VIETNAM NATIONAL UNIVERSITY OF HOCHIMINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

# Project Report: Text-Based Clash Royale Game (TCR)

**Course:** Network Programming (Net-Centric Programming) – IT096IU
**Instructor:** Le Thanh Son
**Group:** Group 41
**Submission Date:** 30/05/2025

## Member List

| Full Name | ID Student |
|---|---|
| Tran Vu Khanh Hung | ITCSIU21182 |
| Vo Hoai Bao | ITITIU21038 |

Ho Chi Minh City, Vietnam

2025

# Contents

# 1. Introduction

Text-Based Clash Royale (TCR) is a simplified, text-only adaption of the popular strategy game - Clash Royale. It offers players a gameplay experience without the need for complex graphics, focusing on core mechanics like attacking, defending, and mana management.

This project aims to implement a networked, text-based version of the popular game Clash Royale using Golang. The main goal is to provide hands-on experience in network programming using TCP protocol, concurrency, and real-time communication. It also serves as a practical exercise in designing communication protocols, synchronizing game states, and managing player data. This project implements a simplified, text-based multiplayer version of Clash Royale, designed to explore network programming concepts using Golang.

Built with the Go programming language and modern networking techniques, TCR makes a great effort to be a scalable, maintainable, and extended platform. It offers a dynamic learning experience that bridges the gap between networking concepts and game development in a client-server architecture.

# 2. Objectives

The primary objectives of the Text-Based Clash Royale (TCR) project are to:

- Develop a network-based two-player game with client-server architecture.
- Enable hands-on implementation of TCP communication protocol at the programming level.
- Reinforce theoretical knowledge of networking and communication through real-world application in a real-time gaming scenario.
- Develop skills in client-server architecture, including managing concurrent connections and synchronizing game state across two clients.
- Implement game mechanics such as turn-based play, real-time continuous play. mana management, and progression systems.

# 3. System Architecture

## 3.1 Overview

The system architecture of TCR follows client-server model designed to support multiplayer gameplay over a network. The architecture consists of a central game server that maintains the game state and applies game rules, while multiple clients connect to this server to participate in matches.

The server handles all game logic, including player authentication, troop deployment, attack calculations, and state synchronization. It manages multiple concurrent client connections using Go’s concurrency features to ensure responsiveness and smooth gameplay.

Clients have a role as interfaces for players, allowing them to log in, send commands, and receive real-time updates about the state of gaming. Communication between clients and the server is established over TCP protocol, ensuring reliable transmission of messages.

This architecture promotes scalability and maintainability by separating concerns between game logic processing on the server and user interaction on the client side.

## 3.2 Components

### 3.2.1 Server

The server component represents a robust TCP-based implementation that functions as the core infrastructure for the Text Combat Royale (TCR) game. Its architecture is designed to efficiently manage multiple concurrent client connections while ensuring real-time synchronization of gameplay and maintaining the integrity of the game state. The server maintains collections of active clients and ongoing game sessions, employing mutex locks to guarantee thread-safe operations during concurrent access.

A sophisticated matchmaking subsystem is incorporated, featuring both basic and enhanced matchmaking queues that facilitate player pairing according to individual preferences. The server oversees the complete game lifecycle, encompassing player authentication and registration, match creation, gameplay management, and the orderly conclusion of matches.

To ensure connection stability and responsiveness, the server implements a ping/pong mechanism to monitor client connectivity, alongside automated procedures for the cleanup of inactive clients. Furthermore, the architecture supports graceful shutdown processes to maintain system reliability.

Communication between the server and clients is conducted through a structured message protocol that encompasses a variety of message types, including those related to authentication, matchmaking, in-game actions, and system notifications. The server also interfaces with a persistent data management system responsible for maintaining player profiles and game configuration data.

Overall, the server is engineered for scalability and maintainability, incorporating comprehensive logging and rigorous error handling throughout its operation to facilitate robust performance and ease of maintenance.

### 3.2.2 Clients

The client component constitutes a sophisticated TCP-based implementation that functions as the primary user interface and interaction layer for the Text Combat Royale (TCR) game. Its architecture integrates multiple specialized modules that collaborate to deliver seamless gameplay experience. The client establishes and maintains a persistent connection with the server, facilitating real-time synchronization of the game state.

The display component provides a rich, color-coded output that categorizes information into server status updates, game events, combat actions, and other relevant notifications, enhancing user comprehension and engagement. User input is managed through an intuitive handling system designed to process commands efficiently.

Supporting two distinct game modes, Simple TCR (turn-based) and Enhanced TCR (real-time), the client implements corresponding gameplay mechanics and manages state accordingly for each mode. Robust error handling mechanisms are incorporated, alongside automatic reconnection strategies and graceful shutdown procedures to ensure reliability and continuity of the user experience.

The client maintains detailed tracking of the game state, including troop deployments, attack counts, tower health status, and mana resource management. A comprehensive debugging subsystem is integrated to provide in-depth analysis of the game state and actionable recommendations.

User authentication is facilitated through a sophisticated system supporting both login and registration workflows, with rigorous input validation and error handling to maintain security and usability. Additional features include automatic mana regeneration, real-time combat status updates, and detailed tracking of player statistics.

Overall, the client architecture is engineered to be highly responsive and user-friendly, offering clear visual feedback and an intuitive command interface that enables efficient execution of all game-related actions.

### 3.2.3 Data Storage

The data storage component constitutes a robust and systematically organized subsystem responsible for managing all persistence operations within the Text Combat Royale (TCR) game. It is implemented via a dedicated DataManager class that oversees three principal JSON-based data repositories: player data (**players.json**), troop specifications (**troops.json**), and tower specifications (**towers.json**).

This component incorporates an advanced player database that records comprehensive player statistics, including player level, experience points, troop and tower levels, total games played, and win rates. A progressive leveling mechanism is employed, wherein the experience points required for advancement increase by 15% per level, accompanied by a corresponding 10% scaling in the statistics of troops and towers.

The DataManager facilitates critical functions such as player authentication, registration, and session management, maintaining active player sessions, and ensuring thread-safe access through appropriate synchronization mechanisms. It also provides methods for calculating experience gains attributable to various in-game actions, including victories, defeats, draws, inflicted damage, and tower destruction. Additionally, the component autonomously adjusts troop and tower statistics based on player progression levels.

Emphasizing data integrity, the system incorporates comprehensive error handling and rigorous validation processes, guaranteeing the proper initialization of new player accounts with default parameters. Designed to be thread-safe, the data storage component supports concurrent access by multiple game instances while preserving consistency and reliability of the underlying data.

## 3.3 Communication Flow

The communication flow within the Text Combat Royale (TCR) system is implemented through a TCP-based client-server architecture utilizing a structured, JSON-based message protocol.

### 3.3.1 Message Protocol

Messages are defined and structured in the **network/protocol.go** file and comprise the following standardized fields: message type, player ID, game ID, timestamp, and data payload. The protocol supports multiple message categories, including:

- **Authentication messages:** LOGIN, REGISTER, AUTH_OK, AUTH_FAIL
- **Matchmaking messages:** FIND_MATCH, MATCH_FOUND, GAME_START
- **Game action messages:** SUMMON_TROOP, ATTACK, END_TURN, SURRENDER
- **Game state messages:** GAME_STATE, GAME_EVENT, GAME_END
- **System messages:** ERROR, PING, PONG, DISCONNECT

### 3.3.2 Client-Server Connection

Clients establish persistent TCP connections to the server, each maintaining buffered readers and writers to facilitate message transmission. The server manages a mapping of connected clients and their respective states. Connection health is actively monitored using periodic **PING** and **PONG** messages to detect any disruptions.

### 3.3.3 Authentication Flow

Clients initiate authentication by sending either a **LOGIN** or **REGISTER** message containing user credentials. The server validates these credentials through the data management subsystem and responds with either **AUTH_OK** or **AUTH_FAIL**. Upon successful authentication, the client receives relevant player data and is marked as authenticated.

### 3.3.4 Matchmaking Flow

Authenticated clients send **FIND_MATCH** messages specifying their preferred game mode. The server places clients into appropriate matchmaking queues and, upon pairing two players, instantiates a new game session. Both clients receive a **MATCH_FOUND** message containing opponent details, followed by game initialization information delivered via the **GAME_START** message.

### 3.3.5 Game Action Flow

During gameplay, clients transmit action messages such as **SUMMON_TROOP**, **ATTACK**, and **END_TURN**. The server validates all actions against game rules and subsequently broadcasts **GAME_EVENT** messages to both players. Game state synchronization is maintained through periodic **GAME_STATE** messages. For the Enhanced game mode, **MANA_UPDATE** messages are also dispatched at regular intervals.

### 3.3.6 Error Handling

The protocol incorporates comprehensive error handling mechanisms. Invalid or disallowed actions prompt the server to return **ERROR** messages containing specific error codes. Connection issues trigger automatic reconnection attempts. The server maintains connection state information and manages disconnections gracefully.

### 3.3.7 Real-time Updates

All significant game events are broadcast promptly to relevant clients, ensuring immediate synchronization of state changes. The Enhanced mode supports real-time mana regeneration updates, while turn transitions are communicated using **TURN_CHANGE** messages.

### 3.3.8 Disconnection Handling

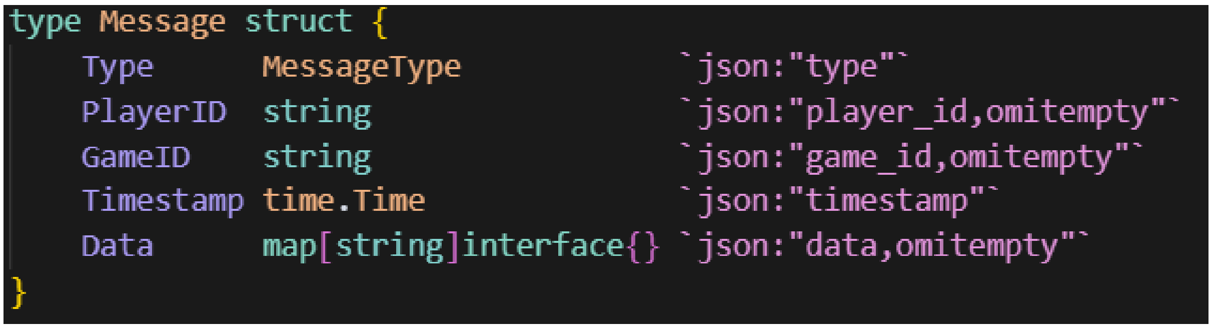
The server detects client disconnections in a timely manner, properly terminating any active games involving the disconnected player. Opponents are notified accordingly, and matchmaking queues are updated to reflect the change in player availability.

# 4. Application PDU (Protocol Data Unit) Description

The communication between the client and server in the Text Combat Royale (TCR) game is facilitated through a well-defined application-layer Protocol Data Unit (PDU) structure implemented using a JSON-based message protocol. This section outlines the design, structure, and handling of these PDUs.

## 4.1 Message Structure

All messages exchanged between client and server adhere to a common base structure defined by the Message struct:



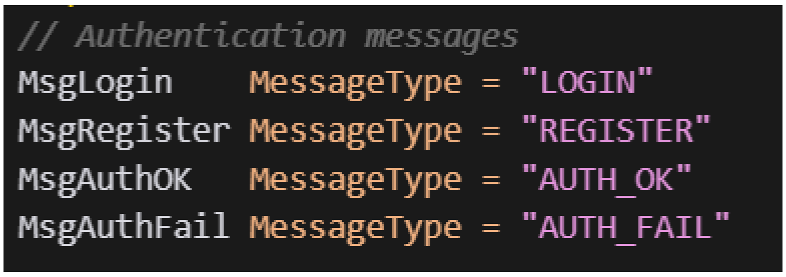
```
type Message struct {
    Type      MessageType            `json:"type"`
    PlayerID  string                 `json:"player_id,omitempty"`
    GameID    string                 `json:"game_id,omitempty"`
    Timestamp time.Time              `json:"timestamp"`
    Data      map[string]interface{} `json:"data,omitempty"`
}
```

**Figure 4.1**: Message struct

- **Type**: Specifies the purpose of the message (e.g., authentication, game action).
- **PlayerID**: Identifies the player involved, if applicable.
- **GameID**: Identifies the game session, if relevant.
- **Timestamp**: Records the time the message was created.
- **Data**: Carries the payload specific to the message type.

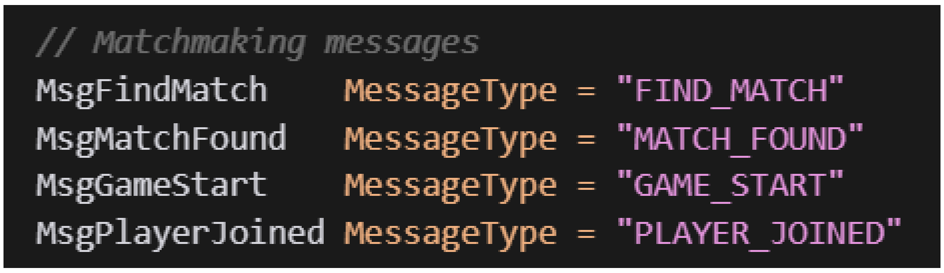
## 4.2 Message Types:

PDUs are categorized by type, represented as constants



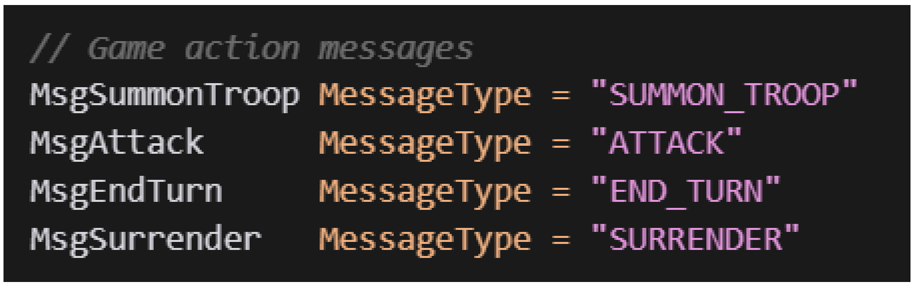
```
// Authentication messages
MsgLogin     MessageType = "LOGIN"
MsgRegister  MessageType = "REGISTER"
MsgAuthOK    MessageType = "AUTH_OK"
MsgAuthFail  MessageType = "AUTH_FAIL"
```

**Figure 4.2:** Authentication messages



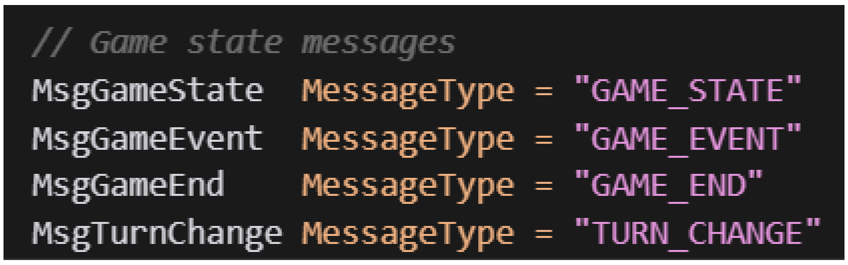
```
// Matchmaking messages
MsgFindMatch     MessageType = "FIND_MATCH"
MsgMatchFound    MessageType = "MATCH_FOUND"
MsgGameStart     MessageType = "GAME_START"
MsgPlayerJoined  MessageType = "PLAYER_JOINED"
```

**Figure 4.3:** Matchmaking messages



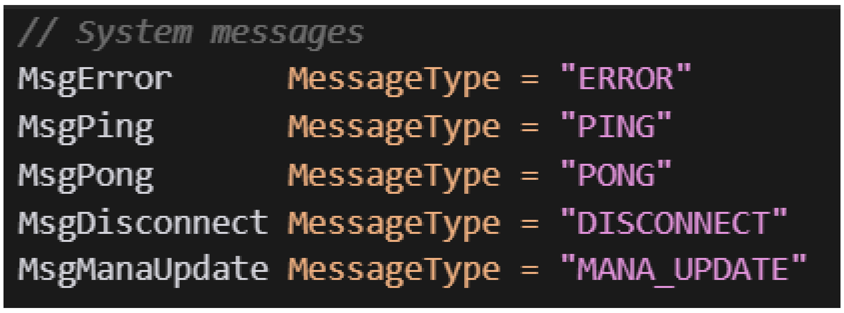
```
// Game action messages
MsgSummonTroop MessageType = "SUMMON_TROOP"
MsgAttack      MessageType = "ATTACK"
MsgEndTurn     MessageType = "END_TURN"
MsgSurrender   MessageType = "SURRENDER"
```

**Figure 4.4:** Game action messages



```
// Game state messages
MsgGameState  MessageType = "GAME_STATE"
MsgGameEvent  MessageType = "GAME_EVENT"
MsgGameEnd    MessageType = "GAME_END"
MsgTurnChange MessageType = "TURN_CHANGE"
```

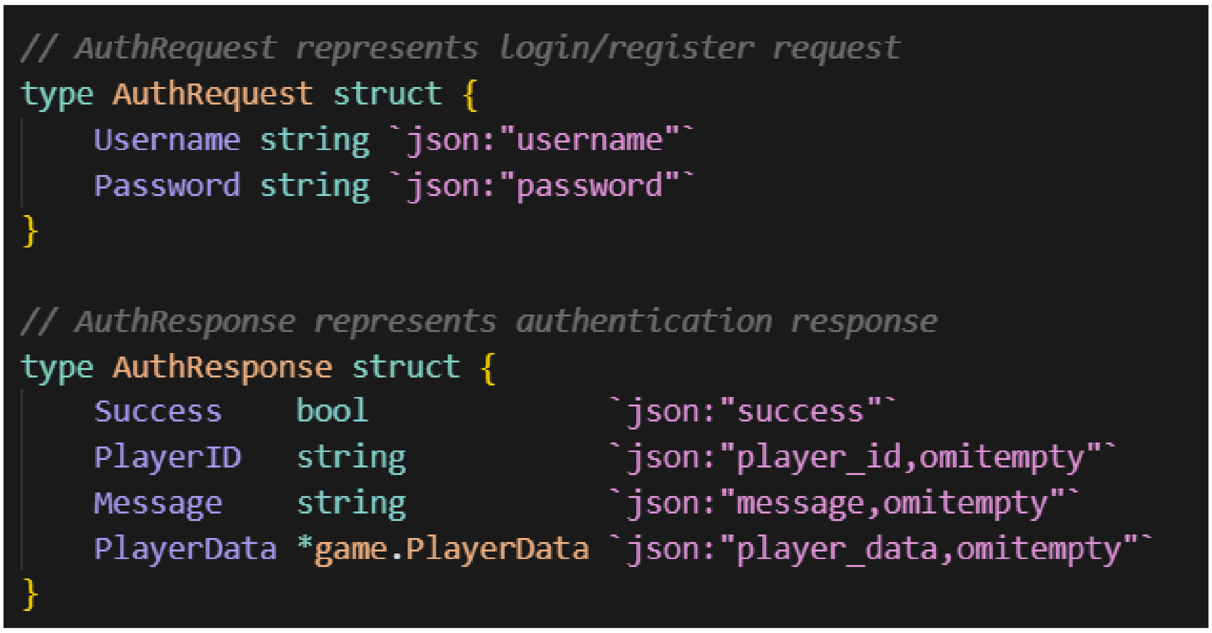
**Figure 4.5**: Game state messages



```
// System messages
MsgError      MessageType = "ERROR"
MsgPing       MessageType = "PING"
MsgPong       MessageType = "PONG"
MsgDisconnect MessageType = "DISCONNECT"
MsgManaUpdate MessageType = "MANA_UPDATE"
```

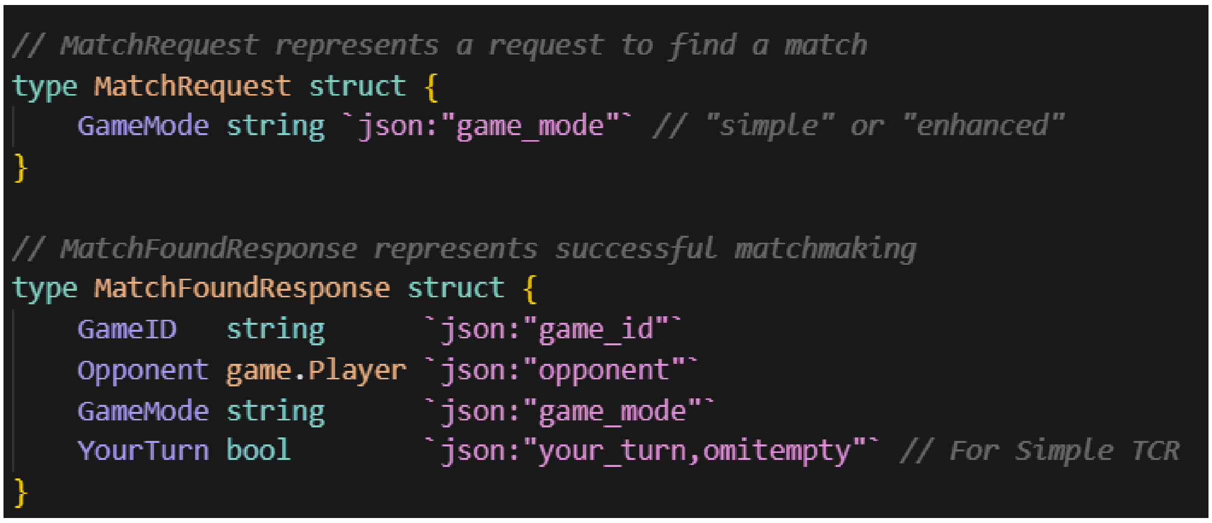
**Figure 4.6**: System messages

## 4.3 PDU Data Structures



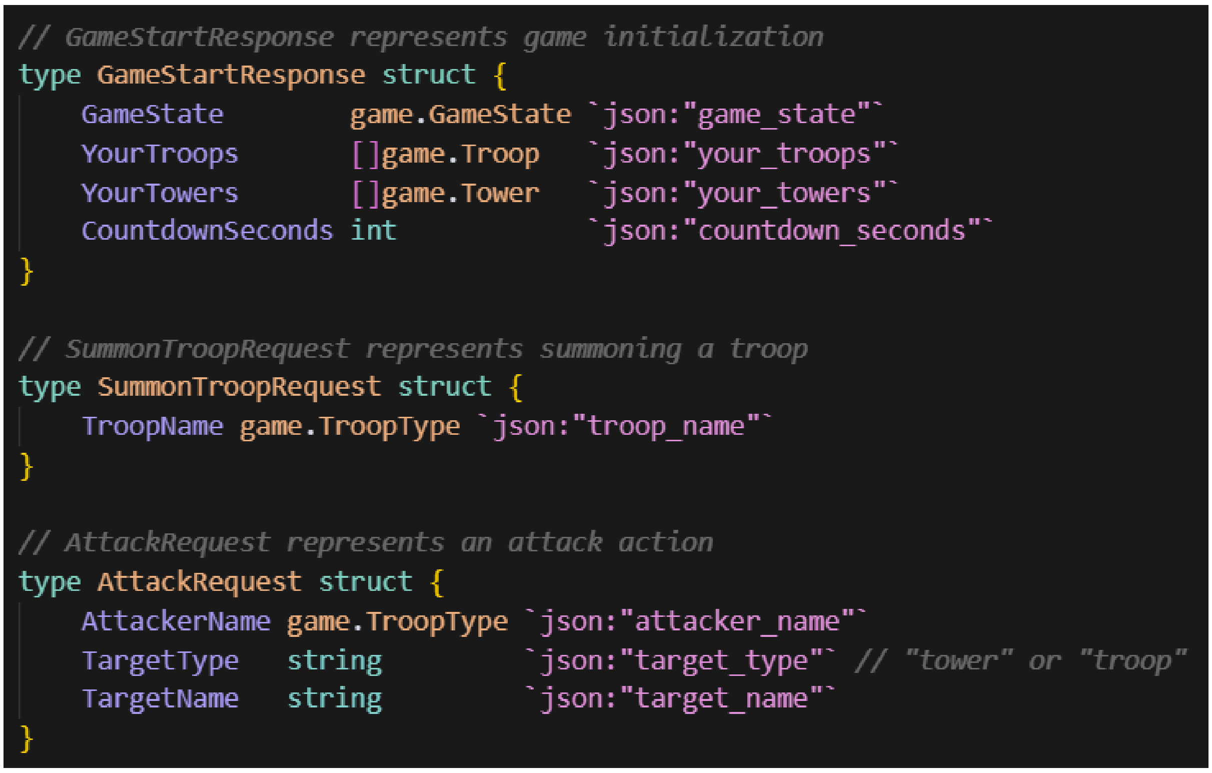
```
// AuthRequest represents login/register request
type AuthRequest struct {
    Username string `json:"username"`
    Password string `json:"password"`
}

// AuthResponse represents authentication response
type AuthResponse struct {
    Success    bool             `json:"success"`
    PlayerID   string           `json:"player_id,omitempty"`
    Message    string           `json:"message,omitempty"`
    PlayerData *game.PlayerData `json:"player_data,omitempty"`
}
```

**Figure 4.7**: Data Structure of Authentication



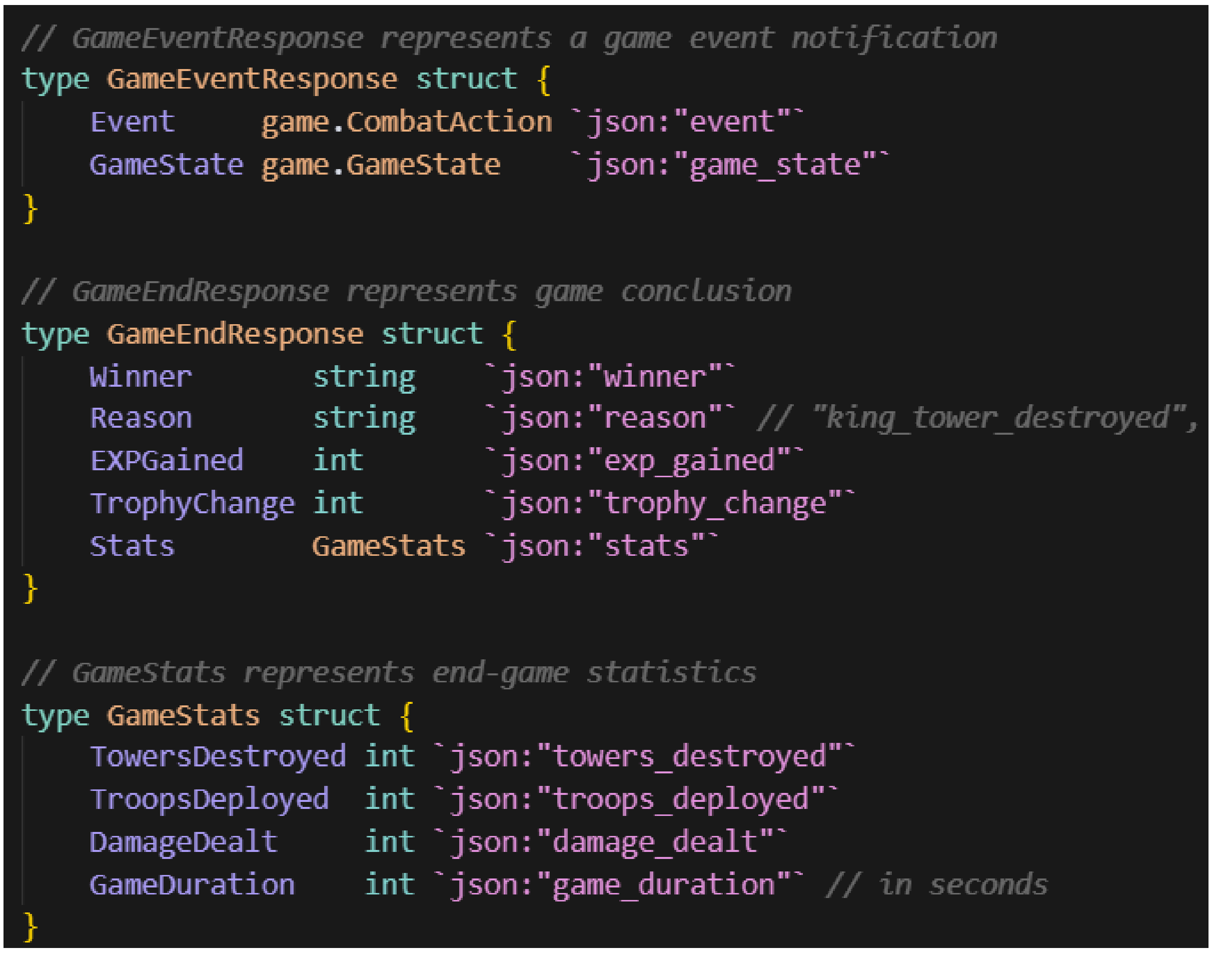
```go
// MatchRequest represents a request to find a match
type MatchRequest struct {
    GameMode string `json:"game_mode"` // "simple" or "enhanced"
}

// MatchFoundResponse represents successful matchmaking
type MatchFoundResponse struct {
    GameID   string      `json:"game_id"`
    Opponent game.Player `json:"opponent"`
    GameMode string      `json:"game_mode"`
    YourTurn bool        `json:"your_turn,omitempty"` // For Simple TCR
}
```

**Figure 4.8**: Data Structure of Matchmaking



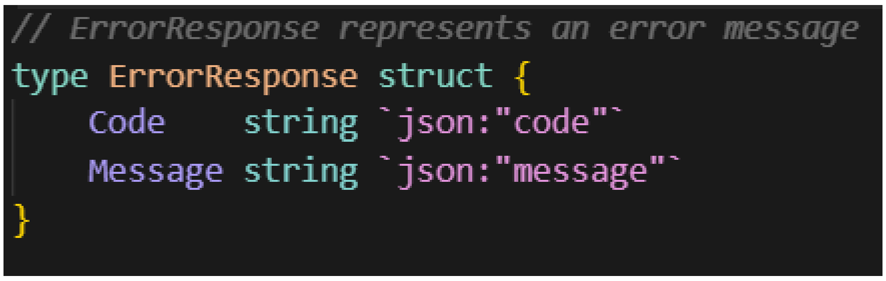
```go
// GameStartResponse represents game initialization
type GameStartResponse struct {
    GameState        game.GameState `json:"game_state"`
    YourTroops       []game.Troop   `json:"your_troops"`
    YourTowers       []game.Tower   `json:"your_towers"`
    CountdownSeconds int            `json:"countdown_seconds"`
}

// SummonTroopRequest represents summoning a troop
type SummonTroopRequest struct {
    TroopName game.TroopType `json:"troop_name"`
}

// AttackRequest represents an attack action
type AttackRequest struct {
    AttackerName game.TroopType `json:"attacker_name"`
    TargetType   string         `json:"target_type"` // "tower" or "troop"
    TargetName   string         `json:"target_name"`
}
```

**Figure 4.9**: Data Structure of GameAction



```go
// GameEventResponse represents a game event notification
type GameEventResponse struct {
    Event     game.CombatAction `json:"event"`
    GameState game.GameState    `json:"game_state"`
}

// GameEndResponse represents game conclusion
type GameEndResponse struct {
    Winner       string    `json:"winner"`
    Reason       string    `json:"reason"` // "king_tower_destroyed",
    EXPGained    int       `json:"exp_gained"`
    TrophyChange int       `json:"trophy_change"`
    Stats        GameStats `json:"stats"`
}

// GameStats represents end-game statistics
type GameStats struct {
    TowersDestroyed int `json:"towers_destroyed"`
    TroopsDeployed  int `json:"troops_deployed"`
    DamageDealt     int `json:"damage_dealt"`
    GameDuration    int `json:"game_duration"` // in seconds
}
```

**Figure 4.10**: Data Structure of GameState

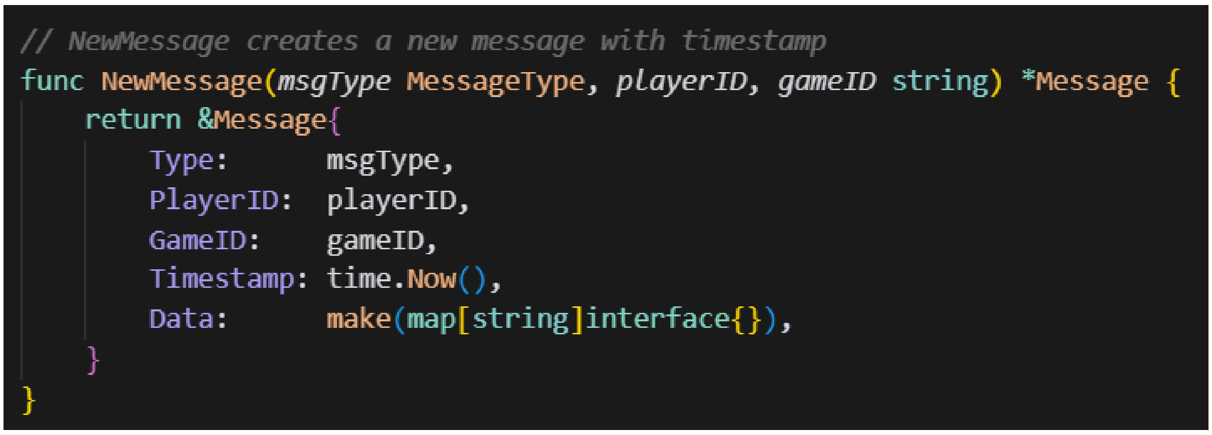


```go
// ErrorResponse represents an error message
type ErrorResponse struct {
    Code    string `json:"code"`
    Message string `json:"message"`
}
```

**Figure 4.11**: Data Structure of System

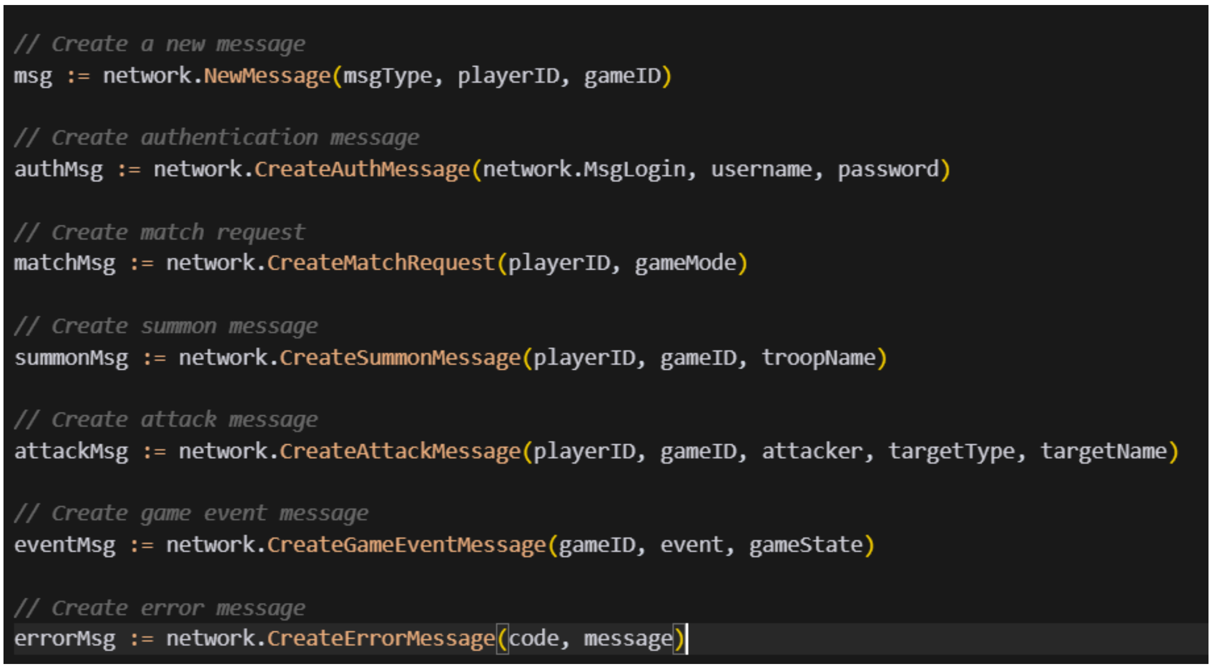
## 4.4 PDU Creation and Handling

Helper functions facilitate the creation of PDUs tailored to specific purposes, such as authentication or game actions.



```go
// NewMessage creates a new message with timestamp
func NewMessage(msgType MessageType, playerID, gameID string) *Message {
    return &Message{
        Type:      msgType,
        PlayerID:  playerID,
        GameID:    gameID,
        Timestamp: time.Now(),
        Data:      make(map[string]interface{}),
    }
}
```

