# Clash Royale

by

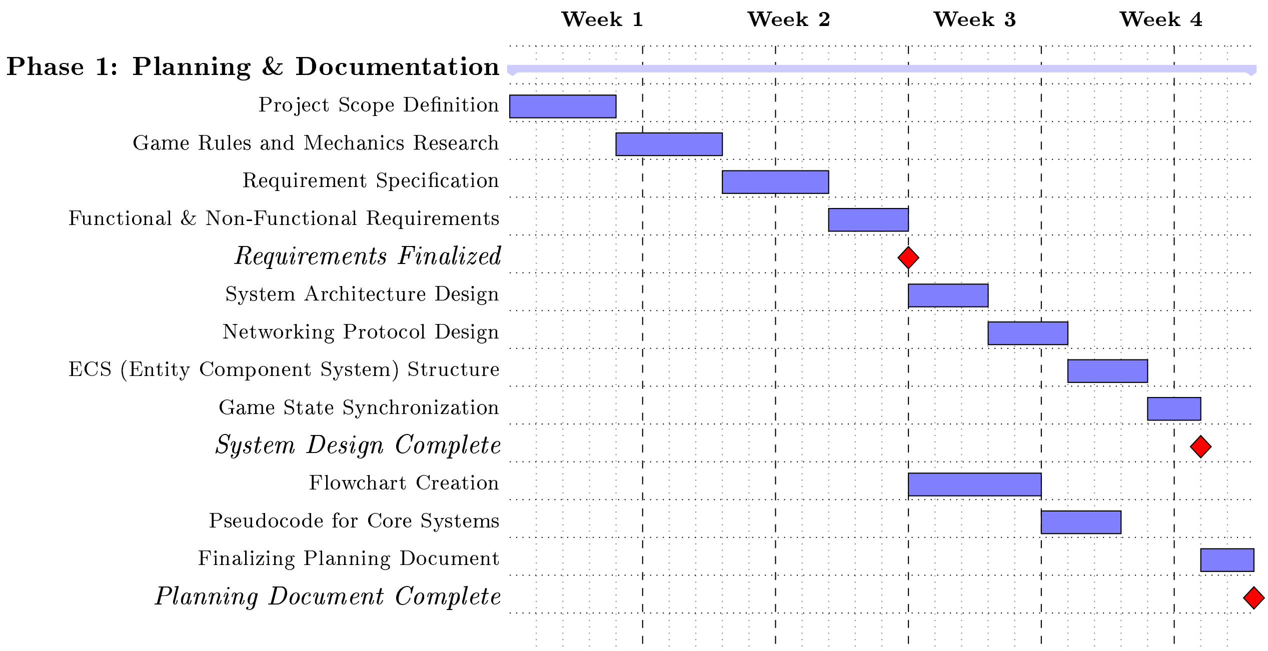
Paul Hodges

March 27, 2025

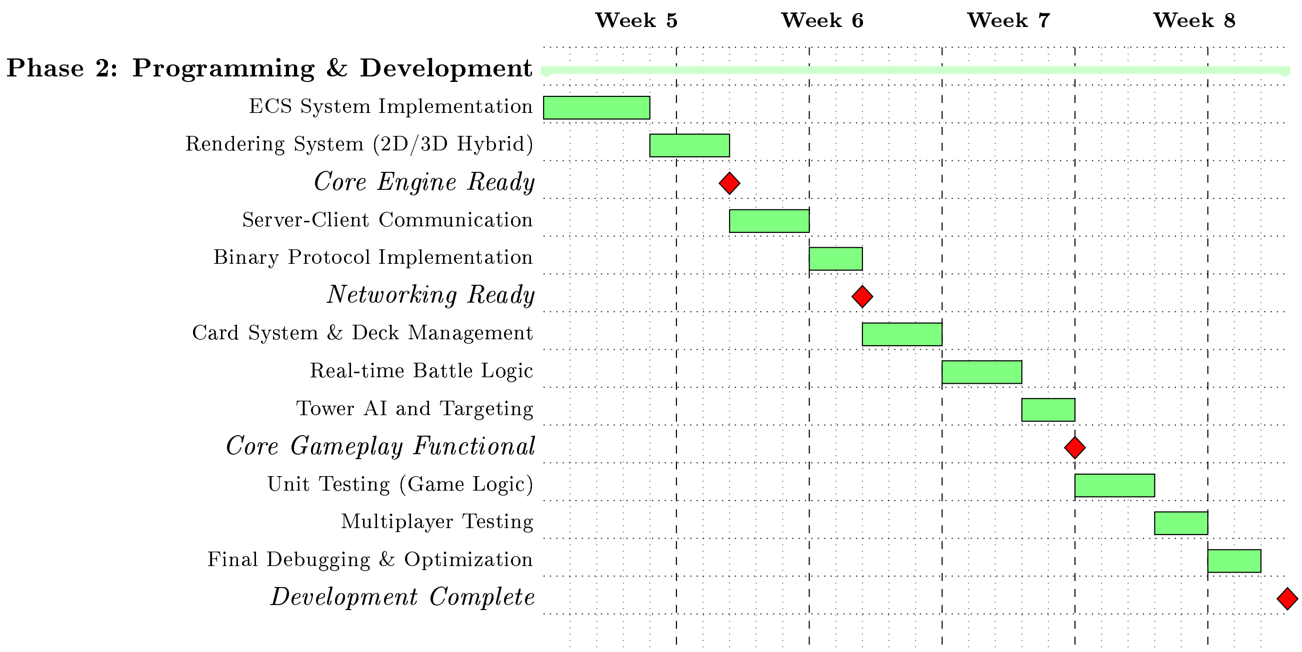
# Contents

# 1 Development Schedule

## 1.1 Planning and Documentation




## 1.2 Programming and Development




# 2 Problem Outline

The purpose of this project is to develop a simplified but functional online version of Clash Royale using Python. This real-time strategy game implementation will focus on core gameplay mechanics and network functionality, demonstrating practical software development skills in a gaming context.
The game will allow players to:

- Engage in real-time, online strategic battles against other players
- Build and customize card decks from their collection
- Progress through a skill-based arena system
- Earn rewards through a chest system to expand their card collection
- Experience the core strategic elements that make Clash Royale engaging

This implementation serves as both an educational tool for understanding game development principles and a demonstration of networking, database management, and software architecture skills in a cohesive application.

# 3 Problem Description

## 3.1 Game Overview

Clash Royale is a **fast-paced, real-time strategy** (RTS) game that incorporates elements from multiple genres:

- **Collectible Card Game (CCG):** Players build and customize decks from a pool of collectible cards. Each card corresponds to a troop, spell, or building, each with its own unique attributes and effects.
- **Tower Defense:** Players must defend their own towers (two Princess Towers and one King Tower) from enemy attacks while simultaneously attempting to destroy opponent towers.
- **Multiplayer Online Battle Arena (MOBA):** Two players face off in a real-time battle, managing resources (elixir) and positioning units tactically on a symmetrical arena.

Key aspects include:

- **Short Match Durations:** Standard matches last 3 minutes, potentially extended by sudden-death overtime.
- **Focus on Tactics and Timing:** Players must carefully manage elixir and deploy units at optimal moments.
- **Intense Head-to-Head Combat:** Each battle is a direct confrontation with another human opponent.

## 3.2 Game Objectives

- **Primary Goal:** Destroy more towers than the opponent before time runs out, or immediately win by destroying the enemy King Tower.
- **Crown System:**
  - Destroying a Princess Tower yields 1 crown.
  - Destroying the King Tower yields 3 crowns, resulting in an instant victory.
- **Overtime:** If both players have an equal number of crowns at the end of 3 minutes, the match enters a sudden-death overtime (up to 2 minutes). The first player to gain a lead in crowns during overtime wins immediately.
- **Draw:** If neither player gains an advantage by the end of overtime, the match ends in a draw.

## 3.3 Core Game Rules

1. **Deck and Card Mechanics:**
   - Each player prepares an **8-card deck** to bring into a match.
   - At the start of a match, players see **4 random cards** from their deck.
   - Once a card is played, it is immediately replaced from the remaining deck cards to maintain a 4-card hand.
   - Cards have different rarities (Common, Rare, Epic, Legendary) and can be upgraded to increase stats.
2. **Elixir Resource:**
   - Elixir is the primary resource used to deploy cards.
   - Elixir regenerates at a base rate of **1 elixir per 2.8 seconds**.
   - The maximum elixir capacity is **10 units**.
   - During the first minute of overtime, elixir generation doubles (about 1 elixir per 1.4 seconds), and it can triple in the second minute (about 1 elixir per 0.9 seconds), depending on the game mode.
3. **Deployment Rules:**
   - Troops and buildings must generally be deployed on the player's side of the arena.
   - Certain spells can be deployed anywhere (e.g., damage spells) as allowed by individual card mechanics.
   - Cards cannot be deployed if the player does not have enough elixir; partial elixir usage is not possible.
4. **Unit Behavior:**
   - Troops move automatically toward enemy units or towers following predefined paths, typically targeting the nearest threat or specific priority targets (e.g., building-focused troops).
   - Air and ground units have separate movement planes, meaning air troops may bypass ground-only defenses if not countered by anti-air.
   - Troops have attributes such as health, damage per second (DPS), movement speed, attack speed, attack range, and special abilities (e.g., area damage).
5. **Match Duration and Victory:**
   - The standard match duration is **3 minutes**.
   - If the crowns are tied after 3 minutes, **overtime** begins with sudden-death rules.
   - Destroying an opponent's King Tower results in an immediate victory regardless of time or remaining towers.

## 3.4 Gameplay Flow

1. **Matchmaking:**
   - Players are paired based on trophy count, ensuring comparable skill levels (within acceptable matchmaking ranges).
   - Additional factors (e.g., hidden matchmaking rating, player level) can be considered to improve match quality.
2. **Battle Start:**
   - Each player begins with around **5 elixir** (default) and 4 randomly drawn cards from their 8-card deck.
   - A short **countdown** (e.g., 3 seconds) precedes the start of the match, allowing players to view their starting hand.

