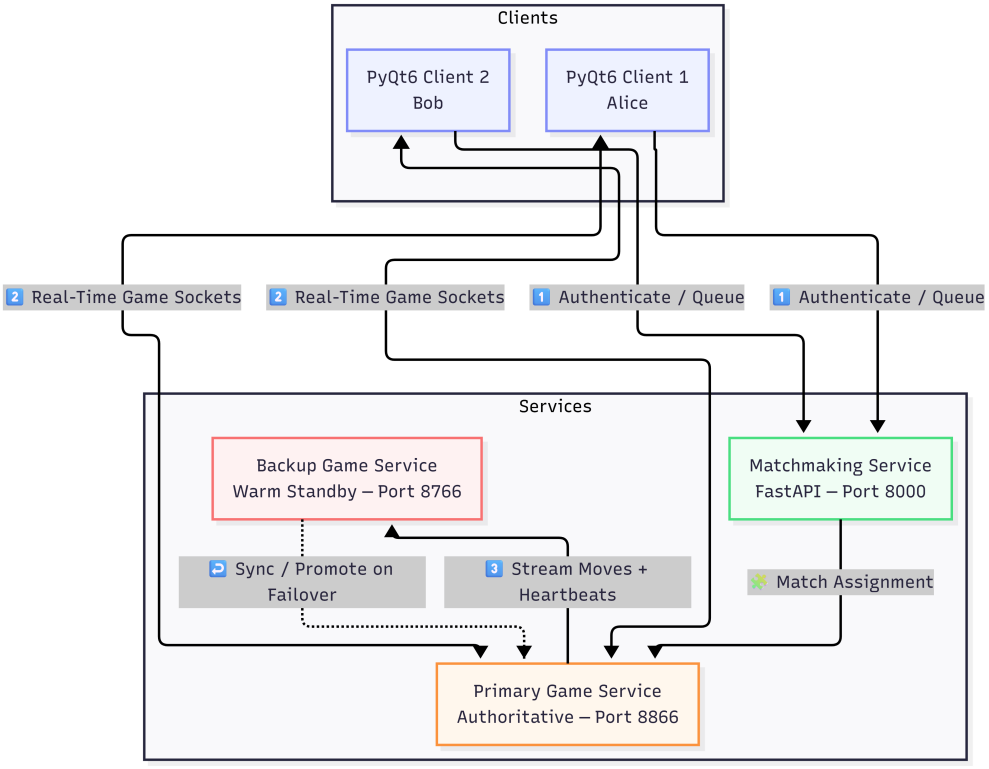

Figure 1: System Architecture (Section 2)

- **Interaction Frequency**: Users interact dynamically via real-time click inputs during a match, exchanging state payloads every few seconds.
- **Interaction Interfaces**: Interactions are managed entirely through a desktop GUI rendered on personal computers.
- **Data Storage**: Temporary session mappings, authentication pairings, and operational move logs are maintained concurrently within the transient storage of active backend worker services.
- **System Roles**: The architecture segregates participants into three distinct operational entities: *Unauthenticated Client*, *Authenticated Player Client*, and *Authoritative Server Daemon*.

2.2 Rationale for Distribution

A centralized local execution model is entirely unfeasible for multi-agent remote environments. Distribution is strictly mandated to solve several critical engineering challenges:

1. **Resource and State Sharing**: The board state matrix cannot reside on a singular user device, as this invites state divergence and client-side manipulation.
2. **Fault Tolerance and High Availability**: In single-server environments, a socket crash or hardware drop instantly destroys all concurrent volatile match sessions. Distribution enables the deployment of hot/warm standby nodes that absorb operational capacity transparently upon primary component dropouts.

3 Requirements Elicitation and Analysis

3.1 Functional Requirements

- **FR-1 (Authentication)**: The system must allow users to authenticate using pre-allocated credentials profiles.
  - *Acceptance Criterion*: Inputting valid combinations (e.g., `alice/123`) returns an HTTP 200 token response allowing lobby entry.
- **FR-2 (Matchmaking Queue)**: Authenticated users must be capable of signaling entry into a global queue.
  - *Acceptance Criterion*: The system pairs two waiting clients into an isolated session matrix as soon as both declare active queue states.
- **FR-3 (Turn Enforcement)**: The system must strictly enforce alternating turn execution boundaries across both clients.
  - *Acceptance Criterion*: Client interface column-drop inputs are blocked when the local player's state is evaluated as out-of-turn.

- **FR-4 (Win Validation)**: The system must identify horizontal, vertical, or diagonal four-in-a-row connections.
  - *Acceptance Criterion*: Instantly locking board states and broadcasting victory alert windows when alignment conditions evaluate to true.

3.2 Non-Functional Requirements

- **NFR-1 (Availability)**: The active game environment must survive unexpected primary server crashes.
  - *Acceptance Criterion*: If the active server process terminates mid-match, clients must resume play on an identical matrix board state within 12 seconds.
- **NFR-2 (Consistency)**: The state configuration rendered across Player 1 and Player 2 viewports must never diverge.
  - *Acceptance Criterion*: State distribution updates must confirm serialization lock-step parity before unlocking user interactions.