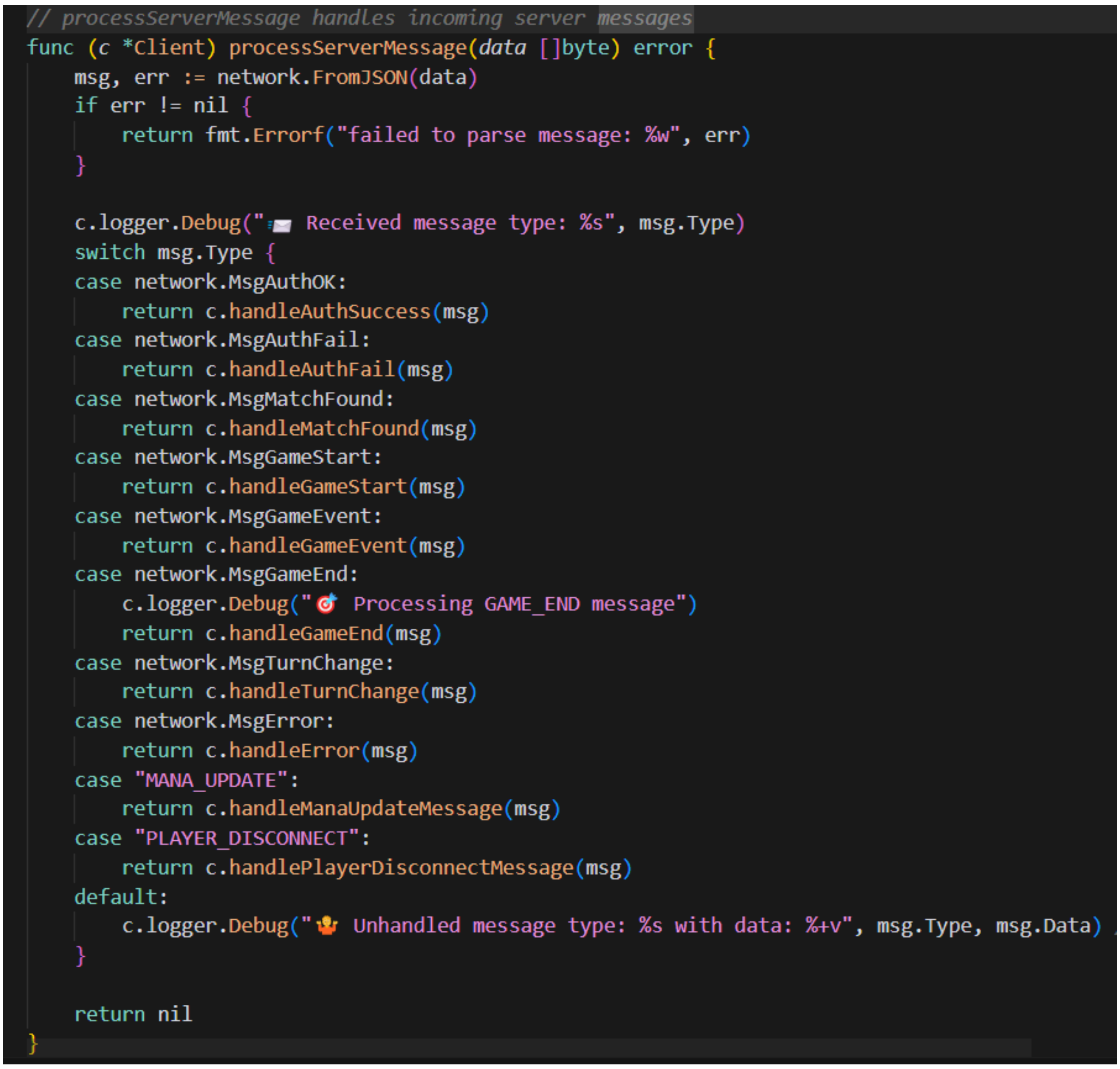
**Figure 4.12**: Helper function for creating a new message



```go
// Create a new message
msg := network.NewMessage(msgType, playerID, gameID)

// Create authentication message
authMsg := network.CreateAuthMessage(network.MsgLogin, username, password)

// Create match request
matchMsg := network.CreateMatchRequest(playerID, gameMode)

// Create summon message
summonMsg := network.CreateSummonMessage(playerID, gameID, troopName)

// Create attack message
attackMsg := network.CreateAttackMessage(playerID, gameID, attacker, targetType, targetName)

// Create game event message
eventMsg := network.CreateGameEventMessage(gameID, event, gameState)

// Create error message
errorMsg := network.CreateErrorMessage(code, message)
```

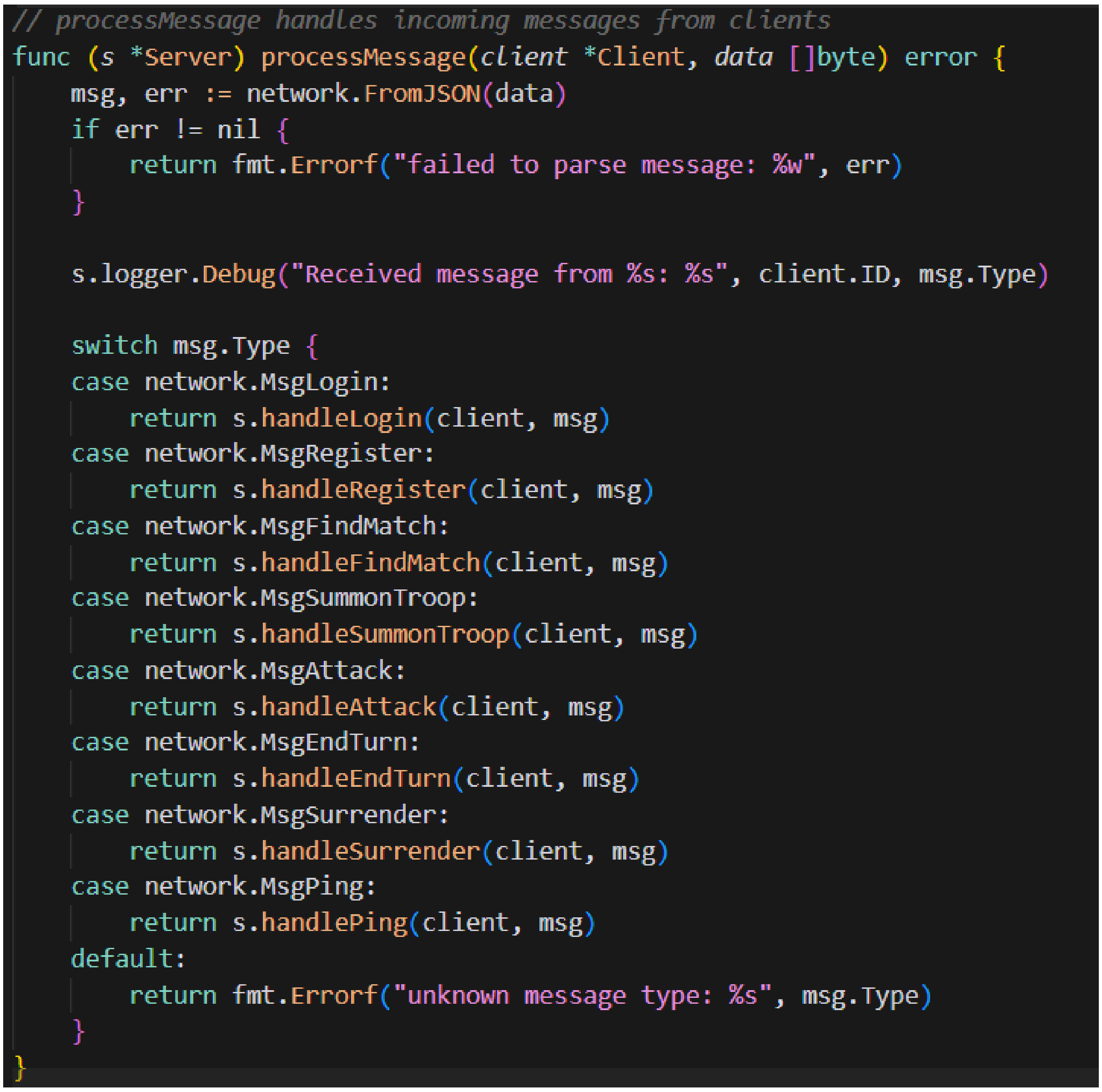
**Figure 4.13**: Examples of creating PDUs

## 4.5 Handling PDUs



```go
// processServerMessage handles incoming server messages
func (c *Client) processServerMessage(data []byte) error {
	msg, err := network.FromJSON(data)
	if err != nil {
		return fmt.Errorf("failed to parse message: %w", err)
	}

	c.logger.Debug("📨 Received message type: %s", msg.Type)
	switch msg.Type {
	case network.MsgAuthOK:
		return c.handleAuthSuccess(msg)
	case network.MsgAuthFail:
		return c.handleAuthFail(msg)
	case network.MsgMatchFound:
		return c.handleMatchFound(msg)
	case network.MsgGameStart:
		return c.handleGameStart(msg)
	case network.MsgGameEvent:
		return c.handleGameEvent(msg)
	case network.MsgGameEnd:
		c.logger.Debug("🎯 Processing GAME_END message")
		return c.handleGameEnd(msg)
	case network.MsgTurnChange:
		return c.handleTurnChange(msg)
	case network.MsgError:
		return c.handleError(msg)
	case "MANA_UPDATE":
		return c.handleManaUpdateMessage(msg)
	case "PLAYER_DISCONNECT":
		return c.handlePlayerDisconnectMessage(msg)
	default:
		c.logger.Debug("🤷 Unhandled message type: %s with data: %+v", msg.Type, msg.Data)
	}

	return nil
}
```

**Figure 4.14**: Function of CLIENT for handling incoming server messages

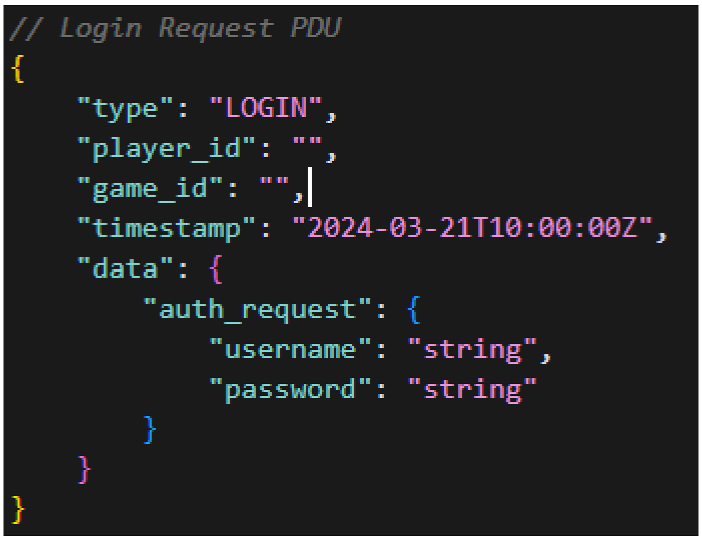


```go
// processMessage handles incoming messages from clients
func (s *Server) processMessage(client *Client, data []byte) error {
    msg, err := network.FromJSON(data)
    if err != nil {
        return fmt.Errorf("failed to parse message: %w", err)
    }

    s.logger.Debug("Received message from %s: %s", client.ID, msg.Type)

    switch msg.Type {
    case network.MsgLogin:
        return s.handleLogin(client, msg)
    case network.MsgRegister:
        return s.handleRegister(client, msg)
    case network.MsgFindMatch:
        return s.handleFindMatch(client, msg)
    case network.MsgSummonTroop:
        return s.handleSummonTroop(client, msg)
    case network.MsgAttack:
        return s.handleAttack(client, msg)
    case network.MsgEndTurn:
        return s.handleEndTurn(client, msg)
    case network.MsgSurrender:
        return s.handleSurrender(client, msg)
    case network.MsgPing:
        return s.handlePing(client, msg)
    default:
        return fmt.Errorf("unknown message type: %s", msg.Type)
    }
}
```

**Figure 4.15**: Function of SERVER for handling incoming client messages

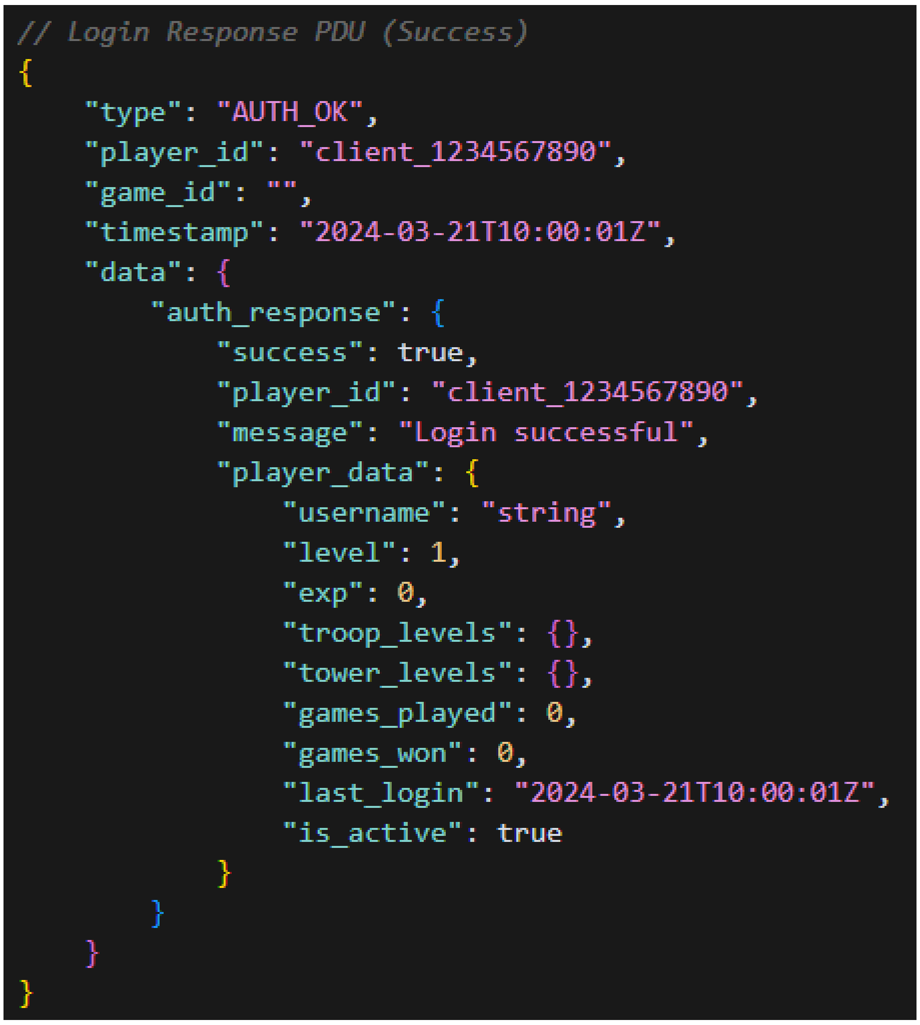
## 4.6 Working with PDUs

### 4.6.1 Authentication PDUs



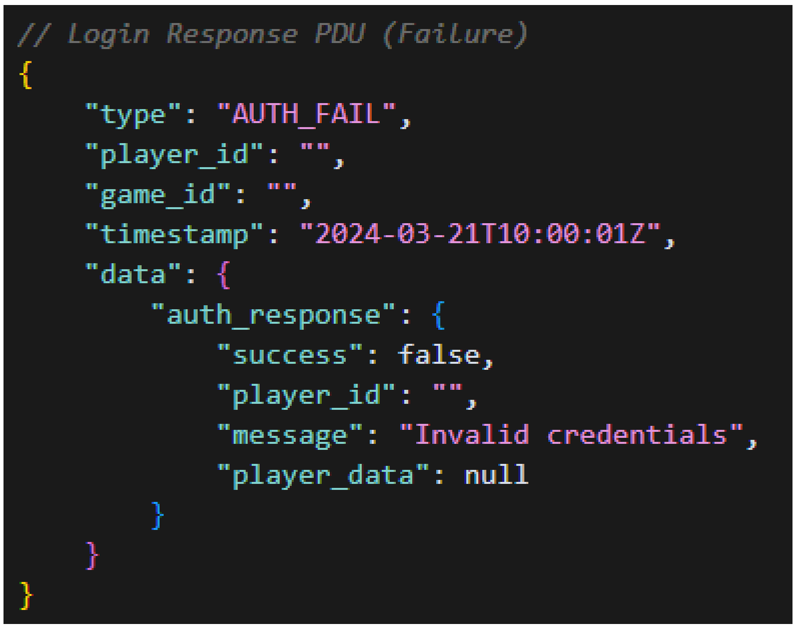
```
// Login Request PDU
{
    "type": "LOGIN",
    "player_id": "",
    "game_id": "",
    "timestamp": "2024-03-21T10:00:00Z",
    "data": {
        "auth_request": {
            "username": "string",
            "password": "string"
        }
    }
}
```

**Figure 4.16**: Login Request PDU



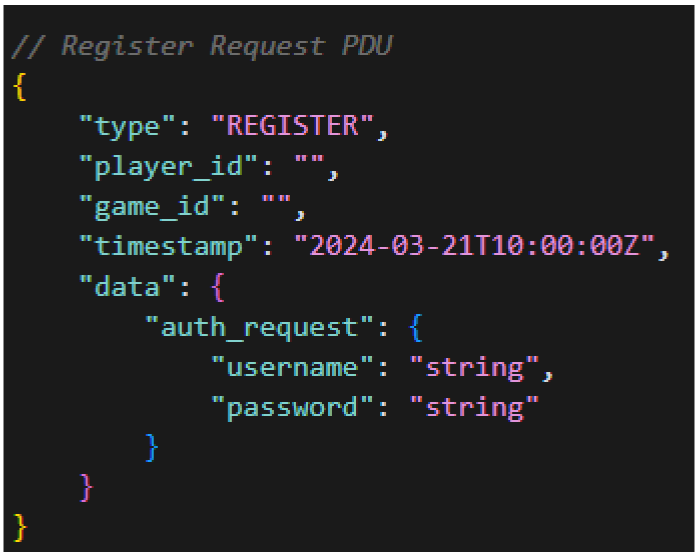
```
// Login Response PDU (Success)
{
    "type": "AUTH_OK",
    "player_id": "client_1234567890",
    "game_id": "",
    "timestamp": "2024-03-21T10:00:01Z",
    "data": {
        "auth_response": {
            "success": true,
            "player_id": "client_1234567890",
            "message": "Login successful",
            "player_data": {
                "username": "string",
                "level": 1,
                "exp": 0,
                "troop_levels": {},
                "tower_levels": {},
                "games_played": 0,
                "games_won": 0,
                "last_login": "2024-03-21T10:00:01Z",
                "is_active": true
            }
        }
    }
}
```

**Figure 4.17**: Login Response PDU (Success)



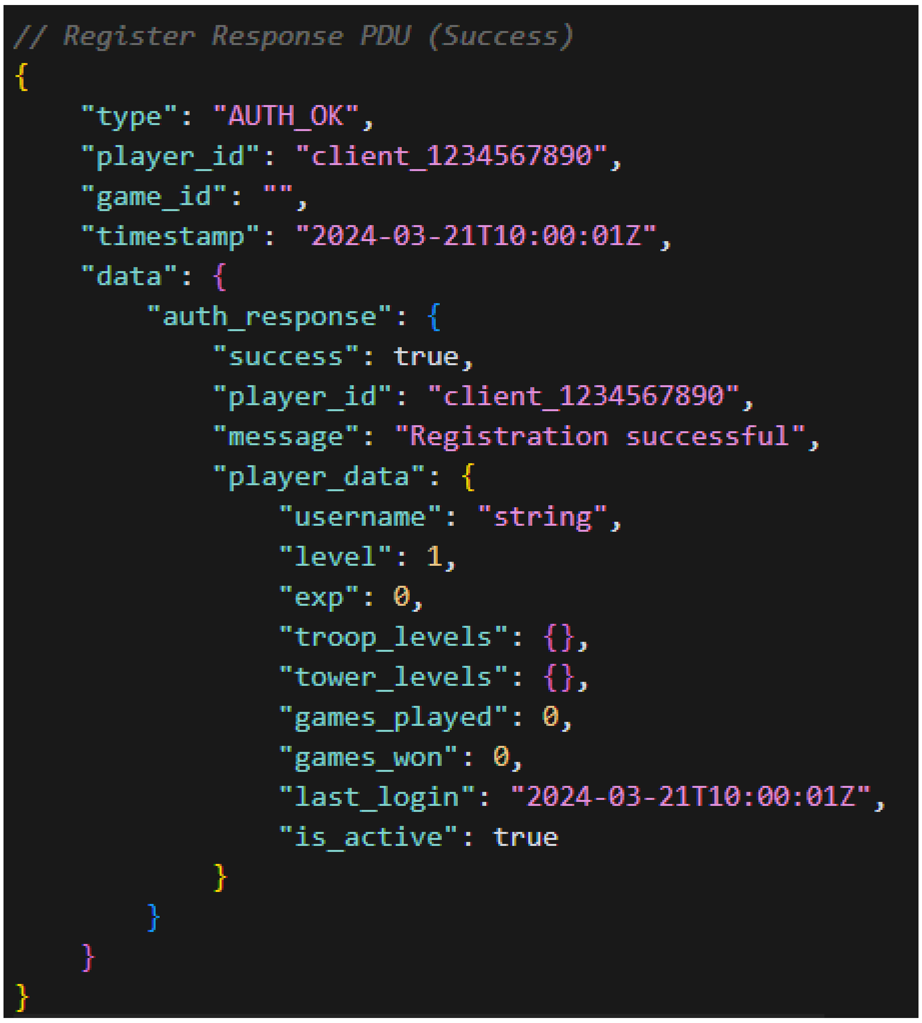
```
// Login Response PDU (Failure)
{
    "type": "AUTH_FAIL",
    "player_id": "",
    "game_id": "",
    "timestamp": "2024-03-21T10:00:01Z",
    "data": {
        "auth_response": {
            "success": false,
            "player_id": "",
            "message": "Invalid credentials",
            "player_data": null
        }
    }
}
```

**Figure 4.18**: Login Response PDU (Failure)



```
// Register Request PDU
{
    "type": "REGISTER",
    "player_id": "",
    "game_id": "",
    "timestamp": "2024-03-21T10:00:00Z",
    "data": {
        "auth_request": {
            "username": "string",
            "password": "string"
        }
    }
}
```

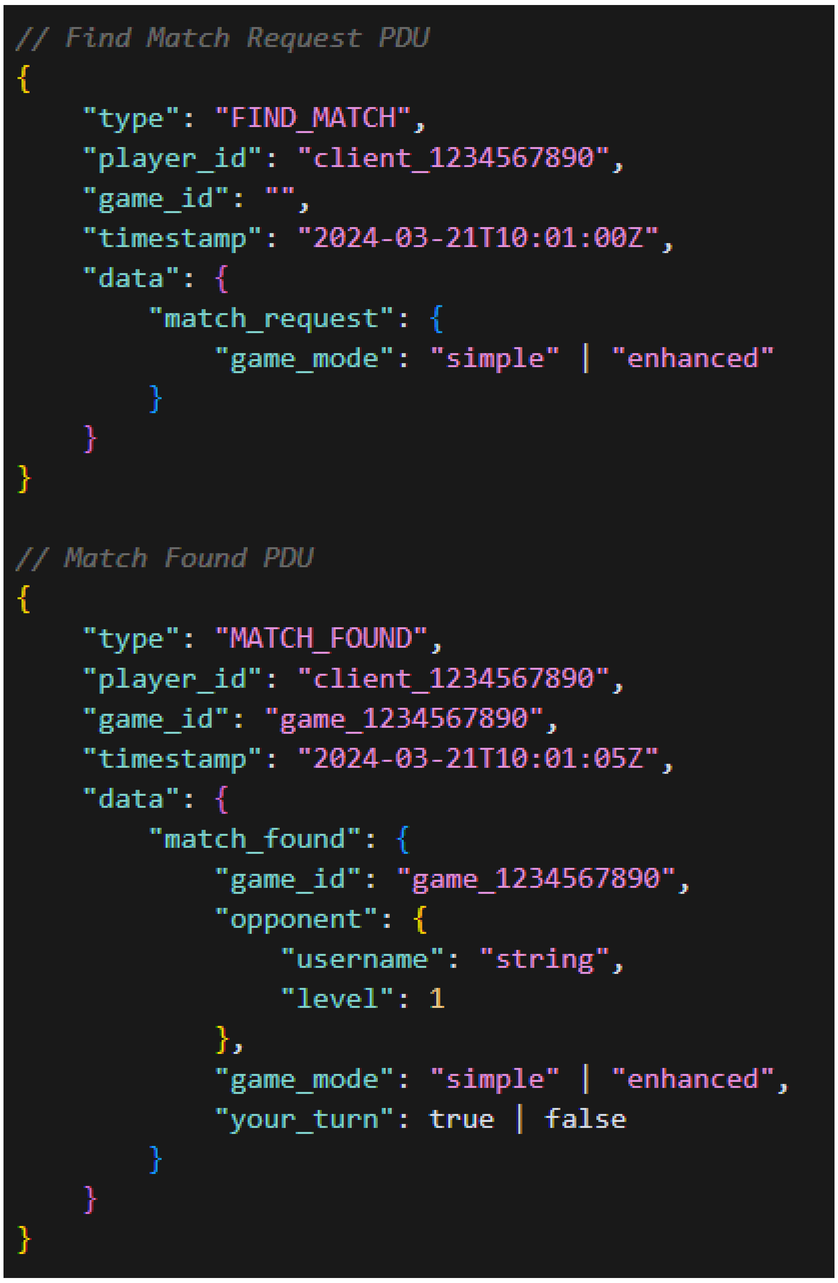
**Figure 4.19**: Register Request PDU



```
// Register Response PDU (Success)
{
    "type": "AUTH_OK",
    "player_id": "client_1234567890",
    "game_id": "",
    "timestamp": "2024-03-21T10:00:01Z",
    "data": {
        "auth_response": {
            "success": true,
            "player_id": "client_1234567890",
            "message": "Registration successful",
            "player_data": {
                "username": "string",
                "level": 1,
                "exp": 0,
                "troop_levels": {},
                "tower_levels": {},
                "games_played": 0,
                "games_won": 0,
                "last_login": "2024-03-21T10:00:01Z",
                "is_active": true
            }
        }
    }
}
```

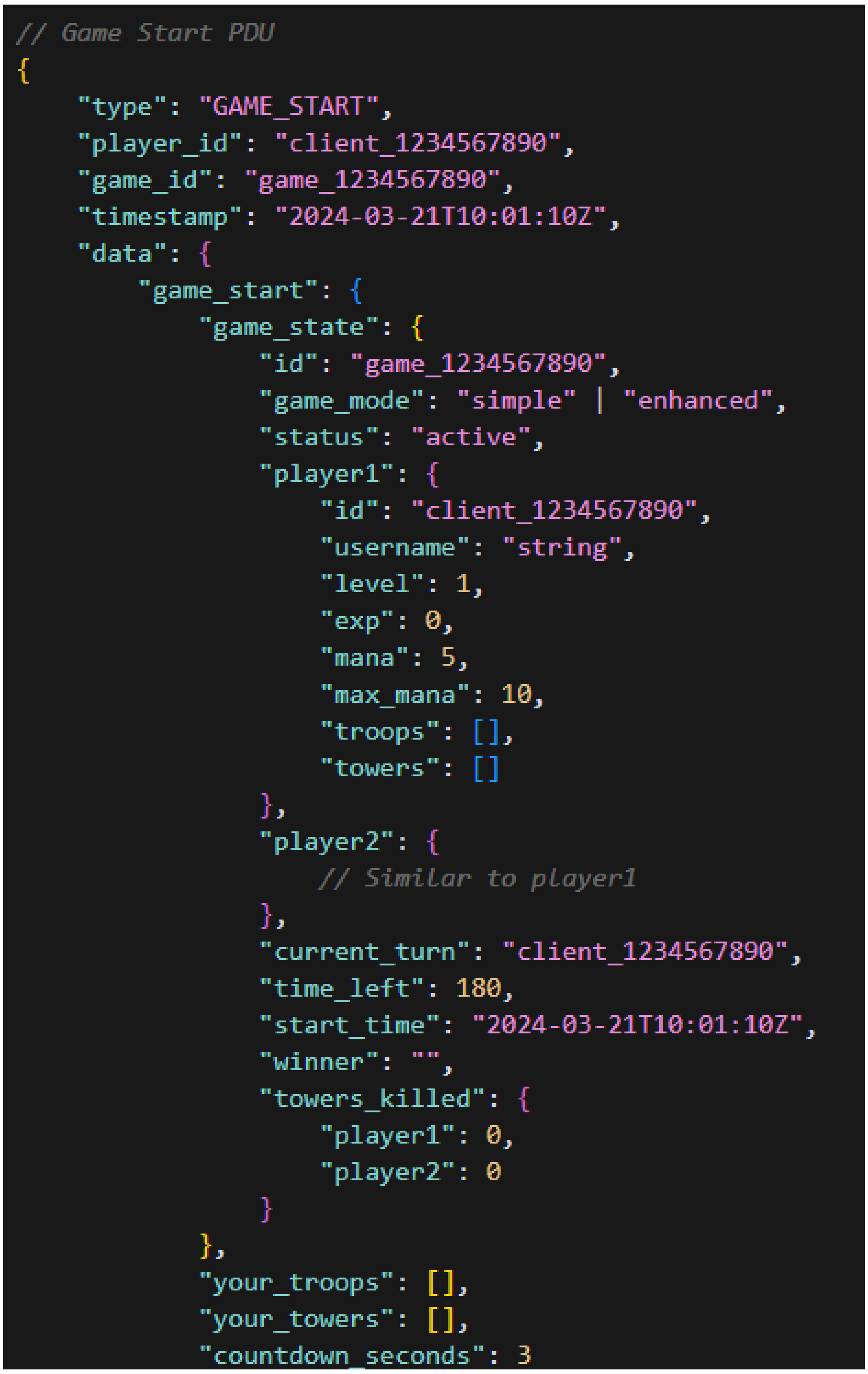
**Figure 4.20**: Register Response PDU

## 4.6.2 Matchmaking PDUs



```
// Find Match Request PDU
{
    "type": "FIND_MATCH",
    "player_id": "client_1234567890",
    "game_id": "",
    "timestamp": "2024-03-21T10:01:00Z",
    "data": {
        "match_request": {
            "game_mode": "simple" | "enhanced"
        }
    }
}

// Match Found PDU
{
    "type": "MATCH_FOUND",
    "player_id": "client_1234567890",
    "game_id": "game_1234567890",
    "timestamp": "2024-03-21T10:01:05Z",
    "data": {
        "match_found": {
            "game_id": "game_1234567890",
            "opponent": {
                "username": "string",
                "level": 1
            },
            "game_mode": "simple" | "enhanced",
            "your_turn": true | false
        }
    }
}
```

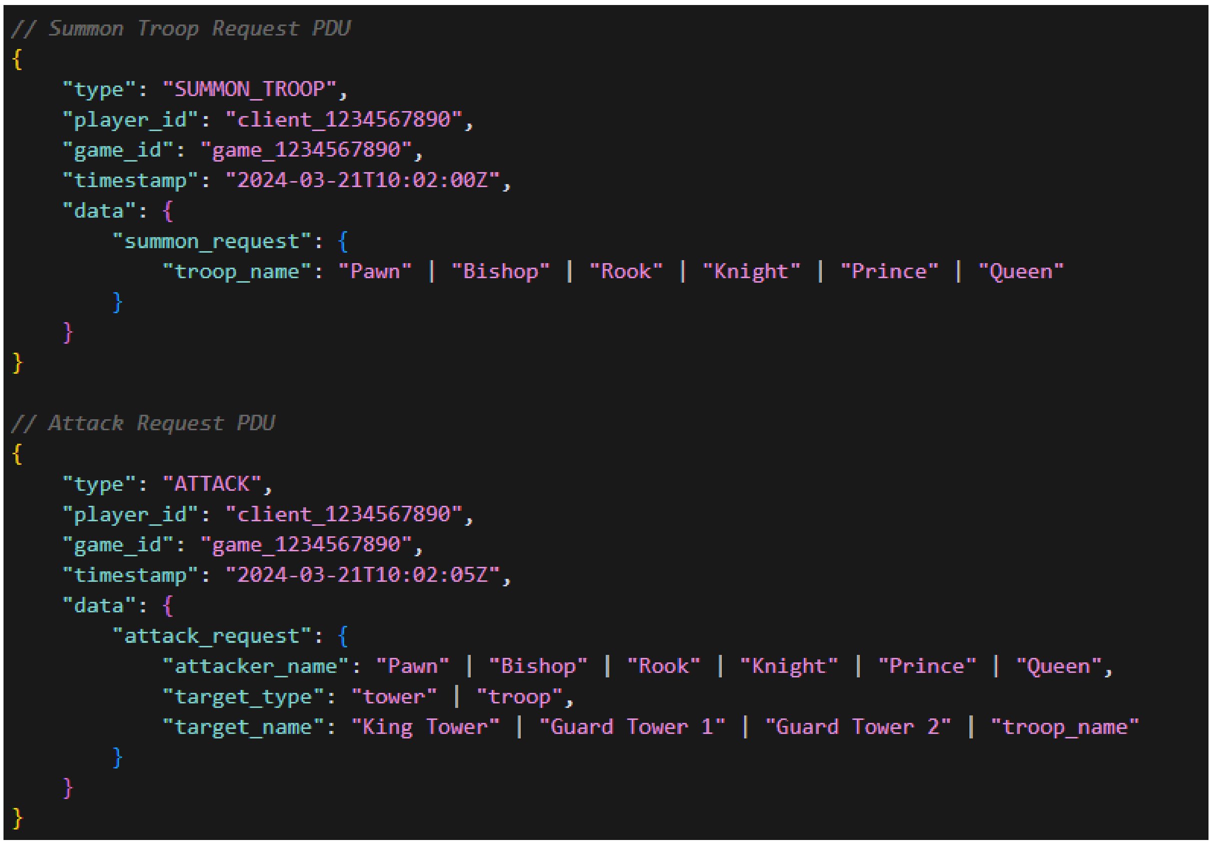
**Figure 4.21**: Find Match and Match Found PDUs



```
// Game Start PDU
{
    "type": "GAME_START",
    "player_id": "client_1234567890",
    "game_id": "game_1234567890",
    "timestamp": "2024-03-21T10:01:10Z",
    "data": {
        "game_start": {
            "game_state": {
                "id": "game_1234567890",
                "game_mode": "simple" | "enhanced",
                "status": "active",
                "player1": {
                    "id": "client_1234567890",
                    "username": "string",
                    "level": 1,
                    "exp": 0,
                    "mana": 5,
                    "max_mana": 10,
                    "troops": [],
                    "towers": []
                },
                "player2": {
                    // Similar to player1
                },
                "current_turn": "client_1234567890",
                "time_left": 180,
                "start_time": "2024-03-21T10:01:10Z",
                "winner": "",
                "towers_killed": {
                    "player1": 0,
                    "player2": 0
                }
            },
            "your_troops": [],
            "your_towers": [],
            "countdown_seconds": 3
```