3. **Real-Time Combat:**
   - Players deploy troops, buildings, or spells using elixir.
   - Units spawn at the chosen deployment location and automatically engage enemy targets.
   - Spell effects apply immediately if valid (e.g., damage, stun, or buff).
4. **Tower Destruction:**
   - When a tower's HP reaches zero, it is destroyed. The opposing player gains crowns accordingly.
   - Destruction of the King Tower instantly ends the match with a 3-crown victory for the attacker.
5. **Overtime and Conclusion:**
   - If crowns are tied at the end of regulation, the match enters **overtime**.
   - Elixir generation rate may **double or triple** during overtime for faster-paced final engagements.
   - If the tie persists after overtime, the match concludes in a **draw**.
   - Players are shown a **results screen** detailing crowns, trophies gained/lost, and any rewards.

## 3.5 Scoring System

- **Princess Tower Destruction:** +1 crown per destroyed Princess Tower.
- **King Tower Destruction:** +3 crowns; match ends immediately.
- **Time Expiration:** If no King Tower is destroyed, the player with the most crowns wins.
- **Draw:** If the match remains tied after overtime, both players neither gain nor lose trophies (in most modes).
- **Trophy Adjustment:** Winners gain trophies; losers lose trophies. Draws typically result in no or minimal trophy change.

## 3.6 Winner Determination

- **Instant Victory:** Destroying the opponent's King Tower at any time.
- **Crown Comparison:** If neither King Tower is destroyed, the number of Princess Towers destroyed determines the winner.
- **Overtime Mechanic:** Sudden death rules apply, where the next destroyed tower (or difference in crowns) decides the match outcome.
- **Draw:** If neither player secures a lead by the end of overtime, the match is declared a draw.

## 3.7 Advanced Game Mechanics and Strategies

- **Elixir Management:** Maintaining a balance between offense and defense, ensuring players do not overspend or waste potential elixir generation.
- **Counter Deployment:** Identifying key counters for popular units (e.g., swarm troops counter single-target tanks, ranged troops counter slow-moving units, etc.).
- **Card Synergies:** Combining cards (e.g., tank + high-DPS unit) for effective pushes. Air-ground synergy is particularly important.
- **Lane Pressure:** Forcing opponents to split their defenses by attacking both left and right lanes strategically.
- **Elixir Counting:** Skilled players track the approximate elixir of the opponent to gauge when to launch a heavy push.

- **Card Cycle Management:** Players cycle quickly to return to **core cards** (win conditions or essential counters).
- **Tower Trading:** Allowing a weaker tower to fall can free elixir for a stronger, faster offensive counter-push.
- **Spell Value:** Using damaging spells to hit multiple units at once, or a unit plus a tower, maximizes elixir efficiency.
- **Unit Placement:** Subtle differences in troop deployment positions can dramatically change the outcome of an interaction (kiting, path manipulation).
- **Timing:** Well-timed deployments catch opponents low on elixir or out of defensive options, leading to big elixir swings.

# 4 Requirements

## 4.1 Functional Requirements

### FR1. Server-Client Architecture

- **Real-Time Multiplayer Connection:**
  - The system shall use a **persistent connection protocol** (e.g., WebSockets) to ensure minimal latency and real-time data exchange.
  - Each client connection shall be **encrypted** (e.g., TLS/SSL) to protect sensitive information.
- **Centralized Game Logic:**
  - The server shall manage all core gameplay elements (e.g., elixir ticks, card interactions, damage calculations) for fairness and anti-cheat purposes.
  - Clients shall send deployment/spell actions, and the server shall validate and execute these actions within game state updates.
- **Scalability and Load Balancing:**
  - The architecture must allow horizontal scaling with load balancers to distribute matchmaking requests and battle servers as the player base grows.
  - A **master server** may handle matchmaking, while additional **battle servers** simulate individual matches.
- **Connection Handling:**
  - The system must manage client **disconnections** gracefully, allowing rejoin within a defined grace period (e.g., 30 seconds).
  - Idle sessions shall be terminated to free server resources.

### FR2. Card Management System

- **Card Collection:**
  - Each player has a **personal inventory** of collected cards, stored and displayed in a dedicated UI (e.g., “Card Collection” screen).
  - Cards are grouped by rarity (Common, Rare, Epic, Legendary) and may also be sorted by elixir cost, usage frequency, or level.
- **Deck Building:**
  - Players must be able to **create, edit, and save multiple decks** of exactly 8 cards.
  - The deck-building interface shall validate that the deck has no illegal combinations if such constraints exist (e.g., no duplicates in standard modes).
  - Players shall have the option to **rename** their decks for quick identification.
- **Card Upgrades:**
  - Each card has **level-based** attributes. Upgrades require gold and a certain number of card copies.
  - Players shall receive a notification or prompt when they have enough copies and gold to perform an upgrade.
  - Upgrading a card increases **hitpoints, damage, or other relevant stats** depending on the card type.
- **Card Unlocking:**
  - New cards become available as players reach certain **trophy thresholds** (arena levels).
  - Players can obtain these cards through **chests**, shop purchases (if implemented), or special events.

### FR3. Real-time Battle Simulation

- **Game State Updates:**

- The server shall send synchronized updates (positions, health, elixir, etc.) at a rate of at least **10 ticks per second**.
- Clients shall interpolate or render the game state for smooth animations on the front end.

- **Deployment and Combat Resolution:**
  - Each deployment request from the client must include **card ID, position, and current elixir count**.
  - The server validates and executes the deployment if the player has sufficient elixir and the placement is valid.
  - Combat is automatically resolved by the server, computing **damage** based on unit stats, interactions, and spells.
- **Collisions and Pathfinding:**
  - The system shall implement a simplified pathfinding algorithm to prevent units from occupying the same space.
  - If two units meet head-on, one or both may adjust pathing slightly to avoid overlap, or a priority system may determine which unit yields ground.
- **Overtime Mechanics:**
  - Once regulation time ends in a tie, the system triggers **sudden death** and adjusts the elixir regeneration rate.
  - If a tower is destroyed during sudden death, the match ends instantly.
  - If the tie persists after the defined overtime period, the match is declared a draw.

**FR4. Progression System**

- **Trophy-Based Ranking:**
  - Each victory adds trophies to the player's total, and each loss subtracts trophies.
  - The trophy adjustment may consider the **trophy difference** between opponents, awarding or deducting more/less accordingly.
- **Arena Unlocks:**
  - Each trophy milestone corresponds to a new **arena**, with unique visuals and an expanding pool of unlockable cards.
  - The UI shall **highlight** newly unlocked features or cards when a player enters a new arena.
- **Matchmaking and Leagues:**
  - At higher trophy counts, players may enter **league divisions** (e.g., Challenger, Master, Champion), each conferring distinct rewards.
  - Seasonal resets may occur in leagues, reducing trophies to a certain baseline and distributing **end-of-season rewards**.

**FR5. Reward System**

- **Chest Types and Mechanics:**
  - Victory chests are acquired after each win, each having a **timer** to unlock (ranging from a few hours to a full day).
  - Chest rewards scale with **arena level**, offering more cards and gold at higher tiers.
  - Special chest types (e.g., Giant, Magical, Legendary) have different **probability distributions** for rare cards.
- **Unlocking Process:**
  - A player may only have a limited number of chest slots (e.g., 4). Additional wins while slots are full do not grant new chests.
  - Players can spend **premium currency** (gems) to speed up or instantly finish chest unlocks.
  - Upon unlock, players receive the content immediately, with any new cards indicated visually in the UI.

- **Additional Rewards:**
  - Daily login bonuses, quests, or **event chests** may provide extra gold, cards, or currency.
  - The system can incorporate **battle pass** mechanics with tiered rewards for active players.

**FR6. Clan System**

- **Clan Creation and Membership:**
  - Any player meeting a minimum requirement (e.g., certain trophy threshold) can **create a clan**.
  - Clans can have a maximum capacity (e.g., 50 members), with join requests requiring **leader or co-leader approval**.
- **Clan Roles and Permissions:**
  - **Leader** can promote/demote members, accept/kick members, and modify clan settings.
  - **Co-leader** has similar privileges but may not be able to transfer leadership.
  - **Elder** typically can accept or reject join requests and assist in management.
  - **Members** can donate/request cards and participate in clan events.
- **Clan Chat and Donations:**
  - A **real-time chat** allows members to communicate, share strategies, and coordinate.
  - Players can **request** specific card donations, and other clan members can donate from their own card pool, earning gold or experience in return.
- **Clan Wars or Clan Events (Optional Advanced Feature):**
  - Clans can compete in organized **clan wars** or events for collective rewards (cards, gold, clan trophies).
  - Each participant's contributions in a clan war might be limited (e.g., a certain number of attacks per day).

**FR7. User Authentication**

- **Registration and Secure Login:**
  - Users must register with a unique username/email. A secure password mechanism with **complexity rules** (minimum length, character diversity) is required.
  - Passwords shall be stored using **salted hashing** algorithms (e.g., bcrypt, Argon2) to protect credentials in case of database breaches.
- **Session Management:**
  - After login, the server issues a **session token** (e.g., JWT) which must be presented with each subsequent request.
  - Session tokens expire after a predefined period (e.g., 24 hours or configurable), or upon explicit logout.
  - Failed login attempts shall be **rate-limited** or locked after repeated failures to mitigate brute-force attacks.

**FR8. Database Integration**

- **Data Storage:**
  - A relational or NoSQL database will store **player profiles**, trophies, card inventories, deck configurations, clan information, and transaction logs.
  - Battle replays or partial replay data (moves, timestamps) may be stored for a set period to allow replay features.
- **Atomic Transactions:**
  - High-value operations (e.g., purchases, chest openings, card upgrades) shall be **transactional** to ensure data consistency (ACID properties).
  - Rollback logic should revert partial updates if an operation fails partway.
- **Backup and Recovery:**
  - Automated **incremental and full backups** of the database must be performed regularly.

