Distributed Chess Team Battle

Final Report for the Distributed Systems Course 2025/26

Matteo Raggi
matteo.raggi3@studio.unibo.it

January 21, 2026

This report presents 'Distributed Team Chess', a cooperative version of Chess [1] designed to demonstrate fundamental principles of Distributed Systems within an interactive environment.

Unlike traditional turn-based games, this system introduces a new gameplay mechanic based on **Role Sharding** and **Distributed Consensus** [2]. Players are organized into teams where each node holds exclusive write permissions over a specific subset of game resources (specific chess pieces). To execute a state transition (a move), the team must reach a consensus through a voting protocol, transforming the game into a distributed coordination challenge. This way to play chess encourages collaboration, communication, and strategic planning among team members, giving the game a new dynamic and engaging dimension.

From an architectural perspective, the system adopts a **Broker-Based Hybrid Peer-to-Peer** model [3]. A scalable .NET [4] backend, orchestrating communication via SignalR and persisting state on Redis [5], acts as a coordination bus and transaction log. This design choice prioritizes **Strong Consistency** and **Partition Tolerance** (CP in CAP [6] terms), ensuring a single source of truth while offloading preliminary move validation logic to client nodes to simulate P2P filtering.

The project places a strong emphasis on **Fault Tolerance** and **High Availability**. It implements automatic failure detection: in the event of a node crash or disconnection, a failover mechanism triggers a dynamic **Rebalancing** of the shards, transferring control of the 'orphaned' pieces to surviving teammates to ensure system liveness.

Furthermore, a timeout-based supervisor prevents distributed deadlocks during the voting phase. This report details the architectural decisions, the consistency models adopted, and the resilience strategies implemented to guarantee a seamless user experience in a volatile distributed environment.

1 Concept

This project focuses on the development of **Distributed Team Chess**, a real-time multiplayer platform that reimagines the classic game through the lens of distributed coordination.

1.1 Type of Product

The system is designed as a composite software solution consisting of:

- A **Web Application** (Single Page Application) developed in Angular [7], serving as the interactive client for players.
- A **Distributed Backend Service** developed in C# (.NET 8), acting as the coordination bus and state manager.
- An infrastructure layer orchestrated via Docker Compose [8], including Redis for state persistence.

1.2 Use Case Description

- *What is the software doing?* The system allows users to create and join chess matches in two distinct modes: Classic (1v1) or **Team Consensus** (e.g., 2v2). In Team mode, the control of the pieces is *sharded* among teammates (e.g., Player A controls Pawns, Player B controls Rooks). To execute a move, a player must propose it, and the team must reach a consensus via voting.
- *Where are the users?* Users are physically distributed across different network nodes. They effectively act as peer nodes in a logical cluster (the Team) that must coordinate to make progress in the game.
- *When do they interact?* Interactions occur in **Real-Time**. The system requires low-latency communication to synchronize board states, proposals, votes, and timers (synchronous interaction model).
- *How do they interact?* Users interact via modern web browsers (desktop or mobile) supporting WebSockets. The UI provides visual feedback for role permissions (which pieces can be moved) and voting procedures (accept/reject proposals).
- *Data Storage:* The system stores **Session Data** (Game State, FEN strings, Active Proposals, Player Roles). Data is stored in a distributed in-memory store (**Redis**) to ensure high availability and fast access for all backend replicas.
- *Roles:*
  - **Proposer:** A player initiating a state change (move).
  - **Voter:** A teammate validating the proposal against their local logic engine.
  - **Coordinator:** The backend acting as the ordering service for transactions.

1.3 Why is distribution needed

Distribution is not merely a deployment detail but a core functional requirement of this project:

- **Resource Sharing & Sharding:** The game state (the board) is a shared resource. Access to write operations on specific parts of this resource (specific pieces) is partitioned (sharded) among different users to force coordination.
- **Coordination and Consensus:** Unlike standard chess, a move is valid only if validated by the distributed consensus of the team nodes. This requires a protocol to propose, vote, and commit changes across the network.
- **Fault Tolerance:** The system is designed to handle node failures (crash failures). If a player disconnects during a match, the system must detect the failure and trigger a **Failover** mechanism, automatically redistributing the disconnected player's permissions (shards) to the surviving teammates to ensure the game's liveness.
- **Consistency:** Maintaining a coherent view of the board across multiple clients and backend instances requires a centralized persistence layer (Redis Backplane) to prevent Split-Brain scenarios.

2 Requirements Elicitation and Analysis

This section outlines the functional and non-functional requirements of the Distributed Team Chess system. The focus is on the distributed coordination aspects rather than the standard rules of chess.

2.1 Functional Requirements

1. **[FR-01] Game Session Management** The system must allow users to create and join game sessions with configurable parameters (Game Mode: Classic 1v1 or Team Consensus).
    - *Acceptance Criterion:* A user can create a lobby specifying the number of players per team. Other users can see the lobby in the list and join until capacity is reached.
2. **[FR-02] Pre-Game Coordination (Ready State)** Once inside a game room ('Waiting Room'), the system must allow users to signal their availability to start. The system must display the list of connected players and their current status (Ready / Not Ready).
    - *Acceptance Criterion:* Players can toggle their status. The game officially starts (broadcasting the initial board state) only when **all** connected players have set their status to 'Ready' and the room capacity is full.

3. **[FR-03] Dynamic Role Sharding** In Team Consensus mode, the system must partition the control of game resources (chess pieces) among team members.
    - *Acceptance Criterion:* Upon game start, each player is assigned a specific subset of pieces (e.g., Pawns vs. Non-Pawns). Players are visually restricted from interacting with pieces they do not own.
4. **[FR-04] Distributed Move Proposal** To execute a move, a player must initiate a proposal sequence rather than modifying the state directly.
    - *Acceptance Criterion:* When a player attempts a move on an owned piece, a 'Move Proposal' is broadcast to teammates. The game state does not change immediately.
5. **[FR-05] Consensus Voting Mechanism** The system must collect votes from team members to validate a proposed state transition.
    - *Acceptance Criterion:* Teammates can vote to Approve or Reject a proposal. The move is applied to the global state only if a majority quorum is reached.
6. **[FR-06] Synchronized Gameplay Visualization** All connected nodes must maintain a synchronized view of the game board and temporal events.
    - *Acceptance Criterion:* Updates such as moves, timer ticks, and player disconnections are reflected on all clients with eventual consistency (visual updates) and strong consistency (turn management).
7. **[FR-07] Automatic Failover (Rebalancing)** The system must detect node failures (disconnections) and ensure game continuity.
    - *Acceptance Criterion:* If a player disconnects, their assigned piece permissions must be automatically transferred to a surviving teammate.
8. **[FR-08] Deadlock Resolution** The system must prevent the game from stalling if a team fails to reach a decision.
    - *Acceptance Criterion:* If a turn timer expires without a consensus, the system automatically enforces a resolution (e.g., executing the most voted move or a random valid move) to progress the game state.

2.2 Non-Functional Requirements

1. **[NFR-01] Strong Consistency** All connected nodes must eventually view the exact same board state (FEN) after a move is committed.
    - *Acceptance Criterion:* No 'Split-Brain' scenarios allowed. The state stored in the central persistence layer (Redis) acts as the single source of truth.
2. **[NFR-02] Real-Time Responsiveness** The system must propagate state changes and votes with minimal latency to ensure playability.

3. [**NFR-03**] **Horizontal Scalability** The system must support multiple backend instances serving different subsets of users.
    - *Acceptance Criterion:* Two users connected to different backend nodes (containers) must be able to play in the same match seamlessly.

2.3 Implementation Requirements

1. [**IR-01**] **Tech Stack:** The backend must be developed in **C# (.NET 8)** and the frontend in **Angular**.
    - *Justification:* C# provides native support for high-performance WebSocket management via SignalR. Angular's reactive model (RxJS) is ideal for event-driven UIs.
2. [**IR-02**] **Containerization:** The system must be deployable via **Docker Compose**.
    - *Justification:* Essential to simulate a distributed environment with multiple nodes and a shared data store on a single development machine.
3. [**IR-03**] **Persistence:** The system must use **Redis**.
    - *Justification:* Required to implement the 'Backplane' pattern for distributed pub/sub and state persistence across volatile backend instances.

2.4 Relevant Distributed System Features

The following analysis motivates which distributed systems features are critical for the success of the project:

- **Resource Sharing:** This is the core concept of the project. The 'Chessboard' is a shared resource accessed concurrently by multiple nodes. The system implements a locking strategy via *Sharding* (partitioning write permissions) to manage concurrent access and prevent conflicts.
- **Fault Tolerance:** Given the session-based nature of the game, a crash of a single client node must not terminate the service for others. The implementation of **Failover strategies** (reassigning pieces) and **Timeouts** (for voting deadlocks) ensures that the system remains Available even in the presence of partial failures.
- **Transparency:**
    - *Location Transparency:* Users are unaware of which backend instance they are connected to or where the Redis store is located.
    - *Failure Transparency:* While the user is notified of a teammate's disconnection (to understand why they have new pieces), the mechanism of rebalancing happens automatically without requiring a game restart.

- **Scalability:** The architecture is designed to be horizontally scalable. By using Redis as a Backplane, the system can scale the number of backend nodes to handle an increasing number of concurrent matches.
- **Security & Trust:** In a P2P context, trust is a major issue. The system implements a 'Defense in Depth' strategy:
  - *Distributed Validation:* Teammates act as validators for proposals.
  - *Authoritative Fallback:* The backend acts as a final oracle to prevent collusion attacks (e.g., a team hacking their clients to vote for an illegal move).
- **Concurrency & Performance:** The system relies on asynchronous message passing (SignalR) to handle multiple votes arriving simultaneously. The eventual consistency of the UI is acceptable as long as the final state (Commit) is strongly consistent.