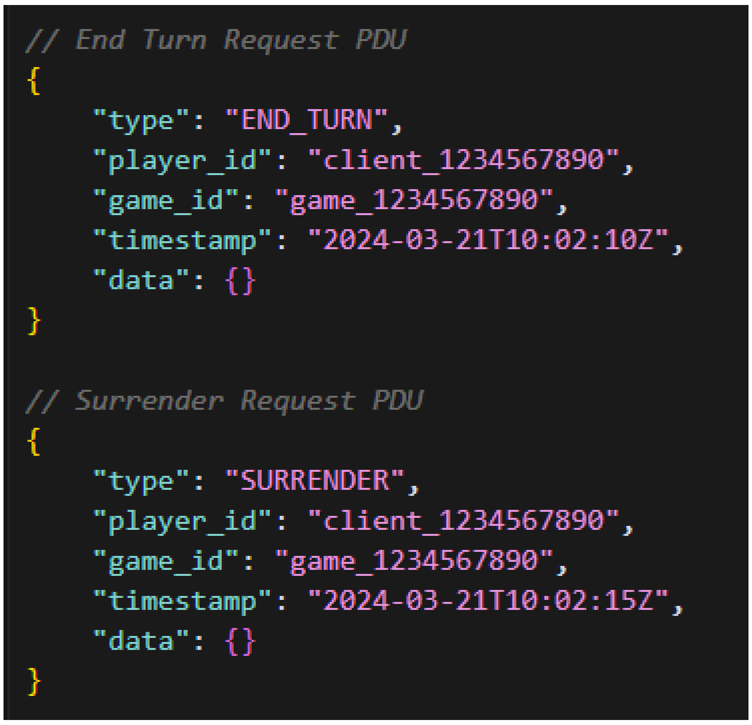
**Figure 4.22**: GameStart PDU

## 4.6.3 Game Action PDUs



```
// Summon Troop Request PDU
{
    "type": "SUMMON_TROOP",
    "player_id": "client_1234567890",
    "game_id": "game_1234567890",
    "timestamp": "2024-03-21T10:02:00Z",
    "data": {
        "summon_request": {
            "troop_name": "Pawn" | "Bishop" | "Rook" | "Knight" | "Prince" | "Queen"
        }
    }
}

// Attack Request PDU
{
    "type": "ATTACK",
    "player_id": "client_1234567890",
    "game_id": "game_1234567890",
    "timestamp": "2024-03-21T10:02:05Z",
    "data": {
        "attack_request": {
            "attacker_name": "Pawn" | "Bishop" | "Rook" | "Knight" | "Prince" | "Queen",
            "target_type": "tower" | "troop",
            "target_name": "King Tower" | "Guard Tower 1" | "Guard Tower 2" | "troop_name"
        }
    }
}
```

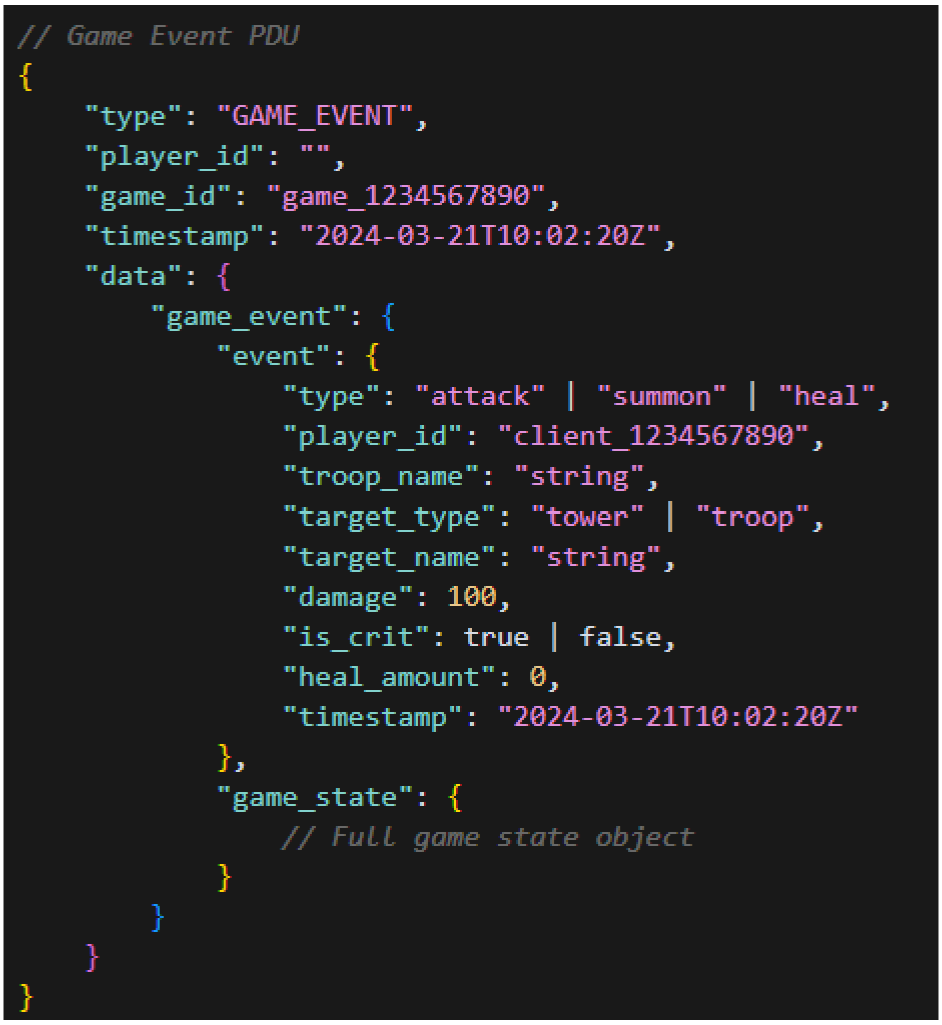
**Figure 4.23**: Summon and Attack Request PDUs



```
// End Turn Request PDU
{
    "type": "END_TURN",
    "player_id": "client_1234567890",
    "game_id": "game_1234567890",
    "timestamp": "2024-03-21T10:02:10Z",
    "data": {}
}

// Surrender Request PDU
{
    "type": "SURRENDER",
    "player_id": "client_1234567890",
    "game_id": "game_1234567890",
    "timestamp": "2024-03-21T10:02:15Z",
    "data": {}
}
```

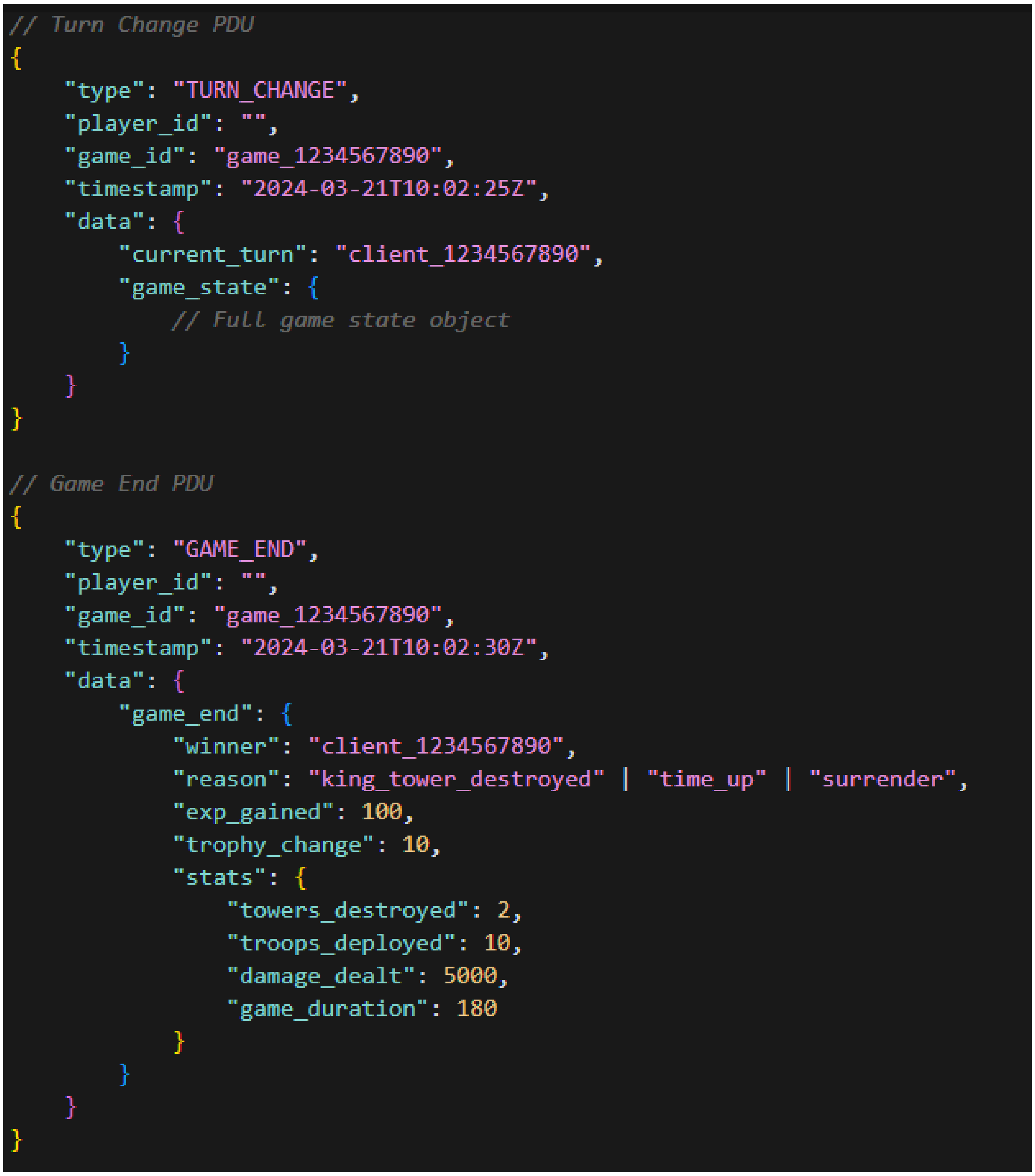
**Figure 4.24**: End Turn and Surrender Request PDUs

## 4.6.4 Game State PDUs



```
// Game Event PDU
{
    "type": "GAME_EVENT",
    "player_id": "",
    "game_id": "game_1234567890",
    "timestamp": "2024-03-21T10:02:20Z",
    "data": {
        "game_event": {
            "event": {
                "type": "attack" | "summon" | "heal",
                "player_id": "client_1234567890",
                "troop_name": "string",
                "target_type": "tower" | "troop",
                "target_name": "string",
                "damage": 100,
                "is_crit": true | false,
                "heal_amount": 0,
                "timestamp": "2024-03-21T10:02:20Z"
            },
            "game_state": {
                // Full game state object
            }
        }
    }
}
```

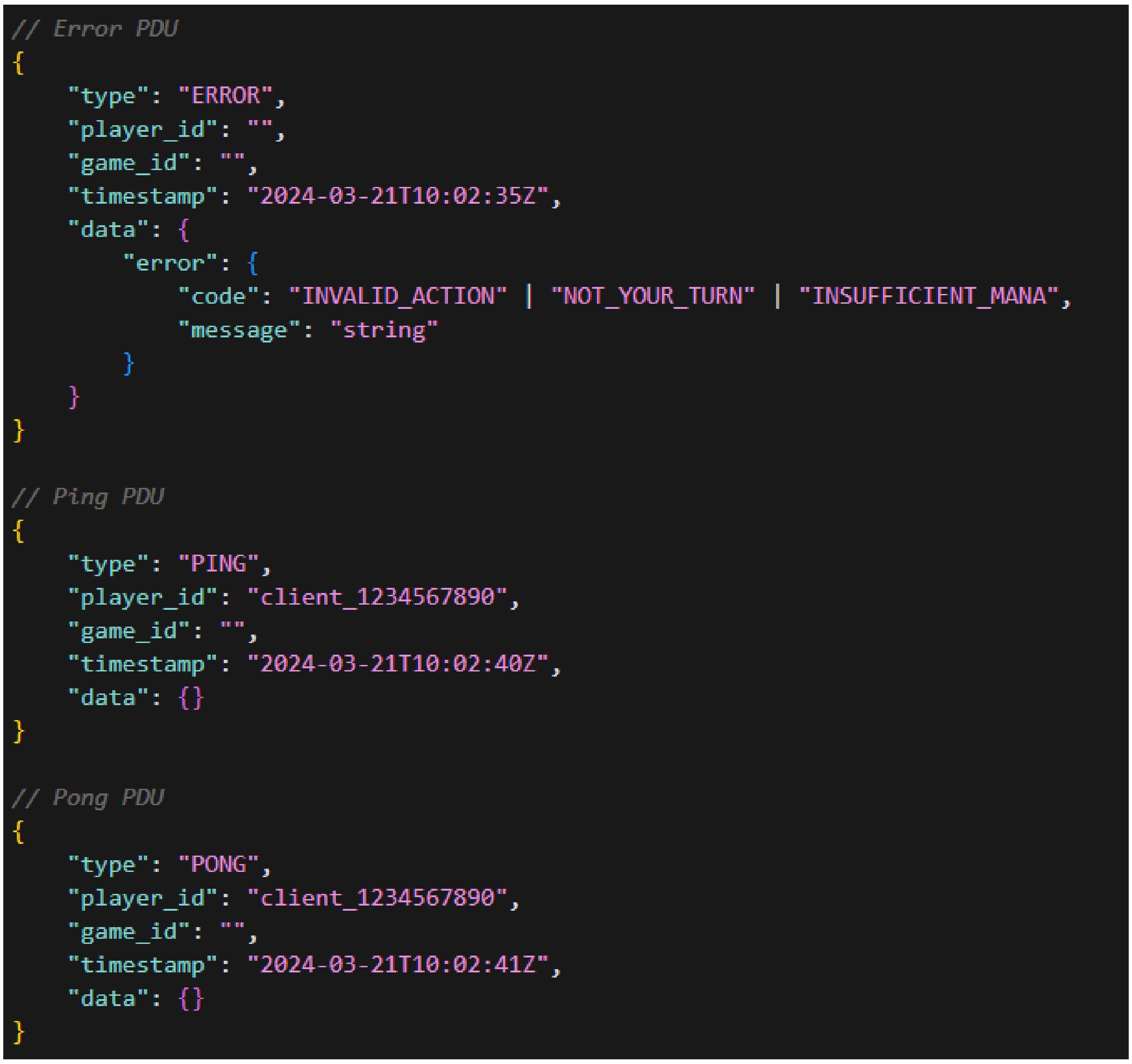
**Figure 4.25**: Game Event PDU



```
// Turn Change PDU
{
    "type": "TURN_CHANGE",
    "player_id": "",
    "game_id": "game_1234567890",
    "timestamp": "2024-03-21T10:02:25Z",
    "data": {
        "current_turn": "client_1234567890",
        "game_state": {
            // Full game state object
        }
    }
}

// Game End PDU
{
    "type": "GAME_END",
    "player_id": "",
    "game_id": "game_1234567890",
    "timestamp": "2024-03-21T10:02:30Z",
    "data": {
        "game_end": {
            "winner": "client_1234567890",
            "reason": "king_tower_destroyed" | "time_up" | "surrender",
            "exp_gained": 100,
            "trophy_change": 10,
            "stats": {
                "towers_destroyed": 2,
                "troops_deployed": 10,
                "damage_dealt": 5000,
                "game_duration": 180
            }
        }
    }
}
```

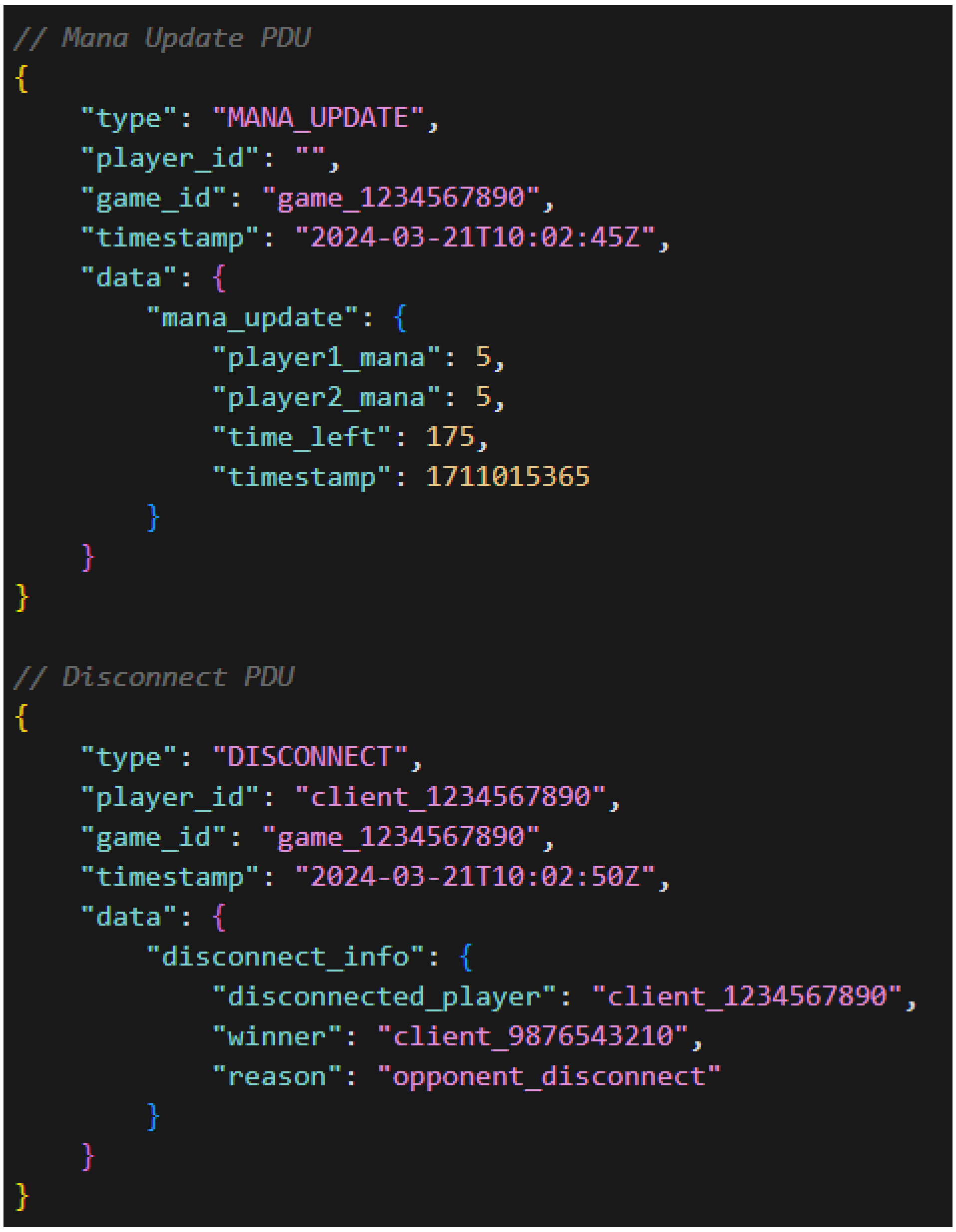
**Figure 4.26**: Turn Change and Game End PDUs

### 4.6.5 System PDUs



```
// Error PDU
{
    "type": "ERROR",
    "player_id": "",
    "game_id": "",
    "timestamp": "2024-03-21T10:02:35Z",
    "data": {
        "error": {
            "code": "INVALID_ACTION" | "NOT_YOUR_TURN" | "INSUFFICIENT_MANA",
            "message": "string"
        }
    }
}

// Ping PDU
{
    "type": "PING",
    "player_id": "client_1234567890",
    "game_id": "",
    "timestamp": "2024-03-21T10:02:40Z",
    "data": {}
}

// Pong PDU
{
    "type": "PONG",
    "player_id": "client_1234567890",
    "game_id": "",
    "timestamp": "2024-03-21T10:02:41Z",
    "data": {}
}
```

**Figure 4.27:** Error, Ping and Pong PDUs



```
// Mana Update PDU
{
    "type": "MANA_UPDATE",
    "player_id": "",
    "game_id": "game_1234567890",
    "timestamp": "2024-03-21T10:02:45Z",
    "data": {
        "mana_update": {
            "player1_mana": 5,
            "player2_mana": 5,
            "time_left": 175,
            "timestamp": 1711015365
        }
    }
}

// Disconnect PDU
{
    "type": "DISCONNECT",
    "player_id": "client_1234567890",
    "game_id": "game_1234567890",
    "timestamp": "2024-03-21T10:02:50Z",
    "data": {
        "disconnect_info": {
            "disconnected_player": "client_1234567890",
            "winner": "client_9876543210",
            "reason": "opponent_disconnect"
        }
    }
}
```

**Figure 4.28**: Mana Update, and Disconnect PDUs

Each Protocol Data Unit (PDU) in the Clash Royale TCR communication protocol follows a consistent and well-defined structure, comprising the following fields:

- **type**: Specifies the message type to identify its purpose.
- **player_id**: Identifies the sender of the message, when applicable.
- **game_id**: Identifies the associated game session, when applicable.
- **timestamp**: Records the creation time of the message.
- **data**: Contains the message-specific payload, varying by message type.

These PDUs are designed to be JSON-serializable, self-contained, and type-safe, facilitating straightforward parsing and validation. Their consistent format ensures seamless interoperability between client and server components, while their extensibility supports future enhancements and additional features. Together, this comprehensive PDU design underpins all necessary client-server communications for the Clash Royale TCR game.

# 5. Sequence Diagram

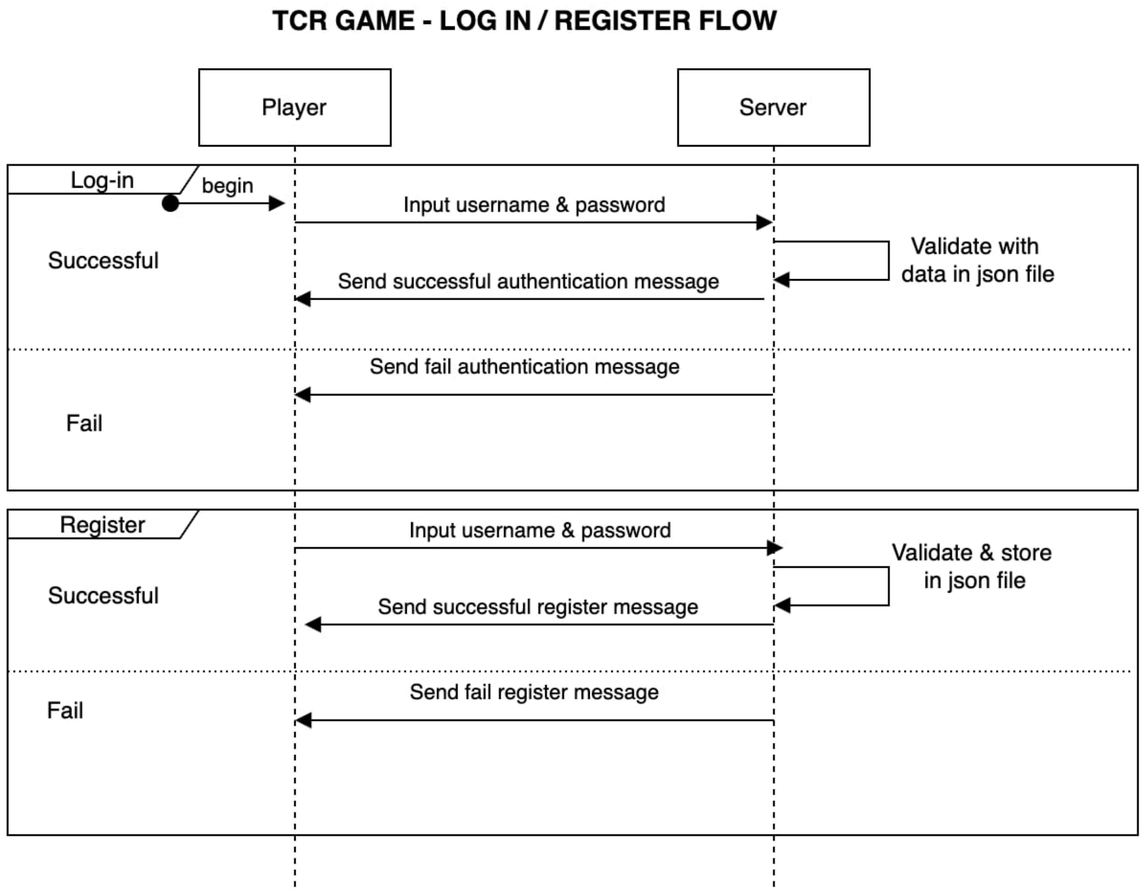

TCR GAME - LOG IN / REGISTER FLOW
Player
Server
Log-in
begin
Input username & password
Successful
Validate with data in json file
Send successful authentication message
Send fail authentication message
Fail
Register
Input username & password
Validate & store in json file
Successful
Send successful register message
Send fail register message
Fail

**Figure 5.1**: Authentication Flow

The sequence diagram illustrates the interaction flow between the Player and the Server during the login and registration processes for the TCR game. In the login process, the Player initiates the flow by submitting a username and password. This information is transmitted to the Server, which then validates the credentials against stored data maintained in a JSON file. Upon successful validation, the Server returns a confirmation message indicating successful authentication. Conversely, if the credentials are invalid, the Server responds with a failure message signaling unsuccessful login.

In the registration process, the Player provides a new username and password to create an account. This information is sent to the Server, which validates the data and subsequently stores it within the JSON file. If the registration is successful, the Server communicates a positive registration confirmation to the Player. Should the registration fail—due to reasons such as duplicate usernames or invalid input—the Server returns a failure notification.

Overall, this diagram presents a straightforward client-server communication model, where the Server is responsible for data validation and persistence, and the Player awaits feedback on the success or failure of their login or registration attempts. It is worth noting, from a security perspective, that storing sensitive authentication data such as passwords in a JSON file is not advisable for production environments. Best practices would involve using secure databases with proper encryption or hashing techniques to safeguard user credentials. Nonetheless, the depicted flow serves as a basic framework suitable for demonstration or prototyping purposes.

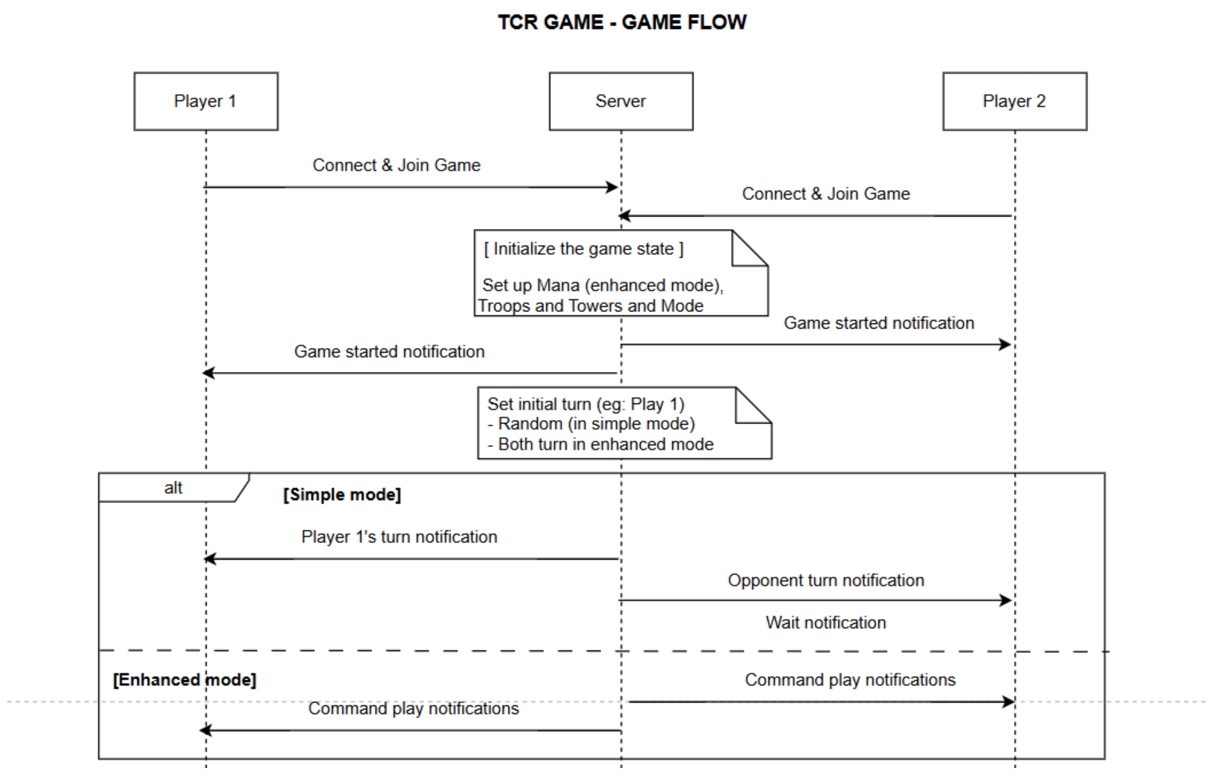

TCR GAME - GAME FLOW
Player 1
Server
Player 2
Connect & Join Game
Connect & Join Game
[ Initialize the game state ]
Set up Mana (enhanced mode), Troops and Towers and Mode
Game started notification
Game started notification
Set initial turn (eg: Play 1)
- Random (in simple mode)
- Both turn in enhanced mode
alt
[Simple mode]
Player 1's turn notification
Opponent turn notification
Wait notification
[Enhanced mode]
Command play notifications
Command play notifications

**Figure 5.2**: Game Start Flow Section

The sequence diagram presents a detailed representation of the interaction between three entities—**Player 1**, **Server**, and **Player 2**—during the initialization and progression phases of a multiplayer game. This diagram serves to clarify the communication protocols and game-state transitions that occur from the point of connection to the beginning of actual gameplay, accommodating two distinct game modes: **Simple** and **Enhanced**.

Initially, both Player 1 and Player 2 independently connect to the server and request to join the game. Once both players are connected, the server initiates the game state setup. This includes establishing the appropriate mode of play (either Simple or Enhanced), setting up Mana (specifically for Enhanced mode), and initializing troops and towers accordingly. Following this initialization, the server dispatches a "Game started notification" to both players, signifying that the game setup is complete and the match is ready to commence.

Subsequently, the server determines and sets the initial turn conditions. In Simple mode, the initial turn is assigned randomly, typically beginning with one of the players (e.g., Player 1).

In contrast, Enhanced mode allows both players to take their turns concurrently. This distinction in turn-taking rules is crucial to the structural flow of gameplay, as it influences how players interact with the game system and with each other.

The diagram utilizes an "alt" fragment to depict the divergence in behavior between Simple and Enhanced modes. In **Simple mode**, the server sends a notification to Player 1 indicating it is their turn, while Player 2 receives a waiting notification. Upon the completion of Player 1's action, the server will reverse these roles. This turn-based interaction ensures a clear and orderly progression of actions. On the other hand, in **Enhanced mode**, both players receive command play notifications simultaneously, allowing them to issue commands at the same time. This parallel processing model emphasizes a more dynamic and fast-paced gaming experience.

Overall, the sequence diagram provides a structured and comprehensive overview of the game flow, clearly distinguishing between the procedural differences inherent in the two game modes. It highlights the server's central role in managing state transitions and coordinating communication between players, ensuring synchronization and fairness in gameplay.

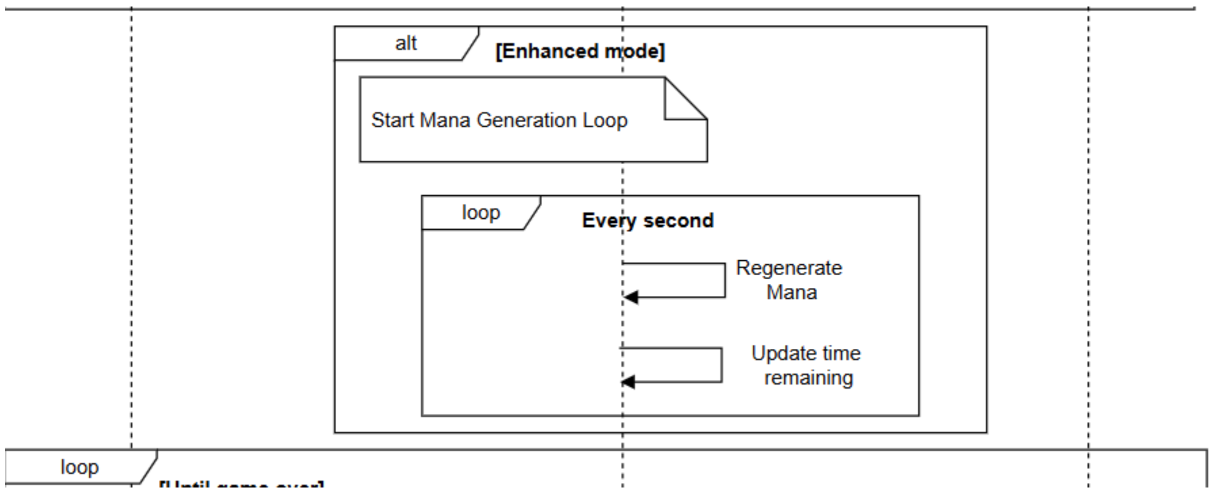


**Figure 5.3**: Enhanced Mode Mana Generation Section

This section of the sequence diagram extends the **"TCR Game - Game Flow"** and focuses specifically on the **Enhanced mode** gameplay mechanics. It describes the initiation and continuous operation of the **Mana Generation Loop**, a key system responsible for resource management and time tracking in real-time gameplay.

Under the **Enhanced mode**, an alternative flow (alt) is activated to initiate the **Mana Generation Loop**. This loop is unique to Enhanced mode and is absent in Simple mode, emphasizing the mode's more complex and dynamic nature. Once initiated, this loop executes a series of operations at fixed time intervals, specifically **every second**.