- Disaster recovery procedures shall be documented, ensuring minimal downtime and data loss in case of critical failures.

- **Scalability Considerations:**
  - The database shall support **sharding or replication** to accommodate a growing user base.
  - Indexing on frequently queried fields (e.g., user IDs, trophy counts) shall be optimized to reduce query latency.

## 4.2 Non-Functional Requirements

**NFR1. Performance**

- **Concurrent Users:** The system shall support at least **100 concurrent** players seamlessly, with a roadmap to scale to thousands if the user base grows.
- **Response Time:** Average server response time for high-frequency operations (e.g., deploying a card) shall not exceed **200 ms** under typical load.
- **Tick Rate:** The battle server shall **update** game state at least **10 times per second** (ideally 20+ for smoother experience).
- **Client Frame Rate:** The client shall maintain a minimum of **30 FPS** on recommended device specifications for fluid animations.

**NFR2. Reliability**

- **Uptime Guarantee:** The system should achieve **99%** or higher uptime annually, excluding scheduled maintenance.
- **Failover and Recovery:** In case of server crashes, automated restart processes must **resume** the service with minimal downtime and no data corruption.
- **Reconnection Logic:** If the client disconnects due to network issues, it should automatically attempt to **reconnect**, rejoining an active match if still in progress.
- **Data Integrity Checks:** The database shall regularly run integrity checks (checksums or similar) to detect and fix corruption.

**NFR3. Scalability**

- **Horizontal Scaling:** The system architecture must allow adding more servers to handle increased load without significant refactoring.
- **Matchmaking Algorithm:** The matchmaking system must handle larger player populations efficiently, maintaining reasonable queue times and matching fairness.
- **Database Sharding/Replication:** As the player base grows, the database layer should scale by distributing data across multiple nodes or using replication strategies.

**NFR4. Maintainability**

- **Coding Standards:** The codebase shall conform to PEP 8 (Python) or a recognized standard (if other languages are used) to ensure **readability and uniformity**.
- **Modular Architecture:** Features like matchmaking, battle simulation, user authentication, and clan management should be **modularized** or separated into microservices.
- **Documentation and Comments:**
  - Detailed **API documentation** (using tools like Swagger/OpenAPI) should be provided for each service endpoint.
  - Inline comments and **design notes** shall clarify complex logic for future maintainers.
- **Logging and Monitoring:**
  - A standardized **logging format** with log levels (debug, info, warning, error) must be used across services.
  - Monitoring tools (e.g., Prometheus, Grafana) shall be in place to track CPU usage, memory, network latency, etc.

**NFR5. Security**

- **Credential Storage:** All user passwords must be stored using **salted hashing** (bcrypt, Argon2, PBKDF2). Plaintext passwords are strictly forbidden.
- **Encryption:** All communication between client and server must be secured with **TLS/SSL** to protect data in transit.
- **Input Validation:** The server must rigorously **validate** client inputs (e.g., deployment commands, user IDs, trophy updates) to prevent injection attacks or exploits.

- **Attack Mitigation:**
  - Protect against **common web vulnerabilities** (SQL injection, cross-site scripting, CSRF) via standard frameworks and best practices.
  - Rate-limit login attempts and critical actions to prevent **brute force** or **DDoS** style attacks.

# 5 Additional Considerations

## 5.1 Cross-Platform Support

- The client software should be compatible with both **iOS** and **Android** platforms.
- A browser-based or desktop client can be considered for broader accessibility.
- The UI must be optimized for various **screen sizes** and different performance capabilities (lower-end vs. high-end devices).

## 5.2 Analytics and Telemetry

- **Player Behavior Data:** Collect anonymized usage statistics (most-played cards, average match duration, win/loss ratios) to inform balancing decisions.
- **System Performance Metrics:** Implement real-time monitoring for server load, memory consumption, and network bandwidth to scale effectively.
- **Event Logging:** All significant in-game actions (card deployments, spell usage, tower destruction) should be logged for debugging or replay features.

## 5.3 Live Operations and Balancing

- **Card Balance Changes:** A live operations team may frequently update card stats to maintain a fair meta, requiring **backend tools** to adjust stats without full server redeployments.
- **Seasonal Events and Modes:** Periodic special events (e.g., draft modes, 2v2 modes, tournaments) can increase engagement and test new mechanics.
- **Patch Management:** A versioning system should ensure clients are notified to update to the latest build when major changes occur.

## 5.4 User Engagement Features

- **Daily Login Rewards:** Offering small incentives (gold, gems, or cards) encourages consistent login behavior.
- **Quests and Challenges:** Short-term or medium-term goals (e.g., “Play 20 spells” or “Win 3 battles in a row”) provide variety and a sense of progression.
- **Push Notifications:** Notify players of chest unlock completions, clan invitations, or special events to encourage re-engagement.

## 5.5 Compliance and Data Protection

- **GDPR and Privacy:** If operating in regions with strict data protection laws, ensure user data can be **exported** or **deleted** upon request.
- **Age Restrictions:** Comply with **COPPA** or similar regulations for younger players, requiring parental consent or restricted in-app purchases.
- **Policy Documents:** Provide a clear **terms of service** and **privacy policy** within the application, detailing data usage and rights.

# 6 Conclusion

This document outlines a comprehensive set of functional and non-functional requirements for developing a **Clash Royale-style real-time strategy game**. The specification includes detailed mechanics for card management, real-time battle simulation, progression systems, social (clan) features, and robust security considerations.

By adhering to these requirements, the resulting product should provide:

- A stable and **engaging** gameplay experience with minimal latency.
- **Scalable** infrastructure to accommodate a growing player base.
- Strong **security** and **reliability** measures for player data and transactions.
- A balanced and **replayable** multiplayer environment that encourages long-term player engagement.

# 7 Design

## 7.1 Core System Design

The game will be built using a custom Entity Component System (ECS) engine and a custom network implementation. The visual style will be 2D with a simulated 3D look achieved through layered sprites and shadow rendering.

### 7.1.1 Engine

The engine is written in Python, based on the design off of one of my previous Rust engines, "Oxidized." Key features include:

#### 7.1.1.1 ECS Architecture

The Entity Component System (ECS) architecture separates logic (components) from entities (system). Entities are simple IDs that contain components, and components execute logic when told to through the pipeline (see 7.1.1.6). Components operate through entities that have the component attached. Components a separate, even from other components of the same type. This design promotes data-oriented programming, improving cache efficiency and making the code more modular and maintainable. In this Python implementation, I use dictionaries and lists to store and manage entities and their related components. This decoupling of different logic systems and containers allows for flexible entity composition and avoids the complexities of traditional inheritance-based object oriented programming.

#### 7.1.1.2 2D/3D Cameras

The engine will support both orthographic and perspective cameras. Orthographic cameras provide a parallel projection, useful for classic 2D views where depth isn't emphasized. Perspective cameras simulate real-world vision, where objects appear smaller as they recede into the distance. This is crucial for creating the simulated 3D look. The camera implementation will likely involve a transformation matrix that converts world coordinates to screen coordinates. For perspective cameras, this matrix will include a perspective projection component, while orthographic cameras will use a simple scaling and translation matrix. I have functions to easily switch between camera types, and to adjust the field of view of the perspective camera.

#### 7.1.1.3 GameObjects and Components

GameObjects serve as containers for components. By assigning different combinations of components to a GameObject, I can create a wide variety of entities with unique behaviors. For example, a "Player" GameObject might have "Position," "Velocity," "Sprite," "Input," and "Health" components. A "Tree" GameObject might have "Position" and "Sprite" components. This modularity allows for easy creation and modification of game entities without modifying core engine code. The component data will be stored in data structures that are easily accessible by the systems.

#### 7.1.1.4 Layered Sprite Rendering

To create the illusion of depth in a 2D environment, I will use layered sprites. Sprites representing objects closer to the "camera" will be rendered on top of sprites representing objects further away. I will implement a sorting mechanism based on the Y-coordinate (or a custom depth value) of the entities to determine the rendering order. This will allow for overlapping sprites to create a sense of depth. Furthermore, I will have the capability to assign a layer number to each sprite, allowing for more manual control over the draw order, and enabling the creation of backgrounds, midgrounds, and foregrounds.

#### 7.1.1.5 Rendering

The rendering for the python engine are not done using conventional methods like `pygame`. I use a custom library that I created myself which uses SDL2 to draw to the screen and provides more complex methods for other things like textures, images and keyboard input. I create the python bindings (for the cpython module) using `pybind11`. The library is compiled, the binds are compiled and then the library is linked with the binds to produce a cpython library. The stub file for the LSP is then generated using stubgen, which is shipped with python.

#### 7.1.1.6 Shared State

Throughout the codebase, the way that state is shared between threads and different classes is through a custom made "pipeline". The pipeline follows the same structure as a data bus, in which data can be sent down the pipe by anyone through just having a reference to the class and "consumed" by any listener appended to it. The state is managed by two pipelines, one which communicates state data, and one that functions as an event loop, the event loop is used for many other things but in the case of program state management it is used to communcate a request for a state update and then the other pipeline will contain a message of the new state, with a correlating id. The messages are transmitted as "frames" in which a single frame contains:

```python
@total_ordering
@dataclass(order=False)
class Frame(Generic[T]):
    data: T = field(compare=False)

    available_at: float = field(default_factory=time.time)
    priority: int = 0
    id: int = 0

    metadata: Optional[Dict[str, Any]] = field(default=None, compare=False)
    annotations: Dict[str, Any] = field(default_factory=dict, compare=False)
    timestamp: float = field(default_factory=time.time, compare=False)
    sender_id: Optional[str] = field(default=None, compare=False)
    expire_at: Optional[float] = field(default=None, compare=False)
    retry_count: int = field(default=0, compare=False)
    delivered_to: List[str] = field(default_factory=list, compare=False)
    correlation_id: Optional[str] = field(default=None, compare=False)
    topic: Optional[str] = field(default=None, compare=False)
    is_response: bool = field(default=False, compare=False)
```

A frame contains data of type `T`, which is a generic saying that the frame can only contain data that is a specific type. This is used because one pipeline can only transmit frames of one type to avoid typemismatch and runtime errors. Python doesn't natively support typesafety as it is not compiled and is interperated on the fly. There is a large amount of typehints within the code so I can use external tools to check for mismatches and prevent a large amount of common runtime errors. There is other data contained in a frame that is managed by the pipeline, like delivered to. Frames check for acknowlegement by other attached listeners to make sure that there is not a mismatch between threads.