3 Design

This chapter explains the strategies used to meet the requirements identified in the analysis.

3.1 Architecture

The architectural style used in this project is named **Broker-Based Hybrid Peer-to-Peer**. This architecture is characterized by the presence of a central *broker* component that facilitates communication and coordination among distributed *peer* components. The broker is the central intermediary, the backend, that manages message routing, state persistence, and coordination logic. The 'hybrid peer-to-peer' aspect comes from the fact that the clients (peers) communicate through the broker, but the security checks for move validation and vote are performed locally on each peer before sending messages to the broker and when receiving messages from it.

The Broker-Based Hybrid Peer-to-Peer architecture is particularly well-suited for the Distributed Team Chess system for several reasons:

- **Centralized Coordination:** The broker (backend) provides a centralized point for managing game state, enforcing rules, and coordinating actions among distributed players. This is crucial for maintaining consistency in a game where multiple players must agree on moves. This adds an additional layer of trust and validation, as the broker can act as an authoritative source of truth.
- **Flexibility:** The hybrid peer-to-peer aspect allows clients to perform local validations and optimizations.
- **Simplified Client Logic:** Clients can focus on user interaction and local validation, while the broker handles complex coordination tasks, simplifying client implementation.

3.2 Infrastructure

The infrastructure of the system is designed to simulate a scalable distributed environment using containerization technology. The architecture is composed of the following infrastructural components:

- **Clients (N instances):** The user entry points. These are Single Page Applications (SPA) running in the users' web browsers. They act as the edge nodes of the distributed system, responsible for local state visualization and P2P validation.
- **Application Servers (M instances):** The computation nodes (*ChessBackend*). These are stateless containers hosting the .NET runtime. Multiple instances can run simultaneously to handle increased load, as they do not retain any long-term state in their local memory.
- **The State Store / Message Broker (1 instance):** A Redis container acting as the infrastructural backbone. It serves two distinct roles:
  1. **Persistence Layer:** Storing the volatile state of active games (Game Rooms, Players, Proposals).
  2. **SignalR Backplane:** Enabling the Pub/Sub mechanism required for the application servers to synchronize real-time messages across the cluster.

Distribution and Network Topology

The components are distributed over a virtualized network managed by **Docker Compose**. The **Backend Servers** and **Redis** reside within a private virtual network (bridge network). They communicate via high-speed internal TCP connections. And the **Clients** reside on the host machine (or potentially external machines) and connect to the Backend Servers. In a production scenario, the Backend servers would be distributed across different availability zones behind a Load Balancer, while Redis would be deployed as a managed cluster.

Service Discovery

Service discovery is handled differently depending on the communication direction:

- **Server-to-Redis:** Discovery is handled via Docker's internal DNS resolution. The backend services communicate with the hostname `redis`, which Docker resolves to the internal IP of the container.
- **Client-to-Server:** In this simulation environment, clients connect to specific exposed ports (e.g., localhost:5000, localhost:5001). In a real-world deployment, this would be abstracted by a Reverse Proxy (e.g., NGINX [9]) or a DNS Load Balancer providing a single entry point (e.g., `api.chessgame.com`).

- **Server-to-Server:** Application servers do not communicate directly with each other. They interact indirectly through the Redis Backplane, effectively decoupling the nodes.

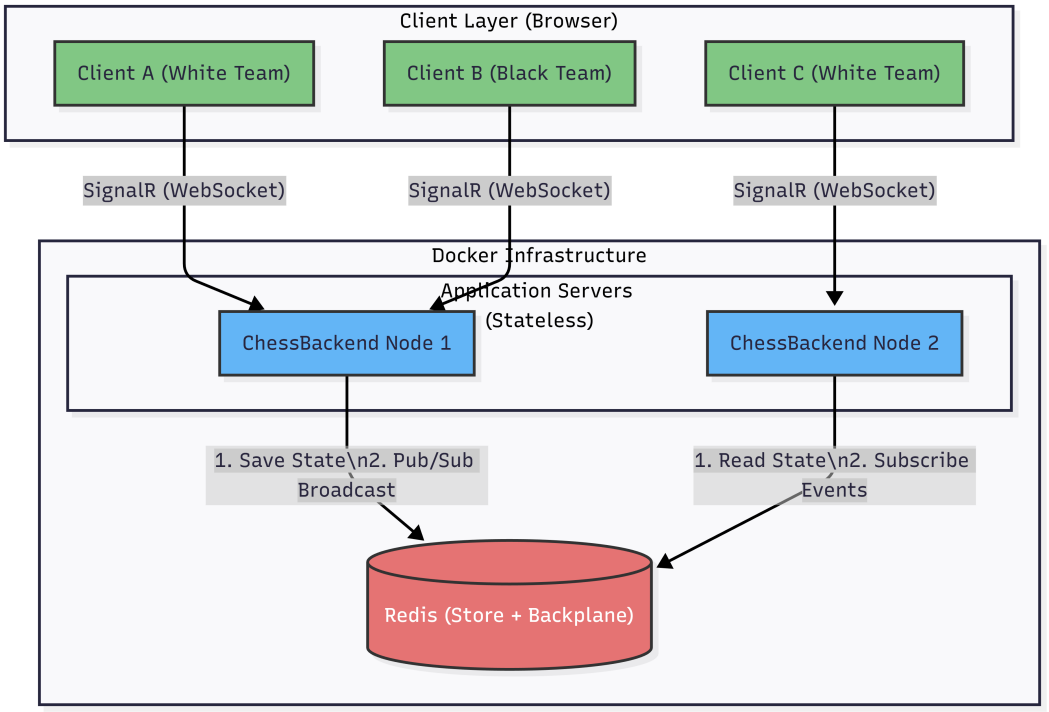

Figure 1: System Infrastructure Diagram. It illustrates the Hybrid architecture where multiple stateless backend nodes synchronize via the Redis Backplane, enabling clients from different teams to communicate seamlessly.

3.3 Modelling

The domain model is designed to support the distributed nature of the game, specifically the requirements for sharding and consensus.

3.3.1 Domain Entities

The core logic revolves around three main entities:

- **GameRoom:** This is the primary entity representing a match. It acts as the consistency boundary. It holds the board state (FEN string), the configuration (Mode, Capacity), and the mapping of teams and permissions.
- **Player:** Represents a user session. It contains identity information (Name, ID) and infrastructural metadata (SignalR ConnectionId) required for routing messages to specific nodes.

- **MoveProposal:** A transient entity representing a 'candidate state transition'. It is created when a player initiates a move and is destroyed when the voting process concludes (Commit or Reject). It tracks the proposal details and the set of cast votes.

3.3.2 Data Structures Detail

The following list details the specific attributes of the domain models, which map directly to the JSON structures stored in Redis.

GameRoom The aggregate root representing the match state.

`GameId` Unique identifier used as the primary key for Redis storage.

`GameName` A user-friendly display name for the lobby.

`Players` A list of `Player` objects currently connected to the room.

`Mode` Enum indicating the game type (`Classic1v1` or `TeamConsensus`).

`Capacity` The maximum number of players allowed (e.g., 4 for a 2vs2 match).

`Teams` A dictionary mapping `PlayerId` to their team color ('w' for White, 'b' for Black).

`Fen` The Forsyth–Edwards Notation string representing the current board state. This is the single source of truth for the game logic.

`PiecePermissions` A dictionary implementing the **Sharding** logic. It maps a `PlayerId` to a list of allowed piece characters (e.g., `['P', 'K']`).

`ActiveProposals` A list of `MoveProposal` objects currently pending validation by the team.

`LastMoveAt` Timestamp of the last committed move, used by the Timeout Service to enforce liveness.

MoveProposal Represents a candidate state transition.

`ProposalId` Unique identifier for the specific voting process.

`ProposerId` The ID of the player initiating the move.

`From, To, Promotion` The payload describing the chess move (UCI format).

`Votes` A `HashSet<String>` containing the IDs of players who approved the proposal. Using a Set ensures voting idempotency (a player cannot vote twice).

`CreatedAt` Timestamp used to calculate the specific expiration of a proposal (if different from the turn timer).

Player Represents the user session data.

`PlayerId` The persistent logical identifier of the user (stored in the client's LocalStorage).

`PlayerName` The display name chosen by the user.

`ConnectionId` The ephemeral SignalR socket ID. This allows the backend to route messages to specific physical connections (e.g., for team-private broadcasts).

`IsReady` Boolean flag for pre-game synchronization.

`CurrentGameId` Stores the reference to the active game. Crucial for the **Failover** mechanism: if a connection drops, the server uses this field to locate the room and redistribute the player's shards.

3.3.3 Infrastructure Mapping

The mapping between domain entities and infrastructural components follows a **Session-Based Persistence** pattern:

- **Persistence Layer (Redis):** The `GameRoom` and its children are serialized as JSON and stored in Redis. Redis acts as the *System of Record*.
- **Broker Layer (Backend):** Entities are deserialized into memory only during the execution of a request (Hub Method) and immediately saved back to Redis. This ensures the backend remains stateless.
- **Client Layer (Frontend):** Clients maintain a 'Shadow Copy' of the state for visualization. This local state is updated via real-time events but is considered non-authoritative.