Inside the loop block, two core actions are executed repeatedly:

1. **Regenerate Mana** – This action incrementally restores a player's mana pool at each interval, allowing for a progressive and balanced use of abilities or unit deployment over time. The regeneration process is essential for enabling continuous interaction and decision-making during gameplay.

2. **Update Time Remaining** – Concurrently, the system updates the remaining time for a given game session or round. This provides real-time feedback to the players and is essential for pacing the match, ensuring that it progresses toward a defined conclusion.

Together, these operations reflect a **time-dependent resource management** system, a hallmark of Enhanced gameplay. The loop continues executing these actions until the game reaches its termination condition, which is defined in another part of the diagram not shown here but likely governed by a separate **loop [Until game over].**

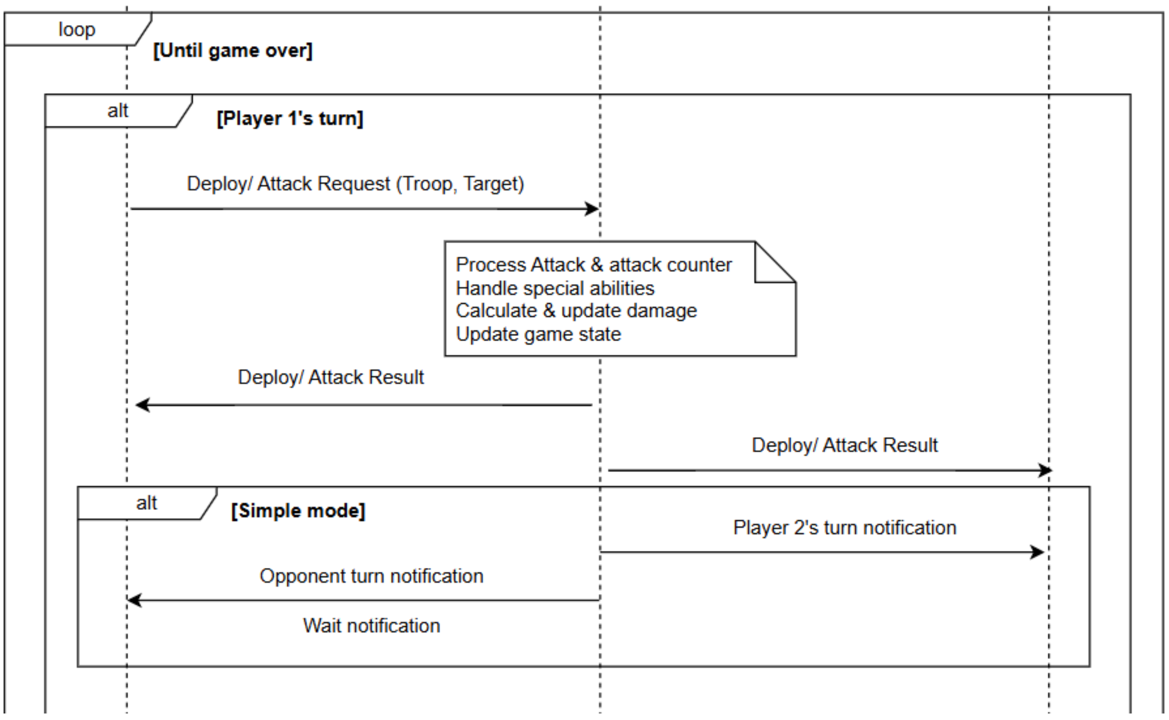


**Figure 5.4:** Player 1's Turn Section

This segment of the sequence diagram represents the **core turn-based gameplay loop** in the **TCR Game**, focusing on player interactions and game state updates until the game concludes.

The entire process operates within a loop labeled **"Until game over"**, indicating that the sequence repeats continuously until a win or loss condition is met. Inside this loop, the flow is divided into alternative fragments depending on the active player's turn.

During **Player 1's turn**, the diagram shows that Player 1 initiates a **Deploy/Attack Request**, which includes details like the chosen troop and its target. This request is sent to the **Server**, which acts as the central authority responsible for game logic. Once the request is received, the server processes the attack, handles any possible counter-attacks, manages any special abilities that might affect the outcome, calculates and updates damage, and finally updates the entire game state to reflect the result of the action.

Once the server completes these tasks, it sends a **Deploy/Attack Result** back to both players, informing them of the updated status of the game after the action.

Next, the diagram enters an alternate flow labeled **"Simple mode"**. In this mode, players take turns one at a time with no overlapping actions. After Player 1 completes their turn, a **"Player 2's turn notification"** is sent to Player 2, while Player 1 receives a **"Wait notification"**. This ensures that each player knows when it's their turn to act and when they need to wait, maintaining clear turn transitions and synchronous gameplay.

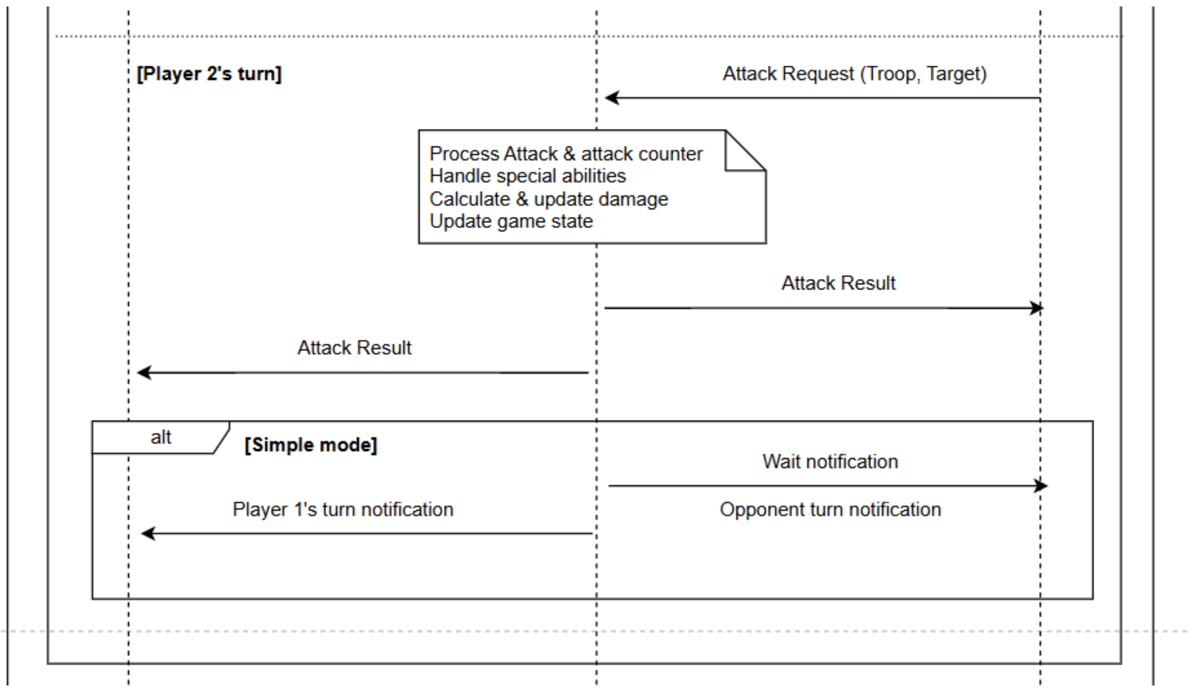

[Player 2's turn]
Attack Request (Troop, Target)
Process Attack & attack counter
Handle special abilities
Calculate & update damage
Update game state
Attack Result
Attack Result
alt
[Simple mode]
Wait notification
Player 1's turn notification
Opponent turn notification

**Figure 5.5**: Player 2's Turn Section

This sequence diagram describes the flow of **Player 2's turn** in a turn-based game system, continuing the cycle initiated by Player 1 in the earlier diagram.

At the beginning of Player 2's turn, they send an **Attack Request**, specifying the troop and its target. This request is directed to the **Server**, which is responsible for executing the game logic. Upon receiving the request, the server processes the attack, checks for any counter-attacks, applies any special abilities involved, calculates the damage inflicted, and updates the game state to reflect the outcome of these actions.

Once this processing is complete, the **Attack Result** is returned from the server to both players. This message contains the results of the attack, including any changes in troop health, special effects applied, or status changes in the game.

After the attack result has been delivered, the flow transitions into a **Simple Mode** segment. In this mode, turn-based control is passed from Player 2 to Player 1. The server sends a **Wait notification** to Player 2, indicating that it is no longer their turn. At the same time, a **Player 1's turn notification** is sent, informing them that they may now take their turn.

This flow continues cyclically between the two players, with each player's turn involving similar request-response patterns and alternating notifications to maintain orderly, turn-based gameplay.

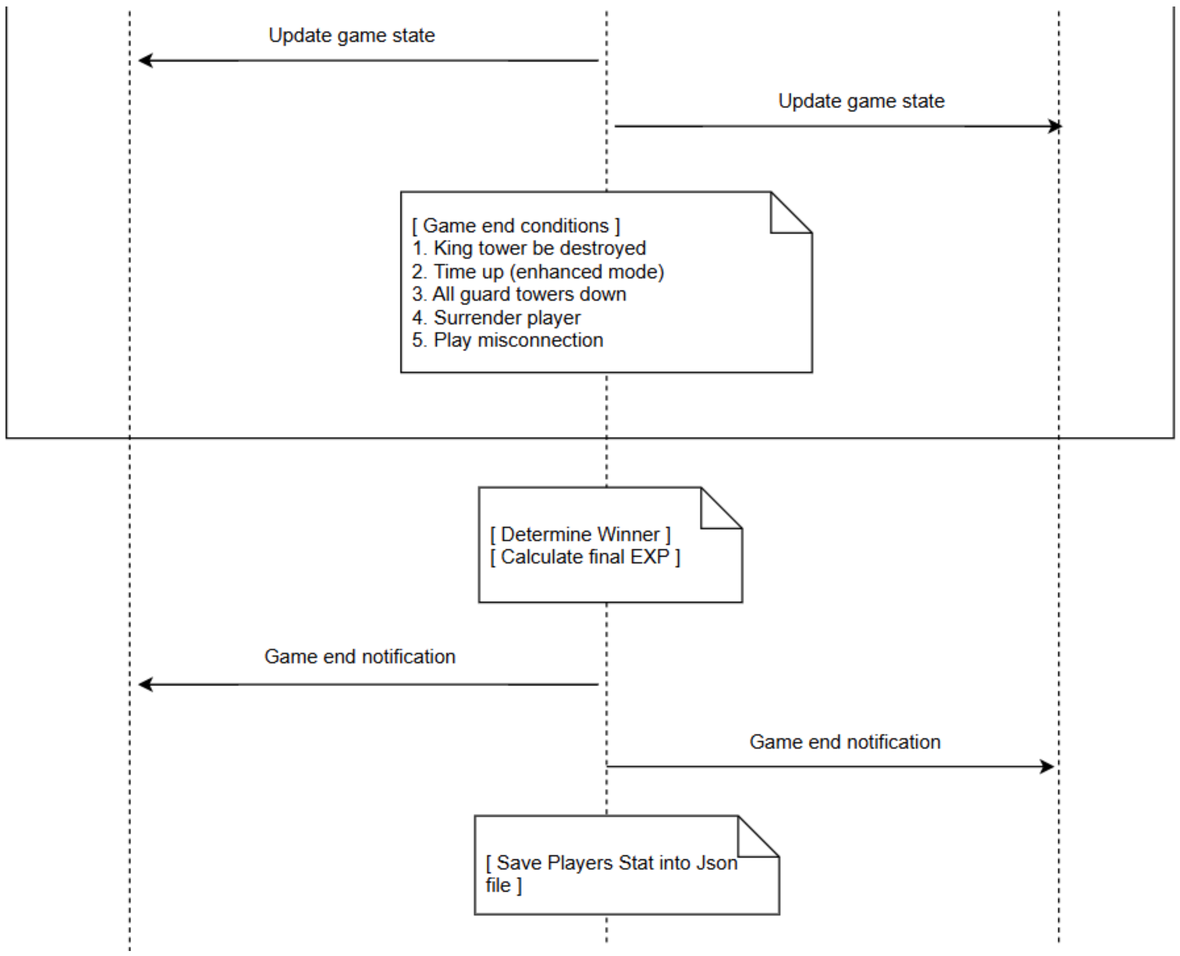


**Figure 5.6**: Game End Flow Section

This sequence diagram outlines the process that occurs at the end of a multiplayer game. The flow begins with both clients and the server updating the game state, which happens routinely during gameplay. At this point, the system evaluates whether any of the predefined game-ending conditions have been triggered.

There are five specific conditions that can cause the game to end. First, the destruction of the king tower—typically a central and most protected structure—automatically results in a game over. Second, the game can end due to a time limit being reached, particularly in enhanced mode where rounds are time-bound. Third, the match can conclude if all guard towers are taken down. Fourth, if a player voluntarily chooses to surrender, the game ends early. Lastly, the game will terminate if a player disconnects or experiences a connection issue during play.

Once one of these conditions is met, the game logic proceeds to determine the winner based on the final state of the match. Simultaneously, the server calculates the final amount of experience points (EXP) for each player. This step likely considers multiple factors, including performance, objectives completed, and whether the player won or lost the match.

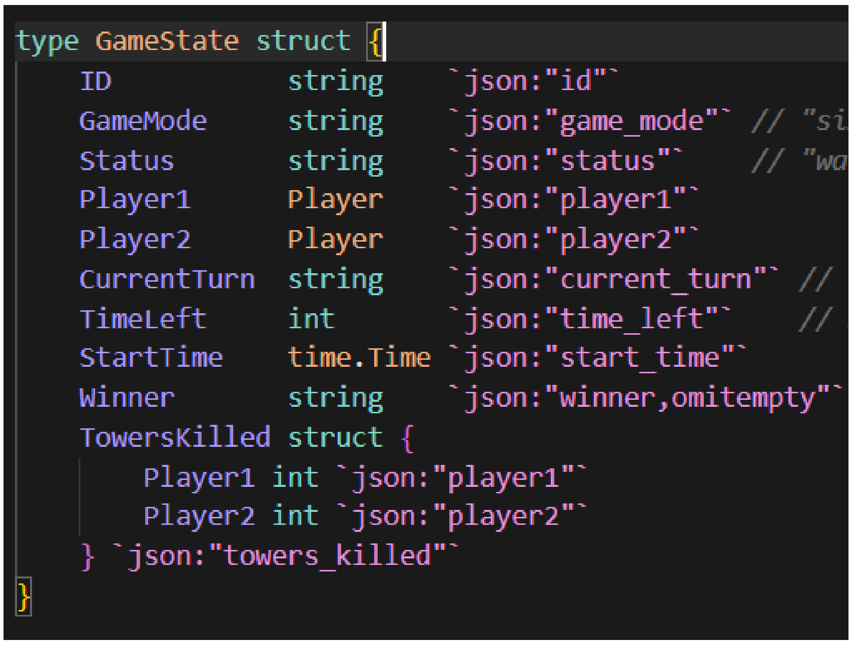
After determining the winner and calculating the EXP, the system sends a game end notification to both players. This ensures that all clients are informed of the match's conclusion. Finally, the system saves each player's statistics into a JSON file. This data could be used later for player profiles, match history, or analytics to improve the gaming experience. This structured sequence guarantees the proper closure of each game session and preserves important data for future reference.

# 6. Game Mechanics

## 6.1 Simple TCR Rules

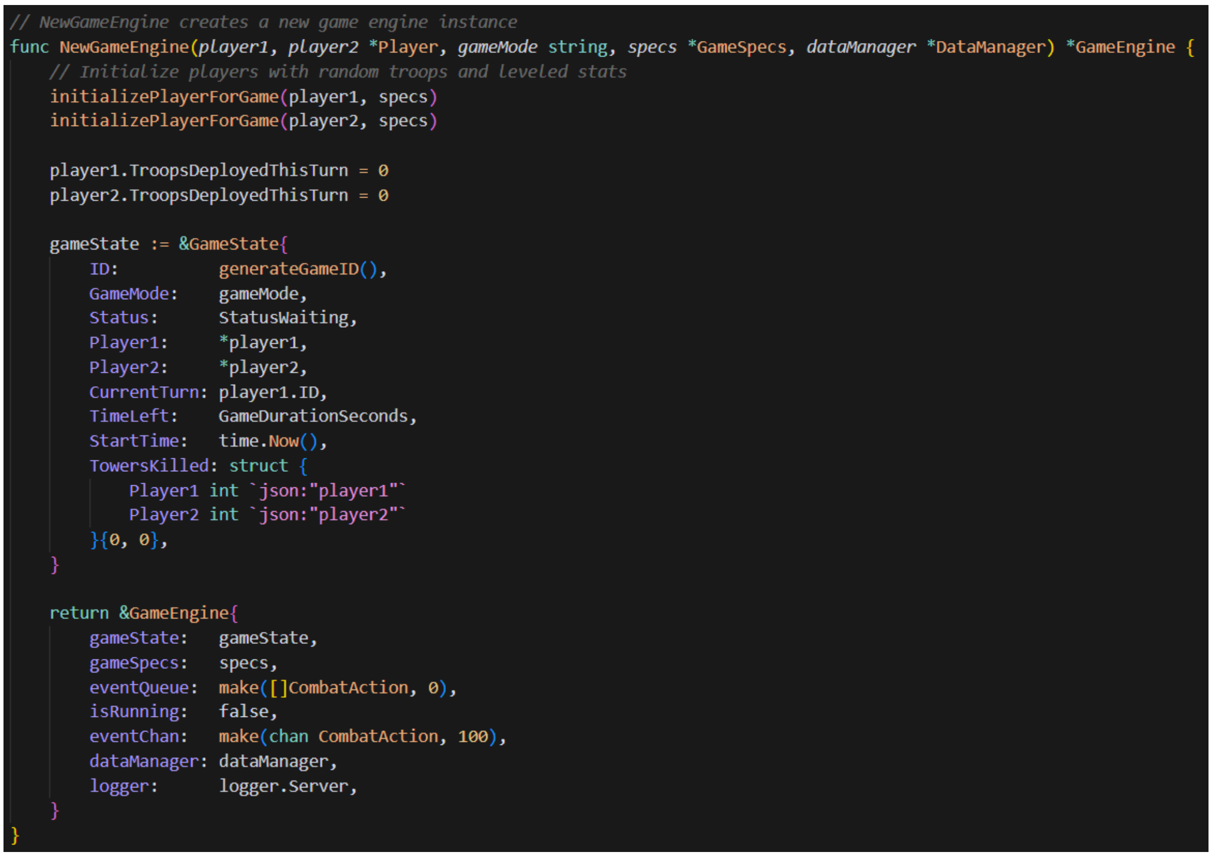
### 6.1.1 Game Initialization

The Simple TCR mode begins with two players, each starting the game with three towers: one King Tower possessing 2000 HP and two Guard Towers with 1000 HP each. The game engine initializes the game state using the **GameState** structure, which tracks the current turn, player details, and the status of all towers. The player who takes the first turn is selected randomly, and this information is stored in the **CurrentTurn** field. The game strictly follows a turn-based format, requiring players to wait for their opponent to finish before taking any actions.



```
type GameState struct {
    ID           string    `json:"id"`
    GameMode     string    `json:"game_mode"` // "si
    Status       string    `json:"status"`    // "wa
    Player1      Player    `json:"player1"`
    Player2      Player    `json:"player2"`
    CurrentTurn  string    `json:"current_turn"` //
    TimeLeft     int       `json:"time_left"`    //
    StartTime    time.Time `json:"start_time"`
    Winner       string    `json:"winner,omitempty"`
    TowersKilled struct {
        Player1 int `json:"player1"`
        Player2 int `json:"player2"`
    } `json:"towers_killed"`
}
```

**Figure 6.1**: Game State Structure

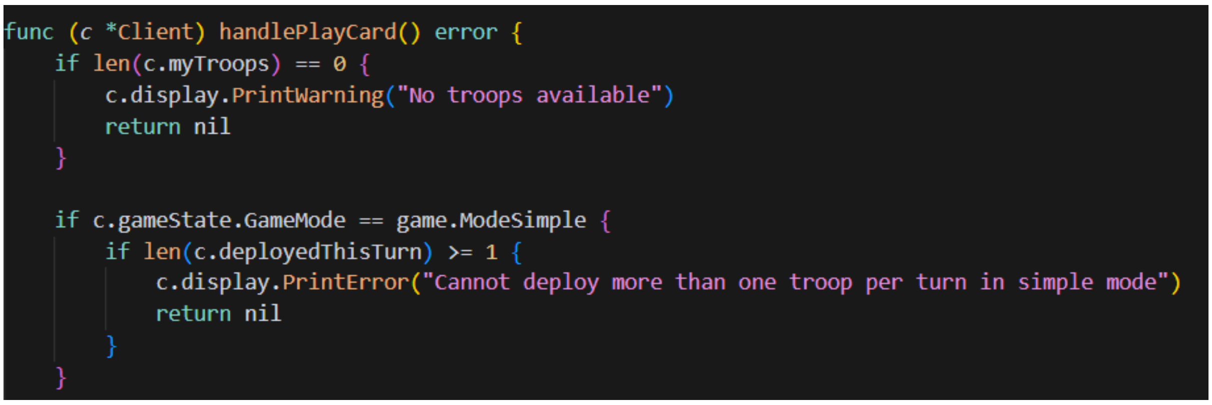


```go
// NewGameEngine creates a new game engine instance
func NewGameEngine(player1, player2 *Player, gameMode string, specs *GameSpecs, dataManager *DataManager) *GameEngine {
    // Initialize players with random troops and leveled stats
    initializePlayerForGame(player1, specs)
    initializePlayerForGame(player2, specs)

    player1.TroopsDeployedThisTurn = 0
    player2.TroopsDeployedThisTurn = 0

    gameState := &GameState{
        ID:          generateGameID(),
        GameMode:    gameMode,
        Status:      StatusWaiting,
        Player1:     *player1,
        Player2:     *player2,
        CurrentTurn: player1.ID,
        TimeLeft:    GameDurationSeconds,
        StartTime:   time.Now(),
        TowersKilled: struct {
            Player1 int `json:"player1"`
            Player2 int `json:"player2"`
        }{0, 0},
    }

    return &GameEngine{
        gameState:   gameState,
        gameSpecs:   specs,
        eventQueue:  make([]CombatAction, 0),
        isRunning:   false,
        eventChan:   make(chan CombatAction, 100),
        dataManager: dataManager,
        logger:      logger.Server,
    }
}
```

**Figure 6.2**: Tower Generation

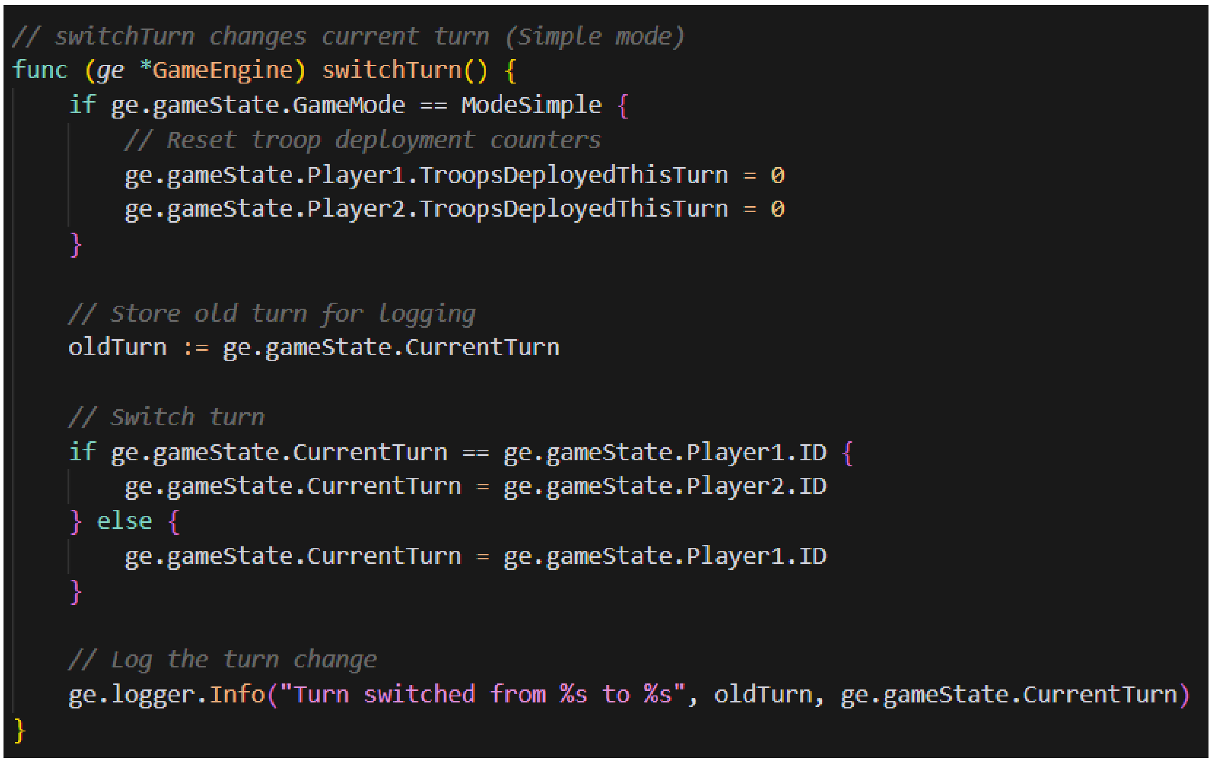
### 6.1.2 Turn-Based Structure

During a player's turn, they may perform several actions in a specific sequence. First, the player can deploy one troop per turn, chosen from their available set of three troops. This limitation is enforced by the **TroopsDeployedThisTurn** counter within the Player structure, which resets at the start of every turn. The **handlePlayCard** function ensures that only one troop deployment per turn is permitted by checking this counter. After deploying troops, players may attack enemy towers with their deployed troops, and finally, they can choose to end their turn.



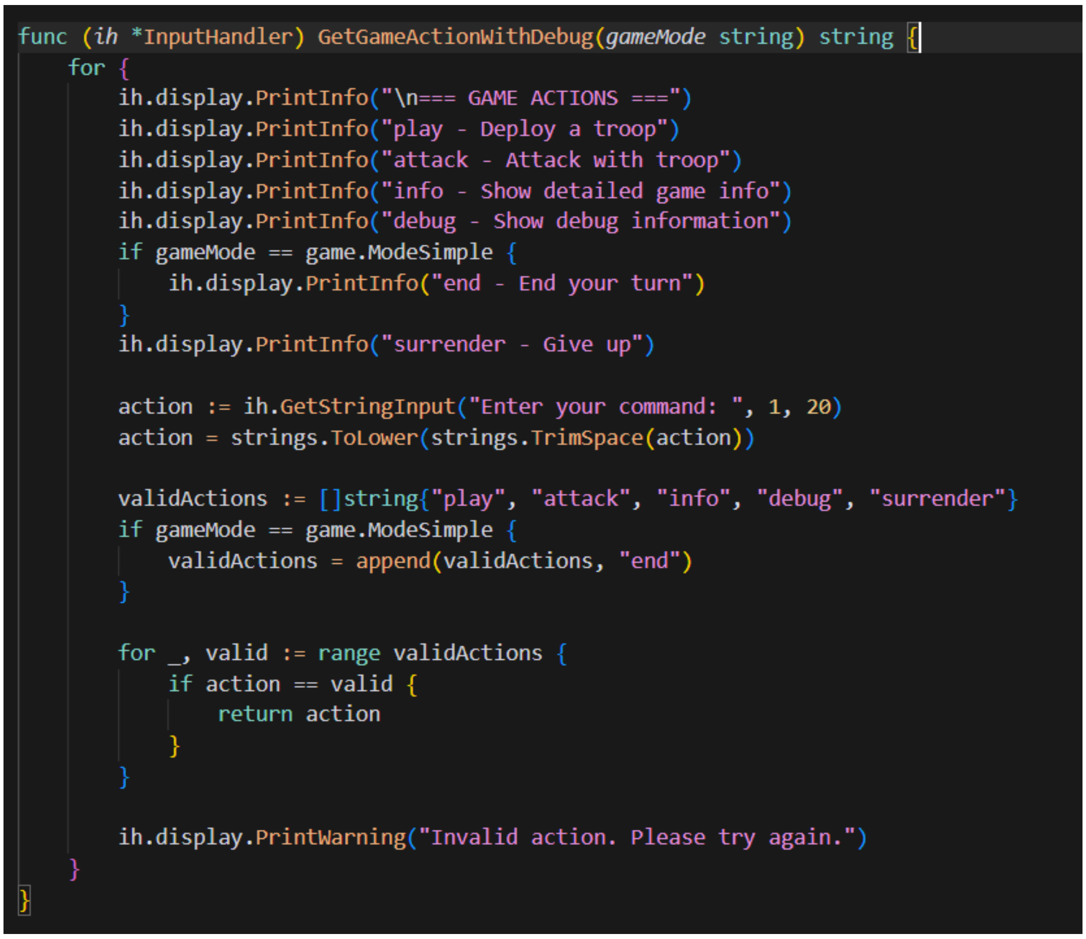
```go
func (c *Client) handlePlayCard() error {
    if len(c.myTroops) == 0 {
        c.display.PrintWarning("No troops available")
        return nil
    }

    if c.gameState.GameMode == game.ModeSimple {
        if len(c.deployedThisTurn) >= 1 {
            c.display.PrintError("Cannot deploy more than one troop per turn in simple mode")
            return nil
        }
    }
```

**Figure 6.3**: Troop Deployment Limit



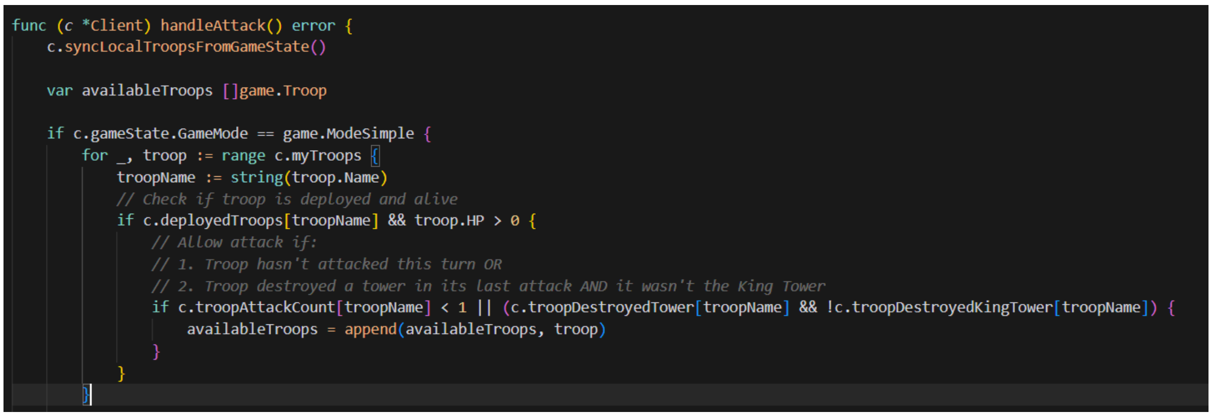
```go
// switchTurn changes current turn (Simple mode)
func (ge *GameEngine) switchTurn() {
    if ge.gameState.GameMode == ModeSimple {
        // Reset troop deployment counters
        ge.gameState.Player1.TroopsDeployedThisTurn = 0
        ge.gameState.Player2.TroopsDeployedThisTurn = 0
    }

    // Store old turn for logging
    oldTurn := ge.gameState.CurrentTurn

    // Switch turn
    if ge.gameState.CurrentTurn == ge.gameState.Player1.ID {
        ge.gameState.CurrentTurn = ge.gameState.Player2.ID
    } else {
        ge.gameState.CurrentTurn = ge.gameState.Player1.ID
    }

    // Log the turn change
    ge.logger.Info("Turn switched from %s to %s", oldTurn, ge.gameState.CurrentTurn)
}
```

**Figure 6.4**: Turn State Management



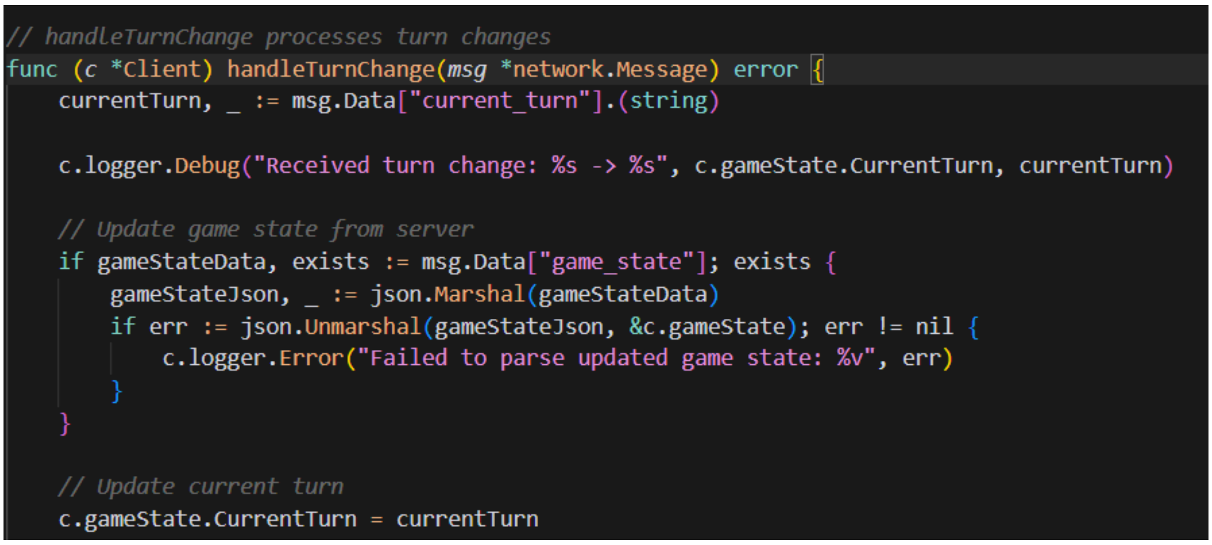
```go
func (ih *InputHandler) GetGameActionWithDebug(gameMode string) string {
    for {
        ih.display.PrintInfo("\n=== GAME ACTIONS ===")
        ih.display.PrintInfo("play - Deploy a troop")
        ih.display.PrintInfo("attack - Attack with troop")
        ih.display.PrintInfo("info - Show detailed game info")
        ih.display.PrintInfo("debug - Show debug information")
        if gameMode == game.ModeSimple {
            ih.display.PrintInfo("end - End your turn")
        }
        ih.display.PrintInfo("surrender - Give up")

        action := ih.GetStringInput("Enter your command: ", 1, 20)
        action = strings.ToLower(strings.TrimSpace(action))

        validActions := []string{"play", "attack", "info", "debug", "surrender"}
        if gameMode == game.ModeSimple {
            validActions = append(validActions, "end")
        }

        for _, valid := range validActions {
            if action == valid {
                return action
            }
        }

        ih.display.PrintWarning("Invalid action. Please try again.")
    }
}
```