### 7.1.2 Network

The network uses custom socket communication with binary data transmission:

#### 7.1.2.1 Custom Binary Protocol

Instead of relying on human-readable text-based protocols, I use a custom binary protocol. This protocol will define the structure of packets sent between the client and server, specifying the types and sizes of data fields. Binary protocols are more compact and efficient, reducing network overhead and latency. I will use python's struct module to pack and unpack data into binary format. I will define packet types, and each packet type will have a defined data structure.

#### 7.1.2.2 Client-Server Architecture

I will implement a client-server architecture, where the server acts as the authoritative source of game state. Clients send input to the server, and the server processes the input, updates the game state, and sends updates back to the clients. This architecture helps prevent cheating and ensures consistency across all clients. The server will maintain the master copy of the game world, and the clients will maintain local copies that are kept in sync with the server.

#### 7.1.2.3 Asynchronous Sockets

To avoid blocking the main game loop, I will use asynchronous sockets. Asynchronous sockets allow the game to continue running while waiting for network data. I will use python's asyncio library or similar, to handle the asynchronous network operations. This will prevent the game from freezing or becoming unresponsive during network communication.

#### 7.1.2.4 Packet Listener and Broadcaster

The server will have a packet listener to receive incoming packets from clients and a packet broadcaster to send packets to all connected clients. The listener will parse incoming packets and dispatch them to the appropriate game logic. The broadcaster will efficiently distribute game state updates to all clients, minimizing network traffic. I will implement packet queuing and throttling to prevent network congestion. The broadcaster will only send data that has changed, to reduce the amount of data sent.

### 7.1.3 2D with 3D Illusion

The 3D look is achieved through:

#### 7.1.3.1 Layered Sprites

As mentioned earlier, layered sprites are crucial for creating the illusion of depth. By rendering sprites in a specific order based on their perceived depth, I can simulate the effect of objects being in front of or behind each other. This will involve sorting sprites by their Y-coordinate or a custom depth value before rendering. I will use a depth buffer, or an equivalent sort algorithm, to ensure that sprites are drawn in the correct order.

#### 7.1.3.2 Shadow Rendering

Adding shadows to objects can significantly enhance depth perception. I will implement shadow rendering by projecting a simplified silhouette of each object onto the ground. The darkness and length of the shadow will vary based on the object's height and the light source's position. This will give objects a sense of grounding and add visual depth to the scene. I will use sprite masks to create the shadow silhouettes.

#### 7.1.3.3 Perspective Scaling

To further enhance the 3D illusion, I will implement perspective scaling. Objects farther away from the "camera" will appear smaller, while objects closer will appear larger. This will be achieved by adjusting the sprite's scale based on its distance from the camera. I will use a perspective projection matrix to calculate the scale factor. This will be done in conjunction with the Layered Sprite rendering, to create a believable 3D effect.

## 8 Pseudocode Implementation

```
####################################################
# 1. Main Server Loop
####################################################

FUNCTION RunServer():
    # 1. Initialization
    server_socket = INITIALIZE_SERVER_SOCKET(HOST, PORT)
    player_registry = {}     # {player_id: Player}
    active_games = {}        # {game_id: Game}
    matchmaking_queue = CREATE_PRIORITY_QUEUE()
    event_dispatcher = CREATE_EVENT_DISPATCHER()
    worker_threads = CREATE_THREAD_POOL(SIZE=MAX_WORKERS)

    # Start a thread responsible for updating all active games
    game_update_thread = START_THREAD(
        UpdateAllGames,
        args=(active_games, event_dispatcher)
    )

    # 2. Main Loop
    WHILE server_running:
        # 2.1 Poll for new connections
        new_connections = server_socket.POLL_CONNECTIONS()
        FOR connection_request IN new_connections:
            connection, address = server_socket.ACCEPT(connection_request)
            player = REGISTER_PLAYER(connection, address)
            player_registry[player.id] = player

            # Assign a worker thread to handle I/O for this player
            ASSIGN_THREAD(HANDLE_PLAYER, args=(player,))

        # 2.2 Process queued requests from each player
        FOR player_id, player IN player_registry.items():
            WHILE player.has_pending_requests():
                request = player.GET_REQUEST()
                HANDLE_PLAYER_REQUEST(
                    player,
                    request,
                    matchmaking_queue,
                    active_games,
                    event_dispatcher
                )

        # 2.3 Handle Matchmaking
        PROCESS_MATCHMAKING(
            matchmaking_queue,
            active_games,
            event_dispatcher
        )

        # 2.4 Execute any cross-system events
        event_dispatcher.EXECUTE_PENDING_EVENTS()

    # 3. Cleanup after server_running is false
    CLEANUP_SERVER_RESOURCES(server_socket, player_registry, active_games)
END FUNCTION


####################################################
# 2. Player Handling and Requests
####################################################
```

```

FUNCTION HANDLE_PLAYER(player):
    """
    Runs in a worker thread, dedicated to a single player's I/O.
    Continues until the player is disconnected or flagged inactive.
    """
    WHILE player.is_active:
        # Potentially blocks until data arrives
        request = player.WAIT_FOR_REQUEST()

        IF request IS NOT NULL:
            # Instead of directly handling logic here, we queue it
            # or handle it in the main loop for concurrency control
            player.ADD_REQUEST(request)

    # Cleanup resources for this player after the loop ends
    CLEANUP_PLAYER(player)
END FUNCTION


###################################################
# 3. Request Routing
###################################################

FUNCTION HANDLE_PLAYER_REQUEST(
    player,
    request,
    matchmaking_queue,
    active_games,
    event_dispatcher
):
    SWITCH request.type:
        CASE "matchmaking":
            rank_value = request.data.get("rank", 0)
            matchmaking_queue.ADD(player, priority=rank_value)
            SEND_INFO(player, "You have been queued for matchmaking.")

        CASE "game_action":
            # Identify which game the player is in
            game = GET_ACTIVE_GAME_FOR_PLAYER(player, active_games)
            IF game IS NOT NULL:
                # The request.data might contain something like
                # {"action": "deploy", "card_id": "knight", "position": (x,y)}
                action_type = request.data.get("action")
                IF action_type == "deploy":
                    card_id = request.data.get("card_id")
                    position = request.data.get("position")
                    IF card_id IS NOT NULL AND position IS NOT NULL:
                        ProcessCardDeployment(game, player, card_id, position)
                    ELSE:
                        SEND_ERROR(player, "Invalid deployment data.")
                ELSE:
                    # Additional game actions can be handled here
                    game.EXECUTE_ACTION(player, request.data)
            ELSE:
                SEND_ERROR(player, "You have no active game.")

        CASE "disconnect":
            LOGOUT_PLAYER(player)
            SEND_INFO(player, "You have disconnected.")

        CASE "custom_event":
            event_data = request.data.get("event_data", {})
```

```
            event_dispatcher.CREATE_EVENT(
                "CustomPlayerEvent",
                data=event_data
            )

        DEFAULT:
            SEND_ERROR(player, "Unknown request type: " + request.type)
END FUNCTION


###################################################
# 4. Matchmaking Process
###################################################

FUNCTION PROCESS_MATCHMAKING(
    matchmaking_queue,
    active_games,
    event_dispatcher
):
    # Keep pairing until we cannot form a valid match
    WHILE matchmaking_queue.HAS_AVAILABLE_PAIRS():
        player1, player2 = matchmaking_queue.GET_MATCHED_PLAYERS()

        IF player1 IS NOT NULL AND player2 IS NOT NULL:
            new_game = INITIALIZE_NEW_GAME(
                player1,
                player2,
                event_dispatcher
            )
            active_games[new_game.id] = new_game

            NOTIFY_PLAYERS_OF_MATCH(player1, player2, new_game)
        ELSE:
            # If we can't form a pair, break out
            BREAK
END FUNCTION


###################################################
# 5. Game Update Thread
###################################################

FUNCTION UpdateAllGames(active_games, event_dispatcher):
    TICK_RATE = 30      # frames per second
    FRAME_DURATION = 1.0 / TICK_RATE

    WHILE server_running:
        current_games = list(active_games.values())

        FOR game IN current_games:
            # Make sure the game hasn't ended
            IF NOT game.is_terminated:
                game.UPDATE(FRAME_DURATION)
                # Optionally dispatch an event after each update
                event_dispatcher.SCHEDULE_EVENT(game, "update_completed")
            ELSE:
                # If the game is terminated, optionally remove from active_games
                active_games.pop(game.id, None)

        # Wait for next frame
        SLEEP(FRAME_DURATION)
END FUNCTION