3.3 Implementation Requirements

- **IR-1 (Language and UI Engine)**: The entire system must execute natively within Python 3.10+, leveraging PyQt6 for all visual assets and interface rendering.
- **IR-2 (Asynchronous Network I/O)**: Network operations must utilize FastAPI for administrative endpoints and the native `websockets` library for persistent bidirectional channels.

3.4 Relevant Distributed System Features

- **Fault Tolerance and Dependability**: Highly Relevant. The system's principal achievement is its resilience against sudden process death using automated node migration.
- **Transparency**: Highly Relevant. The migration of network connections from a dead Primary node to a promoted Backup node must happen implicitly without forcing manual user reconnections or menu navigation.
- **Scalability**: Moderately Relevant. Decoupling the lobby matchmaking server from the game nodes allows additional game clusters to be added horizontally without bottlenecking entry networks.

4 Design

This chapter describes the structural and behavioral design strategies used to fulfill the requirements analyzed in section 3. The design choices outlined below emphasize the

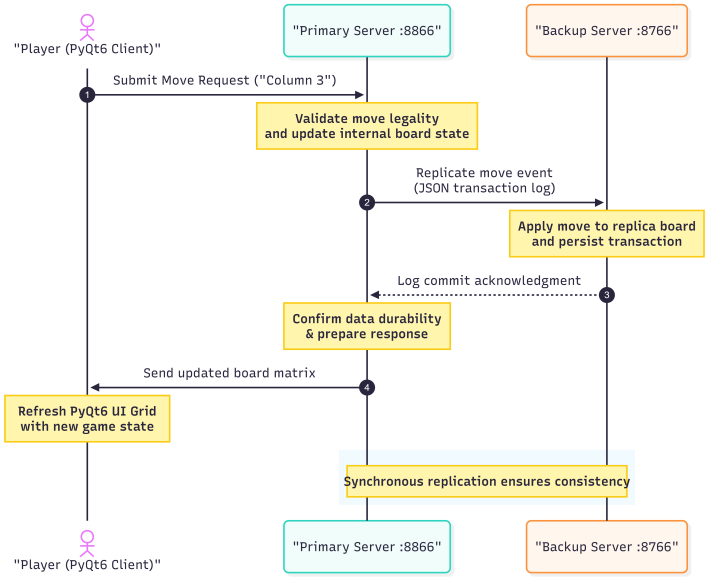

Figure 2: State Machine Replication Flow (Section 3.1)

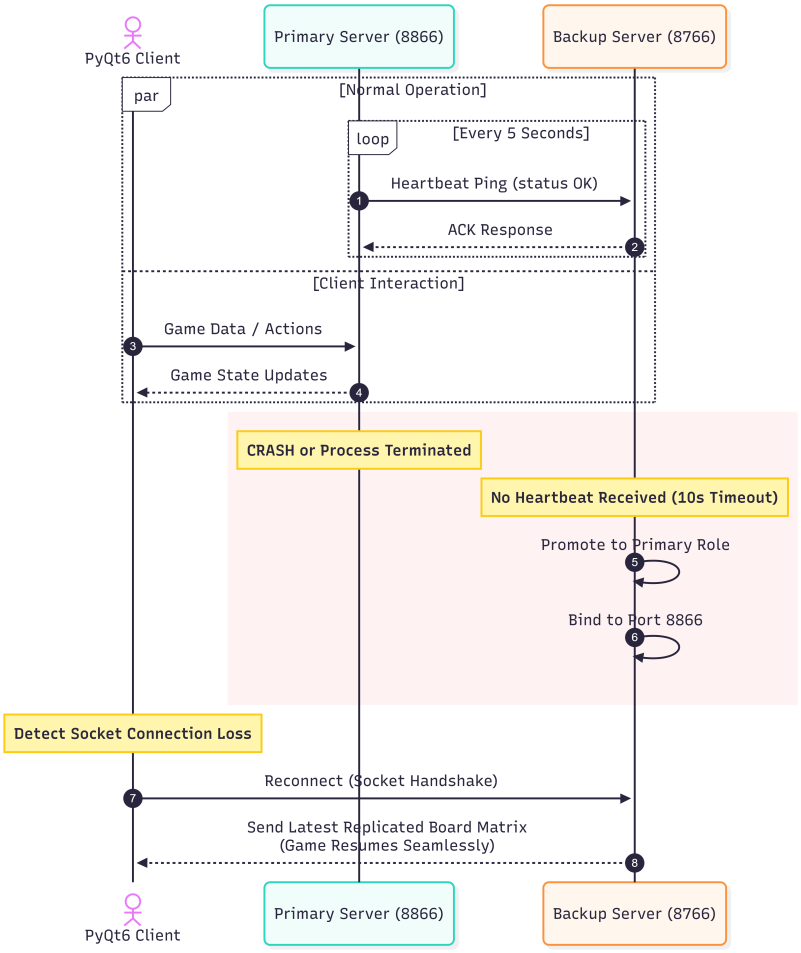

Figure 3: Heartbeat Failover Sequence(section3.2)

decoupling of system concerns, establishing a strict boundary between stateless gateway logistics and stateful, fault-tolerant replication loops that remain independent of final technological realizations.