**Figure 6.5**: Available Actions Display



```go
func (c *Client) handleAttack() error {
    c.syncLocalTroopsFromGameState()

    var availableTroops []game.Troop

    if c.gameState.GameMode == game.ModeSimple {
        for _, troop := range c.myTroops {
            troopName := string(troop.Name)
            // Check if troop is deployed and alive
            if c.deployedTroops[troopName] && troop.HP > 0 {
                // Allow attack if:
                // 1. Troop hasn't attacked this turn OR
                // 2. Troop destroyed a tower in its last attack AND it wasn't the King Tower
                if c.troopAttackCount[troopName] < 1 || (c.troopDestroyedTower[troopName] && !c.troopDestroyedKingTower[troopName]) {
                    availableTroops = append(availableTroops, troop)
                }
            }
        }
```

**Figure 6.6**: Troop Attack Management

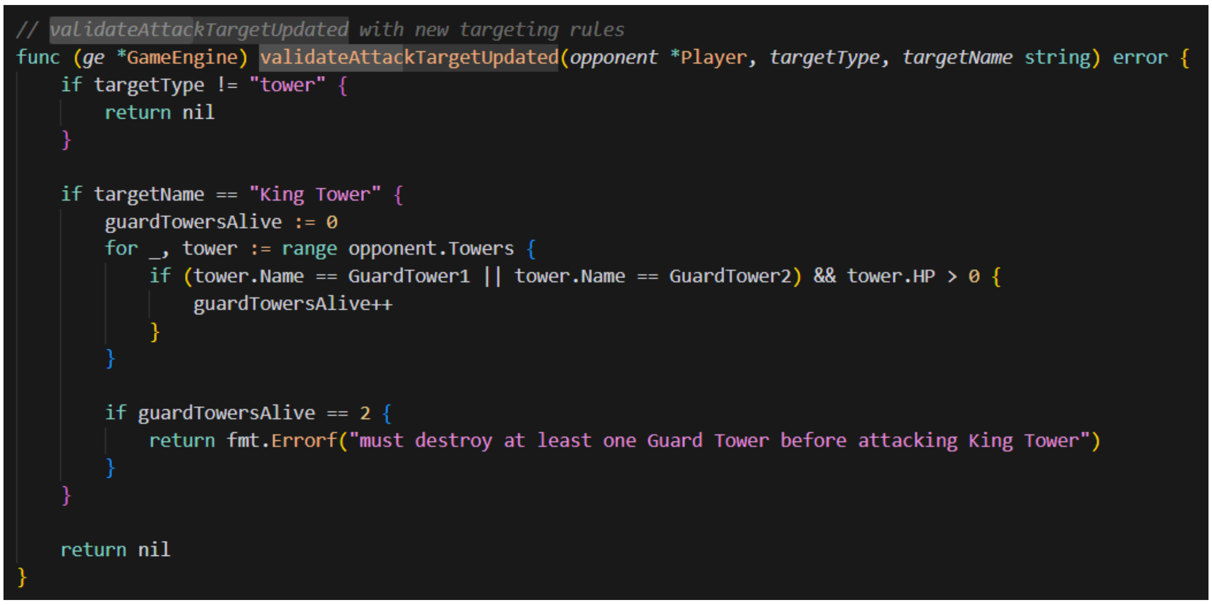


```go
// handleTurnChange processes turn changes
func (c *Client) handleTurnChange(msg *network.Message) error {
    currentTurn, _ := msg.Data["current_turn"].(string)

    c.logger.Debug("Received turn change: %s -> %s", c.gameState.CurrentTurn, currentTurn)

    // Update game state from server
    if gameStateData, exists := msg.Data["game_state"]; exists {
        gameStateJson, _ := json.Marshal(gameStateData)
        if err := json.Unmarshal(gameStateJson, &c.gameState); err != nil {
            c.logger.Error("Failed to parse updated game state: %v", err)
        }
    }

    // Update current turn
    c.gameState.CurrentTurn = currentTurn
```

**Figure 6.7**: Turn Change Handling

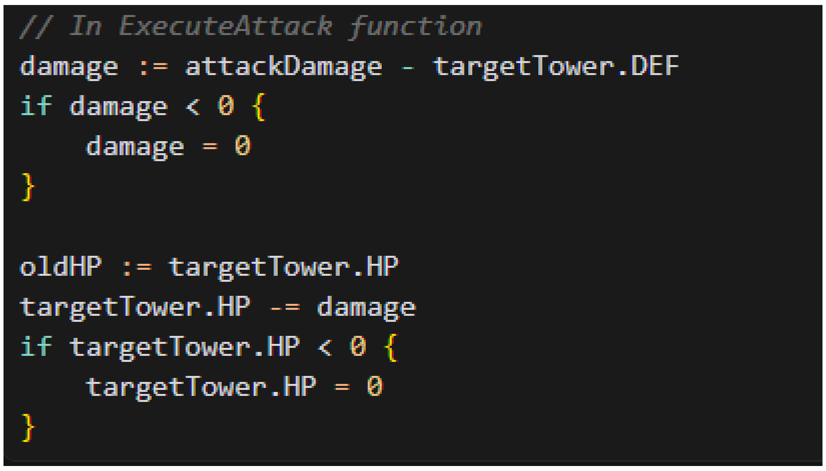
### 6.1.3 Combat and Targeting Rules

The combat and targeting system enforces key rules to maintain game balance. Players must destroy at least one Guard Tower before being allowed to attack the King Tower. Damage inflicted during attacks is calculated using the formula: damage equals the attacker's attack value minus the defender's defense, with a minimum of zero. If a troop successfully destroys a tower, it is granted an additional attack opportunity within the same turn. These rules are implemented through the **validateAttackTargetUpdated** function, which checks tower statuses before allowing attacks on the King Tower, and the **ExecuteAttack** function, which performs damage calculation and updates tower health.



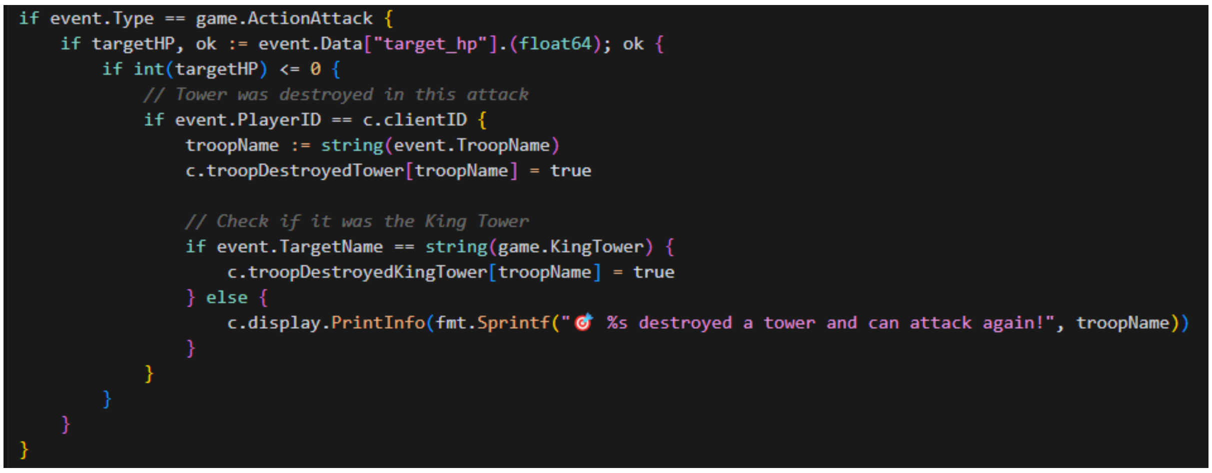
```go
// validateAttackTargetUpdated with new targeting rules
func (ge *GameEngine) validateAttackTargetUpdated(opponent *Player, targetType, targetName string) error {
    if targetType != "tower" {
        return nil
    }

    if targetName == "King Tower" {
        guardTowersAlive := 0
        for _, tower := range opponent.Towers {
            if (tower.Name == GuardTower1 || tower.Name == GuardTower2) && tower.HP > 0 {
                guardTowersAlive++
            }
        }

        if guardTowersAlive == 2 {
            return fmt.Errorf("must destroy at least one Guard Tower before attacking King Tower")
        }
    }

    return nil
}
```

**Figure 6.8**: Tower Targeting Rules



```go
// In ExecuteAttack function
damage := attackDamage - targetTower.DEF
if damage < 0 {
    damage = 0
}

oldHP := targetTower.HP
targetTower.HP -= damage
if targetTower.HP < 0 {
    targetTower.HP = 0
}
```

**Figure 6.9**: Damage Calculation



```go
if event.Type == game.ActionAttack {
    if targetHP, ok := event.Data["target_hp"].(float64); ok {
        if int(targetHP) <= 0 {
            // Tower was destroyed in this attack
            if event.PlayerID == c.clientID {
                troopName := string(event.TroopName)
                c.troopDestroyedTower[troopName] = true

                // Check if it was the King Tower
                if event.TargetName == string(game.KingTower) {
                    c.troopDestroyedKingTower[troopName] = true
                } else {
                    c.display.PrintInfo(fmt.Sprintf("🎯 %s destroyed a tower and can attack again!", troopName))
                }
            }
        }
    }
}
```

**Figure 6.10**: Bonus Attack on Tower Destruction

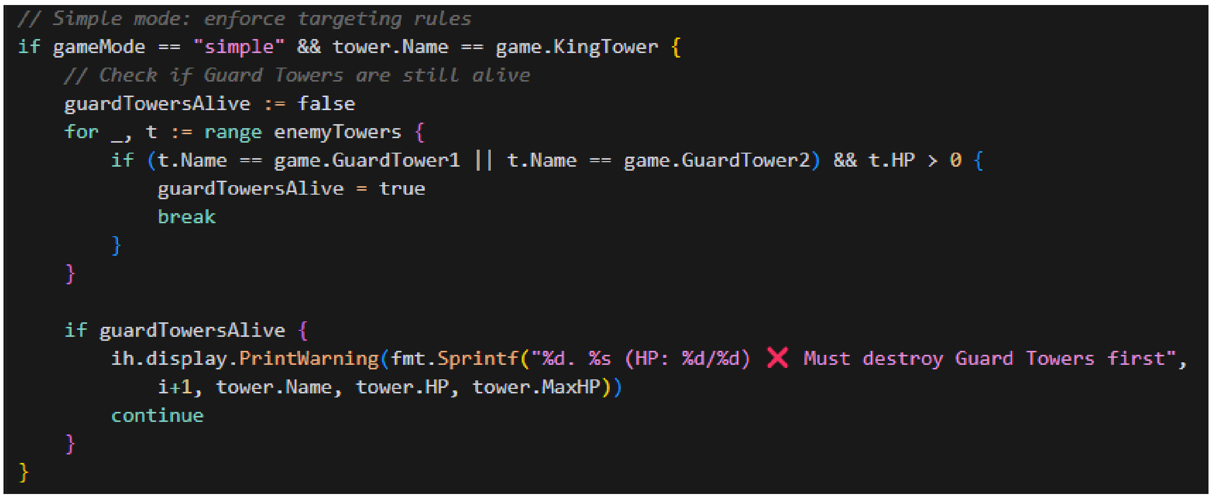
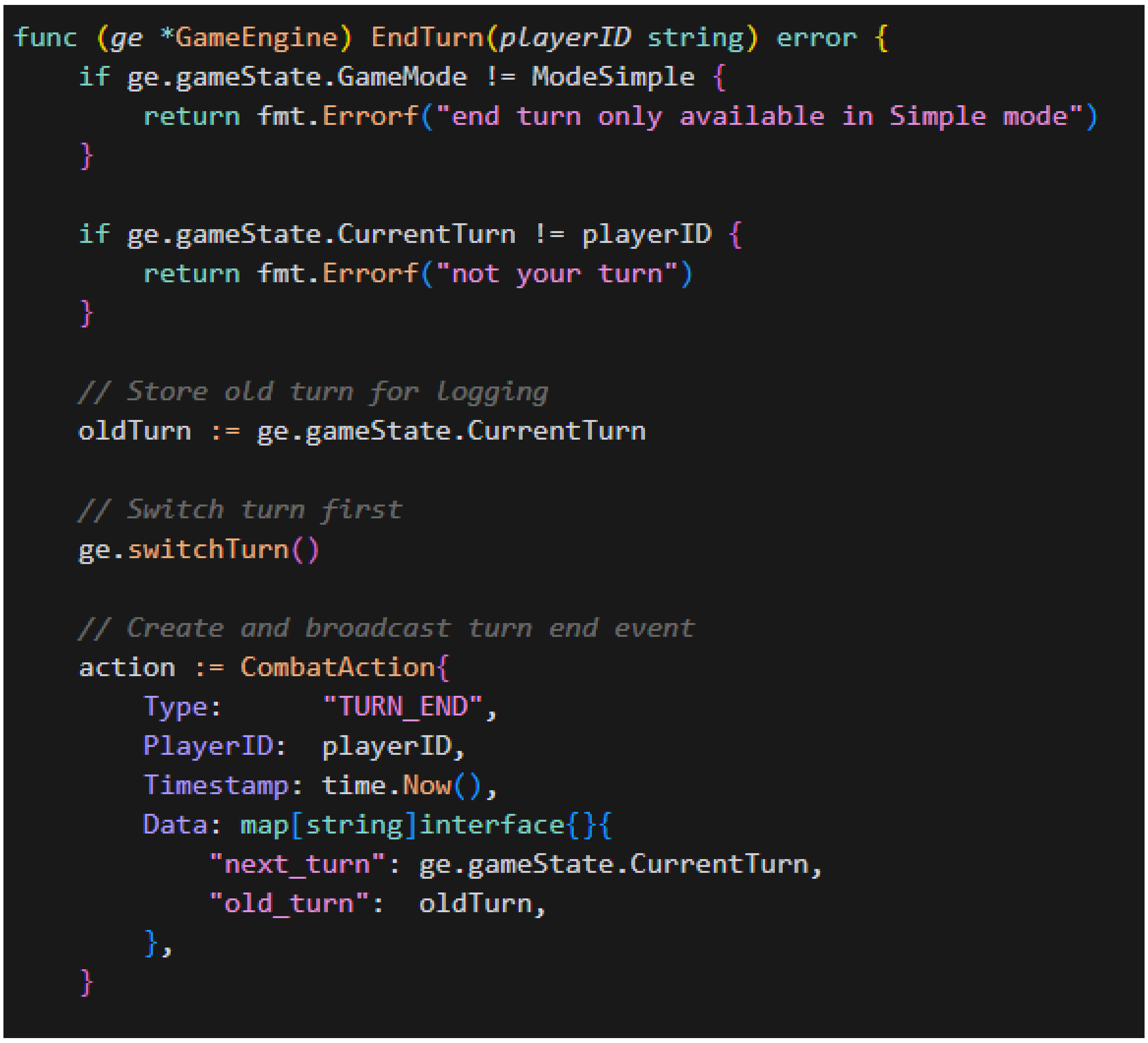

```go
// Simple mode: enforce targeting rules
if gameMode == "simple" && tower.Name == game.KingTower {
    // Check if Guard Towers are still alive
    guardTowersAlive := false
    for _, t := range enemyTowers {
        if (t.Name == game.GuardTower1 || t.Name == game.GuardTower2) && t.HP > 0 {
            guardTowersAlive = true
            break
        }
    }

    if guardTowersAlive {
        ih.display.PrintWarning(fmt.Sprintf("%d. %s (HP: %d/%d) ❌ Must destroy Guard Towers first",
            i+1, tower.Name, tower.HP, tower.MaxHP))
        continue
    }
}
```

**Figure 6.11**: Attack Validation in Simple Mode

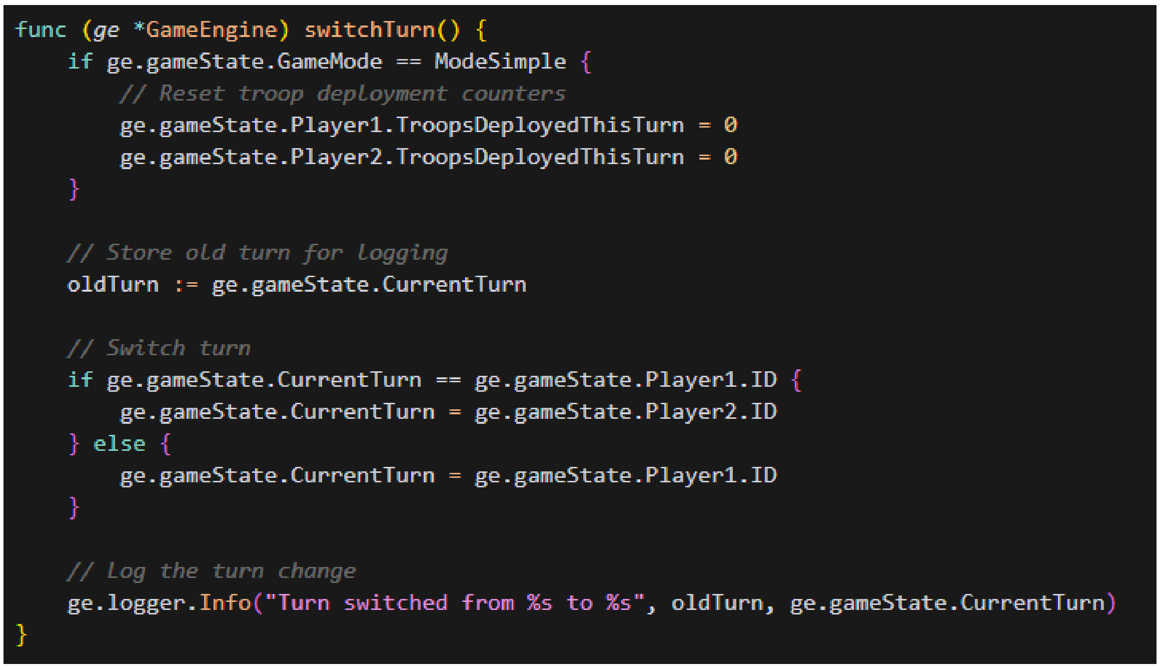
### 6.1.4 Turn Progression and Statement

Turn progression is managed by several core functions. The **EndTurn** function signals the end of a player's turn, while the **switchTurn** function resets deployment counters and switches the **CurrentTurn** to the opposing player. On the client side, the **handleTurnChange** function updates the user interface to indicate the active player and adjust available actions accordingly. When a turn ends, the deployment counter is reset, control passes to the other player, a turn change event is broadcast to all players, and the UI refreshes to reflect the new game state.



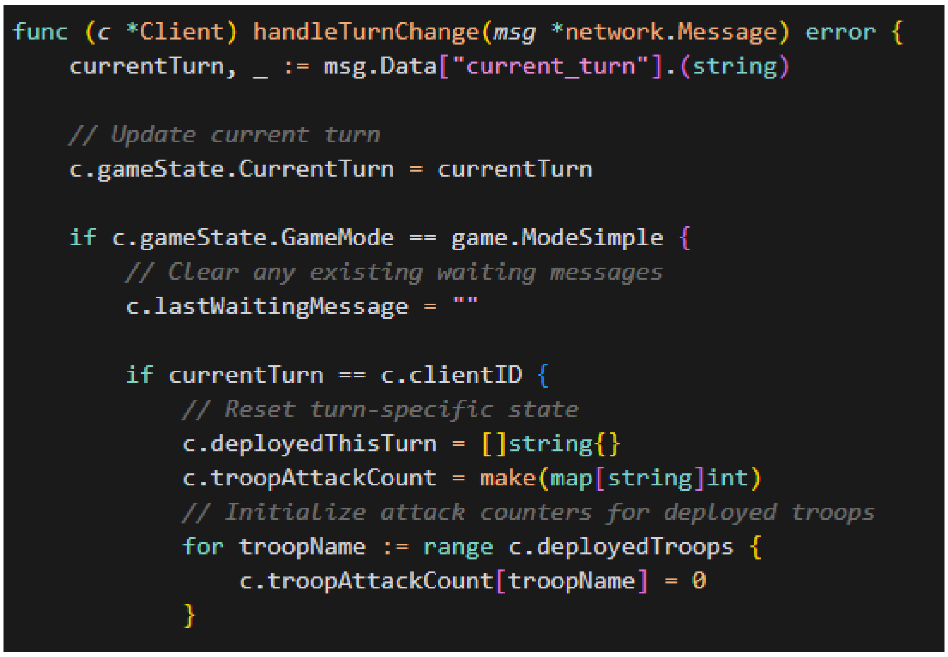
```go
func (ge *GameEngine) EndTurn(playerID string) error {
    if ge.gameState.GameMode != ModeSimple {
        return fmt.Errorf("end turn only available in Simple mode")
    }

    if ge.gameState.CurrentTurn != playerID {
        return fmt.Errorf("not your turn")
    }

    // Store old turn for logging
    oldTurn := ge.gameState.CurrentTurn

    // Switch turn first
    ge.switchTurn()

    // Create and broadcast turn end event
    action := CombatAction{
        Type:      "TURN_END",
        PlayerID:  playerID,
        Timestamp: time.Now(),
        Data: map[string]interface{}{
            "next_turn": ge.gameState.CurrentTurn,
            "old_turn":  oldTurn,
        },
    }
```

**Figure 6.12**: EndTurn function



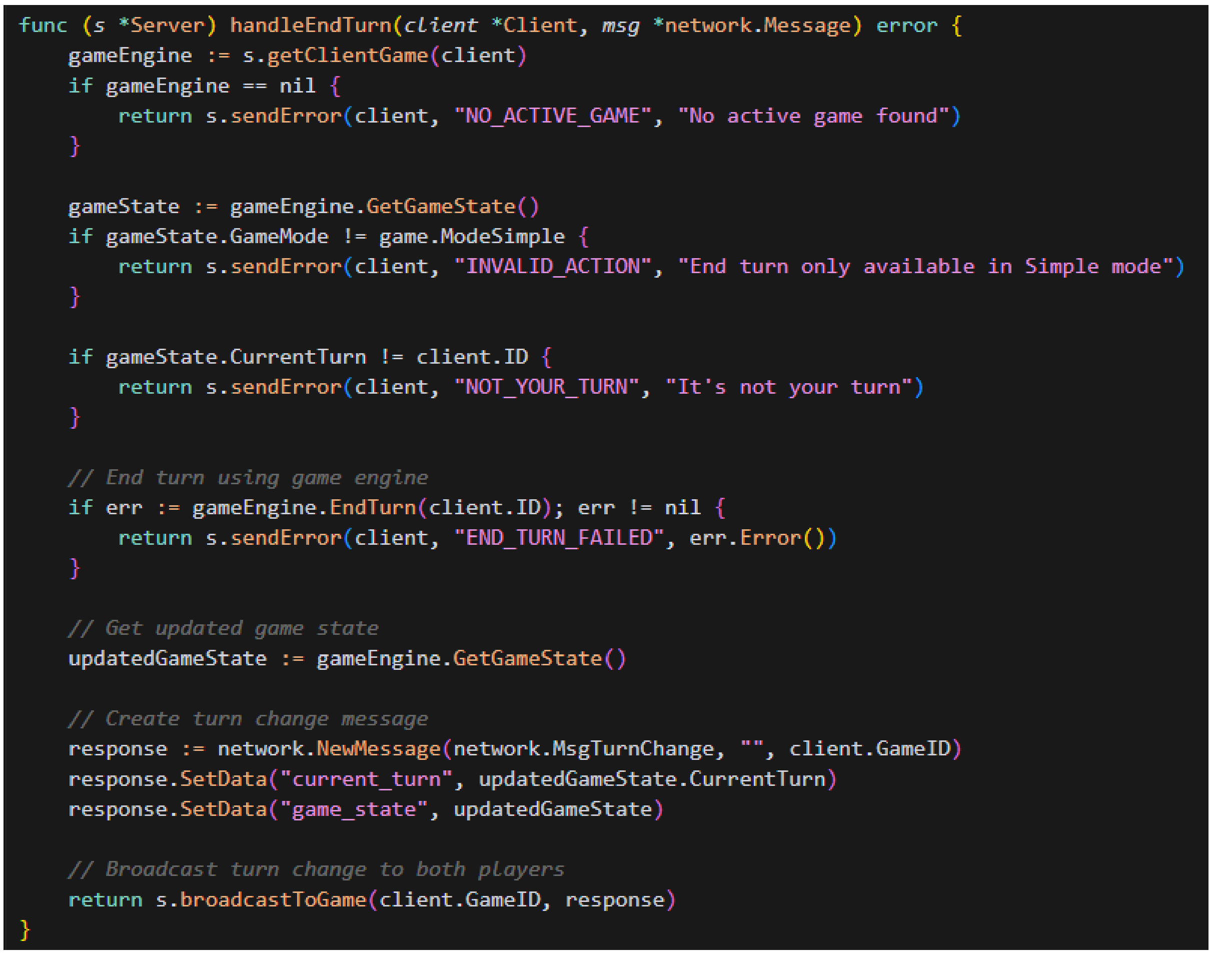
```go
func (ge *GameEngine) switchTurn() {
    if ge.gameState.GameMode == ModeSimple {
        // Reset troop deployment counters
        ge.gameState.Player1.TroopsDeployedThisTurn = 0
        ge.gameState.Player2.TroopsDeployedThisTurn = 0
    }

    // Store old turn for logging
    oldTurn := ge.gameState.CurrentTurn

    // Switch turn
    if ge.gameState.CurrentTurn == ge.gameState.Player1.ID {
        ge.gameState.CurrentTurn = ge.gameState.Player2.ID
    } else {
        ge.gameState.CurrentTurn = ge.gameState.Player1.ID
    }

    // Log the turn change
    ge.logger.Info("Turn switched from %s to %s", oldTurn, ge.gameState.CurrentTurn)
}
```

**Figure 6.13**: switchTurn function



```go
func (c *Client) handleTurnChange(msg *network.Message) error {
    currentTurn, _ := msg.Data["current_turn"].(string)

    // Update current turn
    c.gameState.CurrentTurn = currentTurn

    if c.gameState.GameMode == game.ModeSimple {
        // Clear any existing waiting messages
        c.lastWaitingMessage = ""

        if currentTurn == c.clientID {
            // Reset turn-specific state
            c.deployedThisTurn = []string{}
            c.troopAttackCount = make(map[string]int)
            // Initialize attack counters for deployed troops
            for troopName := range c.deployedTroops {
                c.troopAttackCount[troopName] = 0
            }
```

**Figure 6.14**: handleTurnChange function

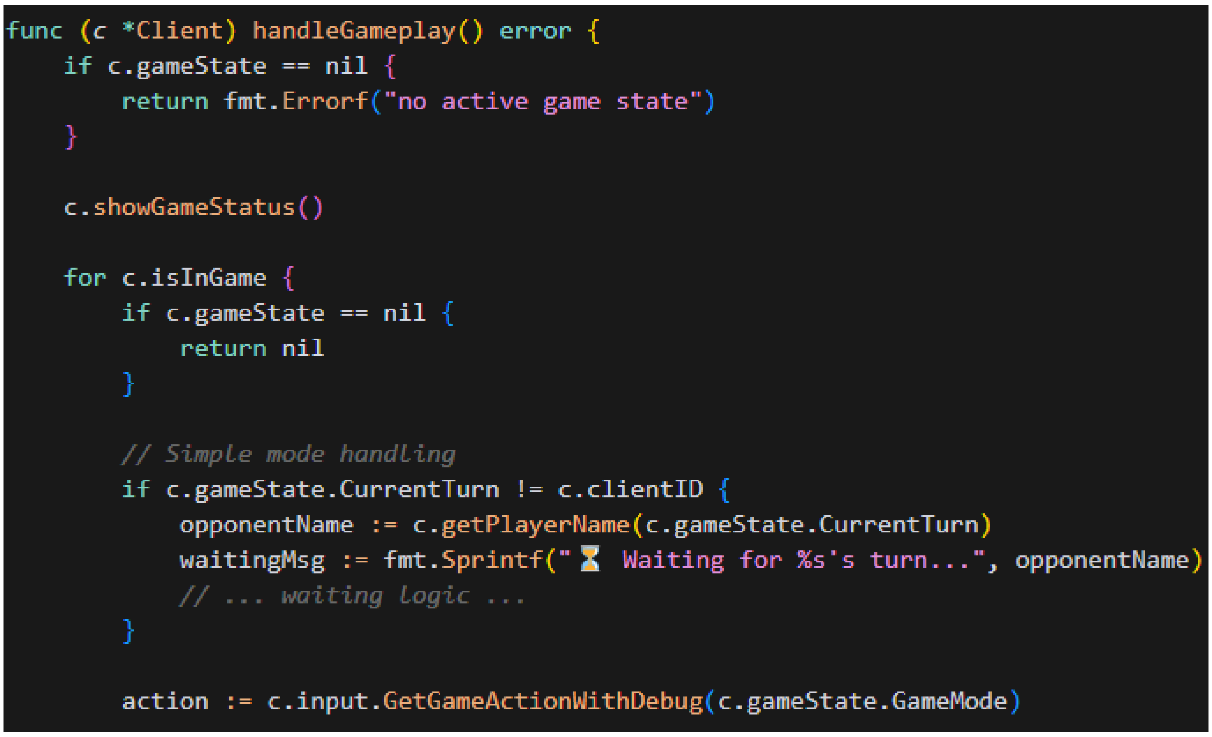


```go
func (s *Server) handleEndTurn(client *Client, msg *network.Message) error {
    gameEngine := s.getClientGame(client)
    if gameEngine == nil {
        return s.sendError(client, "NO_ACTIVE_GAME", "No active game found")
    }

    gameState := gameEngine.GetGameState()
    if gameState.GameMode != game.ModeSimple {
        return s.sendError(client, "INVALID_ACTION", "End turn only available in Simple mode")
    }

    if gameState.CurrentTurn != client.ID {
        return s.sendError(client, "NOT_YOUR_TURN", "It's not your turn")
    }

    // End turn using game engine
    if err := gameEngine.EndTurn(client.ID); err != nil {
        return s.sendError(client, "END_TURN_FAILED", err.Error())
    }

    // Get updated game state
    updatedGameState := gameEngine.GetGameState()

    // Create turn change message
    response := network.NewMessage(network.MsgTurnChange, "", client.GameID)
    response.SetData("current_turn", updatedGameState.CurrentTurn)
    response.SetData("game_state", updatedGameState)

    // Broadcast turn change to both players
    return s.broadcastToGame(client.GameID, response)
}
```

**Figure 6.15:** Server-side Turn Change Handling

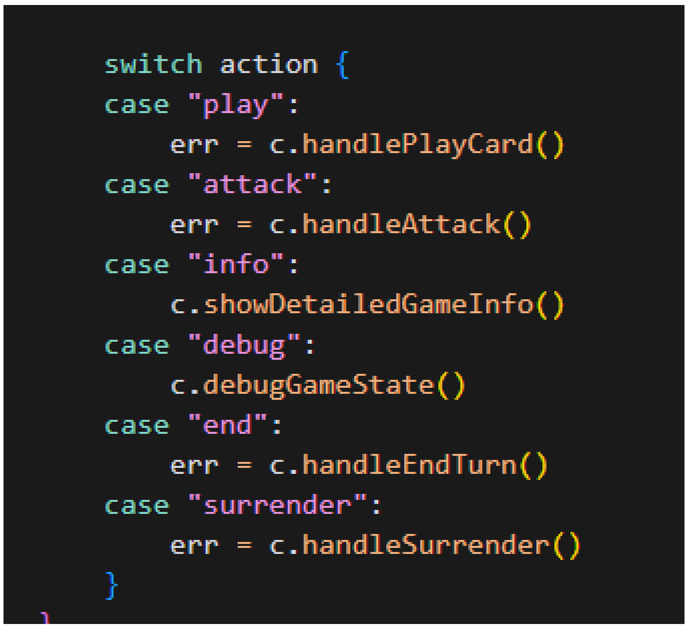
### 6.1.6 Game Flow and Player Interaction

The overall game flow is coordinated by the client's **handleGameplay** function, which checks whether it is the player's turn, displays available actions, processes player commands, updates the game state, and broadcasts changes to all participants. Players can deploy troops (one per turn), attack, view game information, end their turn, or surrender. The game continues until a player's King Tower is destroyed, a player surrenders, or the game times out. This turn-based system encourages strategic planning, requiring players to manage troop deployments and attacks effectively while defending their own towers.



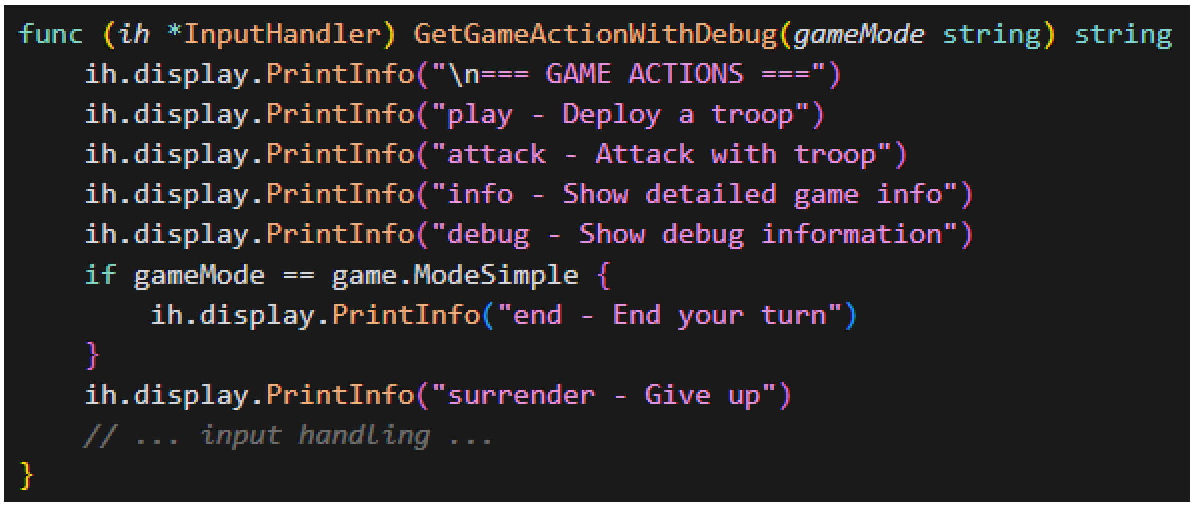
```go
func (c *Client) handleGameplay() error {
    if c.gameState == nil {
        return fmt.Errorf("no active game state")
    }

    c.showGameStatus()

    for c.isInGame {
        if c.gameState == nil {
            return nil
        }

        // Simple mode handling
        if c.gameState.CurrentTurn != c.clientID {
            opponentName := c.getPlayerName(c.gameState.CurrentTurn)
            waitingMsg := fmt.Sprintf("⏳ Waiting for %s's turn...", opponentName)
            // ... waiting logic ...
        }

        action := c.input.GetGameActionWithDebug(c.gameState.GameMode)
```

**Figure 6.16:** Main Game Loop



```
    switch action {
    case "play":
        err = c.handlePlayCard()
    case "attack":
        err = c.handleAttack()
    case "info":
        c.showDetailedGameInfo()
    case "debug":
        c.debugGameState()
    case "end":
        err = c.handleEndTurn()
    case "surrender":
        err = c.handleSurrender()
    }
```

**Figure 6.17**: Actions of game mechanics



```
func (ih *InputHandler) GetGameActionWithDebug(gameMode string) string
    ih.display.PrintInfo("\n=== GAME ACTIONS ===")
    ih.display.PrintInfo("play - Deploy a troop")
    ih.display.PrintInfo("attack - Attack with troop")
    ih.display.PrintInfo("info - Show detailed game info")
    ih.display.PrintInfo("debug - Show debug information")
    if gameMode == game.ModeSimple {
        ih.display.PrintInfo("end - End your turn")
    }
    ih.display.PrintInfo("surrender - Give up")
    // ... input handling ...
}
```

**Figure 6.18**: Available Actions in player's turn

## 6.2 Enhanced TCR Rules

### 6.2.1 Critical Hit Chance Feature

The game incorporates a real-time combat system that operates without turns and lasts for a fixed duration of three minutes (180 seconds). Unlike the turn-based mode, players continuously deploy troops and engage in combat throughout the entire period, creating a fast-paced and dynamic gameplay experience.