```

```

##################################################
# 6. Game Update Logic
##################################################

FUNCTION UPDATE_GAME(game, delta_time):
    # Advance the simulation
    game.INCREMENT_TIME(delta_time)

    # Check if the game is over
    IF CHECK_GAME_END_CONDITIONS(game):
        game.TERMINATE()
        RETURN

    # Replenish resources like elixir
    game.REPLENISH_ELIXIR(delta_time)

    # Update all entities in the game
    UPDATE_ENTITIES(game)

    # Broadcast the latest game state to players
    game.BROADCAST_GAME_STATE()
END FUNCTION

FUNCTION CHECK_GAME_END_CONDITIONS(game):
    # Check whether towers are destroyed or time has elapsed
    # Simple example: If any player's Crown Tower HP <= 0, game ends
    IF game.time_elapsed >= game.max_duration:
        DETERMINE_GAME_WINNER_BY_HP(game)
        RETURN TRUE

    FOR tower IN game.towers:
        IF tower.hp <= 0:
            # We have a winner
            ASSIGN_WINNER(game, tower.opposing_player_id)
            RETURN TRUE

    RETURN FALSE

FUNCTION ASSIGN_WINNER(game, winning_player_id):
    game.winner_id = winning_player_id
    game.is_terminated = TRUE
    # Additional logic: update stats, ranks, etc.

FUNCTION DETERMINE_GAME_WINNER_BY_HP(game):
    # Compares tower HP sums for all players to determine a winner
    best_score = -1
    winning_player = None
    FOR eachPlayerId IN game.players:
        total_hp = SUM_OF_ALL_TOWERS_HP(game, eachPlayerId)
        IF total_hp > best_score:
            best_score = total_hp
            winning_player = eachPlayerId
    ASSIGN_WINNER(game, winning_player)

FUNCTION REPLENISH_ELIXIR(game, delta_time):
    # For each player in the game
    FOR p IN game.players:
        current_elixir = p.elixir
        regen_rate = game.get_elixir_regen_rate()
        p.elixir = MIN(
            10,
            current_elixir + regen_rate * delta_time
```

```
        )

FUNCTION UPDATE_ENTITIES(game):
    entities_copy = game.entities[:]
    FOR entity IN entities_copy:
        IF entity.IS_DESTROYED():
            game.REMOVE_ENTITY(entity)
        ELSE:
            entity.PROCESS_BEHAVIOR(game)
            entity.UPDATE_POSITION(game)
END FUNCTION


###################################################
# 7. Card Deployment Process
###################################################

FUNCTION ProcessCardDeployment(game, player, card_id, position):
    IF NOT VALIDATE_DEPLOYMENT_POSITION(game, player, position):
        SEND_ERROR(player, "Invalid position for deployment.")
        RETURN

    card = player.GET_CARD_FROM_HAND(card_id)
    IF card IS NULL:
        SEND_ERROR(player, "Card not found in hand.")
        RETURN

    IF NOT VERIFY_ELIXIR_COST(player, card):
        SEND_ERROR(player, "Insufficient elixir.")
        RETURN

    # Spend elixir and remove card from hand
    player.SPEND_ELIXIR(card.elixir_cost)
    player.DISCARD_CARD(card_id)
    player.DRAW_NEXT_CARD()

    entity = CREATE_ENTITY_FROM_CARD(card, position, player.side)
    game.REGISTER_ENTITY(entity)

    BROADCAST_DEPLOYMENT_UPDATE(game, player, card, position)
END FUNCTION

FUNCTION VALIDATE_DEPLOYMENT_POSITION(game, player, position):
    # Example: position must be on the player's side of the arena
    # Simple check: x between 0 and game.arena_width,
    #               y in [0, half_arena_height] for one side
    # In practice, more advanced boundary checks are needed
    IF position.x < 0 OR position.x > game.arena_width:
        RETURN FALSE
    IF player.side == "bottom":
        IF position.y < 0 OR position.y > (game.arena_height / 2):
            RETURN FALSE
    ELSE:
        IF position.y < (game.arena_height / 2) OR position.y > game.arena_height:
            RETURN FALSE
    RETURN TRUE

FUNCTION VERIFY_ELIXIR_COST(player, card):
    RETURN (player.elixir >= card.elixir_cost)

FUNCTION CREATE_ENTITY_FROM_CARD(card, position, side):
    # Instantiate a Troop/Spell/Building entity
    # with properties from the card definition
```

```
    new_entity = Entity()
    new_entity.type = card.type       # e.g., "troop"
    new_entity.hp = card.hp
    new_entity.damage = card.damage
    new_entity.range = card.range
    new_entity.position = position
    new_entity.side = side
    return new_entity

FUNCTION BROADCAST_DEPLOYMENT_UPDATE(game, player, card, position):
    # Notifies all players about the new entity
    update_message = {
        "type": "deployment",
        "player_id": player.id,
        "card_id": card.id,
        "position": position
    }
    FOR p IN game.players:
        p.SEND_MESSAGE(update_message)


###################################################
# 8. Target Selection Algorithm
###################################################

FUNCTION DetermineBestTarget(attacker, game_state):
    potential_targets = game_state.GET_ENEMY_ENTITIES(attacker.side)
    best_target = NULL
    minimum_threat_score = INFINITY

    FOR target IN potential_targets:
        threat_score = COMPUTE_THREAT_SCORE(attacker, target)
        IF threat_score < minimum_threat_score:
            minimum_threat_score = threat_score
            best_target = target

    RETURN best_target
END FUNCTION

FUNCTION COMPUTE_THREAT_SCORE(attacker, target):
    # Weighted factors
    DISTANCE_FACTOR = 1.0
    HEALTH_FACTOR   = 0.01
    PRIORITY_FACTOR = 5.0

    distance_val = CALCULATE_DISTANCE(
        attacker.position,
        target.position
    )
    distance_weight = distance_val * DISTANCE_FACTOR

    health_weight = target.health * HEALTH_FACTOR
    priority_weight = target.PRIORITY() * PRIORITY_FACTOR

    total = distance_weight + health_weight + priority_weight
    RETURN total
END FUNCTION


###################################################
# 9. Chest Reward System
###################################################

```

```
FUNCTION GenerateRewardChest(player, chest_type):
    chest = CREATE_CHEST_INSTANCE(chest_type)
    chest.ASSIGN_UNLOCK_TIME()
    chest.ALLOCATE_GOLD_REWARDS(player)
    chest.ALLOCATE_CARD_REWARDS(player)

    player.INCREMENT_CHEST_COLLECTION(chest)
    SEND_REWARD_NOTIFICATION(player, chest)
END FUNCTION

FUNCTION CREATE_CHEST_INSTANCE(chest_type):
    chest = Chest()
    chest.type = chest_type
    # Example: define base times or vary by type
    chest.unlock_time = 0    # assigned later
    chest.rewards = []
    RETURN chest

FUNCTION chest.ASSIGN_UNLOCK_TIME():
    # Simple logic:
    SWITCH self.type:
        CASE "silver":
            self.unlock_time = 3 * 3600  # 3 hours
        CASE "gold":
            self.unlock_time = 8 * 3600  # 8 hours
        CASE "epic":
            self.unlock_time = 12 * 3600
        DEFAULT:
            self.unlock_time = 1 * 3600   # fallback

FUNCTION chest.ALLOCATE_GOLD_REWARDS(player):
    # Reward formula can be based on player's arena
    base_gold = 100
    arena_factor = player.arena
    total_gold = base_gold + (arena_factor * 10)
    # Add to chest
    self.rewards.append({
        "type": "gold",
        "amount": total_gold
    })

FUNCTION chest.ALLOCATE_CARD_REWARDS(player):
    ALLOCATE_CARD_REWARDS(self, player)

FUNCTION ALLOCATE_CARD_REWARDS(chest, player):
    card_distribution = DETERMINE_CARD_DISTRIBUTION(player.arena, chest.type)
    available_cards = FETCH_AVAILABLE_CARDS(player.arena)

    FOR rarity, count IN card_distribution.items():
        selected_cards = PICK_RANDOM_CARDS(available_cards, rarity, count)
        chest.ADD_CARDS(selected_cards)

FUNCTION DETERMINE_CARD_DISTRIBUTION(arena, chest_type):
    # Basic table-based distribution
    # For example:
    IF chest_type == "silver":
        RETURN {"common": 10, "rare": 2}
    ELIF chest_type == "gold":
        RETURN {"common": 20, "rare": 5, "epic": 1}
    ELSE:
        RETURN {"common": 5, "rare": 1}

FUNCTION FETCH_AVAILABLE_CARDS(arena):
```

```
    # In a real system, fetch from a database or config
    # For simplicity, pretend we have these cards
    return [
        {"id": "knight", "rarity": "common"},
        {"id": "archer", "rarity": "common"},
        {"id": "giant",  "rarity": "rare"},
        {"id": "dragon", "rarity": "epic"}
    ]

FUNCTION PICK_RANDOM_CARDS(cards, rarity, count):
    # Filter by rarity
    same_rarity_cards = []
    FOR c IN cards:
        IF c["rarity"] == rarity:
            same_rarity_cards.append(c)

    selected = []
    # Example: random choice logic
    FOR i IN RANGE(count):
        IF same_rarity_cards IS NOT EMPTY:
            r = RANDOM_INDEX(0, LEN(same_rarity_cards)-1)
            selected_card = same_rarity_cards[r]
            selected.append(selected_card)
    RETURN selected

FUNCTION chest.ADD_CARDS(cards_list):
    FOR c IN cards_list:
        self.rewards.append({
            "type": "card",
            "card_id": c["id"],
            "rarity": c["rarity"]
        })

FUNCTION player.INCREMENT_CHEST_COLLECTION(chest):
    # Add chest to player's chest inventory
    self.chests.append(chest)

FUNCTION SEND_REWARD_NOTIFICATION(player, chest):
    notification = {
        "type": "chest_reward",
        "chest_type": chest.type,
        "estimated_unlock_time": chest.unlock_time
    }
    player.SEND_MESSAGE(notification)


###################################################
# 10. Detailed Definitions of Referenced Functions
###################################################

FUNCTION INITIALIZE_SERVER_SOCKET(host, port):
    socket_obj = CREATE_SOCKET()
    socket_obj.BIND(host, port)
    socket_obj.LISTEN(MAX_PENDING_CONNECTIONS)
    RETURN socket_obj
END FUNCTION

FUNCTION server_socket.POLL_CONNECTIONS():
    # Implementation depends on the environment.
    # Typically uses select/poll to find new connection requests.
    # Returns a list of pending connection objects.
    pending_connections = OS_SPECIFIC_POLL(self.internal_socket)
    RETURN pending_connections
```

```
END FUNCTION

FUNCTION server_socket.ACCEPT(connection_request):
    # Accepts a connection, returns a (connection, address) tuple
    conn, addr = OS_ACCEPT(connection_request)
    RETURN (conn, addr)
END FUNCTION

FUNCTION CREATE_THREAD_POOL(SIZE):
    pool = ThreadPool(SIZE)
    # Initialize worker threads
    pool.init_workers()
    RETURN pool
END FUNCTION

FUNCTION ASSIGN_THREAD(function, args=()):
    # Queues the task into the pool
    GLOBAL_THREAD_POOL.add_task(function, args)
END FUNCTION

FUNCTION START_THREAD(function, args=()):
    # Creates a single dedicated thread for a specific function
    t = Thread(target=function, args=args)
    t.start()
    RETURN t
END FUNCTION

FUNCTION REGISTER_PLAYER(connection, address):
    global PLAYER_ID_COUNTER
    PLAYER_ID_COUNTER += 1
    new_player_id = PLAYER_ID_COUNTER
    new_player = Player(
        id=new_player_id,
        connection=connection,
        address=address
    )
    new_player.elixir = 0
    new_player.is_active = True
    new_player.arena = 1
    new_player.chests = []
    # Possibly send a welcome message
    SEND_INFO(new_player, "Welcome! Your player ID is " + STR(new_player_id))
    RETURN new_player
END FUNCTION

FUNCTION Player.WAIT_FOR_REQUEST():
    # Blocks or non-blocks reading from socket
    # parse data into a Request object
    raw_data = self.connection.READ()
    IF raw_data IS EMPTY:
        RETURN NULL
    parsed_request = PARSE_REQUEST(raw_data)
    RETURN parsed_request
END FUNCTION

FUNCTION Player.ADD_REQUEST(request):
    self.request_queue.append(request)
END FUNCTION

FUNCTION Player.has_pending_requests():
    RETURN (LEN(self.request_queue) > 0)
END FUNCTION