3.3.4 Messages and Interaction Flow

The communication of the system is based on messages. These messages can be send from Client to server or vice versa. They can be classified into three categories: Commands, Events, and Queries. Commands are used to request an action or change in the system state. Events are notifications about changes that have occurred. Queries are requests for information without causing any state change.

3.3.5 Messages and Interaction Flow

The communication relies on a strict distinction between **Commands** (intent), **Events** (facts), and **Queries** (state requests).

1. **Commands (Client → Server):**
    - `JoinLobby / JoinGame`: Requests to enter the global lobby or a specific game room.

- `CreateGame`: Request to instantiate a new match with specific parameters (Mode, Team Size).
- `LeaveGame`: Request to disconnect from a room (triggers Failover logic).
- `ReadyGame`: Signals that the user is ready to start. Used to synchronize the game start.
- `ProposeMove`: A request to initiate a state transition (instead of a direct `MakeMove`, which is deprecated in Team Mode).
- `VoteMove`: A confirmation to support a specific proposal.

2. **Events (Server → Clients):**
   - `Lobby Updates`:
     - `PlayerJoinedLobby / PlayerLeftLobby`: Updates the online users list.
     - `GameCreated / DeletedGame`: Updates the available rooms list in real-time.
   - `Room Updates`:
     - `PlayerJoinedGame / PlayerLeftGame`: Notifies roommates of presence changes (triggers Rebalancing).
     - `PlayerReadyStatus`: Updates the "Ready/Not Ready" indicators.
   - `Gameplay Events`:
     - `GameStart / GameOver`: Lifecycle events marking the session boundaries.
     - `ActiveProposalsUpdate`: Broadcasts the full list of pending proposals and votes. Crucial for P2P validation visibility.
     - `ProposalResult`: Notifies if a proposal was rejected (e.g., due to timeout or veto).
     - `MoveMade`: Signals that consensus was reached, carrying the new FEN state.
3. **Queries (Client → Server):**
   - `RequestGameState`: Used upon reconnection (F5) or failover to retrieve the full snapshot of the `GameRoom` (FEN, Teams, Permissions) and resynchronize the local state.

3.3.6 System State

The global state of the system at any given time t is persisted in Redis as a `GameRoom` object and synchronized to clients via the `GameStateMessage` DTO. The state comprises four logical areas:

- **Session Configuration:** Includes static or semi-static metadata defining the match rules.
  - `GameId`: The unique correlation identifier.
  - `Mode`: The architectural mode (`Classic1v1` or `TeamConsensus`).
  - `Capacity`: The strict limit on concurrent connections allowed.
- **Participation & Roles:** Defines who is in the room and their allegiance.
  - `Players`: The list of currently connected user sessions (for presence awareness).
  - `Teams`: The mapping associating each User ID to a side (White or Black).
- **Game Logic Layer:**
  - `Board Configuration (FEN):` The Forsyth–Edwards Notation string representing the absolute truth of the chess board (piece positions, active color, castling rights).
  - `Sharding Map (PiecePermissions):` The dictionary enforcing the distributed role logic, defining which specific subset of pieces a node is authorized to write to (move).
- **Coordination Layer:**
  - `Consensus State (ActiveProposals):` The list of pending state transitions currently under voting.
  - `Temporal State (LastMoveAt):` The synchronization timestamp used by the Timeout Service to detect distributed deadlocks and enforce liveness.

3.4 Interaction

The interaction between components is strictly event-driven and asynchronous. The system moves away from the traditional Request-Response cycle typical of REST APIs, adopting instead a persistent full-duplex communication model to minimize latency and support unsolicited server pushes.

3.4.1 Communication Patterns

The interaction between system components is designed to be strictly event-driven and asynchronous, moving away from the traditional request-response cycle typical of REST APIs. To achieve low-latency synchronization and state consistency, the system orchestrates two complementary interaction patterns: **Remote Procedure Call (RPC)** and **Publish-Subscribe (Pub/Sub)**.

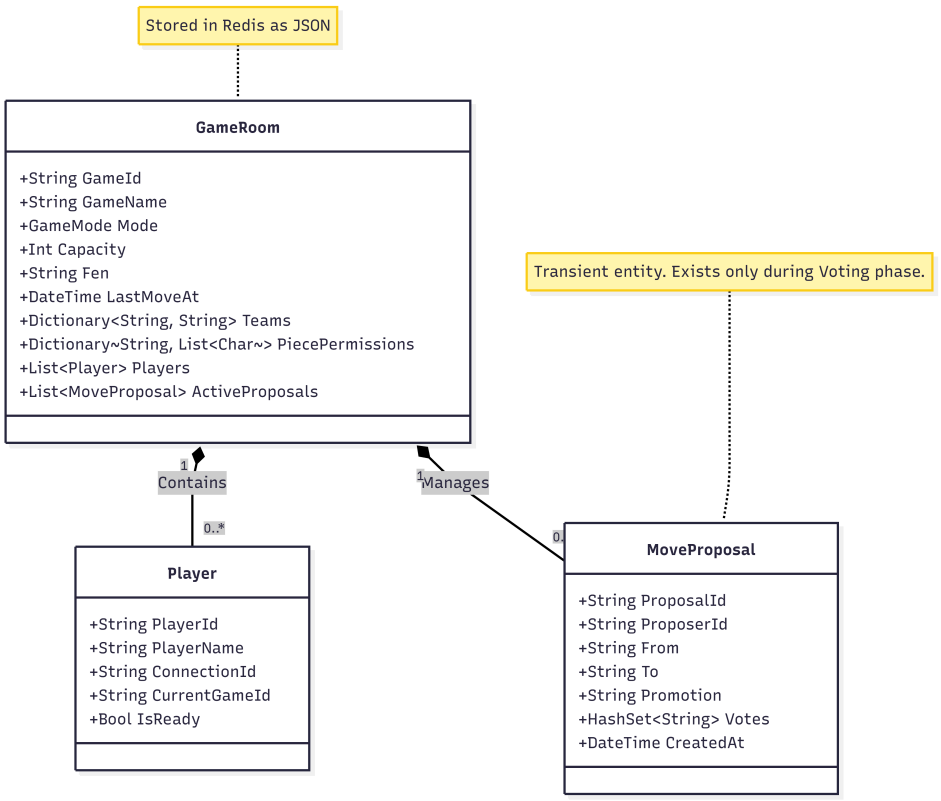

Figure 2: Class Diagram illustrating the relationship between the GameRoom, Players, and the consensus mechanism (MoveProposal).

Client-Initiated Commands (RPC) To initiate state transitions, the client utilizes a Remote Procedure Call pattern. When a user performs an active operation such as joining a lobby or proposing a move, the client application directly invokes a specific method exposed by the server-side Hub (e.g., `ProposeMove`). Unlike a simple message push, this invocation follows a command semantics: the server processes the logic and returns a promise to the client, which resolves upon success or rejects with an error. This ensures that the client receives immediate feedback regarding the validity of their intent.

State Propagation (Pub/Sub) Once a command is validated and processed, the system relies on the Publish-Subscribe pattern to propagate the changes. This mechanism operates at two distinct layers of the infrastructure to guarantee scalability and responsiveness:

First, an **Internal Backplane** is established via Redis. When a backend node publishes an event, it is not retained locally but is propagated through Redis to all other active backend nodes subscribed to the specific channel. This internal synchronization is crucial for the distributed architecture, as it ensures that a client connected to 'Node A' can seamlessly interact with a teammate connected to 'Node B'.

Simultaneously, the system employs an **External Broadcast** mechanism to push updates to the clients. The backend publishes events such as `MoveMade` to specific SignalR Groups acting as topics. All clients subscribed to these groups receive the payload in real-time, allowing the user interface to reactively update its state without the need for inefficient polling strategies.

3.4.2 The Consensus Workflow

The most significant interaction sequence is the **Move Consensus Protocol**. This flow demonstrates how the system coordinates state transitions in a distributed manner without a single node having absolute authority. The interaction phases are:

1. **Proposal:** A client initiates a state transition request. The server validates only the user's permissions (Sharding) but not the move logic.
2. **Propagation:** The server persists the proposal to Redis (Shared State) and broadcasts it to the team.
3. **Distributed Validation:** Peer nodes receive the proposal. They execute a local check using the client-side engine (`chess.js`).
4. **Voting:** If valid, peers send a vote command back to the server.
5. **Commit:** Upon reaching a majority quorum, the server does a final validation, commits the change to Redis and notifies all clients of the new global state.