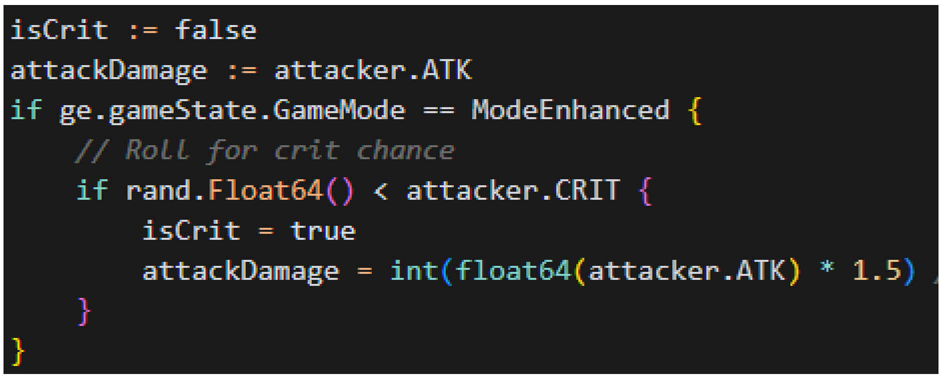
Damage calculation in this mode includes the possibility of critical hits. The damage dealt by an attacker is computed using the formula

Base damage formula: damage = attackDamage - targetTower.DEF

Critical hit formula: damage = (attackDamage * 1.5) - targetTower.DEF

Each troop and tower possesses a base critical hit chance, typically 10% for most units and 5% specifically for Guard Towers. When a critical hit occurs, the damage inflicted is increased by 50%, applying a 1.5x multiplier to the attacker's base attack value.

This critical hit mechanic adds an element of unpredictability and excitement to combat, requiring players to adapt their strategies in real-time as damage outcomes can vary significantly due to critical strikes.



```
isCrit := false
attackDamage := attacker.ATK
if ge.gameState.GameMode == ModeEnhanced {
    // Roll for crit chance
    if rand.Float64() < attacker.CRIT {
        isCrit = true
        attackDamage = int(float64(attacker.ATK) * 1.5)
    }
}
```

**Figure 6.19**: Critical Hit Mechanics

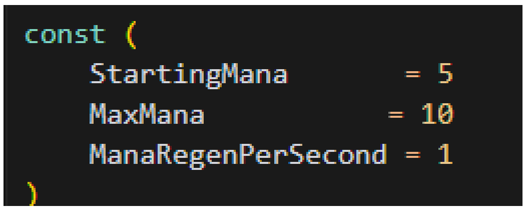
### 6.2.2 Mana Systems

Players begin the game with an initial mana pool of 5 mana points, which they can accumulate up to a maximum of 10 mana. Mana serves as a crucial resource used to summon troops during gameplay.

Mana regenerates continuously at a fixed rate of 1 mana point per second, allowing players to gradually replenish their resource over time and strategize troop deployment accordingly.

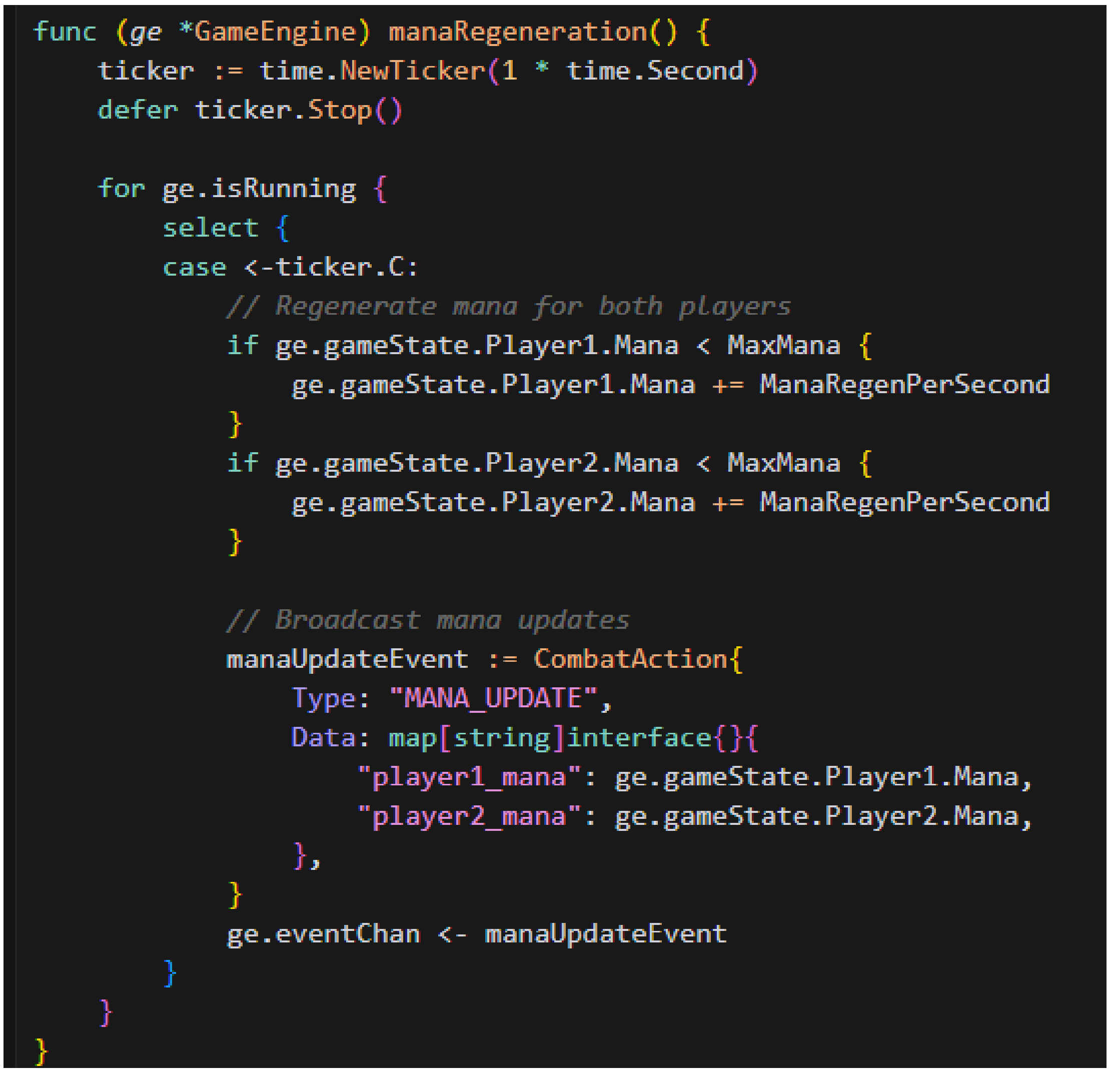
Each type of troop requires a specific amount of mana to be summoned, reflecting their relative strength and utility. The mana costs are as follows: Pawn requires 3 mana, Bishop 4 mana, Rook 5 mana, Knight 5 mana, Prince 6 mana, and Queen 5 mana.

This system encourages players to manage their mana resource carefully, balancing between deploying powerful troops with higher mana costs and more frequent summoning of lower-cost units.



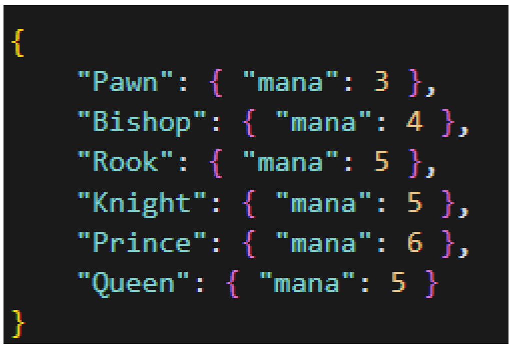
```go
const (
    StartingMana       = 5
    MaxMana           = 10
    ManaRegenPerSecond = 1
)
```

**Figure 6.20**: Mana Constants and Initialization



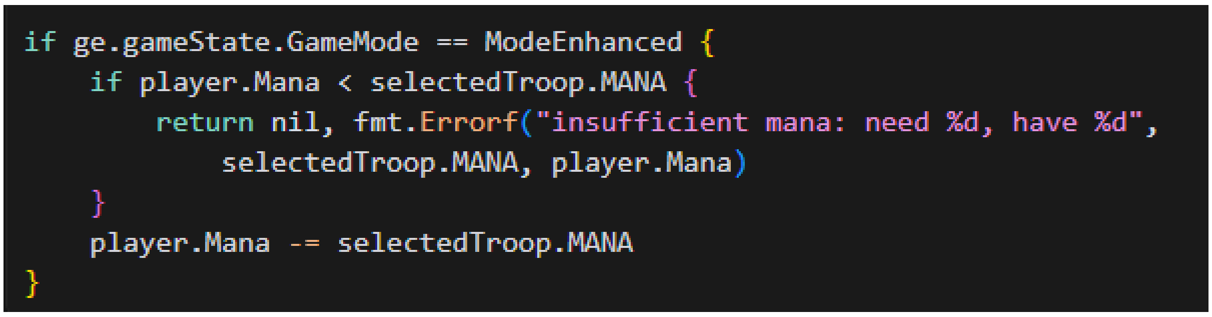
```go
func (ge *GameEngine) manaRegeneration() {
    ticker := time.NewTicker(1 * time.Second)
    defer ticker.Stop()

    for ge.isRunning {
        select {
        case <-ticker.C:
            // Regenerate mana for both players
            if ge.gameState.Player1.Mana < MaxMana {
                ge.gameState.Player1.Mana += ManaRegenPerSecond
            }
            if ge.gameState.Player2.Mana < MaxMana {
                ge.gameState.Player2.Mana += ManaRegenPerSecond
            }

            // Broadcast mana updates
            manaUpdateEvent := CombatAction{
                Type: "MANA_UPDATE",
                Data: map[string]interface{}{
                    "player1_mana": ge.gameState.Player1.Mana,
                    "player2_mana": ge.gameState.Player2.Mana,
                },
            }
            ge.eventChan <- manaUpdateEvent
        }
    }
}
```

**Figure 6.21**: Mana Regeneration System



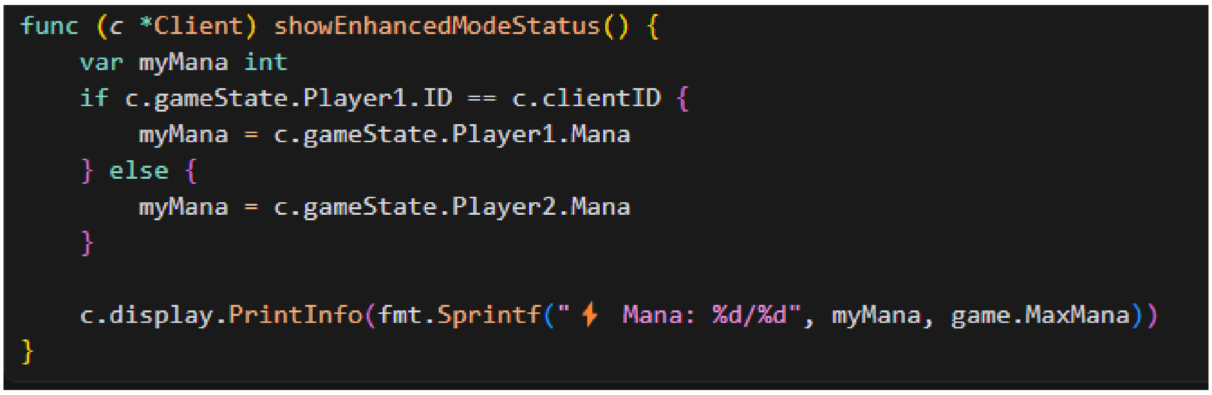
```json
{
    "Pawn": { "mana": 3 },
    "Bishop": { "mana": 4 },
    "Rook": { "mana": 5 },
    "Knight": { "mana": 5 },
    "Prince": { "mana": 6 },
    "Queen": { "mana": 5 }
}
```

**Figure 6.22**: Troop Mana Costs (troops.json)



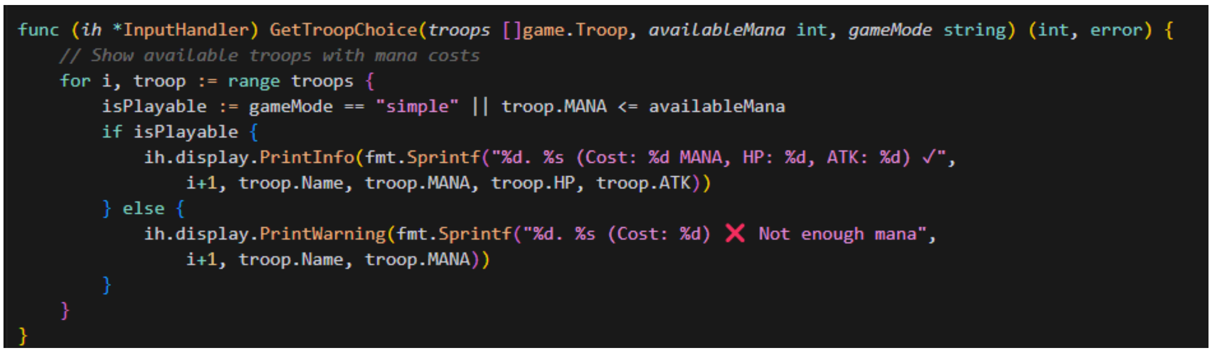
```go
if ge.gameState.GameMode == ModeEnhanced {
    if player.Mana < selectedTroop.MANA {
        return nil, fmt.Errorf("insufficient mana: need %d, have %d",
            selectedTroop.MANA, player.Mana)
    }
    player.Mana -= selectedTroop.MANA
}
```

**Figure 6.23**: Mana Cost Validation when Summon Troop



```go
func (c *Client) showEnhancedModeStatus() {
    var myMana int
    if c.gameState.Player1.ID == c.clientID {
        myMana = c.gameState.Player1.Mana
    } else {
        myMana = c.gameState.Player2.Mana
    }

    c.display.PrintInfo(fmt.Sprintf("⚡ Mana: %d/%d", myMana, game.MaxMana))
}
```

**Figure 6.24**: Mana Display and UI



```go
func (ih *InputHandler) GetTroopChoice(troops []game.Troop, availableMana int, gameMode string) (int, error) {
    // Show available troops with mana costs
    for i, troop := range troops {
        isPlayable := gameMode == "simple" || troop.MANA <= availableMana
        if isPlayable {
            ih.display.PrintInfo(fmt.Sprintf("%d. %s (Cost: %d MANA, HP: %d, ATK: %d) ✓",
                i+1, troop.Name, troop.MANA, troop.HP, troop.ATK))
        } else {
            ih.display.PrintWarning(fmt.Sprintf("%d. %s (Cost: %d) ❌ Not enough mana",
                i+1, troop.Name, troop.MANA))
        }
    }
}
```

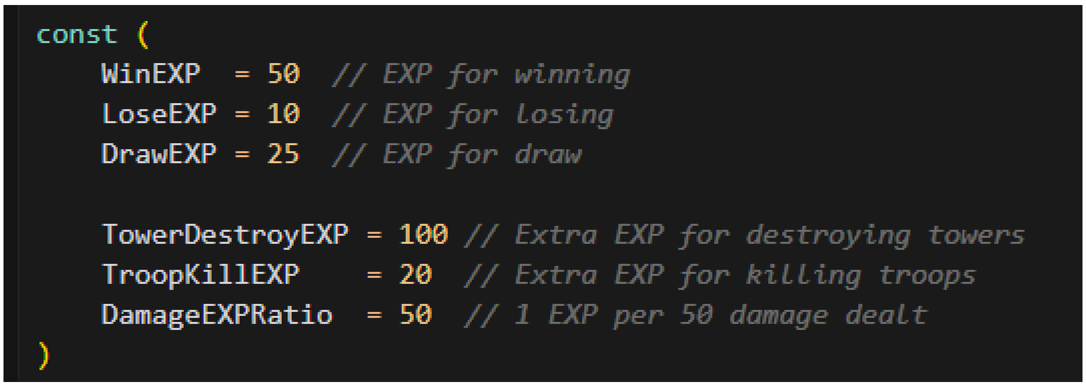
**Figure 6.25**: Troop Selection with Mana Consideration

### 6.2.3 Experience and Leveling System

Players accumulate experience points (EXP) by performing various in-game actions. For every 50 points of damage dealt to opponents, players earn 1 EXP. Additionally, destroying towers grants significant EXP bonuses: 100 EXP for eliminating a Guard Tower and 200 EXP for destroying the King Tower.

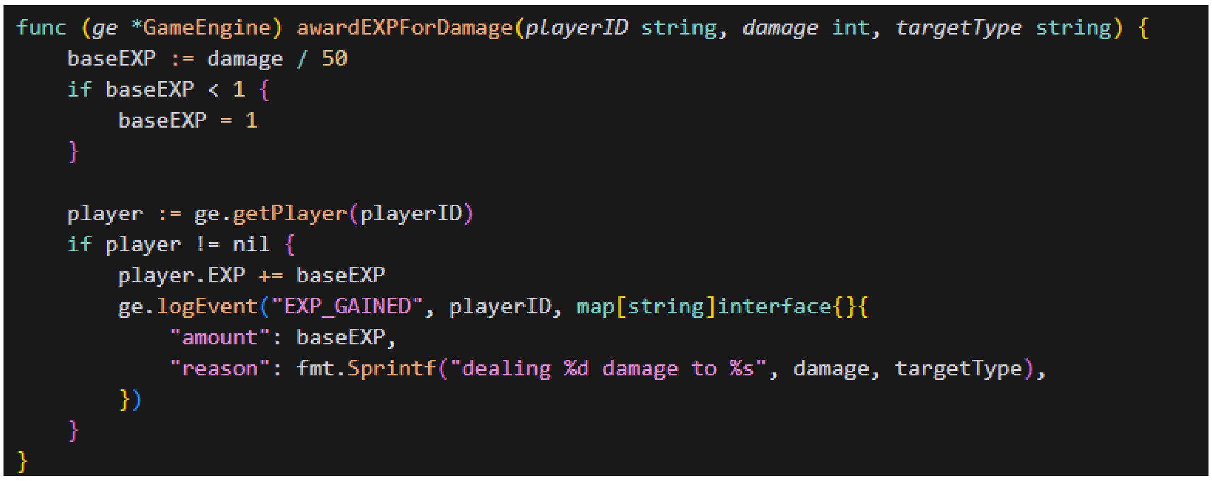
Upon completion of a game, players receive rewards based on the outcome. A victory awards 50 EXP, while a draw grants 25 EXP to each participant. These rewards incentivize competitive and strategic play throughout the match.

The leveling system enhances gameplay progression by increasing troop and tower statistics by 10% with each level gained. To level up, players must meet increasing EXP requirements, which rise by 15% for every subsequent level. The base experience requirement to reach level 2 is set at 100 EXP, establishing a scalable challenge for players as they advance.



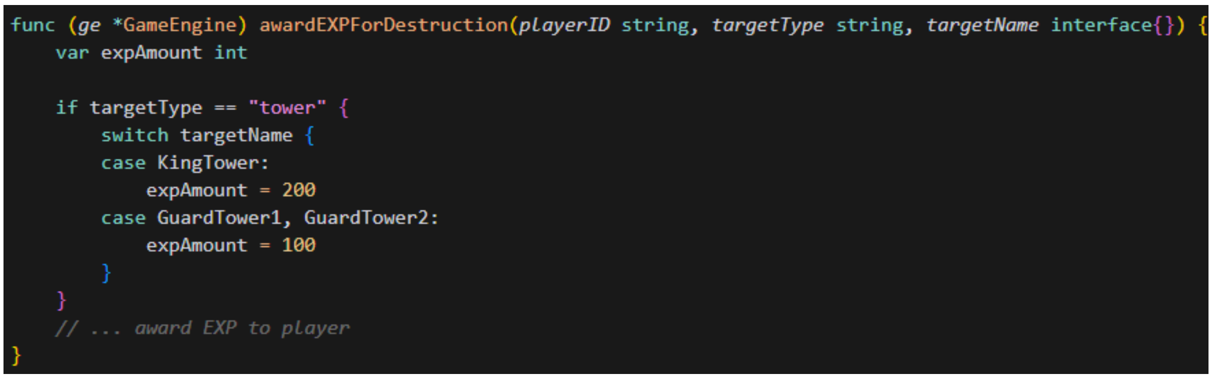
```go
const (
    WinEXP   = 50  // EXP for winning
    LoseEXP  = 10  // EXP for losing
    DrawEXP  = 25  // EXP for draw

    TowerDestroyEXP  = 100 // Extra EXP for destroying towers
    TroopKillEXP     = 20  // Extra EXP for killing troops
    DamageEXPRatio   = 50  // 1 EXP per 50 damage dealt
)
```

**Figure 6.26**: Experience Points Reward Constants



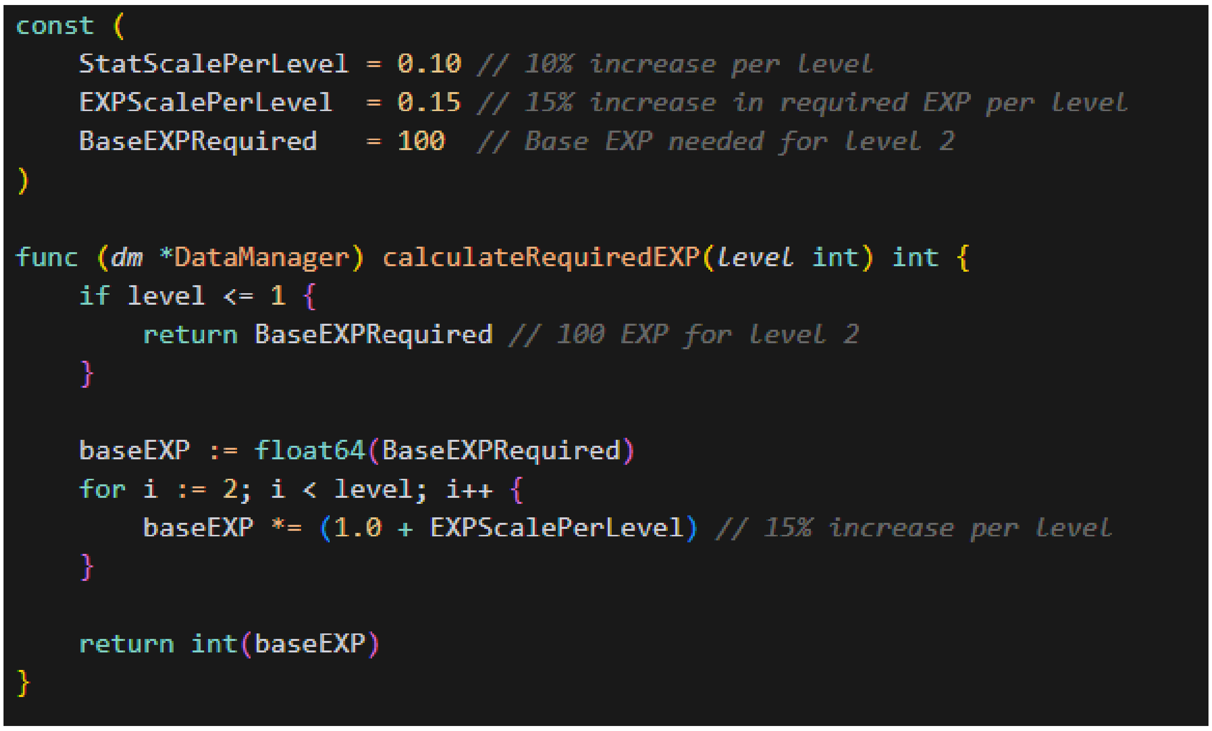
```go
func (ge *GameEngine) awardEXPForDamage(playerID string, damage int, targetType string) {
    baseEXP := damage / 50
    if baseEXP < 1 {
        baseEXP = 1
    }

    player := ge.getPlayer(playerID)
    if player != nil {
        player.EXP += baseEXP
        ge.logEvent("EXP_GAINED", playerID, map[string]interface{}{
            "amount": baseEXP,
            "reason": fmt.Sprintf("dealing %d damage to %s", damage, targetType),
        })
    }
}
```

**Figure 6.27**: Damage-based EXP



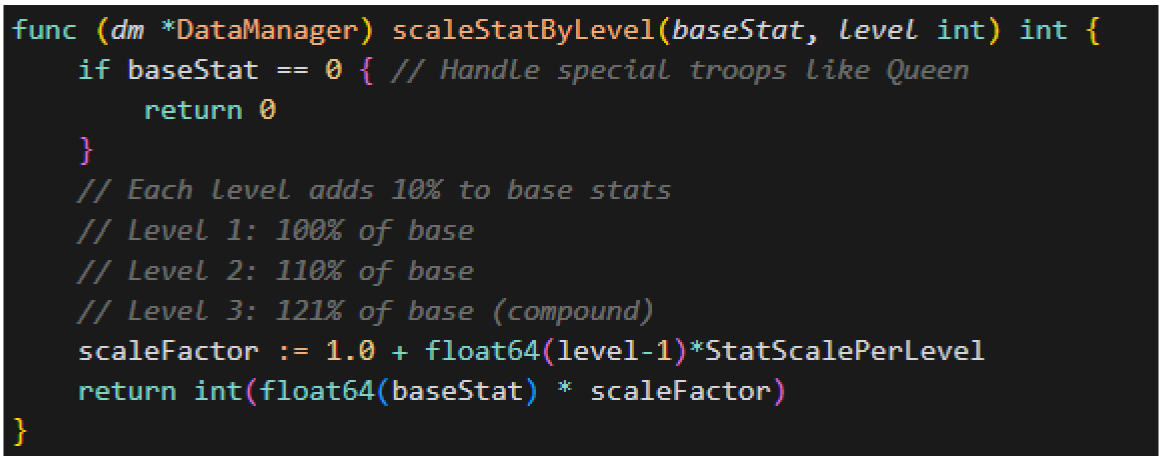
```go
func (ge *GameEngine) awardEXPForDestruction(playerID string, targetType string, targetName interface{}) {
    var expAmount int

    if targetType == "tower" {
        switch targetName {
        case KingTower:
            expAmount = 200
        case GuardTower1, GuardTower2:
            expAmount = 100
        }
    }
    // ... award EXP to player
}
```

**Figure 6.28**: Tower Destruction EXP



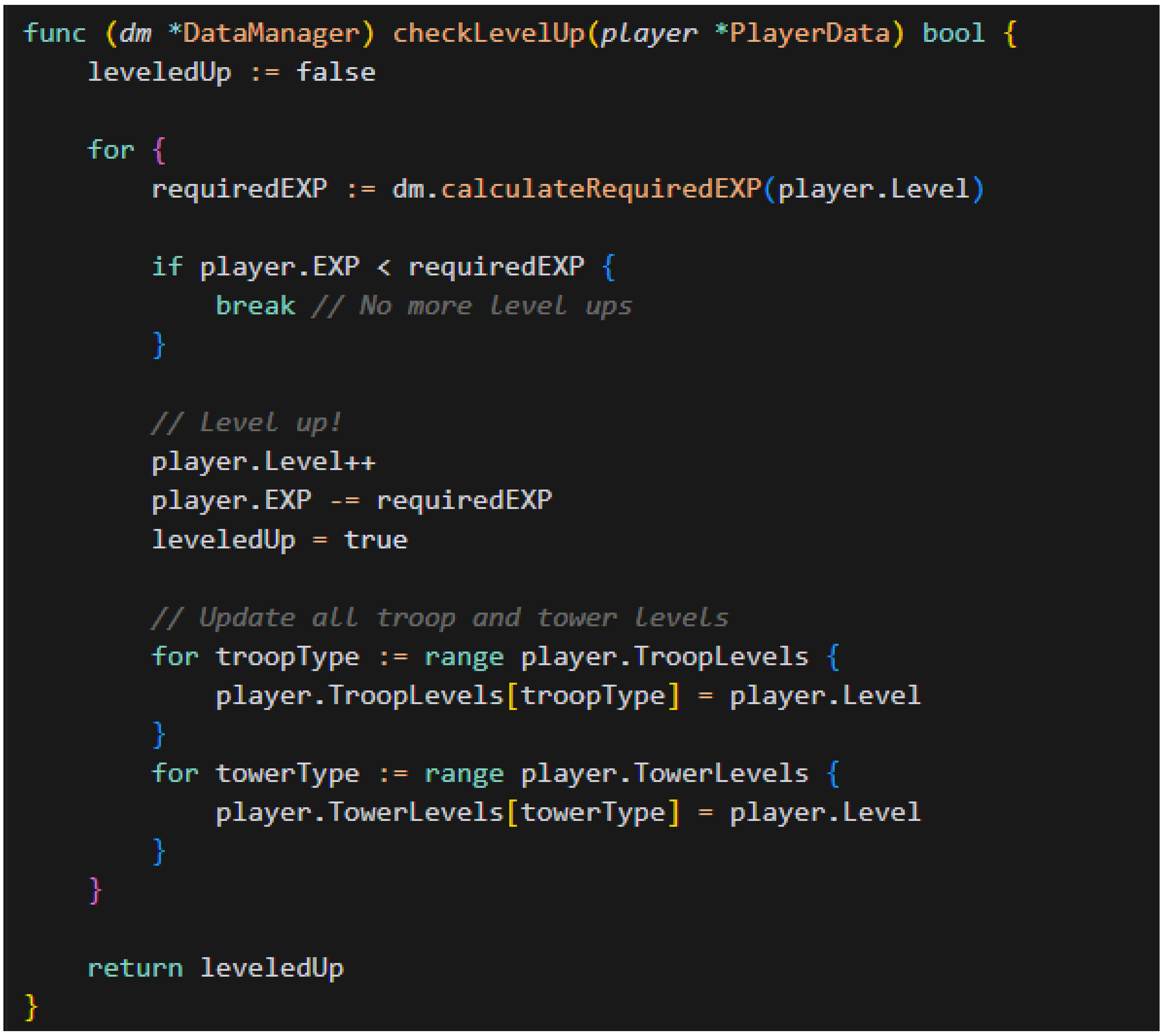
```go
const (
    StatScalePerLevel = 0.10 // 10% increase per level
    EXPScalePerLevel  = 0.15 // 15% increase in required EXP per level
    BaseEXPRequired   = 100  // Base EXP needed for level 2
)

func (dm *DataManager) calculateRequiredEXP(level int) int {
    if level <= 1 {
        return BaseEXPRequired // 100 EXP for level 2
    }

    baseEXP := float64(BaseEXPRequired)
    for i := 2; i < level; i++ {
        baseEXP *= (1.0 + EXPScalePerLevel) // 15% increase per level
    }

    return int(baseEXP)
}
```

**Figure 6.29**: Leveling Systems



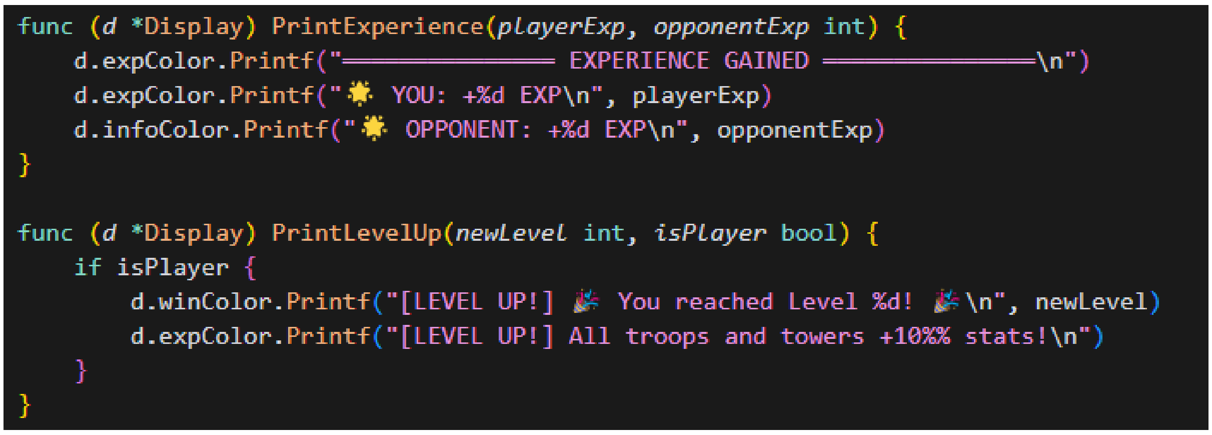
```go
func (dm *DataManager) scaleStatByLevel(baseStat, level int) int {
    if baseStat == 0 { // Handle special troops like Queen
        return 0
    }
    // Each level adds 10% to base stats
    // Level 1: 100% of base
    // Level 2: 110% of base
    // Level 3: 121% of base (compound)
    scaleFactor := 1.0 + float64(level-1)*StatScalePerLevel
    return int(float64(baseStat) * scaleFactor)
}
```

**Figure 6.30**: Stat Scaling with Levels



```go
func (dm *DataManager) checkLevelUp(player *PlayerData) bool {
    leveledUp := false

    for {
        requiredEXP := dm.calculateRequiredEXP(player.Level)

        if player.EXP < requiredEXP {
            break // No more level ups
        }

        // Level up!
        player.Level++
        player.EXP -= requiredEXP
        leveledUp = true

        // Update all troop and tower levels
        for troopType := range player.TroopLevels {
            player.TroopLevels[troopType] = player.Level
        }
        for towerType := range player.TowerLevels {
            player.TowerLevels[towerType] = player.Level
        }
    }

    return leveledUp
}
```

**Figure 6.31**: Level Up Process



```go
func (d *Display) PrintExperience(playerExp, opponentExp int) {
    d.expColor.Printf("═══════════════ EXPERIENCE GAINED ═══════════════\n")
    d.expColor.Printf("🌟 YOU: +%d EXP\n", playerExp)
    d.infoColor.Printf("🌟 OPPONENT: +%d EXP\n", opponentExp)
}

func (d *Display) PrintLevelUp(newLevel int, isPlayer bool) {
    if isPlayer {
        d.winColor.Printf("[LEVEL UP!] 🎉 You reached Level %d! 🎉\n", newLevel)
        d.expColor.Printf("[LEVEL UP!] All troops and towers +10%% stats!\n")
    }
}
```

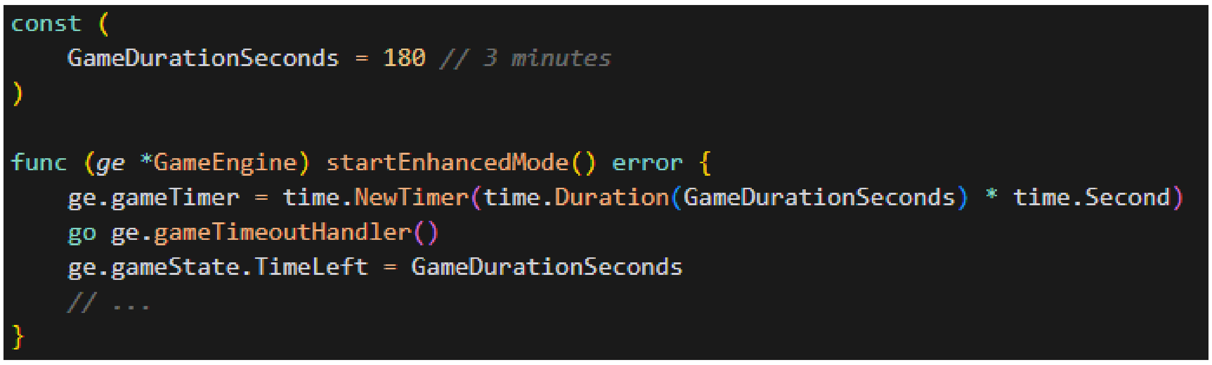
**Figure 6.32**: UI Display for EXP and Leveling

### 6.2.4 Continous Combat & Win Conditions

The primary objective of the game is to destroy the opponent's King Tower. Achieving this immediately concludes the match in favor of the attacking player. If neither player manages to destroy the King Tower within the allotted time, the game evaluates a secondary victory condition based on tower destruction.