```

```
FUNCTION Player.GET_REQUEST():
    IF LEN(self.request_queue) == 0:
        RETURN NULL
    RETURN self.request_queue.pop(0)
END FUNCTION

FUNCTION CLEANUP_PLAYER(player):
    # Closes the socket connection if still open
    IF player.connection IS NOT NULL:
        player.connection.CLOSE()

    # Mark as inactive
    player.is_active = False

    # Remove from matchmaking if needed
    # (Implementation depends on how matchmaking queue is structured)
    matchmaking_queue.REMOVE(player)

    # Additional persistence if necessary
END FUNCTION

FUNCTION GET_ACTIVE_GAME_FOR_PLAYER(player, active_games):
    FOR game_id, game IN active_games.items():
        IF player.id IN game.player_ids:
            RETURN game
    RETURN NULL
END FUNCTION

FUNCTION SEND_ERROR(player, message):
    error_packet = {
        "type": "error",
        "message": message
    }
    player.SEND_MESSAGE(error_packet)

FUNCTION SEND_INFO(player, message):
    info_packet = {
        "type": "info",
        "message": message
    }
    player.SEND_MESSAGE(info_packet)

FUNCTION LOGOUT_PLAYER(player):
    player.is_active = False
    # Additional logic: remove from matchmaking queue if present
    matchmaking_queue.REMOVE(player)
    # Possibly broadcast that this player has disconnected

FUNCTION INITIALIZE_NEW_GAME(player1, player2, event_dispatcher):
    new_game = CREATE_GAME_INSTANCE()
    new_game.ADD_PLAYER(player1)
    new_game.ADD_PLAYER(player2)
    new_game.id = GENERATE_UNIQUE_GAME_ID()
    new_game.event_dispatcher = event_dispatcher
    new_game.is_terminated = FALSE
    new_game.time_elapsed = 0
    new_game.max_duration = 180  # 3 minutes, for example
    # Setup default towers, base elixir, etc.
    new_game.towers = CREATE_DEFAULT_TOWERS(player1, player2)
    # Register them in the game
    FOR tower IN new_game.towers:
        new_game.entities.append(tower)

```

```
    RETURN new_game
END FUNCTION

FUNCTION CREATE_GAME_INSTANCE():
    g = Game()
    g.entities = []
    g.players = []
    RETURN g

FUNCTION Game.ADD_PLAYER(player):
    self.players.append(player)
    self.player_ids.append(player.id)

FUNCTION GENERATE_UNIQUE_GAME_ID():
    global GAME_ID_COUNTER
    GAME_ID_COUNTER += 1
    RETURN GAME_ID_COUNTER

FUNCTION NOTIFY_PLAYERS_OF_MATCH(player1, player2, game):
    match_info = {
        "type": "match_found",
        "opponent_ids": [player1.id, player2.id],
        "game_id": game.id
    }
    player1.SEND_MESSAGE(match_info)
    player2.SEND_MESSAGE(match_info)

FUNCTION UpdateAllGames(active_games, event_dispatcher):
    # Already defined, with a TICK_RATE, etc.

FUNCTION UPDATE_GAME(game, delta_time):
    # Called from game.UPDATE(delta_time)
    game.INCREMENT_TIME(delta_time)
    # rest is same as above
END FUNCTION

FUNCTION game.INCREMENT_TIME(delta_time):
    self.time_elapsed += delta_time

FUNCTION game.TERMINATE():
    self.is_terminated = TRUE
    # Additional logic: mark final states, award trophies, etc.

FUNCTION game.REPLENISH_ELIXIR(delta_time):
    # Implementation shown above

FUNCTION game.BROADCAST_GAME_STATE():
    # Gather positions, HP, etc., and send to all players
    state_snapshot = {
        "type": "game_state",
        "entities": [
            {
                "id": e.id,
                "hp": e.hp,
                "position": e.position
            }
            FOR e IN self.entities
        ],
        "time_elapsed": self.time_elapsed
    }
    FOR p IN self.players:
        p.SEND_MESSAGE(state_snapshot)

```

```
FUNCTION CREATE_DEFAULT_TOWERS(player1, player2):
    # Typically each player has 1 King Tower + 2 Princess Towers
    tower1 = Tower(owner=player1.id, hp=3000, side="bottom")
    tower2 = Tower(owner=player1.id, hp=2000, side="bottom")
    tower3 = Tower(owner=player1.id, hp=2000, side="bottom")
    tower4 = Tower(owner=player2.id, hp=3000, side="top")
    tower5 = Tower(owner=player2.id, hp=2000, side="top")
    tower6 = Tower(owner=player2.id, hp=2000, side="top")
    return [tower1, tower2, tower3, tower4, tower5, tower6]

FUNCTION CLEANUP_SERVER_RESOURCES(server_socket, player_registry, active_games):
    server_socket.CLOSE()
    FOR player_id, player IN player_registry.items():
        CLEANUP_PLAYER(player)
    FOR game_id, game IN active_games.items():
        game.TERMINATE()
END FUNCTION


###################################################
# 11. Error Handling & Validation
###################################################

FUNCTION RAISE_ERROR(message):
    # Throws an exception in actual code
    # For pseudocode, we might just print or log it
    PRINT("ERROR: " + message)
    # Or raise an exception if the language supports it

FUNCTION SEND_ERROR(player, message):
    # Already defined above
```