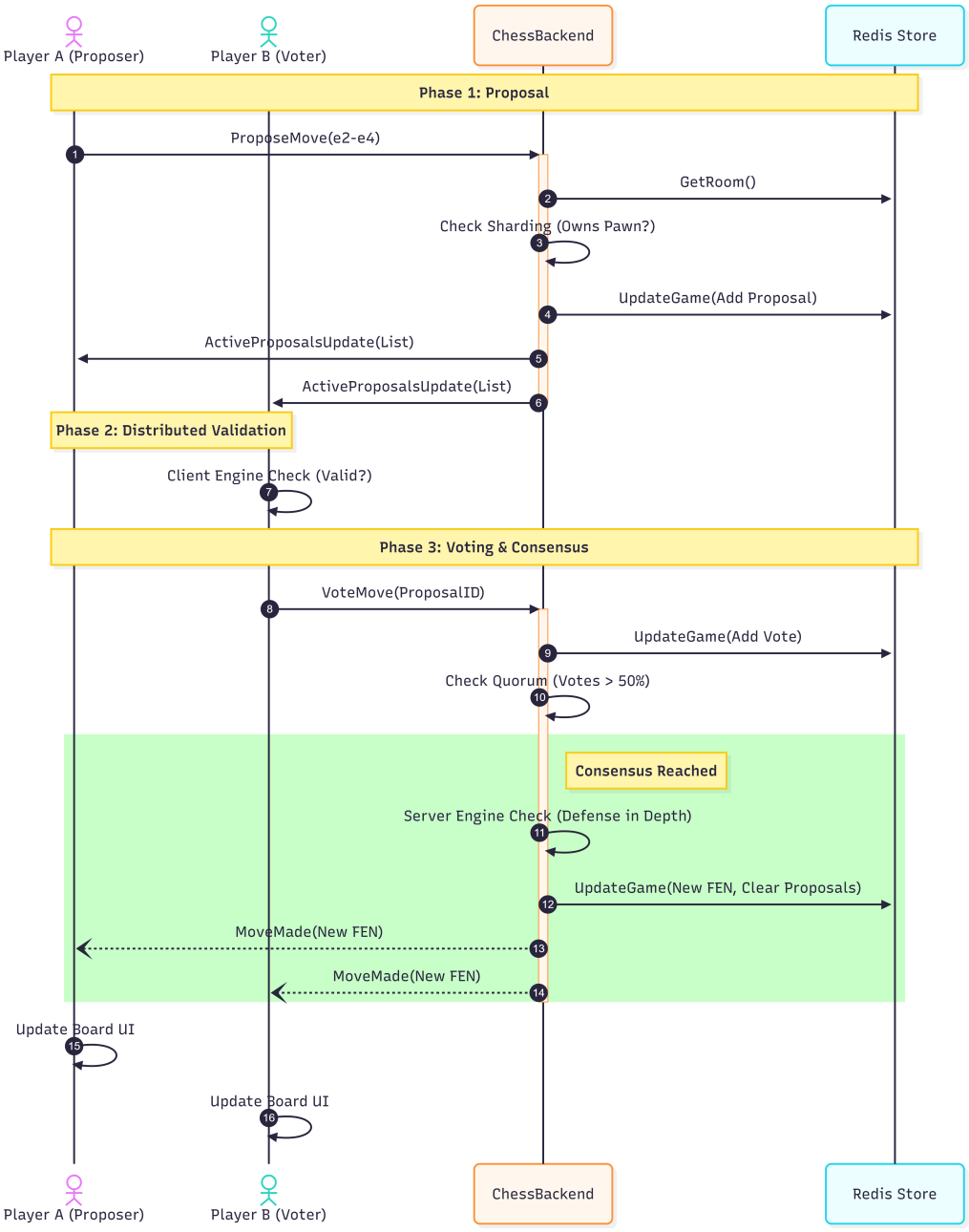

Figure 3: Sequence Diagram of the Consensus Protocol. It depicts the flow from a move proposal by Player A, through the validation and voting by Player B, to the final commitment by the Backend.

3.5 Behaviour

The system behaviour is modeled as an Event-Driven State Machine. A key architectural distinction is made between the processing nodes (Backend) and the data storage (Redis) to ensure scalability and resilience.

3.5.1 Component Behaviour

- **Client Nodes (Stateful UI):** Clients behave as stateful interactive agents. They maintain a local, non-authoritative replica of the game state (the DOM and `chess.js` instance).
  - *Reaction to User Input:* They optimistically validate inputs (e.g., checking piece ownership via Sharding) before sending Commands.
  - *Reaction to Events:* Upon receiving server events, they trigger internal logic (P2P validation) to update the UI or enable voting buttons.
- **Backend Broker (Stateless Coordinator):** The application servers are strictly stateless. They do not retain the game context in local memory between HTTP/Socket requests.
  - *Reaction to Messages:* For every incoming message (Command), the backend fetches the current state from Redis, applies the business logic, persists the new state back to Redis, and broadcasts the resulting Events.
  - *Concurrency:* Multiple backend instances can behave as a single logical entity thanks to this stateless design backed by the Redis shared store.
- **Timeout Service (Active Monitor):** A distinct background worker acts as a system 'watchdog'. It periodically scans the active games in Redis to detect stalled states (e.g., expired turn timers) and triggers forced state transitions to ensure system liveness.

3.5.2 State Management & Transitions

The `GameRoom` entity updates are strictly regulated by a state machine that alternates between an 'Open' phase and a 'Consensus' phase. The components in charge of updating the state are the **Backend** (upon reaching quorum) and the **Timeout Service** (upon expiration).

The lifecycle of a game turn is defined as follows:

1. **Idle State:** The board is stable. No proposals are pending. Players are free to explore or initiate a move.
2. **Voting State:** Triggered by a `ProposeMove` command. In this state, the `ActiveProposals` list is populated. The system waits for incoming `VoteMove` messages.

3. **Resolution State (Transition):** Triggered by reaching a consensus quorum ($Votes > N/2$) or by the Timeout Service. The system commits the move to the FEN string, clears the proposals, and reverts to Idle.

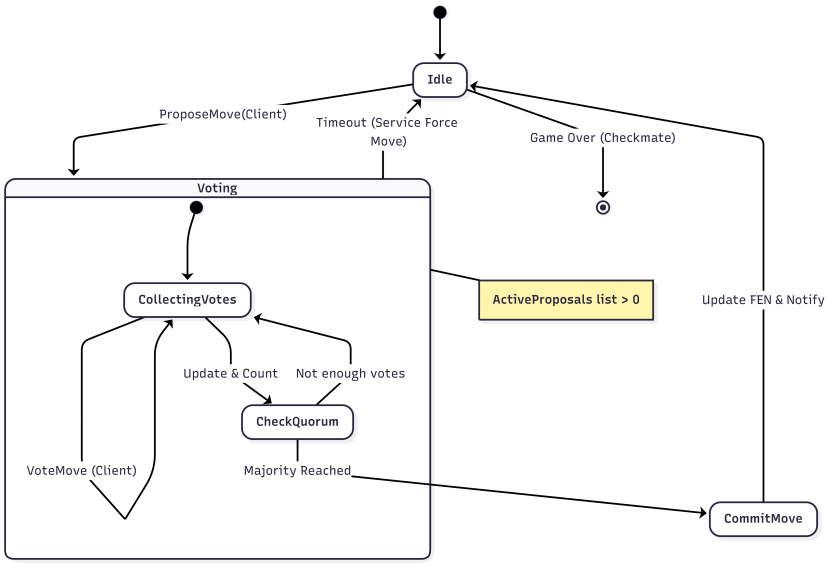

Figure 4: State Diagram of the Game Logic. It illustrates the transition from the Idle state to the Voting phase, and the conditions (Quorum or Timeout) that lead to a State Commit or Rejection.

3.6 Data and Consistency Issues

The system handles stateful data within a stateless distributed architecture. This necessitates a strict separation between processing (Backend) and storage (Redis).

3.6.1 Data Storage Strategy

- **Data to be stored:** The system persists ephemeral session data: the `GameRoom` state (board FEN, teams, proposals) and `Player` metadata (connection mapping).
- **Storage Location:** All state is externalized to **Redis**, a distributed in-memory data store.
- **Motivation:** Since the backend nodes are stateless containers managed by Docker, storing data in the application memory (RAM) is not viable; data would be lost upon container restart or inaccessible to other nodes in the cluster. Redis provides the necessary low-latency shared access required for real-time gaming.

3.6.2 Data Model (Key-Value)

Persistent data is stored using a **Key-Value** paradigm.

- **Format:** Domain objects (`GameRoom`, `Player`) are serialized into JSON strings.
- **Access Pattern:** Data is accessed primarily by primary key (e.g., `game:{guid}` or `player:{id}`).
- **Justification:** A relational model (SQL) was deemed unnecessary because the data schema is hierarchical and self-contained within the Aggregate Root (`GameRoom`). The game does not require complex join queries or long-term historical analysis, making the $O(1)$ access time of a Key-Value store the optimal choice for performance.

3.6.3 Query Patterns & Concurrency

Queries are performed exclusively by the **Backend Components** (Hubs and Services) during the execution of a Command. The typical interaction pattern follows a *Read-Modify-Write* cycle:

1. **Read:** When a command arrives (e.g., `VoteMove`), the backend retrieves the current JSON state from Redis and deserializes it.
2. **Modify:** The business logic is applied in memory (e.g., adding a vote to the HashSet).
3. **Write:** The updated object is serialized and saved back to Redis, overwriting the previous key.

Concurrent Access Handling: Since multiple players can interact simultaneously (e.g., concurrent voting), race conditions are a potential issue. The system mitigates this by using the Backend as a serializer. While Redis supports atomic operations, the current implementation relies on the high throughput of Redis and the logic idempotency to handle near-simultaneous writes, accepting a 'Last-Write-Wins' policy for non-critical metadata, while critical state changes (Commit Move) are protected by the consensus logic itself.

3.6.4 Shared Data

Data sharing is the core requirement that enables the distributed nature of the system:

- **Backend-to-Backend:** Backend nodes share the `GameRoom` state via Redis. This allows a player connected to Node A to play seamlessly with a player connected to Node B.
- **Backend-to-Frontend:** The server shares the 'Official State' with clients via SignalR broadcasts. Clients maintain a local copy for rendering but must always defer to the server's version in case of conflict (Authoritative Server pattern).

3.7 Fault-Tolerance

The system is designed to operate reliably in an environment where components (clients or servers) can fail at any time. Fault tolerance is achieved through state externalization, heartbeat monitoring, and active failover strategies.