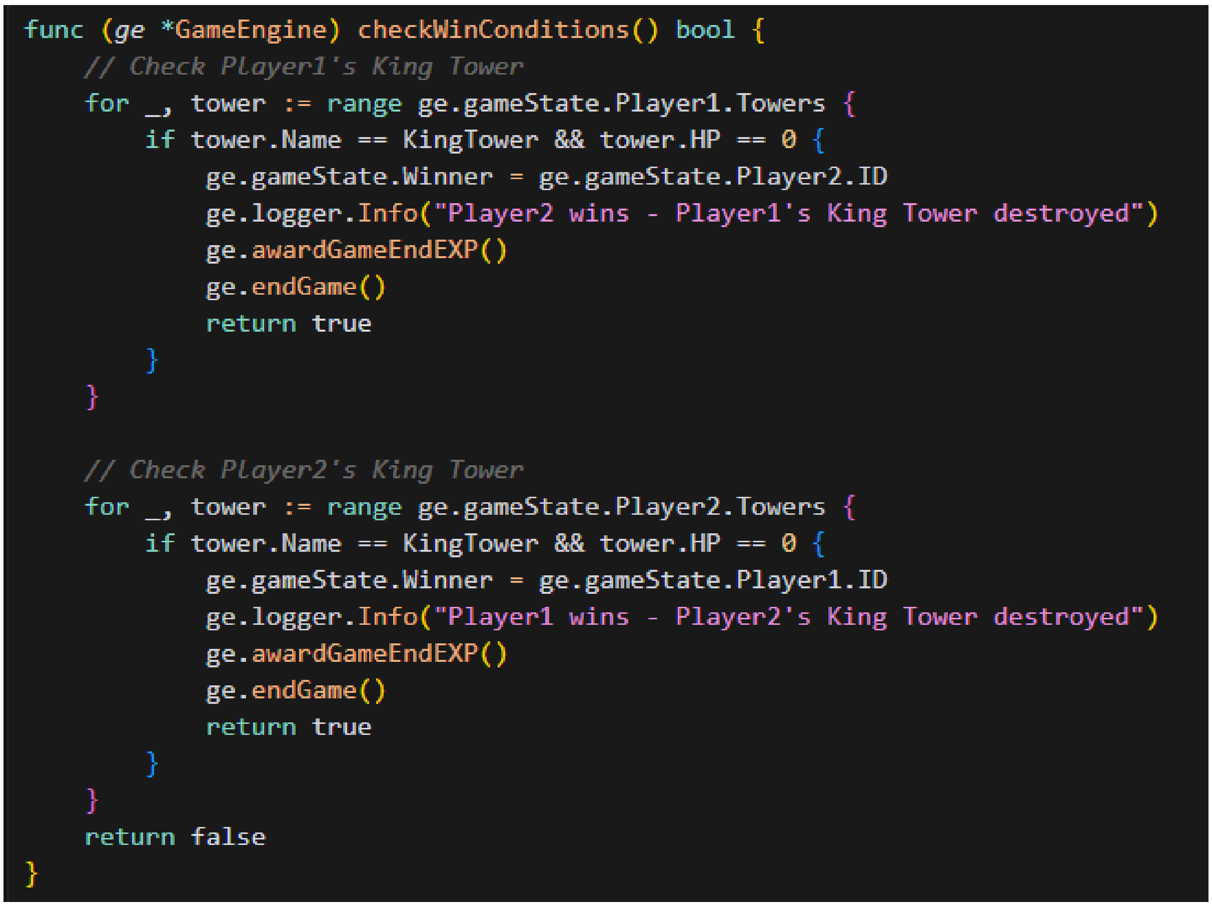
If the three-minute timer expires without a King Tower being destroyed, the player who has destroyed more of the opponent’s towers is declared the winner. Should both players have destroyed an equal number of towers, the game is considered a draw.

Matches are constrained by a strict three-minute time limit, ensuring fast-paced gameplay and strategic urgency. Additionally, players have the option to surrender at any point during the game, which results in an immediate loss for the surrendering player.



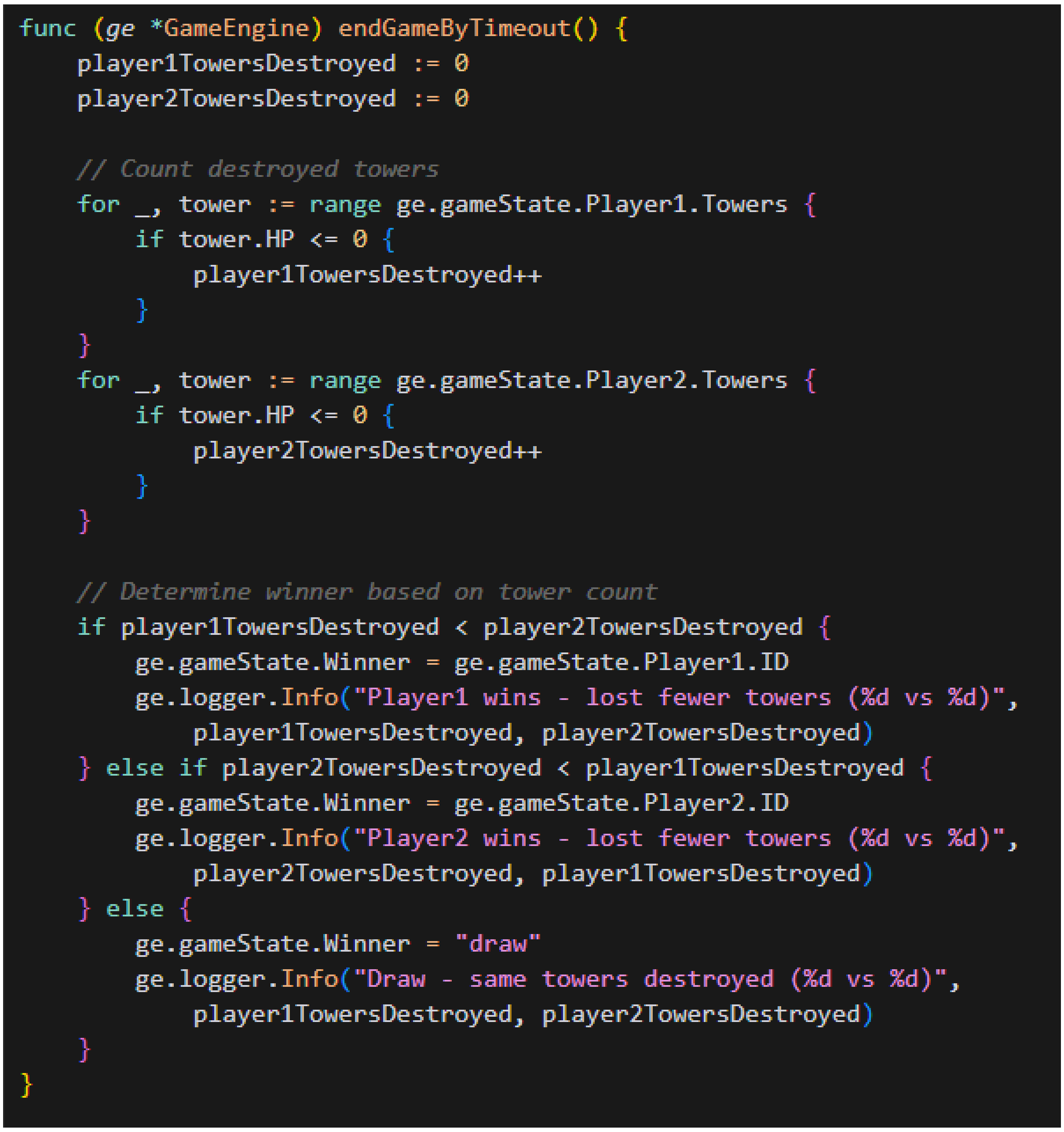
```go
const (
    GameDurationSeconds = 180 // 3 minutes
)

func (ge *GameEngine) startEnhancedMode() error {
    ge.gameTimer = time.NewTimer(time.Duration(GameDurationSeconds) * time.Second)
    go ge.gameTimeoutHandler()
    ge.gameState.TimeLeft = GameDurationSeconds
    // ...
}
```

**Figure 6.33**: Game Duration and Timer



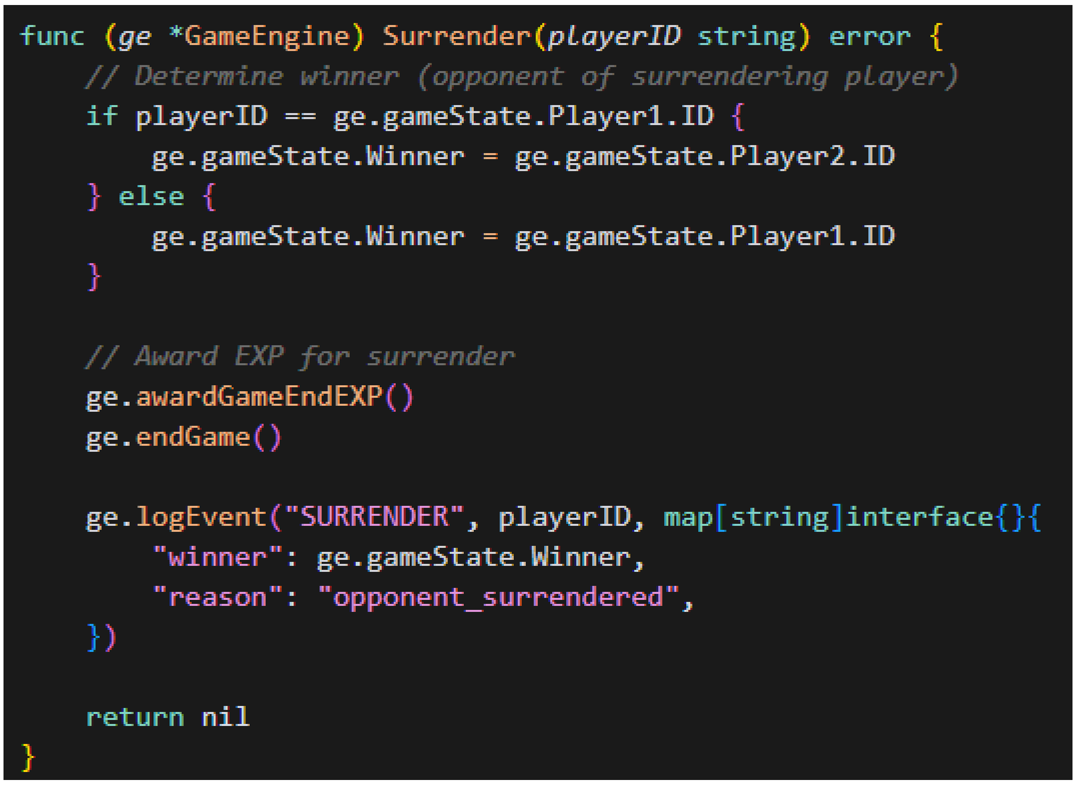
```go
func (ge *GameEngine) checkWinConditions() bool {
    // Check Player1's King Tower
    for _, tower := range ge.gameState.Player1.Towers {
        if tower.Name == KingTower && tower.HP == 0 {
            ge.gameState.Winner = ge.gameState.Player2.ID
            ge.logger.Info("Player2 wins - Player1's King Tower destroyed")
            ge.awardGameEndEXP()
            ge.endGame()
            return true
        }
    }

    // Check Player2's King Tower
    for _, tower := range ge.gameState.Player2.Towers {
        if tower.Name == KingTower && tower.HP == 0 {
            ge.gameState.Winner = ge.gameState.Player1.ID
            ge.logger.Info("Player1 wins - Player2's King Tower destroyed")
            ge.awardGameEndEXP()
            ge.endGame()
            return true
        }
    }
    return false
}
```

**Figure 6.34**: Primary Win Condition (King Tower Destruction)



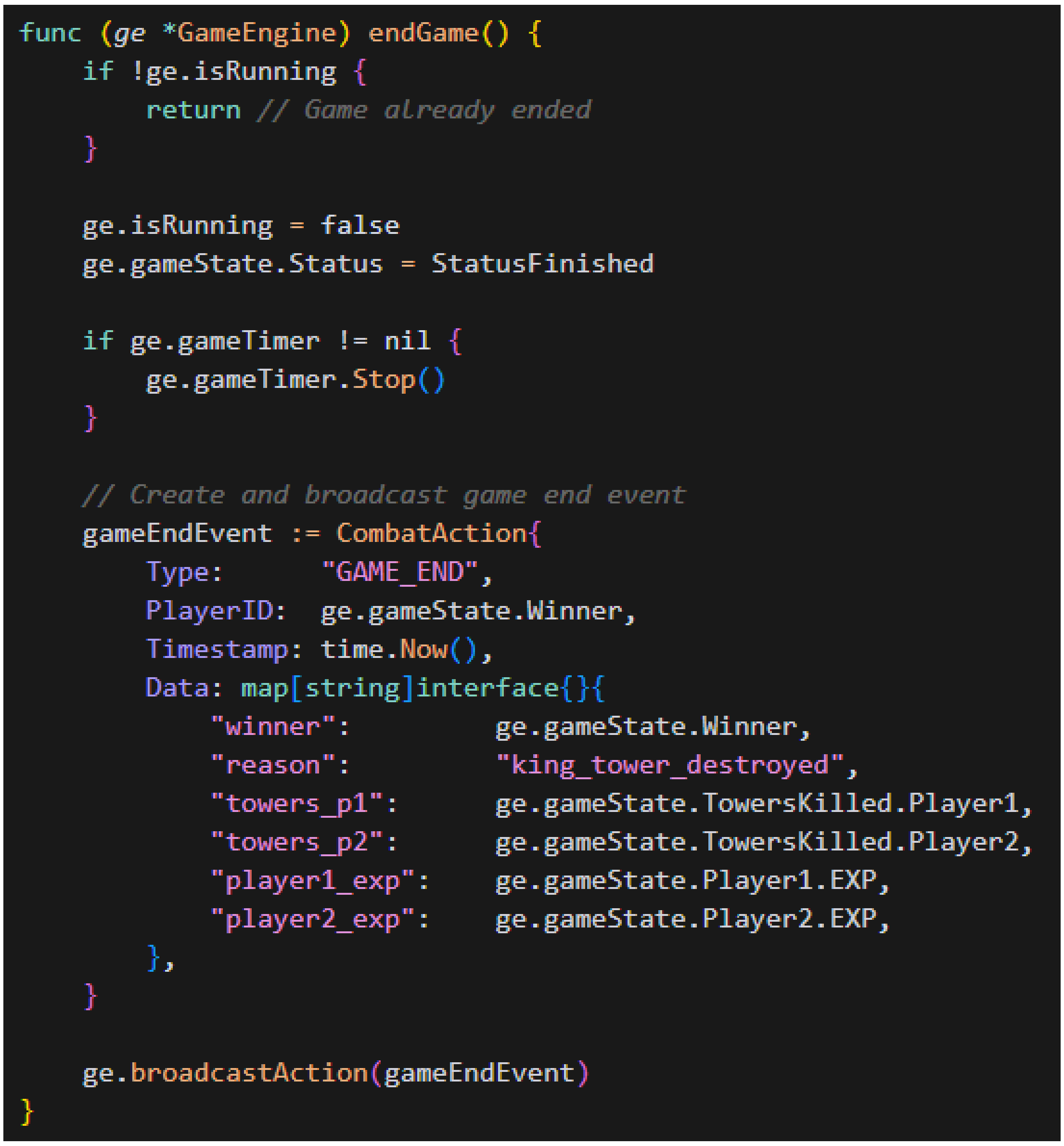
```go
func (ge *GameEngine) endGameByTimeout() {
    player1TowersDestroyed := 0
    player2TowersDestroyed := 0

    // Count destroyed towers
    for _, tower := range ge.gameState.Player1.Towers {
        if tower.HP <= 0 {
            player1TowersDestroyed++
        }
    }
    for _, tower := range ge.gameState.Player2.Towers {
        if tower.HP <= 0 {
            player2TowersDestroyed++
        }
    }

    // Determine winner based on tower count
    if player1TowersDestroyed < player2TowersDestroyed {
        ge.gameState.Winner = ge.gameState.Player1.ID
        ge.logger.Info("Player1 wins - lost fewer towers (%d vs %d)",
            player1TowersDestroyed, player2TowersDestroyed)
    } else if player2TowersDestroyed < player1TowersDestroyed {
        ge.gameState.Winner = ge.gameState.Player2.ID
        ge.logger.Info("Player2 wins - lost fewer towers (%d vs %d)",
            player2TowersDestroyed, player1TowersDestroyed)
    } else {
        ge.gameState.Winner = "draw"
        ge.logger.Info("Draw - same towers destroyed (%d vs %d)",
            player1TowersDestroyed, player2TowersDestroyed)
    }
}
```

**Figure 6.35**: Secondary Win Condition (Tower Count)



```go
func (ge *GameEngine) Surrender(playerID string) error {
    // Determine winner (opponent of surrendering player)
    if playerID == ge.gameState.Player1.ID {
        ge.gameState.Winner = ge.gameState.Player2.ID
    } else {
        ge.gameState.Winner = ge.gameState.Player1.ID
    }

    // Award EXP for surrender
    ge.awardGameEndEXP()
    ge.endGame()

    ge.logEvent("SURRENDER", playerID, map[string]interface{}{
        "winner": ge.gameState.Winner,
        "reason": "opponent_surrendered",
    })

    return nil
}
```

**Figure 6.36**: Surrender Mechanism



```go
func (ge *GameEngine) endGame() {
    if !ge.isRunning {
        return // Game already ended
    }

    ge.isRunning = false
    ge.gameState.Status = StatusFinished

    if ge.gameTimer != nil {
        ge.gameTimer.Stop()
    }

    // Create and broadcast game end event
    gameEndEvent := CombatAction{
        Type:      "GAME_END",
        PlayerID:  ge.gameState.Winner,
        Timestamp: time.Now(),
        Data: map[string]interface{}{
            "winner":         ge.gameState.Winner,
            "reason":         "king_tower_destroyed",
            "towers_p1":      ge.gameState.TowersKilled.Player1,
            "towers_p2":      ge.gameState.TowersKilled.Player2,
            "player1_exp":    ge.gameState.Player1.EXP,
            "player2_exp":    ge.gameState.Player2.EXP,
        },
    }

    ge.broadcastAction(gameEndEvent)
}
```

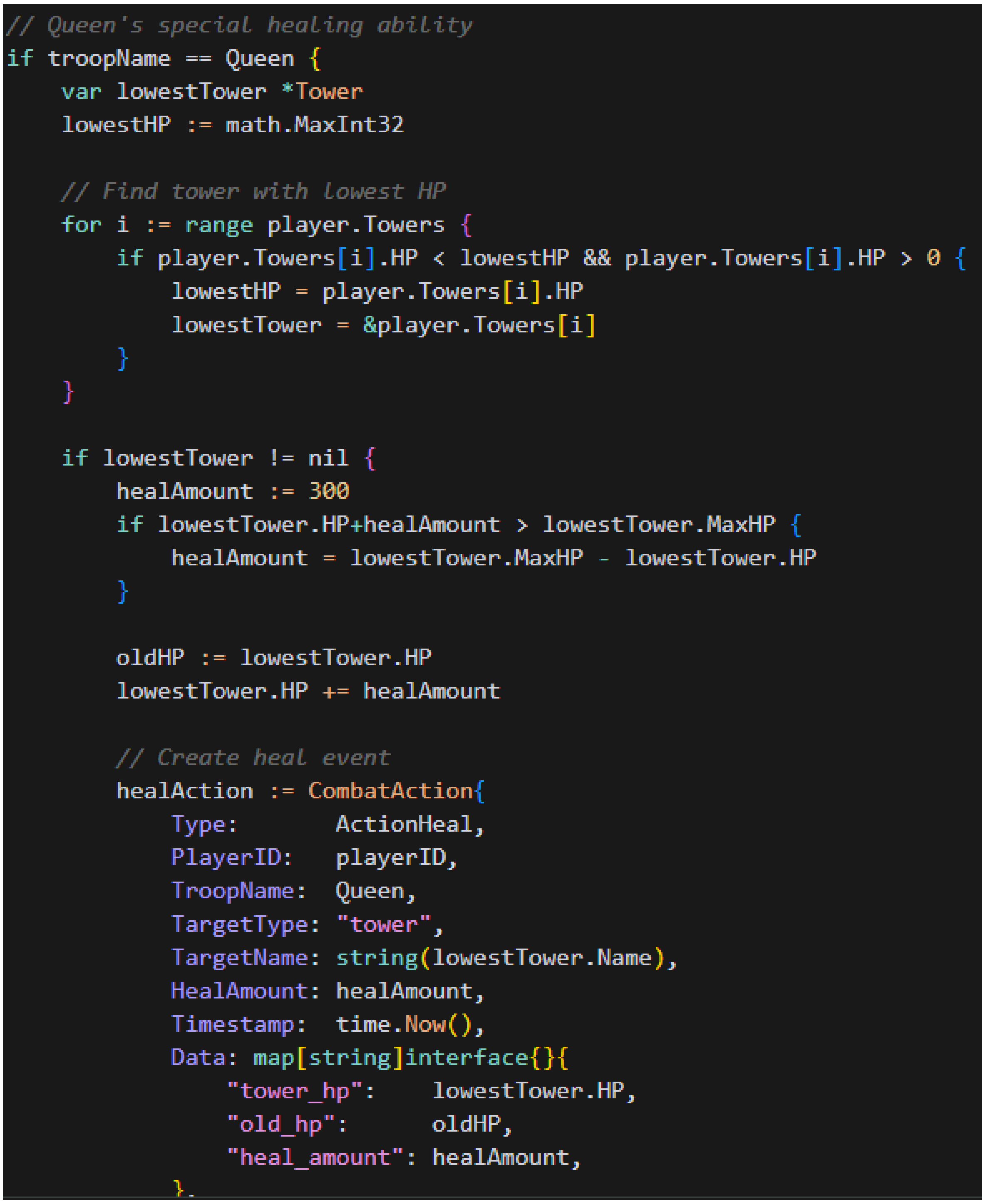
**Figure 6.37**: Game End Handling

### 6.2.5 Troop & Tower Management System

Each player controls a set of three troops that participate in combat. Troops that are destroyed during the game can be respawned, allowing players to maintain a dynamic and continuous offensive presence. Among these units, the Queen possesses a unique special ability: she can heal the friendly tower with the lowest health by restoring 300 HP, providing strategic defensive support.

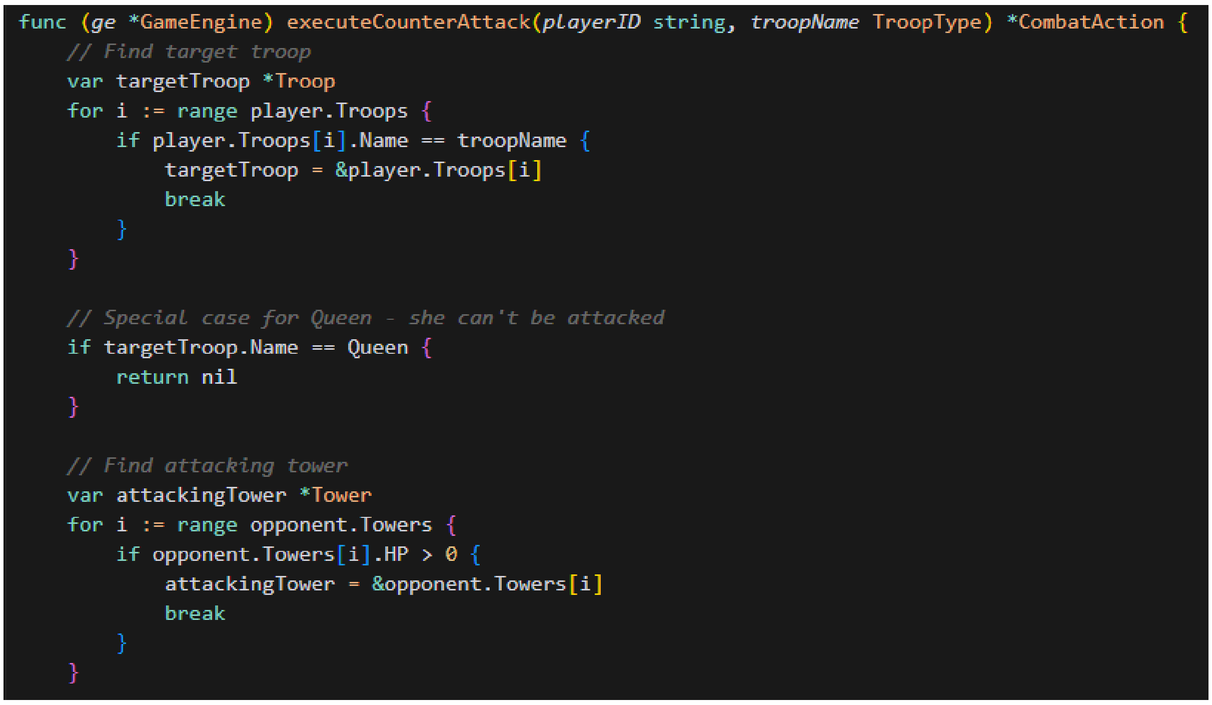
Troop statistics—namely health points (HP), attack power (ATK), and defense (DEF)—scale proportionally with the player’s level, increasing by 10% per level. However, the critical hit chance for troops remains constant regardless of level progression.

Similarly, each player's defensive lineup consists of three towers: one King Tower and two Guard Towers. The King Tower starts with 1500 HP, 400 ATK, 200 DEF, and a 10% critical hit chance. Each Guard Tower begins with 800 HP, 250 ATK, 150 DEF, and a 5% critical hit chance. Tower statistics also scale with player level, increasing by 10% per level. Additionally, towers possess the ability to counter-attack troops deployed against them, adding a defensive layer to the gameplay.



```
// Queen's special healing ability
if troopName == Queen {
    var lowestTower *Tower
    lowestHP := math.MaxInt32

    // Find tower with lowest HP
    for i := range player.Towers {
        if player.Towers[i].HP < lowestHP && player.Towers[i].HP > 0 {
            lowestHP = player.Towers[i].HP
            lowestTower = &player.Towers[i]
        }
    }

    if lowestTower != nil {
        healAmount := 300
        if lowestTower.HP+healAmount > lowestTower.MaxHP {
            healAmount = lowestTower.MaxHP - lowestTower.HP
        }

        oldHP := lowestTower.HP
        lowestTower.HP += healAmount

        // Create heal event
        healAction := CombatAction{
            Type:       ActionHeal,
            PlayerID:   playerID,
            TroopName:  Queen,
            TargetType: "tower",
            TargetName: string(lowestTower.Name),
            HealAmount: healAmount,
            Timestamp:  time.Now(),
            Data: map[string]interface{}{
                "tower_hp":    lowestTower.HP,
                "old_hp":      oldHP,
                "heal_amount": healAmount,
            },
```

**Figure 6.38**: Queen's Healing Ability

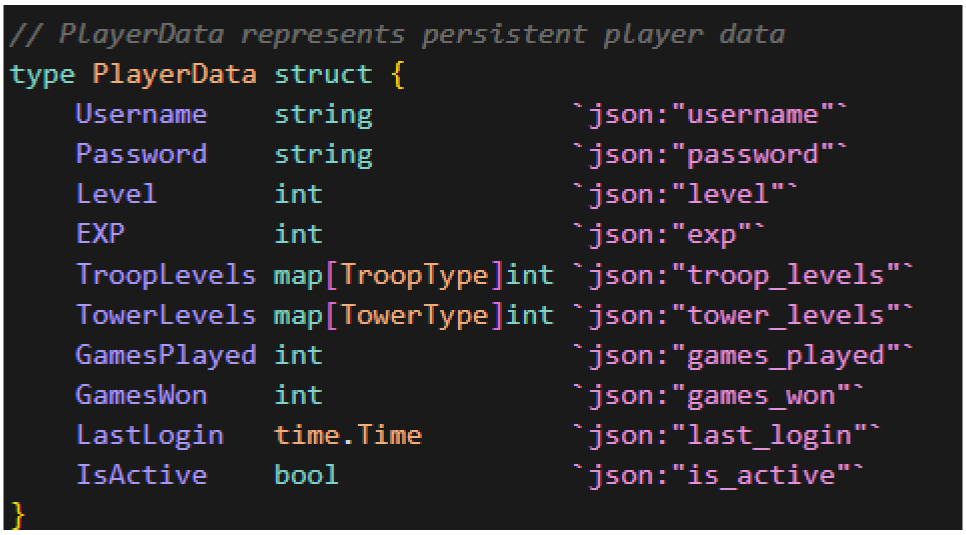


```go
func (ge *GameEngine) executeCounterAttack(playerID string, troopName TroopType) *CombatAction {
    // Find target troop
    var targetTroop *Troop
    for i := range player.Troops {
        if player.Troops[i].Name == troopName {
            targetTroop = &player.Troops[i]
            break
        }
    }

    // Special case for Queen - she can't be attacked
    if targetTroop.Name == Queen {
        return nil
    }

    // Find attacking tower
    var attackingTower *Tower
    for i := range opponent.Towers {
        if opponent.Towers[i].HP > 0 {
            attackingTower = &opponent.Towers[i]
            break
        }
    }
```

**Figure 6.39**: Tower Counter-Attacks

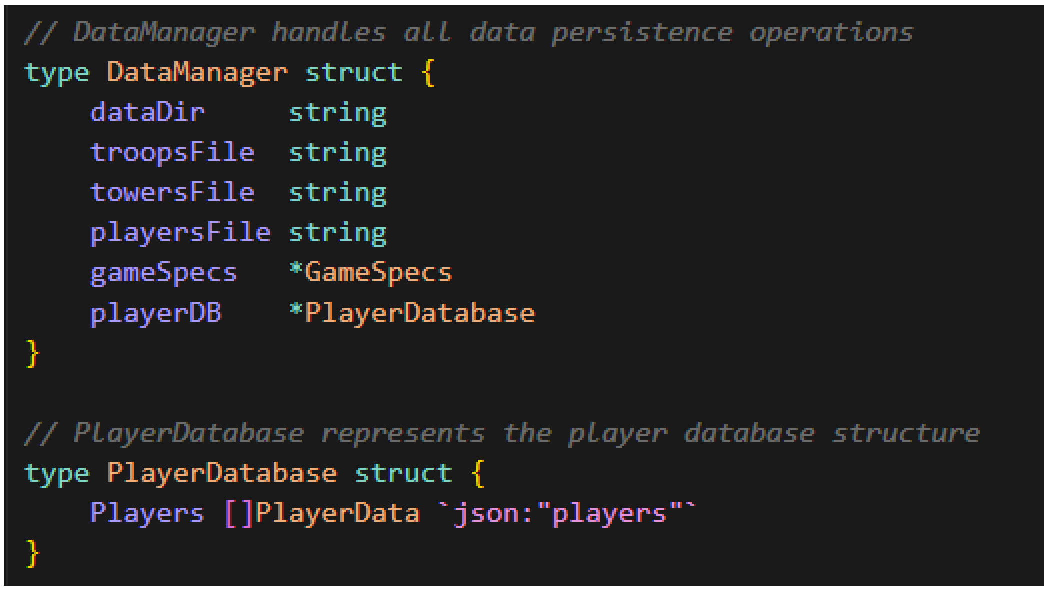
### 6.2.6 Data Persistence

Player data is persistently stored in JSON format to facilitate easy access and modification. The saved information includes essential credentials such as username and password, along with progression metrics like player level and accumulated experience points (EXP). Additionally, individual levels of troops and towers are tracked to reflect player upgrades. The data also records gameplay statistics, including the total number of games played and won, as well as the player's last login time and current active status. This comprehensive data storage ensures that player progress and game state are accurately maintained across sessions.