4.1 Architecture & Infrastructure

The system implements a split-service, decentralized architectural style rather than a traditional monolithic structure. Responsibilities are cleanly isolated across discrete, self-contained network layers to optimize fault containment and availability. The infrastructure consists of four primary nodes distributed across the local network network space:

- **Matchmaking Gate (Port 8000)**: A public-facing gateway acting as the initial entry checkpoint for client registration, authentication, and session pooling.
- **Primary Game Node (Port 8866)**: The active authoritative coordinator responsible for managing live session states, evaluating rule matrices, and resolving turn cycles.
- **Backup Game Node (Port 8766)**: A hot/warm standby replica cluster node that mirrors the active primary state, completely isolated from direct client contact under normal operating conditions.
- **Graphical Thin Clients**: Client nodes that interface over TCP/IP transport loops to capture user input and render broadcast server configurations.

How these components discover each other and distribute across the processing space is explicitly detailed in the structural blueprint shown in fig. 4.

4.2 Modelling

The domain models follow a strict authoritative server configuration. The presentation layer is decoupled from the data state layout. The user-facing software holds simple presentation objects, while the absolute truth of the game session resides inside a centralized `GameState` class on the server side. The object structure encompasses several key properties:

- `board_matrix`: A localized 6×7 nested array representing the playing grid.
- `current_turn`: A unique identification hash tracking the active player.
- `move_history_log`: An append-only linear transaction array recording chronological cell commitments.

fig. 5 outlines the class attributes, operational methods, and structural associations that define the system's codebase.

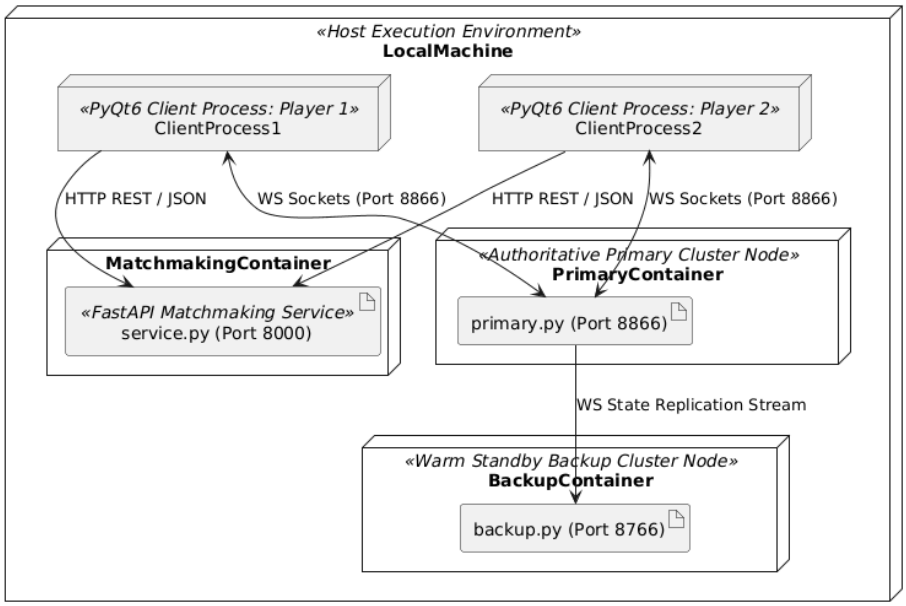

Figure 4: UML Deployment Diagram showing process topology, network boundaries, and interface ports.

4.3 Interaction

Under normal operating parameters, component interaction uses a bidirectional, full-duplex message pipe pattern. The system avoids inefficient HTTP polling mechanisms, relying instead on long-lived transport sockets to minimize latency and ensure immediate updates.

When a user selects a cell, the event is immediately packed as an intention request and pushed over the socket. The message sequence flows as follows:

1. The Client transmits a raw turn coordinate intent frame to the active Primary.
2. The Primary runs structural validation routines against its local `GameState` instance.
3. If the move is legal, the Primary serializes the delta mutation and pushes it down the replication channel to the Backup.
4. Upon receiving a log commitment acknowledgment from the Backup, the Primary commits the turn locally and broadcasts the updated matrix to both player clients.

This precise transaction sequence is mapped out chronologically in fig. 6.

4.4 Behaviour

The system components are divided into stateless and stateful behavior models:

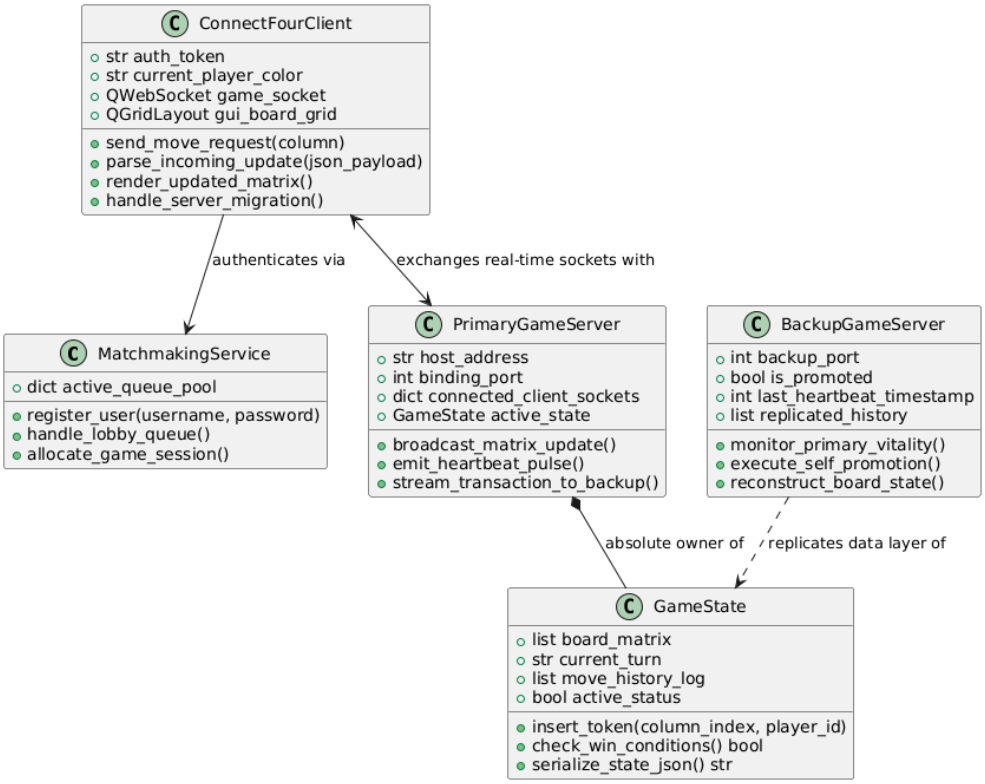

Figure 5: UML Class Diagram detailing object attributes, method definitions, and component scopes.

- **Matchmaking Hub**: Operates as a stateless service. It processes incoming registration, authentication, and token verification requests independently, routing client entries based entirely on transient parameters without keeping permanent socket state.
- **Game Engine Nodes**: Explicitly stateful. They trace the progression of active matches, manage the sequence of turns, and run background heartbeat monitoring loops to check the health of neighboring services.

The complete operational lifecycle of these services, highlighting how the Backup transitions its behavioral state during a critical system error, is illustrated in fig. 7.

4.5 Data and Consistency Issues

To maximize performance without introducing heavy database locking synchronization overhead, the system relies on an append-only state machine replication pattern. Persistent shared file systems are omitted to avoid single points of storage failure.

Instead, the Primary node streams transaction log frames to the Backup over an internal channel. The system guarantees *eventual consistency*; because the state machine

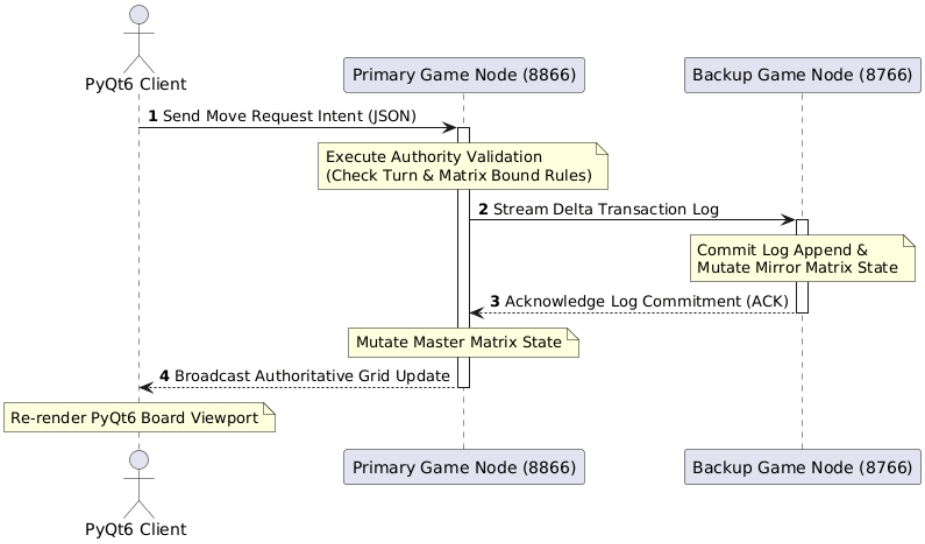

Figure 6: UML Sequence Diagram displaying message exchanges under standard State Machine Replication parameters.

transitions deterministically based on identical input logs, replaying the append-only move log on the Backup guarantees that its local matrix state perfectly matches the Primary's state upon recovery.

4.6 Fault-Tolerance

Fault isolation and automated error handling are managed via active standby network monitoring:

- **Heartbeat Exchange**: The Primary node transmits periodic vitality check packets to the Backup node at a strict **5-second interval**.
- **Timeout Countdown**: The Backup maintains a countdown timer. If a network partition occurs or the Primary encounters an unrecoverable process crash, the Backup notes the absence of pings.
- **Failover Action**: If the silence persists for a continuous **10-second timeout window**, the Backup declares the Primary dead. It terminates replication tracking, executes an unassisted promotion routine to assume the Primary role, re-binds to the main port (8866), and restores the active game loop using its intact mirrored state data.