3.7.1 Data Sharing and Replication

To ensure that the failure of a processing node (Backend) does not result in data loss, the system adopts a Shared State architecture.

- **Redis Backplane:** Data is not replicated inside the application servers' memory but is shared across a distributed bus (Redis). This enables a **Stateless Architecture**: any backend node can service any client request. If a backend container crashes, clients automatically reconnect to a healthy instance, retrieving the game state from Redis without any perceived data loss.
- **Client-Side Replication:** Clients maintain a 'Shadow Replica' of the board. While this is not authoritative, it allows the UI to remain responsive during brief network interruptions before resynchronizing with the master state.

3.7.2 Heart-beating and Timeouts

The system employs both infrastructural and application-level mechanisms to detect failures and preserve **Liveness**.

- **Infrastructural Heartbeat (SignalR):** The WebSocket connection is maintained via automatic Ping/Pong frames. If a client stops responding (e.g., network cable unplugged), the server detects the socket closure within a configured timeout window (default 30s) and triggers the `OnDisconnected` event. Conversely, clients are configured with `.withAutomaticReconnect()` to handle transient server failures.
- **Application-Level Timeout (Deadlock Prevention):** In a consensus-based system, a crash during a voting phase could lead to a deadlock (waiting indefinitely for a vote that will never arrive). To solve this, a **ProposalTimeoutService** runs in the background. It monitors the `LastMoveAt` timestamp of active games. If a turn exceeds the time limit (120 seconds), the service forces a resolution (executing the most voted move or a random move), guaranteeing that the game state always progresses.

3.7.3 Error Handling and Failover

The system implements specific recovery strategies for component failures:

- **Client Node Failure (Crash/Disconnect):** This is a critical scenario in a role-sharded game. If a player controlling specific pieces (e.g., Rooks) disconnects,

those resources would become 'frozen'. *Recovery Strategy:* The system implements an automatic Failover and Rebalancing mechanism.

1. The Backend detects the disconnection via the Heartbeat.
2. The logic identifies the 'orphaned' shards (permissions).
3. The permissions are immediately transferred to the surviving teammate(s).
4. A state update is broadcast, allowing the teammate to control the disconnected player's pieces until they return.

- **Backend Node Failure:** Since the logic is stateless, a backend crash is handled transparently by the client's reconnection logic. The client will re-establish a connection to an available node and request the latest game state (`RequestGameState`), ensuring eventual consistency.

3.8 Availability

Availability is addressed through redundancy and high-performance in-memory data access, while explicitly accepting trade-offs in favor of Consistency in the event of severe network partitions.

3.8.1 Caching Mechanisms

To minimize latency and reduce network overhead, the system implements caching at two levels:

- **Client-Side Caching (Shadow Replica):** The Angular client maintains a local copy of the full game state ('GameRoom' object).
  - *Why:* This allows the UI to render the board and validate potential moves (P2P pre-validation) without querying the server for every interaction (e.g., hovering over a piece).
  - *How:* The cache is invalidated and updated via server-push events.
- **In-Memory Data Store (Redis):** While technically the primary store, Redis operates entirely in RAM.
  - *Why:* It acts as a high-speed shared layer that eliminates the disk I/O bottlenecks associated with traditional SQL databases.
  - *How:* Backend nodes treat Redis as an always availavle cache for session data. Since data is ephemeral (session-scoped), the system relies on Redis speed to guarantee high availability of read/write operations during the game.

3.8.2 Load Balancing

The architecture supports horizontal scalability through the Stateless Backend design pattern.

- **Mechanism:** The system utilizes the **Redis Backplane** to decouple clients from specific server instances.
- **How it works:** Multiple backend containers (e.g., Node A, Node B) run in parallel. A client connecting to Node A can seamlessly interact with a client connected to Node B.
- **Why:** This allows the distribution of the computational load (move validation, vote counting, WebSocket connections) across multiple CPUs or servers. In a production environment, a Layer-7 Load Balancer (e.g., NGINX) would sit in front of the backend cluster to distribute incoming WebSocket connections.

3.8.3 Behavior under Network Partitioning

In the context of the **CAP Theorem**, this system classifies as a **CP (Consistency + Partition Tolerance)** system. Integrity of the game rules takes precedence over raw availability.

- **Scenario:** A network partition occurs between the Backend and Redis, or between the Backend and the Client.
- **System Behavior (Fail-Fast):**
  - If a Backend node loses connection to Redis, it stops accepting 'ProposeMove' or 'VoteMove' commands, throwing internal server errors.
  - It does **not** switch to a local in-memory fallback.
- **Why:** Falling back to local memory would cause a **Split-Brain** scenario, where two different servers might approve conflicting moves for the same game (divergent state).
- **Recovery:** When the partition heals, the stateless nature of the backend allows it to immediately resume serving requests using the authoritative state found in Redis, ensuring eventual recovery without data corruption.

3.9 Security

Security in this system is approached primarily through strict server-side validation and granular access control, prioritizing gameplay integrity over traditional identity management.

3.9.1 Authentication

- **Type:** Anonymous Session-Based Authentication ('Soft Authentication').
- **Mechanism:** Upon the first visit, the client generates a cryptographically strong UUID ('PlayerId') and persists it in the browser's LocalStorage. This ID is sent within the signal context or message payload for every interaction.

- **Why:** Given the ephemeral and casual nature of the application (a lobby-based game), enforcing a username/password registration would introduce unnecessary friction. The UUID acts as a bearer token, sufficient to identify a returning user session (e.g., after a page refresh) without requiring personal data storage.

3.9.2 Authorization (Access Control)

Authorization is the most critical security aspect of this distributed system. Instead of simple roles (Admin/User), the system implements a dynamic, state-aware Attribute-Based Access Control (ABAC).

- **Context-Aware Permissions (Sharding):** Access rights to 'Write Operations' (Moving a piece) are not static. The backend verifies three dynamic conditions before accepting a proposal:
  1. **Turn Authorization:** Is the requesting player's team currently active?
  2. **Resource Ownership:** Does the player hold the specific permission shard for the piece type being moved (e.g., *Permission: ['R', 'N']*)?
  3. **Team Isolation:** Is the player voting on a proposal belonging to their own team? (Preventing sabotage).
- **Execution Authority:** While clients can *propose* state changes, they have **zero authority** to commit them. Only the Backend (acting as the Trusted System) has the authorization to alter the immutable state in Redis, and only after the Consensus condition is met.

3.9.3 Defense in Depth & Cryptography

- **Transport Security:** In a production environment, all WebSocket and HTTP traffic is encrypted via **TLS/SSL** (WSS protocol) to prevent Man-in-the-Middle attacks and packet sniffing (which would reveal opponent strategies).
- **Server-Side Oracle (Anti-Cheating):** The system implements a 'Defense in Depth' strategy. Even if a client is compromised (e.g., modified JavaScript to send illegal moves or bypass sharding checks), the Backend executes an independent validation using the C# Game Engine. This ensures that no invalid state can ever be persisted to Redis, regardless of the consensus reached by potentially malicious clients.

4 Implementation

This chapter details the concrete technological choices, protocols, and frameworks selected to realize the design described in the previous chapter. While the design focused on abstract patterns like Broker-Based coordination and Logical P2P, the implementation leverages the .NET 8 ecosystem and Angular to provide a robust, cross-platform solution.

The development adhered to modern practices, including containerization via Docker to ensure environment consistency and the use of strongly-typed Data Transfer Objects (DTOs) to guarantee contract alignment between the frontend and backend.

4.1 Network Protocols

The system relies on a hybrid protocol stack to ensure both accessibility (via standard web ports) and low-latency performance for the gameplay interactions.

4.1.1 WebSockets (Primary Transport)

The core communication backbone is built upon the WebSocket protocol.

- **Usage:** Used for all real-time events, including move proposals, voting signals, and board updates.
- **Why:** Unlike HTTP, WebSockets provide a persistent, full-duplex TCP connection. This eliminates the overhead of the HTTP handshake for every single interaction, drastically reducing latency. This is crucial for the 'Voting' phase, where multiple team members broadcast votes simultaneously.
- **Implementation:** The protocol is abstracted by the SignalR library, which manages the connection lifecycle, heartbeats (Ping/Pong frames to detect node failures), and automatic reconnection attempts.

4.1.2 HTTP/1.1 (Negotiation & Fallback)

Standard HTTP is used exclusively for the initial connection phase. Before establishing a WebSocket connection, the client performs an HTTP POST handshake to the backend. This allows the server to verify user credentials (if applicable) and determine the most suitable transport capabilities.

4.1.3 RESP (Internal Communication)

Within the Docker infrastructure, the Backend nodes communicate with the Persistence Layer using RESP (Redis Serialization Protocol) over raw TCP. RESP is a binary-safe, human-readable protocol designed for extreme performance. It minimizes the overhead when the Backend needs to serialize/deserialize the Game State thousands of times per minute.

4.1.4 Why TCP and not UDP?

Although many real-time games utilize UDP (User Datagram Protocol) for speed, this project strictly adheres to TCP (Transmission Control Protocol). In Distributed Chess, **Reliability** and **Ordering** are more critical than raw speed. Dropping a packet containing a 'Vote' or receiving moves out of order would corrupt the game state (Split-Brain).

TCP guarantees packet delivery and ordering, ensuring Strong Consistency at the network level.