## 8.1 Algorithm Flowcharts

### 8.1.1 Main Server Loop Flowchart

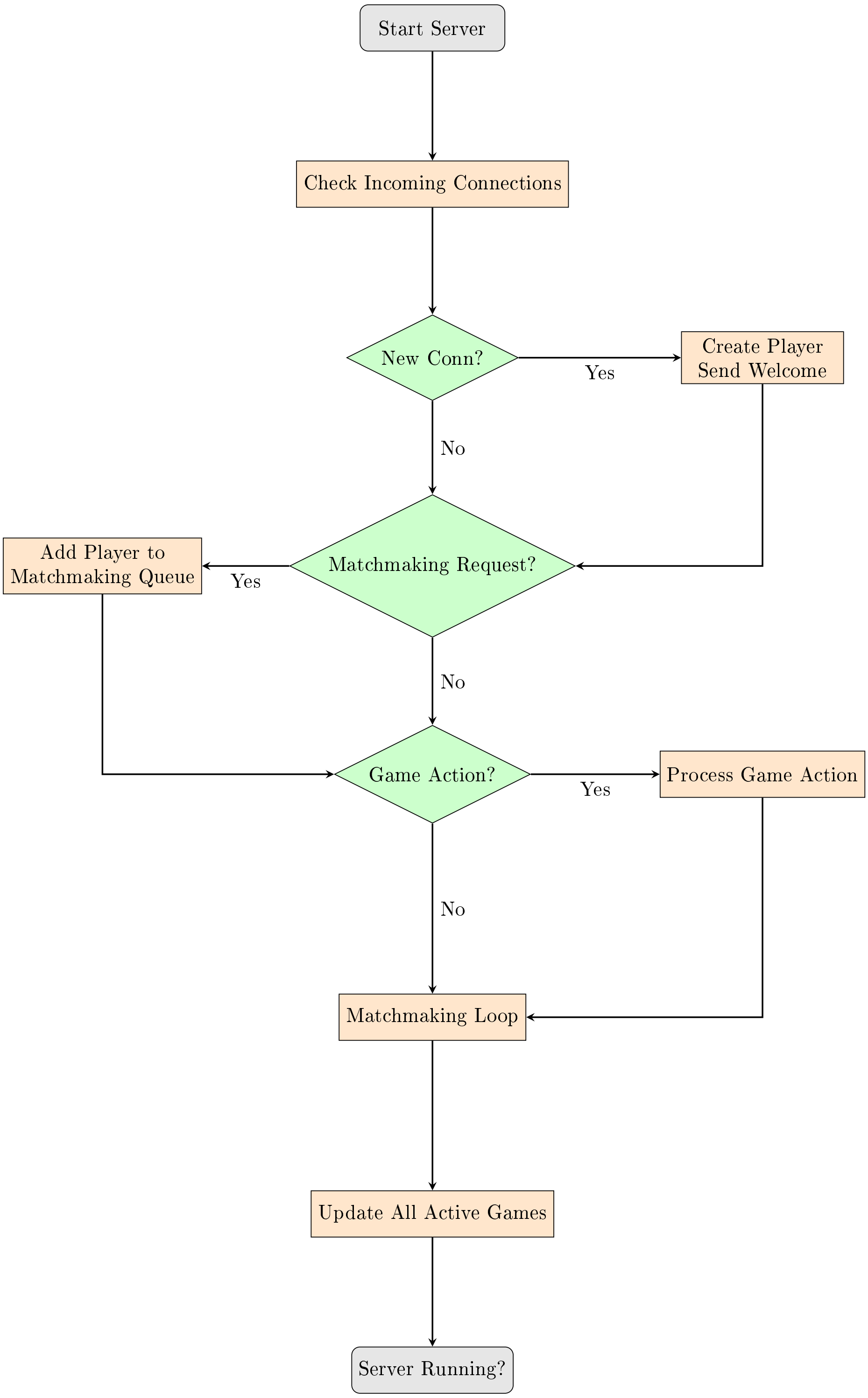



Figure 1: Main Server Loop Flowchart

### 8.1.2 Card Deployment Process Flowchart

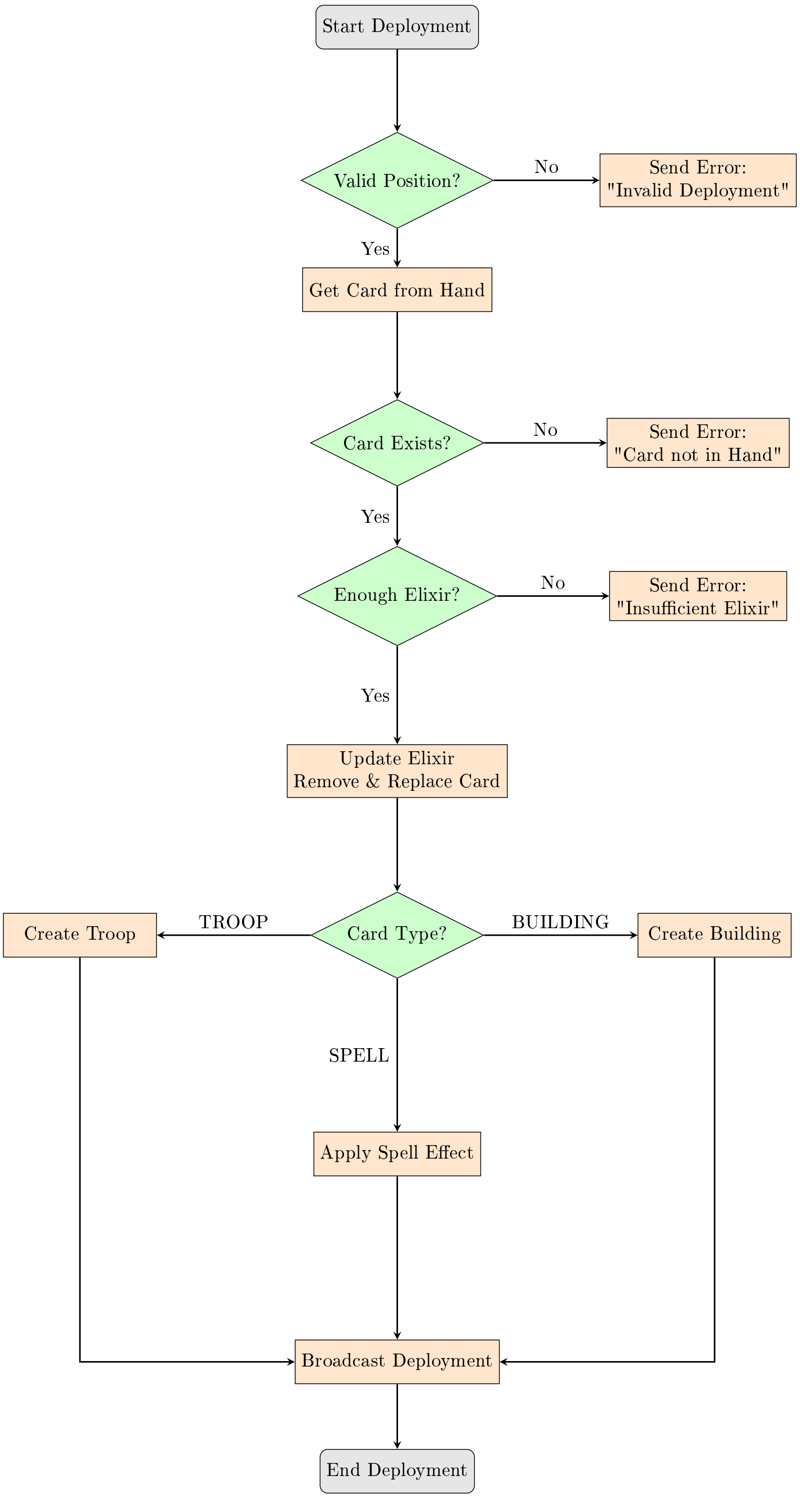



Figure 2: Card Deployment Flowchart

### 8.1.3 Battle Update Logic Flowchart

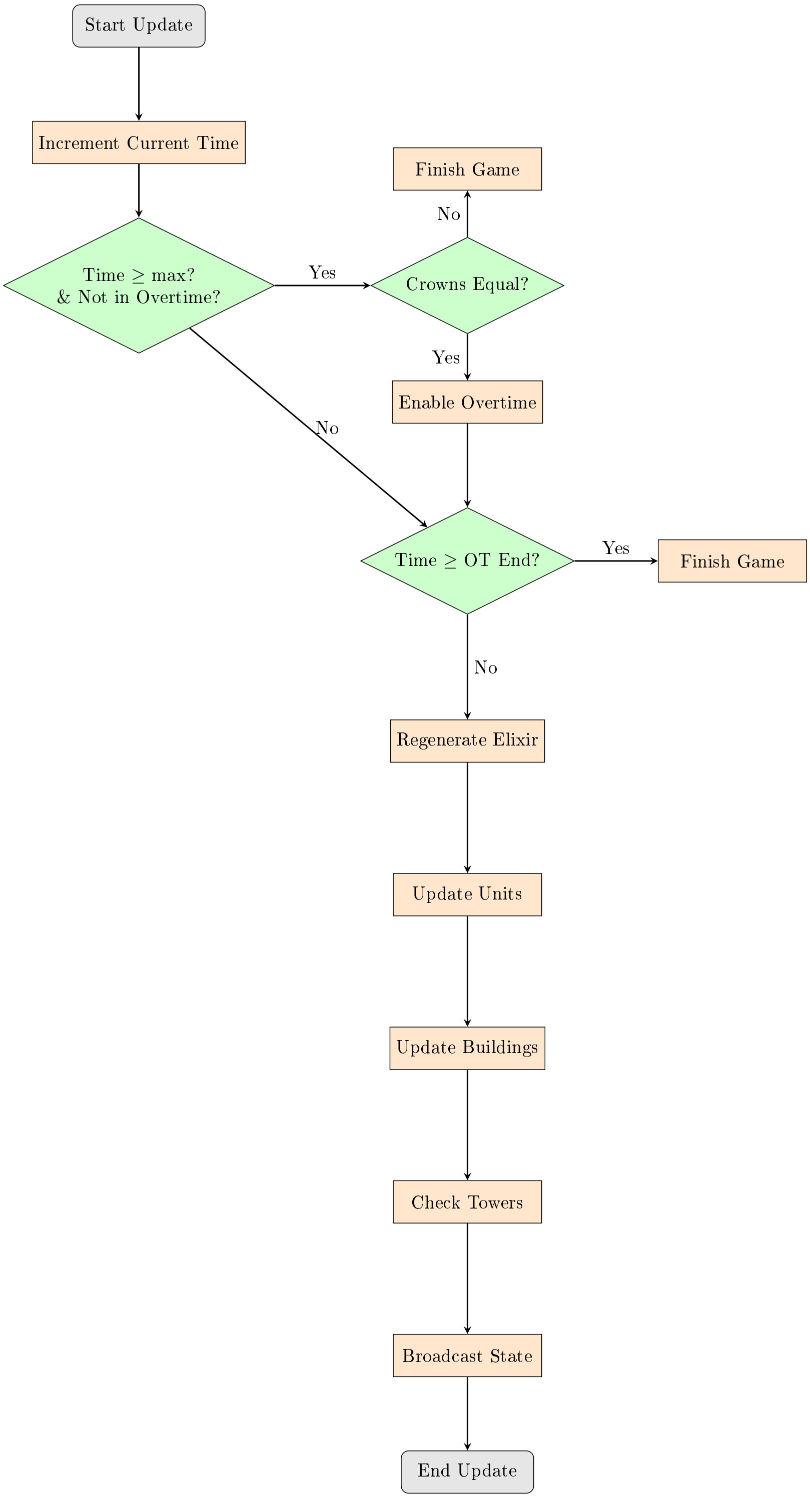



Figure 3: Battle Update Logic Flowchart

### 8.1.4 Target Selection Algorithm Flowchart

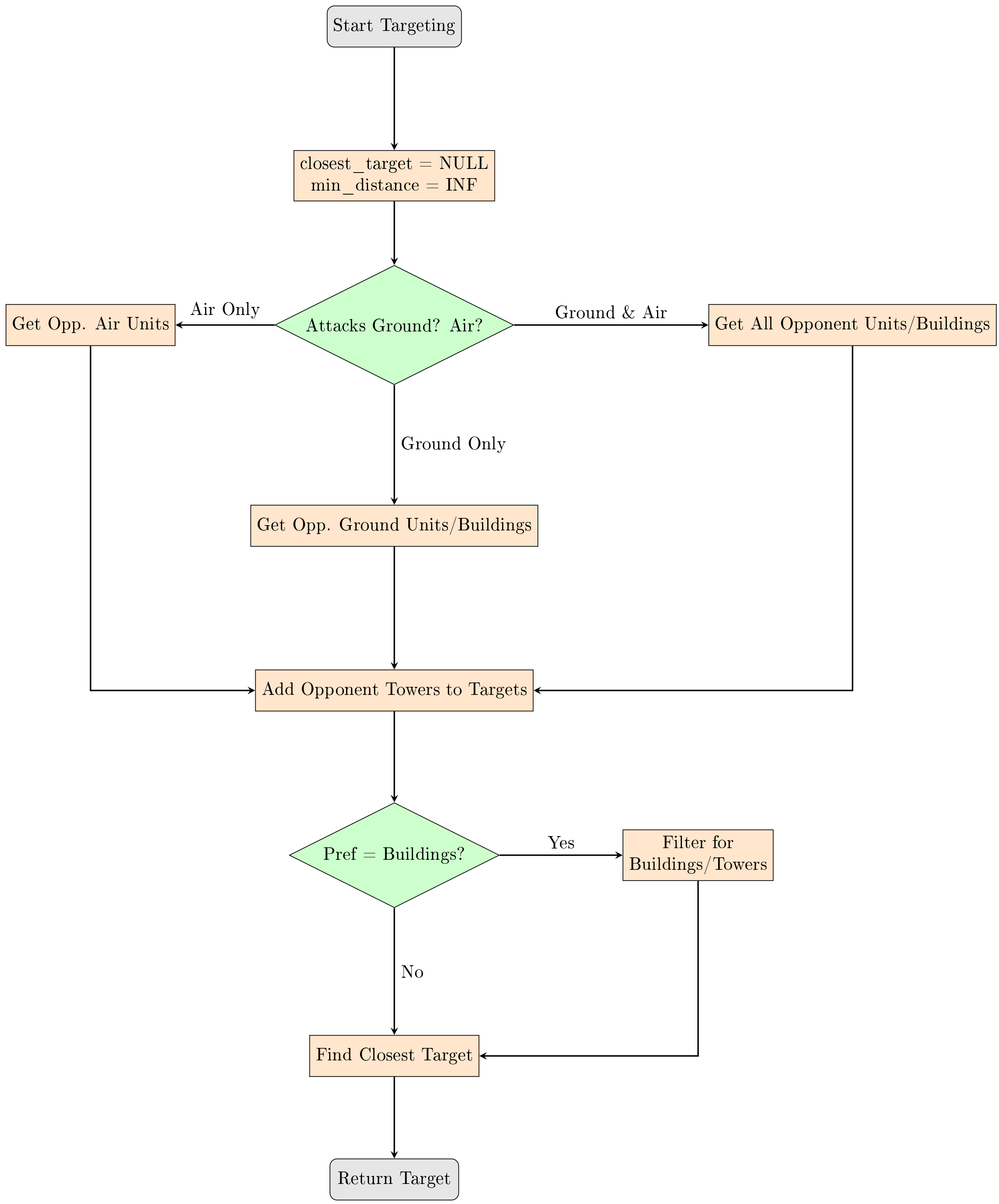



Figure 4: Target Selection Flowchart

#### 8.1.5 Chest Reward System Flowchart

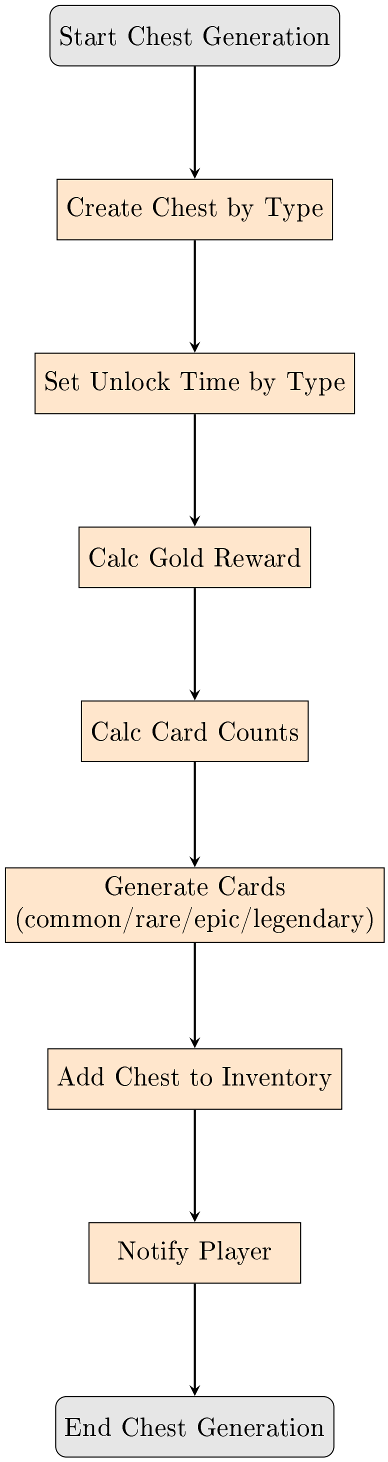



Figure 5: Chest Reward System Flowchart

# 9 Trace Tables

The following subsections provide example trace tables for typical scenarios in each major workflow.

## 9.1 Main Server Loop Trace Table

| Step | Condition | Action | player_registry | active_games |
|---|---|---|---|---|
| 1 | new_connections=[ConnA] | REGISTER_PLAYER(ConnA) | {1: PlayerA} | {} |
| 2 | PlayerA requests matchmaking | matchmaking_queue.ADD(PlayerA) | {1: PlayerA} | {} |
| 3 | new_connections=[ConnB] | REGISTER_PLAYER(ConnB) | {1: PlayerA, 2: PlayerB} | {} |
| 4 | PlayerB requests matchmaking | matchmaking_queue.ADD(PlayerB) | same | {} |
| 5 | PROCESS_MATCHMAKING finds pair | INITIALIZE_NEW_GAME | same | {1001: Game1} |
| 6 | event_dispatcher.EXECUTE | no pending events | same | same |

Table 1: Main Server Loop - Two Players Matching

## 9.2 Player Handling (HANDLE_PLAYER) Trace Table

| Step | Condition | Action | Outcome |
|---|---|---|---|
| 1 | player.is_active = True | WAIT_FOR_REQUEST() | request = None |
| 2 | request is None | loop repeats | handle_player in wait |
| 3 | external event: player.is_active=False | exit loop | CLEANUP_PLAYER(player) |
| 4 | CLEANUP_PLAYER | close socket, remove from queue | player inactivated |

Table 2: HANDLE_PLAYER - Disconnect Scenario

## 9.3 Request Routing (HANDLE_PLAYER_REQUEST) Trace Table

| Step | Input | Check / Action | Outcome |
|---|---|---|---|
| 1 | request.type="game_action" | GET_ACTIVE_GAME_FOR_PLAYER | game=Game1 |
| 2 | action="deploy", card_id="knight" | ProcessCardDeployment | successful deployment |
| 3 | else branch? | not triggered | - |
| 4 | End | - | Request done |

Table 3: Request Routing - Deploy Action

## 9.4 Matchmaking Process Trace Table

| Step | Condition | Action | matchmaking_queue | active_games |