4.7 Availability

In alignment with the CAP Theorem, the system prioritizes an **AP (Availability and Partition Tolerance)** configuration. If a network partition isolates a cluster node, the

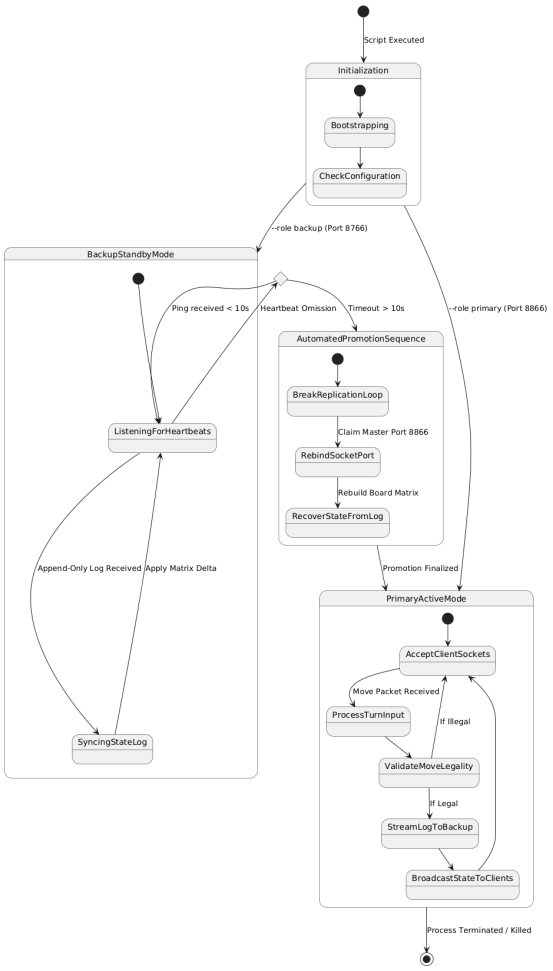

Figure 7: UML Finite State Machine Diagram visualizing node lifecycles and failover conditions.

system favors maintaining high availability for active users over enforcing absolute global consistency.

Rather than halting the game, the system relies on the automated failover sequence to quickly promote the Backup node. This minimizes down-time, allowing clients to automatically migrate their sockets and resume play without dropping connections or losing match progress.

4.8 Security

Security is managed at the entry boundary through a secure user directory interface to prevent malicious state manipulation or cheating:

- **Authentication Boundary**: Clients cannot establish direct socket handshakes

with the game clusters arbitrarily. They must first execute a secure handshake with the Matchmaking Gate using their account profiles.

- **Token Verification**: Successful authorization returns a signed identification token. This token must be included in the header of all subsequent WebSocket connections, allowing the game servers to verify user identities and assign appropriate player roles securely.

5 Implementation

This chapter details the concrete technical choices, programming frameworks, and data serialization formats used to translate the abstract design architectural specifications into an operational, concurrent platform. The implementation relies on asynchronous network communication and standard network transport loops to maximize computational efficiency and ensure non-blocking task processing across all distributed processes.

5.1 Network Communication & Data Representation

To minimize overhead during real-time interaction windows and provide structural support for concurrent connection pools, the network layer utilizes two distinct application-layer transport protocols running on top of a transmission control protocol (TCP) stack:

- **Hypertext Transfer Protocol (HTTP/REST)**: Used exclusively for short-lived, transactional management operations at the boundary gate. The Matchmaking service exposes endpoints utilizing standard HTTP POST and GET methods to securely process account creation, handle user login verification, and negotiate initial authorization handshakes.

- **Asynchronous WebSockets (WS)**: Once a client passes the boundary gate, standard HTTP channels are upgraded to long-lived, full-duplex WebSocket channels. This allows low-latency, real-time message exchange between the clients and the active game services over an existing TCP pipe, completely eliminating the polling overhead that would degrade a live turn-based strategy match.

- **Data Serialization (JSON Streams)**: All data in transit—including game room states, move validation intentions, heartbeat signals, and network migration coordinates—is serialized exclusively using structured **JavaScript Object Notation (JSON)** text streams. This lightweight data representation guarantees human-readable debugging capabilities and ensures open interoperability.

5.2 Technological Details & Directory Map

The complete codebase is written in Python, leveraging a microservice-inspired architecture where components execute as independent modules. Network logic relies on `FastAPI` backed by an ASGI server (`Uvicorn`) for high-concurrency event routing, while

the presentation layer utilizes the event-driven loop within the `PyQt6` toolkit to handle UI changes without freezing network thread queues.

To clarify the relationships between files, classes, and network configurations, table 1 maps each structural code script to its distinct technical role, network access configuration, and underlying software framework.

Table 1: System Directory Map and Component Technology Profile.

File Path Location	Core Architectural Role	Primary Framework	Network Port
`matchmaking/service.py`	Identity Auth & Lobby Queues	FastAPI / Uvicorn	8000 (HTTP)
`game_service/primary.py`	Game Rules & Log Syncing	Asynchronous WebSockets	8866 (WS)
`game_service/backup.py`	Heartbeats & Failover Promos	Asynchronous WebSockets	8766 (WS)
`client/gui.py`	Graphical Viewport Render	PyQt6 QWebSocket Client	Dynamic Client Port

5.3 Functional Source Module Analysis

The internal responsibilities of the files shown in table 1 are structured as follows:

1. **Lobby Coordination (`service.py`)**: Establishes an asynchronous routing pool loop. It tracks waiting users inside an in-memory thread-safe array list. The moment the list count matches two, it fires an internal game-spawning trigger, packages signed room credentials inside a JSON frame, and delivers it to the clients to guide their socket upgrade targets.