4.2 Data Representation

To ensure maximum interoperability between the .NET ecosystem (Backend) and the TypeScript ecosystem (Frontend), JSON (JavaScript Object Notation) was selected as the universal data interchange format.

4.2.1 Serialization Format

All in-transit data messages (Commands and Events) and at-rest data (Redis values) are represented as UTF-8 encoded JSON strings.

- **Why JSON over Binary (e.g., Protobuf):** While binary formats offer smaller payload sizes, JSON provides human readability, which is essential for debugging distributed systems. Furthermore, JSON is natively supported by web browsers and Angular, eliminating the need for complex unpacking logic or external schema files on the client side.
- **Naming Conventions:** The system handles the impedance mismatch between C# naming conventions (`PascalCase`) and JavaScript conventions (`camelCase`). The `System.Text.Json` serializer is configured to automatically convert property names during serialization (e.g., `PlayerId` becomes `playerId`), ensuring that the frontend receives idiomatic JavaScript objects.

4.2.2 Data Transfer Objects (DTOs)

To enforce a strict contract between the client and the server, the system utilizes a Shared Library pattern.

- **Shared Project:** A common C# library contains the class definitions for all messages (e.g., `MoveProposal`, `VoteMessage`).
- **Frontend Interfaces:** These classes are mirrored in TypeScript interfaces on the client. This ensures that both ends of the distributed system agree on the data structure.
- **Validation:** Deserialization acts as a first layer of validation. If a client sends a malformed JSON that does not match the DTO schema, the backend rejects the request before it reaches the business logic.

4.2.3 Redis Storage

Complex domain entities, such as the `GameRoom`, are hierarchical structures containing lists and dictionaries. To store them in Redis (a Key-Value store), they are serialized into a single JSON string.

- **Pros:** Allows storing the entire game state atomically.
- **Cons:** Requires deserializing the full object to read a single property. Given the relatively small size of a chess game state (typically < 2KB), this overhead is negligible compared to the benefits of simplicity.

4.3 Data Storage and Querying

The persistence layer is implemented using Redis, a high-performance in-memory NoSQL data store. This choice aligns with the requirement for low-latency state synchronization in a real-time distributed application.

4.3.1 NoSQL Key-Value Paradigm

Unlike traditional Relational Database Management Systems (RDBMS) like SQL Server or PostgreSQL, the system does not enforce a rigid schema with tables and foreign keys. Instead, it adopts an Aggregate-Oriented approach.

- **Aggregates:** The `GameRoom` is treated as a single atomic unit (Aggregate Root). It contains all nested information (players, board state, proposals) within itself.
- **Why not SQL?** A chess match is an isolated context, a move in Game A never relates to Game B. Therefore, the complex Join capabilities of SQL are unnecessary overhead. Storing the game as a single JSON blob allows for faster retrieval without expensive table traversals.

4.3.2 Query Patterns

All queries performed by the backend are simple Primary Key Lookups, ensuring a time complexity of O(1) regardless of the system load.

4.3.3 Data Access Implementation

The implementation utilizes the `StackExchange.Redis` client library. The interaction follows a strict pattern to ensure statelessness in the application layer:

1. **Hydration:** Upon receiving a request (e.g., `VoteMove`), the backend fetches the raw JSON string from Redis (‘StringGetAsync‘) and deserializes it into C# objects.
2. **Logic Execution:** The business logic processes the objects in memory.
3. **Dehydration:** The modified state is serialized back to JSON and saved to Redis (‘StringSetAsync‘), replacing the old value.

This ‘Load-Process-Save’ cycle ensures that the backend nodes remain purely computational units, delegating all state retention to the Redis cluster.

4.4 Authentication and Authorization

The implementation prioritizes a seamless user experience ('frictionless entry') for authentication, while enforcing strict, context-aware security policies for authorization to protect the integrity of the distributed game state.

4.4.1 Authentication: Anonymous UUIDs

Instead of traditional credential-based systems (OAuth/JWT), which would create unnecessary barriers for a casual lobby-based game, the system implements an Anonymous Persistent Authentication mechanism.

- **Identity Generation:** Upon the first visit, the Angular client generates a cryptographically strong Universally Unique Identifier (UUID v4) using the browser's native `crypto` API.
- **Persistence:** This identifier is stored in the browser's `localStorage`. This ensures that if a user reloads the page (F5) or restarts the browser, they retain their identity and can reconnect to active sessions.
- **Transmission:** The UUID acts as a bearer token. It is transmitted to the backend during the SignalR connection handshake and included in the context of every Hub invocation.

4.4.2 Authorization: Attribute-Based Access Control (ABAC)

Authorization is not handled via static roles (e.g., Admin vs. User) but through a fine-grained ABAC model[10] implemented directly in the C# business logic. Access to write operations (moves) is granted dynamically based on the attributes of the subject (player) and the object (piece).

The `GameHub` enforces the following policy pipeline for every `ProposeMove` command:

1. **Team Attribute Verification:** The system retrieves the `Teams` dictionary from Redis and verifies that the requester's ID belongs to the color currently allowed to move (derived from the FEN string).
2. **Shard Ownership Verification:** If the game is in *Team Consensus* mode, the system checks the `PiecePermissions` map. It verifies that the piece being moved (e.g., 'P' for Pawn) is present in the player's allowed list.
3. **Vote Integrity:** When casting a vote via `VoteMove`, the system verifies that the voter belongs to the same team as the proposer, preventing cross-team interference.

Any violation of these rules triggers a `HubException`, immediately rejecting the request and protecting the system against compromised clients or malicious API calls.

4.5 Technological Details

The implementation leverages a modern, full-stack ecosystem designed for performance and maintainability.

4.5.1 Backend: .NET 8 ecosystem

The server-side logic is built upon ASP.NET Core (.NET 8). This framework was selected for its high-performance Kestrel web server and robust asynchronous programming model (‘async/await‘), which is essential for handling thousands of concurrent WebSocket connections.

- **SignalR Core:** The primary library for real-time communication. It abstracts the underlying WebSocket transport and provides the `Hub` pattern for RPC style calls. Crucially, it handles the **Redis Backplane** integration transparently, allowing the distribution of messages across multiple server instances.
- **Hosted Services:** The system utilizes `BackgroundService` to implement the *ProposalTimeoutService*, ensuring that the game logic (timer checks) runs independently of user requests.

4.5.2 Frontend: Angular 20

The client application is developed using Angular 20. Taking advantage of its latest reactive features to manage the complex state of a distributed UI.

- **RxJS (Reactive Extensions):** Used to model the stream of events coming from the server. Server events (e.g., `ActiveProposalsUpdate`) are exposed as `Subjects`, allowing components to subscribe and react to state changes declaratively.
- **Angular Signals:** Used for local state management within components (e.g., the board UI). Signals provide fine-grained reactivity, updating the DOM efficiently only when specific data changes, ensuring good performance even during rapid updates.

4.5.3 Domain Libraries (Dual-Engine Approach)

To implement the "Defense in Depth" strategy, the system employs two distinct chess libraries:

- **Client-Side: chess.js (TypeScript):** A lightweight library running in the browser. It handles FEN parsing, move generation for UI highlighting, and pre-validation logic (P2P filter).
- **Server-Side: ChessDotNetCore (C#):** A robust .NET library running in the backend. It acts as the authoritative oracle, validating every committed move against the official rules of chess before persistence to prevent cheating.

4.5.4 Infrastructure

- **Docker & Docker Compose:** Used to containerize the application and simulate the distributed cluster locally. The `docker-compose.yml` defines the networking and dependency order between the backend nodes and Redis.
- **Redis:** Utilized via the `redis alpine` image to ensure a lightweight footprint while providing all necessary data structures (Strings, Sets, Pub/Sub channels).

5 Validation

The validation process focused heavily on the distributed nature of the system. Unlike a standalone application, a distributed system must be validated against network partitions, node failures, and concurrency issues. Therefore, the testing strategy prioritized **Integration Testing** and **System Testing** over granular unit testing of UI components.

5.1 Automatic Testing & Deployment Validation

Automation was applied primarily to the **Environment Provisioning** and **Integration** phases using Docker Compose. This ensures that the distributed environment (multiple backend nodes + Redis) can be spun up deterministically to reproduce complex scenarios.

5.1.1 Unit Testing

Unit testing was targeted at the `GameEngine` library, as it contains the authoritative domain logic isolated from infrastructural dependencies.

- **Component:** `ChessLogic` (Server-Side).
- **Rationale:** To ensure that the "Oracle" (the server-side validation) correctly identifies illegal moves, checkmates, and stalemates before the logic is integrated into the Hubs.
- **Automation:** The underlying library `ChessDotNetCore` is thoroughly tested. The validation focused on the adapter logic: ensuring that FEN strings are parsed correctly and that the turn color ('w'/'b') is correctly inferred from the state string.

5.1.2 Integration Testing (Infrastructure)

Integration testing verified the correct interaction between the stateless Backend nodes and the Redis Backplane.

- **Goal:** Verify that messages published by Node A are correctly received by Node B via Redis Pub/Sub.

- **Automation:** The `docker-compose up --build` command acts as the automated integration test suite. It builds the containers, creates the internal network, and establishes the TCP links between services.
- **Success Condition:** The Backend logs "Connected to Redis" and the SignalR Hub accepts WebSocket connections.

5.2 System Testing (Distributed Scenarios)

To validate the critical requirements (Sharding, Consensus, Fault Tolerance), we defined and executed specific **End-to-End (E2E) Scenarios**. These tests replicate production-like conditions including latency and abrupt failures.