|---|---|---|---|---|
| 1 | queue.has_pairs()=True | get_matched_players() | empty after pop | {} |
| 2 | players valid | initialize new game | [] | {2001: GameObj} |
| 3 | queue.has_pairs()=False | break | [] | same |

Table 4: Matchmaking - Single Pair Scenario

## 9.5 Game Update Thread (UpdateAllGames) Trace Table

| Step | Games | Check | Action | Result |
|---|---|---|---|---|
| 1 | {G1(not terminated), G2(terminated)} | G1: not terminated | G1.UPDATE() | G1 remains active |
| 2 | G2: is_terminated=True | remove from active_games | G2 removed | {G1} left |
| 3 | Sleep | 1 / TICK_RATE | wait next frame | - |

Table 5: Game Update Thread - One Terminated, One Active

## 9.6 Game Update Logic (UPDATE_GAME) Trace Table

| Step | Condition | Action | Outcome |
|---|---|---|---|
| 1 | time_elapsed=179, max_duration=180 | INCREMENT_TIME(1s) | time_elapsed=180 |
| 2 | check_end_conditions | time_elapsed >= 180 | DETERMINE_GAME_WINNER_BY_HP, TERMINATE |
| 3 | after termination | skip further steps | game ended |

Table 6: Game Update Logic - Timeout Scenario

## 9.7 Card Deployment Process Trace Table

| Step | Condition | Action | Result |
|---|---|---|---|
| 1 | position=(3,4) | VALIDATE_DEPLOYMENT_POSITION | valid position |
| 2 | card in hand | yes | proceed |
| 3 | player.elixir=6, cost=4 | verify cost | enough elixir |
| 4 | SPEND_ELIXIR(4) | discard card, draw next | elixir=2 |
| 5 | CREATE_ENTITY | new troop in game | entity registered |
| 6 | BROADCAST_DEPLOYMENT | all players see new troop | done |

Table 7: Card Deployment - Valid Scenario

## 9.8 Target Selection Algorithm Trace Table

| Step | Targets | Compute Score | Min Score Target |
|---|---|---|---|
| 1 | [T1, T2, T3] | T1=10, T2=7, T3=12 | T2 is best so far |
| 2 | best_target = T2 | - | final = T2 |

Table 8: Target Selection - Closest Target Example

## 9.9 Chest Reward System Trace Table

| Step | Input | Action | Outcome |
|---|---|---|---|
| 1 | chest_type="gold" | CREATE_CHEST_INSTANCE("gold") | chest.type=gold |
| 2 | ASSIGN_UNLOCK_TIME | 8 hours | chest.unlock_time=28800 |
| 3 | ALLOCATE_GOLD_REWARDS | base=100+(arena*10) | added to chest |
| 4 | ALLOCATE_CARD_REWARDS | {common:20, rare:5, epic:1} | random picks |
| 5 | INCREMENT_CHEST_COLLECTION | chest added to inventory | player has new chest |
| 6 | SEND_REWARD_NOTIFICATION | player alerted | done |

Table 9: Chest Reward System - Gold Chest Scenario