2. **Authoritative Primary Engine (`primary.py`)**: Operates the core state evaluation cycle. It accepts incoming WebSocket move intent integers representing columns, validates them against the current player token, checks cell availability inside the 6×7 list grid, updates the matrix layout, and immediately pushes an append-only delta replication frame down to the backup port channel before updating the clients.

3. **Warm Standby Monitor (`backup.py`)**: Operates on a continuous background listening loop. It consumes real-time replication log updates pushed from the Primary to maintain an identical backup state in local memory. Concurrently, it hosts an active tracking thread that counts down the elapsed time since the last heartbeat; if a silent window persists beyond 10 seconds, it instantly breaks out of standby mode, claims ownership of the master game port, and restores active gameplay.

4. **Graphical Thin Presentation Client (`gui.py`)**: Built as a pure rendering script. It wraps low-level network connections within a dedicated worker thread to prevent interaction lag on the main window canvas. When incoming messages arrive via the socket, the client parses the JSON string, matches the configuration tokens, and immediately maps the values to update color filters on the grid layout.

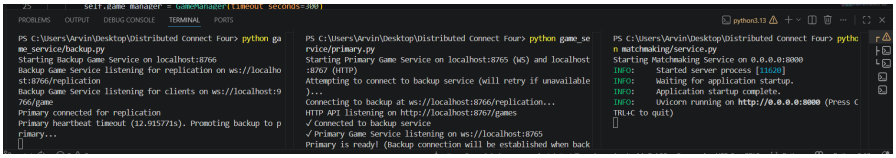

Figure 8: Parallel backend infrastructure daemons initializing ready listening ports across distinct execution environments

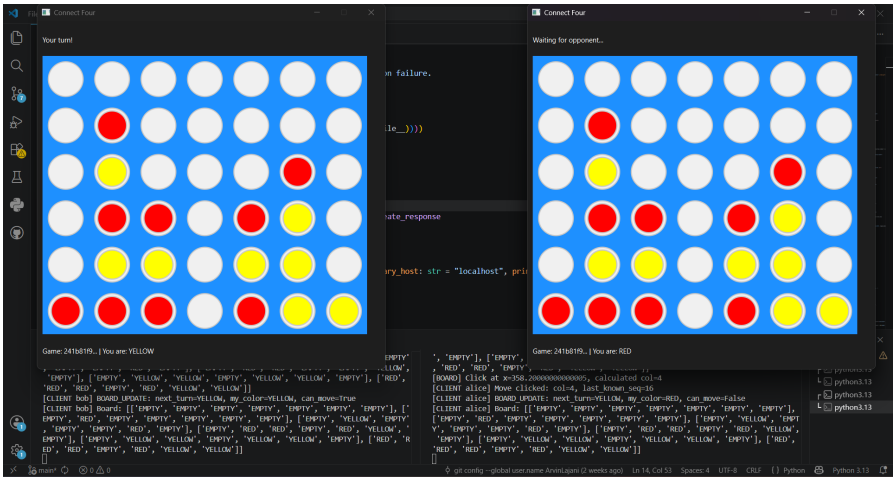

Figure 9: Active turn-by-turn game execution showing real-time state synchronization across dual client viewports

6 Validation

6.1 Automatic Testing

Automated verification routines focus on validating state calculations and protocol operations:

- **Unit Testing**: Scripts independently verify row/column array index limits and validate the win-checking algorithms across horizontal, vertical, and diagonal layouts.
- **Integration Validation**: Tests programmatically establish mock server loops to ensure JSON state frames parse correctly without throwing schema errors.

6.2 Acceptance Test

Manual acceptance tests were conducted to confirm the end-to-end functionality of the distributed cluster. These scenarios verify that network failover actions remain completely transparent to end users.

The empirical validation of the system's fault-tolerant capabilities was evaluated by executing a destructive runtime test scenario across the local cluster network.

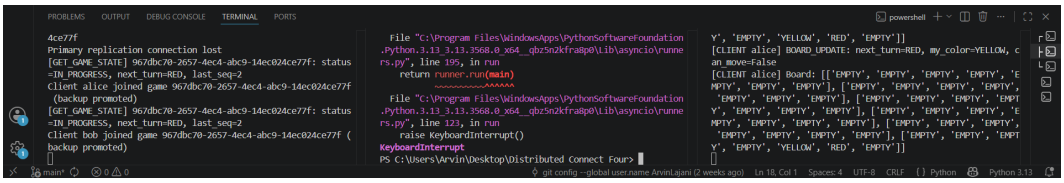

Figure 10: Active console log capture capturing regular 5-second heartbeat traces and the automated promotion sequence triggered by a Primary server failure