5.2.1 Test Scenario 1: Consensus & Sharding [FR-03, FR-05]

- **Rationale:** Verify that the distributed voting logic works and that sharding prevents unauthorized moves.
- **Procedure:**
  1. Start a 2vs2 match with 4 distinct browser sessions (Players A, B, C, D).
  2. Player A (Pawns) attempts to move a Rook (owned by B).
  3. Player A proposes a legal Pawn move.
  4. Player B (Teammate) votes to Approve.
- **Result:**
  - The Rook move is blocked locally by the UI (Sharding validation).
  - The Pawn proposal appears on B's screen. After the vote, the move is committed to all 4 screens.

5.2.2 Test Scenario 2: Node Crash & Failover [FR-07, NFR-02]

- **Rationale:** Verify the system's High Availability and the ability to rebalance resources (Orphaned Shards).
- **Procedure:**
  1. In an active 2vs2 match, Player A (controlling Pawns) closes the browser window (simulating a Node Crash / Network Failure).
  2. The Backend detects the socket disconnection via SignalR heartbeat.
- **Result:**
  - The server detects the crash.
  - The permissions for "Pawns" are automatically transferred to the teammate (Player B).

- Player B receives a state update; their UI updates to show Pawns are now controllable (opacity removed). The game continues without deadlocks.

5.2.3 Test Scenario 3: Liveness & Timeout [FR-08]

- **Rationale:** Verify that the system resolves distributed deadlocks (e.g., a team goes AFK or refuses to vote).
- **Procedure:**
  1. Player A proposes a move.
  2. The teammates do not vote.
  3. Wait for the global turn timer (120s) to expire.
- **Result:**
  - Upon expiration, the `ProposalTimeoutService` forces a resolution (executing the proposal or a random move).
  - The turn passes to the opponent. The system does not hang.

5.2.4 Test Scenario 4: State Recovery (F5 Refresh)

- **Rationale:** Verify the persistence of the session state in Redis.
- **Procedure:** A player refreshes the page in the middle of a voting phase.
- **Result:** The client reconnects, calls `RequestGameState`, and receives the exact FEN, active proposals, and the correct timer value, resuming play seamlessly.

5.2.5 Test Scenario 5: Hacker attack

- **Rationale:** Verify the robustness of server-side validation against malicious clients.
- **Procedure:** A player modifies the JavaScript to propose an illegal move (e.g., moving an opponent's piece).
- **Result:** The other clients nodes refuses to accept the invalid move. Then I canceled the frontend checks to verify the backend one. The backend rejects the proposal via the `ChessDotNetCore` validation, throwing a `HubException`, and the game state remains unchanged.

5.3 Test Environment Setup

To validate the distributed coordination and state synchronization capabilities of the system, a multi-node cluster environment was simulated on a single development machine. The test setup required the orchestration of **two distinct backend instances** and **four distinct client sessions** (to simulate a full 2vs2 match).

5.3.1 Backend Cluster Simulation

The backend nodes were deployed using a hybrid strategy to verify that separate processes could share the same state via Redis:

- **Node A (Containerized):** Deployed via Docker Compose, exposing port `5000`. This node represents a standard production-like instance running in an isolated Linux environment.
- **Node B (Local Process):** Launched directly via Visual Studio (or .NET CLI) on port `5001`. This allowed for real-time debugging (breakpoints) of the distributed logic while interacting with the containerized node.

Both nodes were configured to connect to the same **Redis** container, ensuring that the *Backplane* and *Persistence Layer* were shared.

5.3.2 Client Simulation

Since the authentication logic relies on UUIDs stored in the browser's `LocalStorage`, simulating four distinct players required session isolation:

- **Frontend Instances:** Two separate Angular development servers were launched ('ng serve') on ports `4200` and `4201`, pointing respectively to Backend Node A and Backend Node B.
- **Session Isolation:** To generate four unique `PlayerId`s, the clients were accessed using a combination of standard browser windows and **Incognito/Private** windows across different browsers (e.g., Chrome and Edge). This setup guaranteed that each of the four 'nodes' possessed a unique identity and local state replica.

5.4 Acceptance Test (Manual Validation)

While the infrastructure and core logic were validated through integration testing, the final acceptance phase relied heavily on **Manual Exploratory Testing**. This approach was strictly necessary to validate the user experience (UX) and the visual consistency of the distributed state across multiple screens.

5.4.1 Scope of Manual Testing

Manual testing sessions focused on scenarios that are difficult to model deterministically in automated suites:

- **Visual Feedback of Sharding:** We manually verified that the "Piece Permissions" logic was correctly reflected in the UI. For example, verifying that a player controlling "Pawns" effectively sees the Rooks as semi-transparent and receives the correct "Toast Notification" when attempting to drag a locked piece.

- **Synchronization Latency:** We visually inspected the eventual consistency of the system by placing two screens side-by-side (connected to different backends). We verified that a move committed on Screen A appeared on Screen B with acceptable latency (< 200ms) and without visual glitches (e.g., pieces jumping back and forth).
- **Race Conditions in UI:** We stress-tested the UI by attempting to vote on a proposal exactly as the timer was hitting zero, or by refreshing the page during the animation of a move, to ensure the client state recovered correctly without leaving "ghost pieces" on the board.

5.4.2 Why Manual Testing?

Automating End-to-End (E2E) tests for this specific project presents distinct challenges that made manual validation more efficient and effective:

1. **Multi-Client Coordination Complexity:** Automating a consensus scenario requires coordinating 4 simultaneous browser instances (WebDrivers) that must act in specific temporal sequences (e.g., "Client A proposes, wait 2s, Client B votes"). Configuring a test harness (e.g., Cypress or Selenium) to orchestrate this distributed choreography is exponentially more complex than testing a standard web app.
2. **Drag-and-Drop Interactions:** The gameplay relies heavily on drag-and-drop interactions via the `chess.js` integration. These interactions are notoriously flaky in headless browser environments and often produce false negatives in automated tests.
3. **Fault Injection:** Simulating a "hard crash" (e.g., pulling the plug or killing a process to test Failover) is immediate to perform manually but complex to script within a CI/CD pipeline. Manual testing allowed for rapid iteration on "Chaos Engineering" scenarios (randomly closing tabs) to fine-tune the reconnection logic.

6 Deployment

The deployment strategy is designed to be completely reproducible and platform-agnostic. By leveraging Containerization, the entire distributed cluster (Frontend, Backend nodes, and Redis) can be provisioned on any machine with a single command, eliminating the "it works on my machine" problem.

6.1 Prerequisites

To run the system locally, the host machine requires the following software installed:

- **Git:** To clone the repository.
- **Docker Desktop**

No other dependencies (like .NET SDK, Node.js, or Redis) are required on the host machine, as they are encapsulated within the build images.

6.2 Installation and Execution

The following steps describe how to install and run the system from scratch:

1. **Clone the Repository:**

   ```
   git clone https://github.com/your-username/distributed-team-chess.git
   cd distributed-team-chess
   ```

2. **Build and Run:** Execute the orchestration command. The `--build` flag ensures that the latest code changes are compiled into fresh images.

   ```
   docker compose up -d --build
   ```

3. **Expected Outcome:** Docker will pull the base images (Alpine Linux, ASP.NET Core, Node.js), compile the C# and TypeScript code, and start the containers.
   - **Frontend:** Accessible at `http://localhost:4200`.
   - **Backend API:** Accessible at `http://localhost:5000`.
   - **Redis:** Running in the background on port `6379`.

6.3 Configuration

Configuration is managed via **Environment Variables** injected by Docker Compose, adhering to the *12-Factor App* methodology.

- **Redis Connection:** The backend connection string is not hardcoded but injected via the variable `ConnectionStrings__Redis`.
  - *Value in Docker:* `redis:6379` (Internal DNS hostname).
  - *Value in Development:* `localhost:6379` (Local loopback).
- **Ports:** The mapping between container ports and host ports is defined in the `docker-compose.yml` file, allowing for conflict resolution if host ports are busy.

6.4 Containerization Engineering

The Docker setup was engineered to optimize image size and build times.

6.4.1 Backend Container (Multi-Stage Build)

The `ChessBackend` utilizes a **Multi-Stage Build** process to separate the build environment (heavy, contains SDK) from the runtime environment (lightweight).

1. **Build Stage:** Uses the `mcr.microsoft.com/dotnet/sdk:8.0` image. It copies the solution files, including the `Shared` library and `GameEngine`, restores dependencies via NuGet, and compiles the binaries.
2. **Runtime Stage:** Uses the `mcr.microsoft.com/dotnet/aspnet:8.0` image. This image is significantly smaller as it contains only the runtime (CLR) required to run the DLLs, without the compiler tools.

6.4.2 Frontend Container

The Angular application is containerized using a similar two-step approach:

1. **Node Stage:** Uses a Node.js image to install NPM packages and build the production bundles using the Angular CLI (`ng build --prod`).
2. **Serving Stage:** The static artifacts (HTML, JS, CSS) are copied into a lightweight **NGINX** container (`nginx:alpine`). NGINX is configured to serve the SPA and handle the routing fallback (redirecting 404s to `index.html`).

6.4.3 Orchestration (Docker Compose)

The `docker-compose.yml` file acts as the infrastructure-as-code definition. It handles:

- **Dependency Management:** The `depends_on` directive ensures that the Backend starts only after the Redis container is healthy.
- **Networking:** A default bridge network allows containers to communicate using service names (e.g., `backend` can ping `redis`) while isolating them from the external network, except for the explicitly exposed ports.