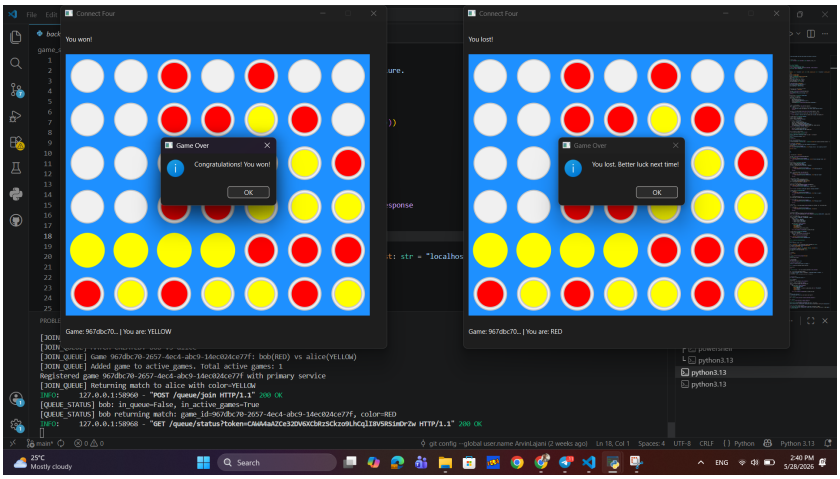

Figure 11: End-game victory validation. The backend rules engine locks inputs and displays modal notifications once 4 matches align.

As demonstrated by the initialization traces in **??**, the three backend service daemons successfully bootstrap inside separate execution windows, concurrently binding to their pre-allocated ports to listen for network traffic. During standard operating parameters, **??** highlights the normal gameplay synchronization phase. In this state, client input events are captured by the presentation layer, marshaled into lightweight JSON transport packages, and parsed dynamically across the authoritative primary service without inducing UI blockages or thread starvation.

The system's core resilience architecture was evaluated by introducing an unannounced process failure, where the Primary node daemon was forcefully killed mid-match. As captured by the real-time console log stream in **??**, the Backup standby node detects the immediate cessation of the 5-second heartbeat loop. Upon exceeding the strict 10-second promotion timeout deadline, the Backup successfully triggers its automated self-promotion routine, breaks the replication tracking loop, re-binds to the primary master port, and recovers the active session matrix from its local log mirrors without losing volatile data.

Finally, **??** confirms the structural integrity of the application's rule validation boundaries. Because the entire validation engine is strictly hosted on the server side, victory configurations are calculated securely on the promoted node, locking down user input streams and broadcasting matching victory modal frames to both thin viewports simultaneously..

7 Deployment

This chapter provides comprehensive, step-by-step documentation detailing the installation parameters, operational prerequisites, and sequence dependencies required to initialize the distributed multiplayer environment from scratch. To preserve runtime stability and prevent port allocation collisions, the deployment steps must be executed in a strict, sequential order across isolated shell instances.

7.1 Environment Prerequisites & Initialization

The software infrastructure is engineered to run as a cross-platform Python application. Prior to executing the orchestration files, the host deployment environment must satisfy the following system requirements:

- **Runtime Interpreter**: Python 3.10 or higher must be installed on the host operating system and appended to the system environment path variables.
- **External Component Libraries**: The external third-party dependencies responsible for GUI window compilation, networking, and asynchronous task loops must be downloaded via the Python Package Index (`pip`).

To initialize the host environment, execute the following command within the root folder directory:

```
pip install fastapi pyqt6 websockets uvicorn
```

7.2 Strict Dependency Execution Sequence

Because the runtime components rely on long-lived transport socket pipelines, they cannot handle arbitrary connection setups before the ports are listening. For instance, the authoritative Primary node must establish a connection with the Backup standby node immediately upon startup. Therefore, the initialization batch scripts (`.bat`) or command terminal instances must be launched in this precise, chronological sequence across separate, isolated console environments:

1. **Terminal Instance 1 (Warm Standby Engine)**: Initialize the isolated backup cluster daemon first by running the following script:

   ```
   .\start_backup.bat
   ```

   Expected Outcome: The console instantiates the service layer, outputs local configuration hashes, and prints a success indicator confirming that the socket loop is actively listening for incoming primary heartbeats on port `8766`.

2. **Terminal Instance 2 (Authoritative Primary Node)**: Once the backup service is listening, open a separate shell window and initialize the primary server daemon:

   ```
   .\start_primary.bat
   ```

   Expected Outcome: The process claims ownership of master port `8866`, establishes an active communication line to port `8766`, and starts transmitting 5-second vital heartbeat signals down the replication stream.

3. **Terminal Instance 3 (Lobby Management Gateway)**: Initialize the central administrative matchmaking hub:

   ```
   .\start_matchmaking.bat
   ```

   Expected Outcome: The `Uvicorn` ASGI server process initializes successfully, binding to port `8000` to handle incoming HTTP/REST registration and authentication traffic.

4. **Terminal Instances 4 & 5 (PyQt6 Graphical Thin Clients)**: Finally, to simulate a full remote match environment, open two separate shell windows and run the client script once in each window to launch Player 1 and Player 2:

   ```
   .\start_client.bat
   ```

   Expected Outcome: Two independent, fully functional PyQt6 desktop GUI application windows will render on the desktop display, establishing immediate network connection links to the matchmaking gate on port `8000`.

7.3 Configuration and Network Environment Profiles

Under standard local evaluation profiles, all service configurations defaults to loopback addresses (`127.0.0.1`). If deploying the architecture across a physically distributed local area network (LAN) with multiple physical machines, you must update the core connection endpoints within a central environment configuration file prior to boot-up. Ensure that the clients point their socket targets to the specific public IP address hosting the primary node, and that the firewall policies on the host machines are configured to open ports `8000`, `8866`, and `8766` for standard TCP traffic.

8 User Guide

Operating the distributed Connect Four application is designed to be highly intuitive, leveraging a responsive visual workflow that completely abstracts the underlying complex network logic and failover routines from the end-user. This section serves as an operational manual, guiding users through the chronological phases of application lifecycle management, match execution, and manual fault-tolerance observation.

8.1 Phase 1: User Identity Authentication

Upon executing the graphical client initialization script ($.$ *client.bat*), *thesystemcompilesandlaunchesthepri*

```
Navigate to the credential input fields rendered within the center panel
of the interface window.

Input a pre-allocated test profile combination into the respective forms.
For standard cluster testing, valid embedded profiles include alice (with
password 123) for Player 1, and bob (with password 123) for Player 2.

Click the Log In interactive button element to dispatch a secure verification
payload to the Matchmaking Gateway listening on Port 8000.

Expected Visual Outcome: Upon successful verification, the authentication
input frame fades out, and the client application smoothly transitions
to expose the global matchmaking pool lobby view.
```

8.2 Phase 2: Matchmaking Queue Entry and Session Spawning

Once authenticated within the global lobby system, players must register their presence within the active pairing pool to instantiate a match container.

1. Click the prominent **Enter Matchmaking Queue** action button located in the center of the viewport window. The client status bar will update to display a `"Searching for Opponent..."` tracking message.

2. Repeat this process on the second independent client window to satisfy the multi-agent requirements of the match container.

3. *Expected Visual Outcome*: The moment both nodes occupy active queue states, the Matchmaking service maps the pairing, allocates the socket endpoints, and closes the lobby view. The GUI dynamically re-renders to reveal the interactive, vertical 6×7 grid board layout initialized to an empty state.

8.3 Phase 3: Real-Time Gameplay Input Execution

The match proceeds under strict authoritative server turn boundaries. Player 1 (Red tokens) and Player 2 (Blue tokens) execute alternate cell inputs until a terminal condition is evaluated.

1. Observe the active turn notification header displayed at the top of the interface window to confirm that the server has unlocked your client input controls.

2. Position your mouse cursor over any of the seven column arrow indicators situated directly above the board matrix canvas.

3. Click the target column arrow indicator to transmit a token drop request coordinate packet across the active WebSocket line.

4. *Expected Visual Outcome*: The piece falls deterministically to the lowest available empty row cell index within that specific column. The turn indicator instantly updates, locking down your interface controls and un-locking the interactive grid elements on your opponent's remote display.

8.4 Phase 4: Live Observation of Fault-Tolerant Failover

To actively observe and validate the high-availability replication mechanics engineered into the backend service layer, users can perform a destructive runtime evaluation mid-game:

1. While a match is actively progressing with multiple pieces visible on the grid canvas, locate the running command terminal instance executing the **Primary Server process** (`primary.py` binding to Port 8866).

2. Forcefully terminate the primary process. This can be accomplished by directly closing the shell window layout or by clicking inside the command console and inputting the interrupt shortcut sequence `Ctrl + C`.

3. Immediately shift attention back to the running client GUI viewports.

4. *Expected Visual Outcome*: The application will briefly freeze interaction lines for a maximum of 10 seconds as the socket recognizes the connection dropout. Behind the scenes, the Backup server tracks the heartbeat timeout and promotes itself to the Primary role on Port 8866.

5. Without any manual button interaction, page refreshes, or menu navigation, the thin clients dynamically re-establish their full-duplex socket handshakes with the promoted server node. The game resumes seamlessly with the exact matrix pattern, turn state, and move histories fully intact.

9 Self-Evaluation

9.1 Arvin Lajani - Evaluation

My primary contribution to the project focused on engineering a resilient, highly available distributed architectural design. The structural goals prioritized absolute backend decoupling, strict authoritative state enforcement, and robust fault-recovery loops over complex graphical gameplay modifications, aligning directly with core distributed system engineering principles.

- **Strengths (Architectural Decoupling and Structural Isolation)**: The system successfully separates operational concerns across completely independent microservice daemons. By isolating the public-facing matchmaking gate from the stateful game clusters, process vulnerabilities are contained, ensuring that high traffic volumes at the entry boundary cannot compromise ongoing match instances.
- **Strengths (Fault Resilience and Transparent Self-Healing)**: The implementation of a State Machine Replication (SMR) pipeline backed by an automated heartbeat verification loop provides robust high availability. The backup node successfully maintains state parity in volatile memory via real-time transaction streaming. Upon experiencing an unannounced primary node failure, the system executes an unassisted standby-to-primary promotion and local port re-binding sequence within strict timeout deadlines, preserving volatile game matrices and allowing transparent client socket migration without dropping active sessions.
- **Strengths (Asynchronous Concurrency Model)**: Leveraging Python's asynchronous event loops combined with standard full-duplex WebSocket connections ensures optimal resource efficiency. The architecture effortlessly handles concurrent multi-user queues and parallel game containers without encountering thread lockages, race conditions during matrix mutations, or presentation lag on the graphical client viewports.