6.4.4 Dockerfile

Listing 1: Backend Dockerfile (Multi-Stage Build)

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY DistributedChess.sln ./
COPY ChessBackend/ChessBackend.csproj ChessBackend/
COPY Shared/Shared.csproj Shared/
COPY GameEngine/GameEngine.csproj GameEngine/

RUN dotnet restore DistributedChess.sln

```

```
COPY . .
RUN dotnet publish ChessBackend/ChessBackend.csproj -c Release -o
    /app/publish


FROM mcr.microsoft.com/dotnet/aspnet:8.0
WORKDIR /app
COPY --from=build /app/publish .
EXPOSE 5000
ENTRYPOINT ["dotnet", "ChessBackend.dll"]
```

Listing 2: Frontend Dockerfile

```
FROM node:20 AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build --prod

FROM nginx:alpine

COPY --from=build /app/dist/distributed-chess/browser /usr/share/
    nginx/html

COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80
```

6.4.5 File docker-compose.yml

Listing 3: Docker Compose Configuration

```
services:
  redis:
    image: redis:7-alpine
    container_name: redis
    ports:
      - "6379:6379"

  backend:
    build: ./backend
    container_name: chessbackend
    environment:
      - REDIS_CONNECTION=redis:6379
    ports:
      - "5000:8080"
    depends_on:
      - redis
```

```
17
18    frontend:
19      build: ./frontend
20      container_name: frontend
21      ports:
22        - "4200:80"
23      depends_on:
24        - backend
```

7 User Guide

This section provides a walkthrough of the application from the end-user's perspective, illustrating how to initiate a distributed session and interact with the consensus mechanism.

7.1 Access and Lobby

Instruction: Open a web browser and navigate to `http://localhost:4200`.

- The user is presented with the **Main Lobby**.
- The button on the left open the modal to join an existing Game Room showing available rooms. Rooms that have reached full capacity are automatically hidden from the list.
- The button on the right opens the modal to create a new Game Room.

7.1.1 Creating a New Game Room

Instruction: Click the "Create Game" button.

- A modal window appears allowing the user to configure the session.
- **Team Size:** Select the number of players per team. Selecting "2" creates a 2vs2 match with *Team Consensus* mode enabled.

Expected Outcome: The game is created on the backend, and the user is redirected to the Waiting Room.

7.2 Preparation Phase (Waiting Room)

Instruction: Wait for other players to join the game using the Lobby list.

- Toggle the **"Ready"** button.
- The game will not start until the room capacity is full (e.g., 4/4 players) and all players have signaled they are Ready.

Figure 5: The Lobby Interface

Expected Outcome: Once the conditions are met, the system broadcasts the `GameStart` event, assigning teams (White/Black) and roles (Shards). The browser automatically redirects to the **Game Board**.

Figure 6: The Waiting Room Interface

7.3 Gameplay: The Sharded Board

Upon entering the board, the user interface adapts to the assigned role.

Visual Feedback:

- **Team Indicator:** The top bar displays the turn timer.
- **Permissions:** A badge displays if it's your turn or the opponent's, and the specific pieces the user is allowed to move (e.g., "You can move: Pawns, King").

- **Locked Pieces:** Pieces that belong to the user's team but are assigned to a teammate (e.g., Rooks) appear **semi-transparent (opaque)**.
- **Board and proposals:** Board appear in the left side. Proposals appear on the right side, and hovering over them highlights the corresponding move on the board.

Instruction: Try to drag a piece.

- If the user tries to move a locked piece, a **Toast Notification** appears: "You cannot control this piece!".
- If the user selects a valid piece, legal moves are highlighted with green dots.

Figure 7: The Game Board view. Note the opaque pieces which indicate the Sharding logic in action: the player can only interact with their assigned subset.

7.4 The Voting Process (Consensus)

In *Team Consensus* mode, moves are not executed immediately.

7.4.1 Proposing a Move

Instruction: Complete a move action on the board (e.g., move Pawn e2 to e4).

- **Outcome:** The piece snaps back to the original position. A "Move Proposal" appears in the sidebar on the right.
- The Global Timer bar at the top continues to run.

7.4.2 Voting

Instruction: Teammates see the new proposal card in the sidebar.

- Hovering over the proposal card highlights the move (Yellow → Green) on the board for review.
- Click the **"Vote"** button to approve the move.

Expected Outcome:

- If the majority is reached, the move is committed: the piece moves on all screens, and the turn passes to the opponent.
- If the timer expires (Timeout), the system forces a resolution to prevent deadlocks.

8 Self-evaluation

As this project was developed individually, I assumed full responsibility for the entire software development lifecycle, from architectural design to deployment.

8.1 Role Description: [Matteo Raggi]

My role encompassed all layers of the distributed stack:

- **Architect:** Defined the Broker-Based Hybrid P2P architecture and selected the CP (Consistency/Partition Tolerance) consistency model to guarantee game integrity.
- **Backend Engineer:** Implemented the .NET Core services, designing the consensus protocol, the sharding logic, and the background timeout monitors.
- **Frontend Developer:** Built the Angular reactive interface, handling the complex state synchronization via RxJS and implementing the visual feedback for the distributed logic (e.g., locked pieces).
- **DevOps:** Orchestrated the local cluster using Docker Compose, ensuring networking isolation and reproducible builds via multi-stage Dockerfiles.

8.2 Product Assessment

8.2.1 Strengths

- **Robust Coordination Logic:** The system successfully implements complex distributed patterns such as *Role Sharding* and *Majority Consensus* without relying on a central authority for the gameplay logic, respecting the project's core vision.

- **Resilience & Liveness:** The implementation of automatic Failover (transferring permissions upon crash) and Timeouts ensures that the system is highly resilient to node failures, solving the "stalled game" problem typical of P2P systems.
- **Architectural Cleanliness:** The separation of concerns between the stateless Backend (Broker), the stateful Redis (Store), and the GameEngine (Logic Library) makes the codebase maintainable and testable.

8.2.2 Weaknesses

- **Single Point of Failure (Infrastructure):** While the application servers are redundant, the architecture relies on a single Redis instance. In a production environment, this should be replaced by a Redis Cluster to ensure high availability of the data layer.
- **Soft Security:** The authentication relies on client-side UUIDs ("Soft Authentication"). While sufficient for a casual game, a malicious user with access to another player's local storage could impersonate them. A robust OAuth flow was out of scope but would be necessary for a commercial release.

9 Future Works

While the current implementation successfully demonstrates the core principles of distributed coordination, several enhancements would be necessary to transition "Distributed Team Chess" from an academic prototype to a production-grade scalable platform.

9.1 Infrastructure Evolution

- **Redis High Availability:** Currently, the system relies on a single Redis instance, representing a Single Point of Failure (SPOF). Future work would involve deploying a **Redis Cluster** or using **Redis Sentinel** to provide automatic failover and data replication for the persistence layer, shifting the CAP balance slightly more towards Availability.
- **Orchestration with Kubernetes:** Moving from Docker Compose to **Kubernetes (K8s)** would allow for dynamic auto-scaling of the Backend pods based on CPU/Memory usage. Implementing an Ingress Controller would also facilitate SSL termination and more sophisticated load balancing strategies (e.g., sticky sessions vs stateless routing).

9.2 Network Optimizations

- **Binary Serialization (Protobuf):** To optimize network throughput, especially for mobile clients with limited bandwidth, JSON messages could be replaced by

Protocol Buffers (gRPC). This would reduce payload size by eliminating the verbose text-based structure of JSON.

9.3 Feature Extensions

- **Matchmaking System:** Replacing the manual Lobby with an automated Matchmaking service based on ELO ratings. This would require a new microservice dedicated to tracking player statistics and forming balanced teams asynchronously using queues (e.g., RabbitMQ).
- **Event Sourcing for Replays:** Currently, only the latest state (FEN) is persisted. Adopting an **Event Sourcing** pattern would involve storing the entire sequence of 'MoveCommitted' events. This would enable features like "Instant Replay", "Undo Move", and advanced anti-cheating auditing by re-simulating the entire match history server-side.

9.4 Security Hardening

- **OAuth2 / OpenID Connect:** Replacing the "Soft Authentication" (UUID) with a standard identity provider (like Google). This would allow players to recover their sessions across different devices and provide a secure foundation for a persistent ranking system.

References

[1] Wikipedia contributors. Chess. *Wikipedia*, 2025.

[2] Laurent L. Njilla Sachin Shetty, Charles A. Kamhoua. *Distributed Consensus*. John Wiley and Sons, 2019.

[3] M. Thilliez, T. Delot, S. Lecomte, and N. Bennani. Hybrid peer-to-peer model in proximity applications. In *17th International Conference on Advanced Information Networking and Applications, 2003. AINA 2003.*, pages 306–309, 2003.

[4] Hoang Lam Thuan L. Thai. Net framework essentials. *.NET Framework Essentials*, 2003.

[5] Josiah Carlson. Redis in action. *Redis in Action*, 2013.

[6] Seth Gilbert and Nancy Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.

[7] Google LLC. Angular - the web development framework for developers, 2025. Accessed: 2025-01-05.

[8] Docker Inc. Docker compose - define and run multi-container applications with docker, 2025. Accessed: 2025-01-05.

[9] NGINX, Inc. Nginx - high performance load balancer, web server and reverse proxy, 2025. Accessed: 2025-01-18.

[10] E. Yuan and J. Tong. Attributed based access control (abac) for web services. In *IEEE International Conference on Web Services (ICWS'05)*, page 569, 2005.