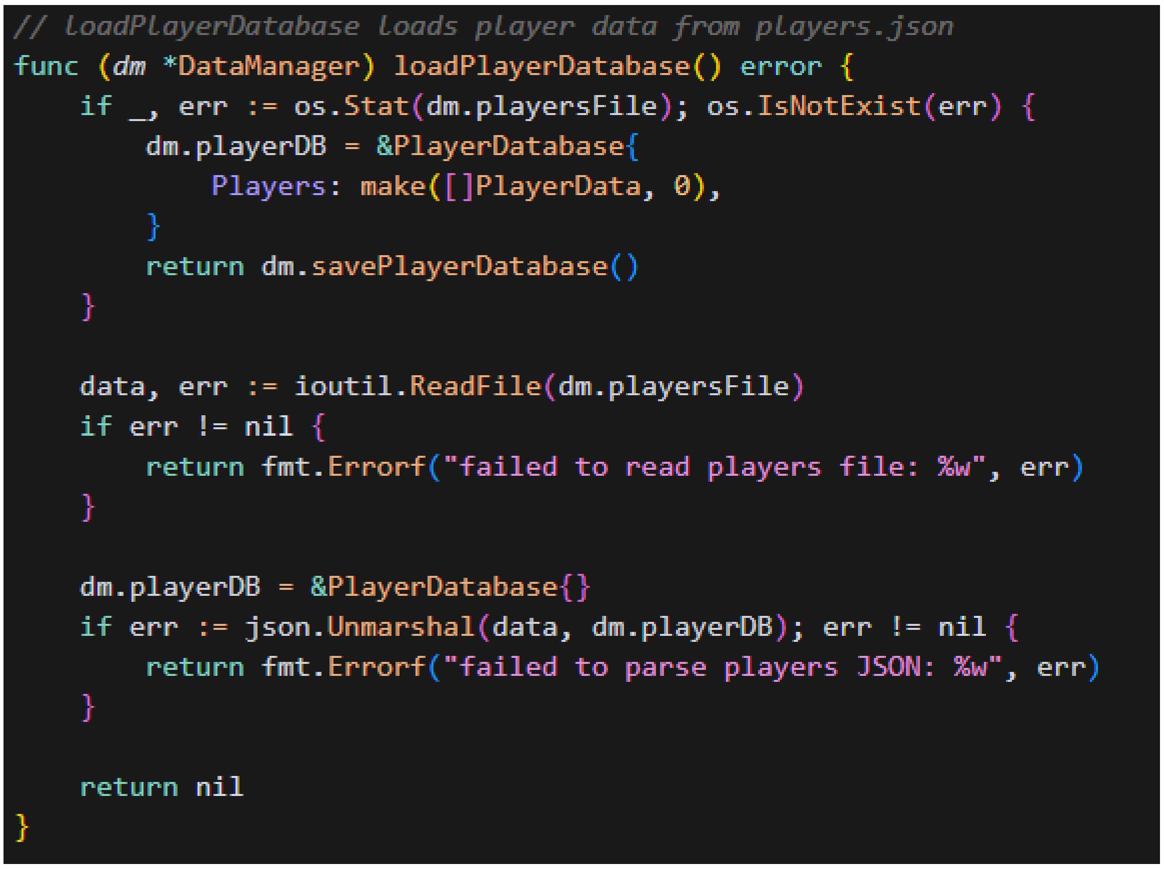
```go
// PlayerData represents persistent player data
type PlayerData struct {
    Username    string            `json:"username"`
    Password    string            `json:"password"`
    Level       int               `json:"level"`
    EXP         int               `json:"exp"`
    TroopLevels map[TroopType]int `json:"troop_levels"`
    TowerLevels map[TowerType]int `json:"tower_levels"`
    GamesPlayed int               `json:"games_played"`
    GamesWon    int               `json:"games_won"`
    LastLogin   time.Time         `json:"last_login"`
    IsActive    bool              `json:"is_active"`
}
```

**Figure 6.40**: Player Data Structure



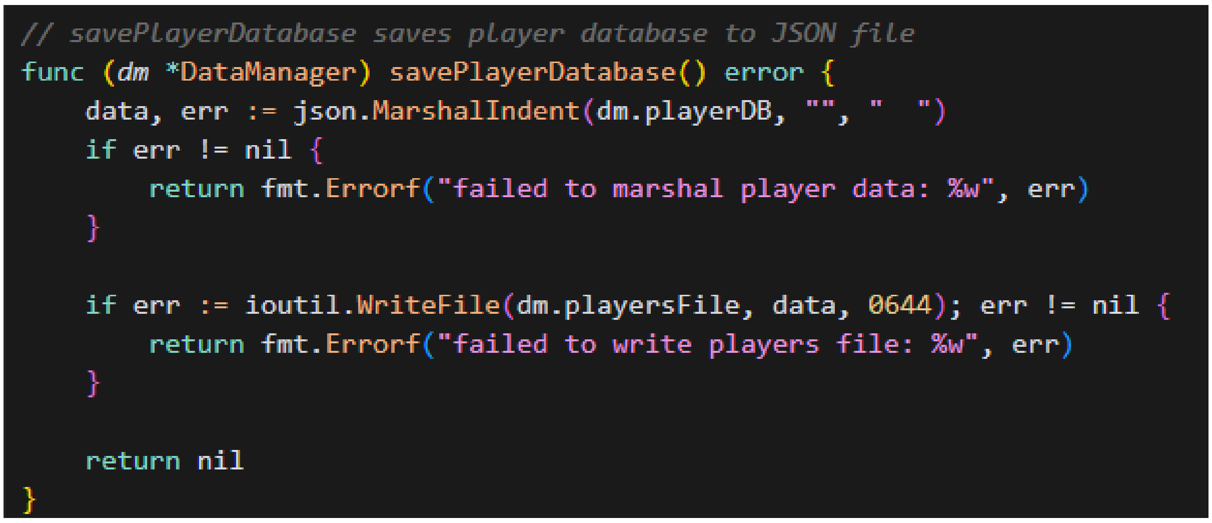
```go
// DataManager handles all data persistence operations
type DataManager struct {
    dataDir     string
    troopsFile  string
    towersFile  string
    playersFile string
    gameSpecs   *GameSpecs
    playerDB    *PlayerDatabase
}

// PlayerDatabase represents the player database structure
type PlayerDatabase struct {
    Players []PlayerData `json:"players"`
}
```

**Figure 6.41**: Data Manager Persistence



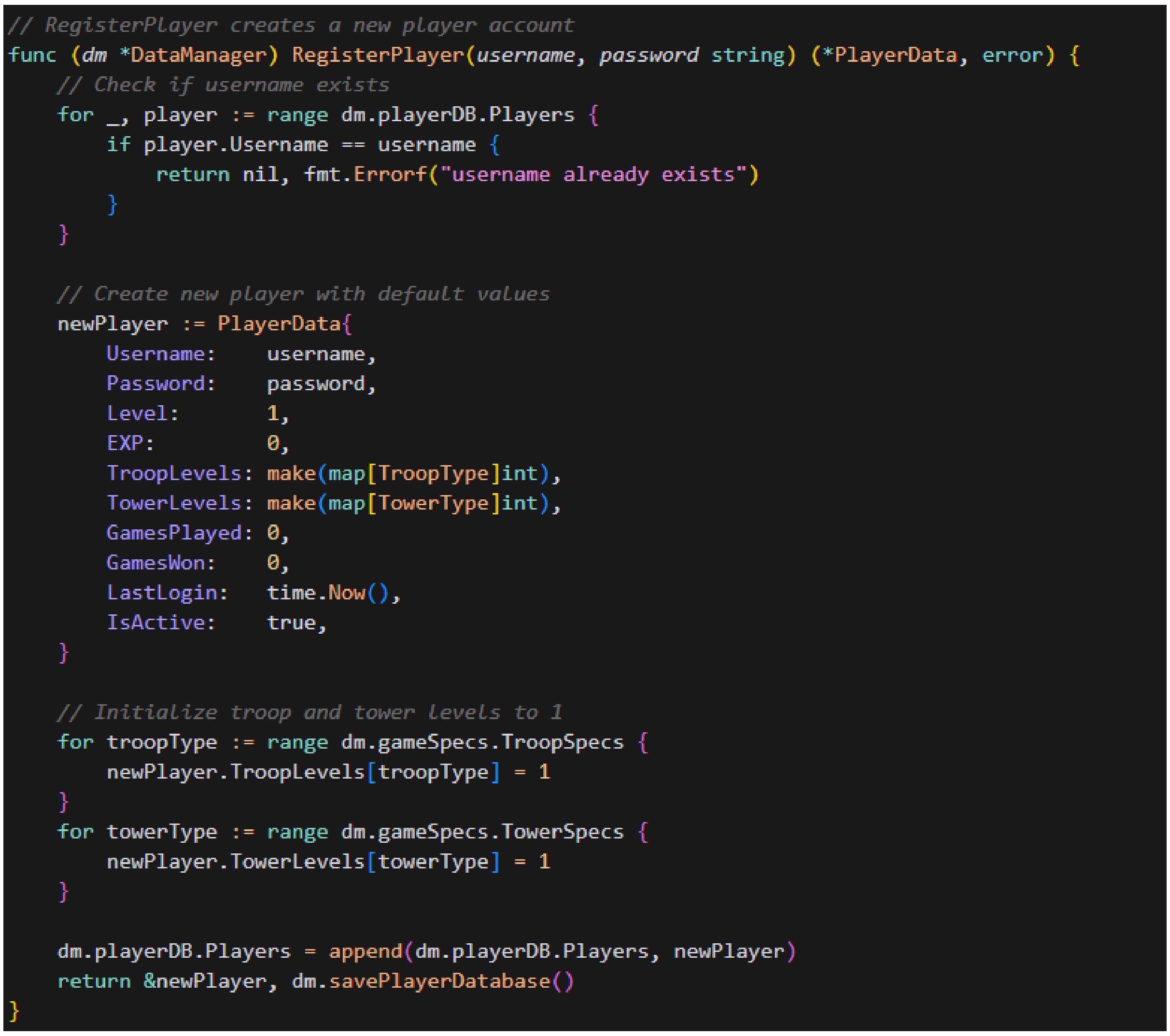
```go
// loadPlayerDatabase loads player data from players.json
func (dm *DataManager) loadPlayerDatabase() error {
    if _, err := os.Stat(dm.playersFile); os.IsNotExist(err) {
        dm.playerDB = &PlayerDatabase{
            Players: make([]PlayerData, 0),
        }
        return dm.savePlayerDatabase()
    }

    data, err := ioutil.ReadFile(dm.playersFile)
    if err != nil {
        return fmt.Errorf("failed to read players file: %w", err)
    }

    dm.playerDB = &PlayerDatabase{}
    if err := json.Unmarshal(data, dm.playerDB); err != nil {
        return fmt.Errorf("failed to parse players JSON: %w", err)
    }

    return nil
}
```

**Figure 6.42**: Loading Player Data



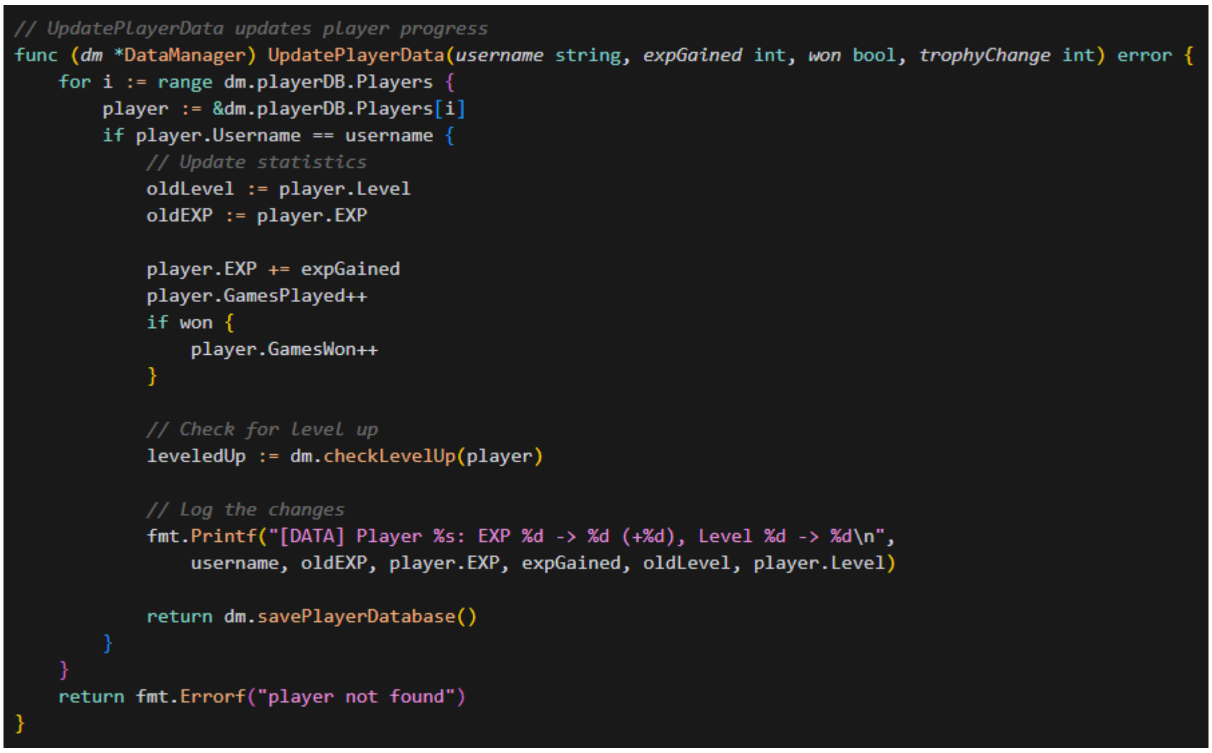
```go
// savePlayerDatabase saves player database to JSON file
func (dm *DataManager) savePlayerDatabase() error {
    data, err := json.MarshalIndent(dm.playerDB, "", "  ")
    if err != nil {
        return fmt.Errorf("failed to marshal player data: %w", err)
    }

    if err := ioutil.WriteFile(dm.playersFile, data, 0644); err != nil {
        return fmt.Errorf("failed to write players file: %w", err)
    }

    return nil
}
```

**Figure 6.43**: Save Player Data



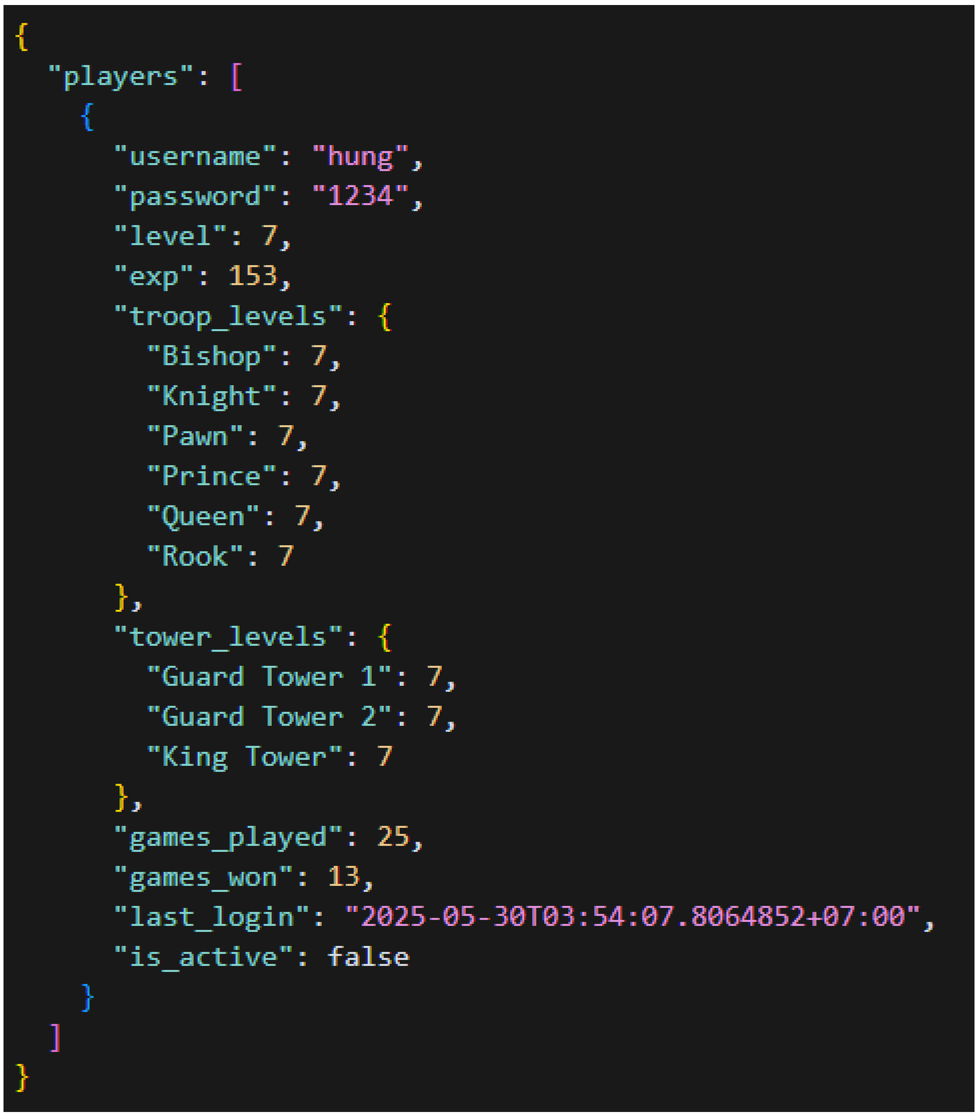
```go
// RegisterPlayer creates a new player account
func (dm *DataManager) RegisterPlayer(username, password string) (*PlayerData, error) {
    // Check if username exists
    for _, player := range dm.playerDB.Players {
        if player.Username == username {
            return nil, fmt.Errorf("username already exists")
        }
    }

    // Create new player with default values
    newPlayer := PlayerData{
        Username:    username,
        Password:    password,
        Level:       1,
        EXP:         0,
        TroopLevels: make(map[TroopType]int),
        TowerLevels: make(map[TowerType]int),
        GamesPlayed: 0,
        GamesWon:    0,
        LastLogin:   time.Now(),
        IsActive:    true,
    }

    // Initialize troop and tower levels to 1
    for troopType := range dm.gameSpecs.TroopSpecs {
        newPlayer.TroopLevels[troopType] = 1
    }
    for towerType := range dm.gameSpecs.TowerSpecs {
        newPlayer.TowerLevels[towerType] = 1
    }

    dm.playerDB.Players = append(dm.playerDB.Players, newPlayer)
    return &newPlayer, dm.savePlayerDatabase()
}
```

**Figure 6.44:** Player Registration



```go
// UpdatePlayerData updates player progress
func (dm *DataManager) UpdatePlayerData(username string, expGained int, won bool, trophyChange int) error {
    for i := range dm.playerDB.Players {
        player := &dm.playerDB.Players[i]
        if player.Username == username {
            // Update statistics
            oldLevel := player.Level
            oldEXP := player.EXP

            player.EXP += expGained
            player.GamesPlayed++
            if won {
                player.GamesWon++
            }

            // Check for level up
            leveledUp := dm.checkLevelUp(player)

            // Log the changes
            fmt.Printf("[DATA] Player %s: EXP %d -> %d (+%d), Level %d -> %d\n",
                username, oldEXP, player.EXP, expGained, oldLevel, player.Level)

            return dm.savePlayerDatabase()
        }
    }
    return fmt.Errorf("player not found")
}
```

**Figure 6.45**: Update Player Data



```json
{
  "players": [
    {
      "username": "hung",
      "password": "1234",
      "level": 7,
      "exp": 153,
      "troop_levels": {
        "Bishop": 7,
        "Knight": 7,
        "Pawn": 7,
        "Prince": 7,
        "Queen": 7,
        "Rook": 7
      },
      "tower_levels": {
        "Guard Tower 1": 7,
        "Guard Tower 2": 7,
        "King Tower": 7
      },
      "games_played": 25,
      "games_won": 13,
      "last_login": "2025-05-30T03:54:07.8064852+07:00",
      "is_active": false
    }
  ]
}
```

**Figure 6.46**: Examples Player Data in JSON

# 7. Deployment & Execution Instructions

## 7.1 Server Component

The server component is implemented as a TCP server responsible for managing core game logic, player management, and matchmaking services. To deploy and run the server, the following configuration and initialization steps are involved:

### 7.1.1 Configuration

The server accepts several command-line flags to customize its runtime behavior:

- **--port**: Specifies the TCP port on which the server listens (default: **"8080"**).
- **--host**: Defines the server host address (default: **"localhost"**).

- **--data-dir**: Sets the directory path for persistent data storage (default: **"data"**).
- **--log-level**: Configures the logging verbosity, supporting levels such as DEBUG, INFO, WARN, and ERROR.
- **--log-file**: Optional flag to specify a file path for log output.

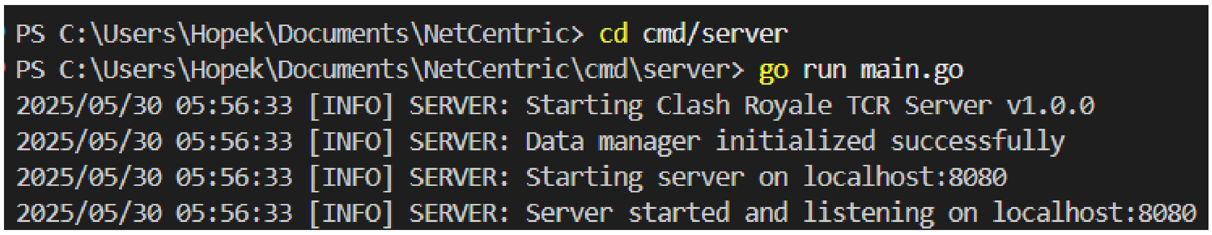
### 7.1.2 Initialization

Upon startup, the server performs the following initialization procedures:

- Instantiates a data manager responsible for player data persistence and retrieval.
- Configures the logging system based on the specified log level and optional log file destination.
- Creates a TCP listener bound to the configured host and port to accept incoming client connections.
- Initializes matchmaking queues tailored for both Simple and Enhanced game modes.

### 7.1.3 Execution

```
1. cd cmd/server
2. go run main.go
```



```
PS C:\Users\Hopek\Documents\NetCentric> cd cmd/server
PS C:\Users\Hopek\Documents\NetCentric\cmd\server> go run main.go
2025/05/30 05:56:33 [INFO] SERVER: Starting Clash Royale TCR Server v1.0.0
2025/05/30 05:56:33 [INFO] SERVER: Data manager initialized successfully
2025/05/30 05:56:33 [INFO] SERVER: Starting server on localhost:8080
2025/05/30 05:56:33 [INFO] SERVER: Server started and listening on localhost:8080
```

**Figure 7.1**: Execute the Server

During execution, the server:

- Listens for incoming client connections on the configured TCP port.
- Manages player authentication and registration workflows.
- Oversees game sessions and matchmaking operations.
- Processes in-game events and broadcasts updates to connected clients.
- Performs periodic cleanup of inactive client connections and ensures graceful shutdown procedures are followed upon termination.

## 7.2 Client

The client component serves as the user interface and communication handler, enabling players to interact with the game server. Deployment and execution follow these steps:

### 7.2.1 Configuration

The client accepts several command-line options:

- **--server**: Specifies the server address in **host:port** format (default: **"localhost:8080"**).
- **--log-level**: Sets the client-side logging level (DEBUG, INFO, WARN, ERROR).
- **--log-file**: Optional log file path for client logs.

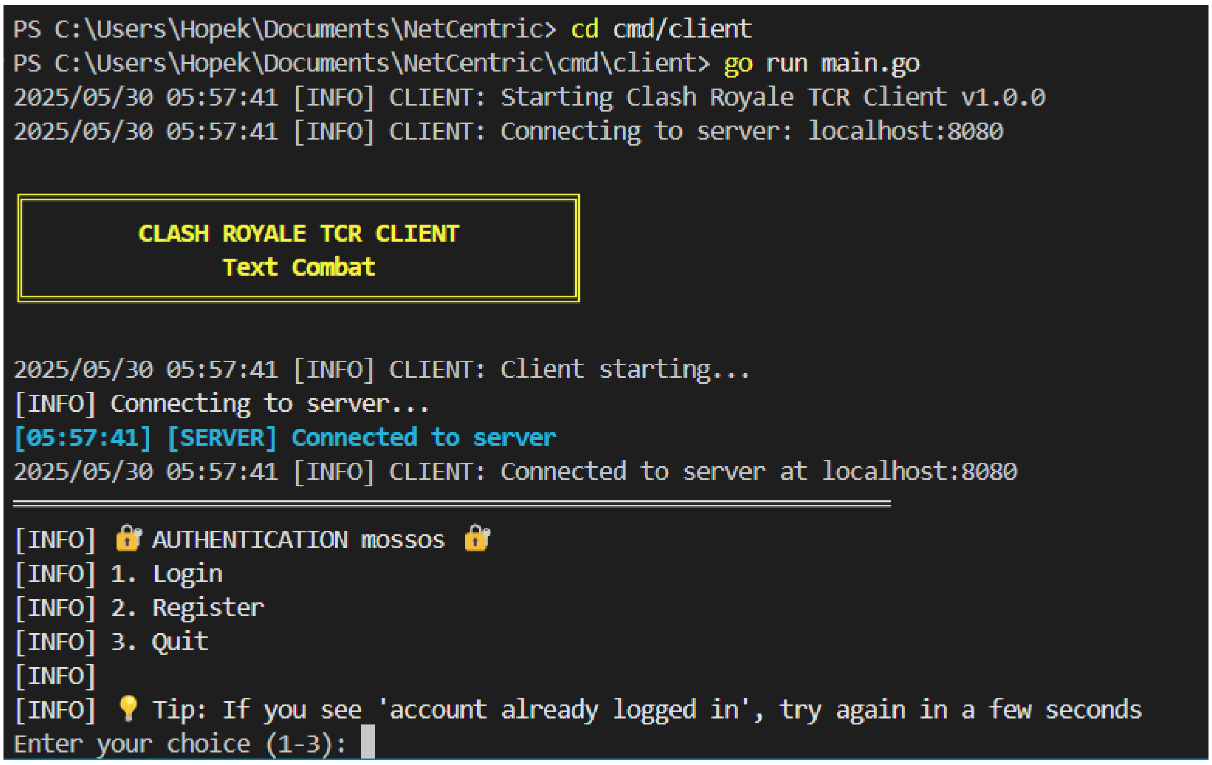
### 7.2.2 Initialization

When started, the client:

- Initializes a display system to present game information and user prompts.
- Configures logging according to the specified level and optional output file.
- Establishes a persistent TCP connection to the server at the provided address.
- Manages user authentication and maintains synchronized game state.

### 7.3.3 Execution

```
1. cd cmd/client
2. go run main.go
```

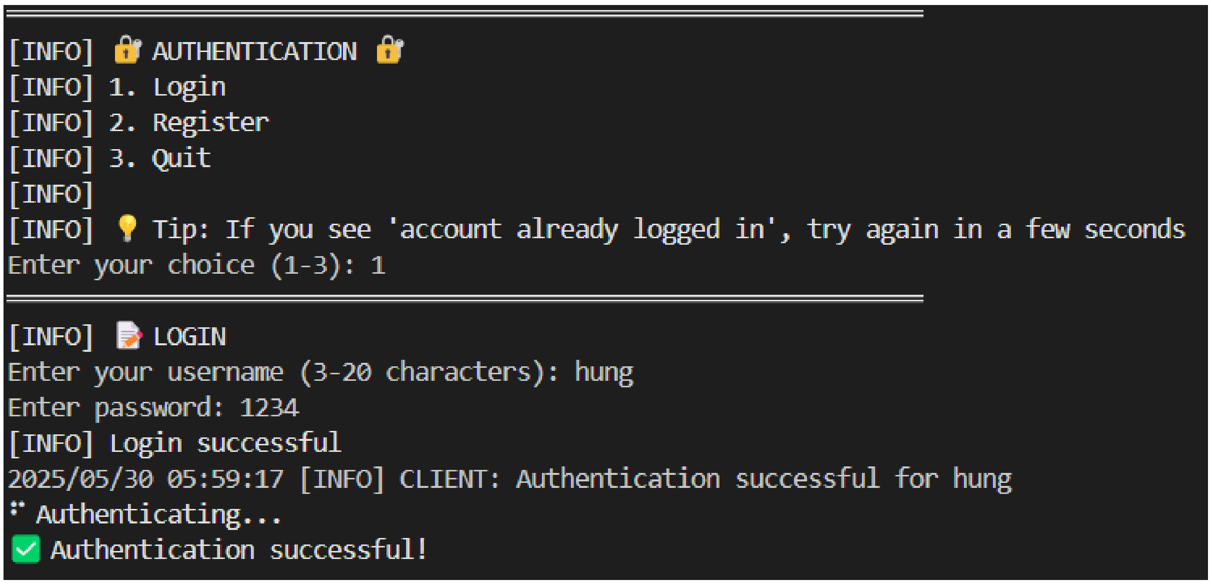


```
PS C:\Users\Hopek\Documents\NetCentric> cd cmd/client
PS C:\Users\Hopek\Documents\NetCentric\cmd\client> go run main.go
2025/05/30 05:57:41 [INFO] CLIENT: Starting Clash Royale TCR Client v1.0.0
2025/05/30 05:57:41 [INFO] CLIENT: Connecting to server: localhost:8080

        CLASH ROYALE TCR CLIENT
              Text Combat

2025/05/30 05:57:41 [INFO] CLIENT: Client starting...
[INFO] Connecting to server...
[05:57:41] [SERVER] Connected to server
2025/05/30 05:57:41 [INFO] CLIENT: Connected to server at localhost:8080
════════════════════════════════════════════════════════════════
[INFO] 🔐 AUTHENTICATION mossos 🔐
[INFO] 1. Login
[INFO] 2. Register
[INFO] 3. Quit
[INFO]
[INFO] 💡 Tip: If you see 'account already logged in', try again in a few seconds
Enter your choice (1-3):
```

**Figure 7.2**: Execute the Client

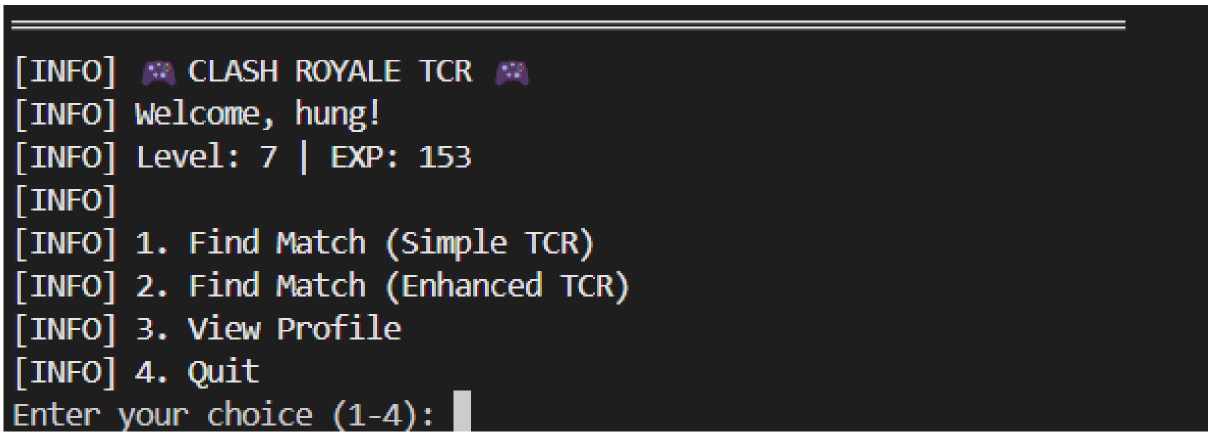
# 8. Testing & Evaluation

Login authentication was thoroughly tested using both valid and invalid credentials. This ensured that the system correctly grants access to authorized users while effectively preventing unauthorized login attempts. When 2 players starting finding the match, then match found, and start the game flow, with 4 options (summon troops, attack, show the information of match, debug for testing, end turn, and ending the game) in Simple mode



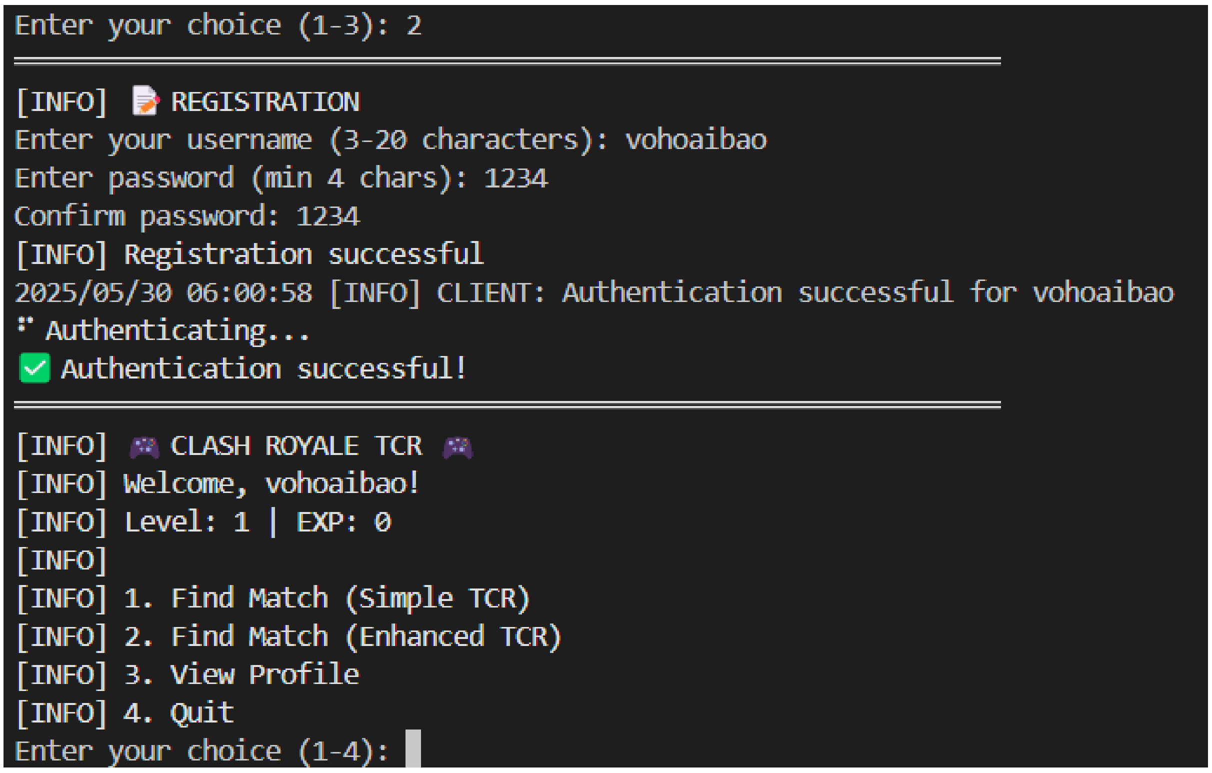
```
══════════════════════════════════════════════════
[INFO] 🔐 AUTHENTICATION 🔐
[INFO] 1. Login
[INFO] 2. Register
[INFO] 3. Quit
[INFO]
[INFO] 💡 Tip: If you see 'account already logged in', try again in a few seconds
Enter your choice (1-3): 1
══════════════════════════════════════════════════
[INFO] 📝 LOGIN
Enter your username (3-20 characters): hung
Enter password: 1234
[INFO] Login successful
2025/05/30 05:59:17 [INFO] CLIENT: Authentication successful for hung
⠋ Authenticating...
✅ Authentication successful!
```

**Figure 8.1**: Login validation testing (SUCCESS)



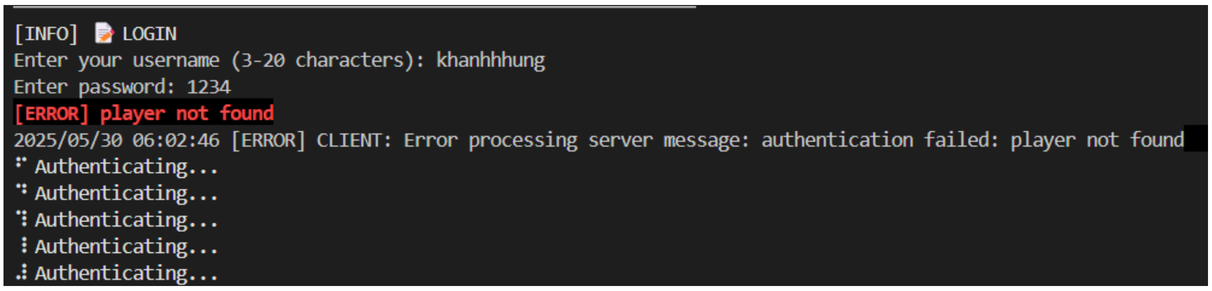
```
══════════════════════════════════════════════════
[INFO] 🎮 CLASH ROYALE TCR 🎮
[INFO] Welcome, hung!
[INFO] Level: 7 | EXP: 153
[INFO]
[INFO] 1. Find Match (Simple TCR)
[INFO] 2. Find Match (Enhanced TCR)
[INFO] 3. View Profile
[INFO] 4. Quit
Enter your choice (1-4):
```

**Figure 8.2**: Profile of Player after successfully logging in



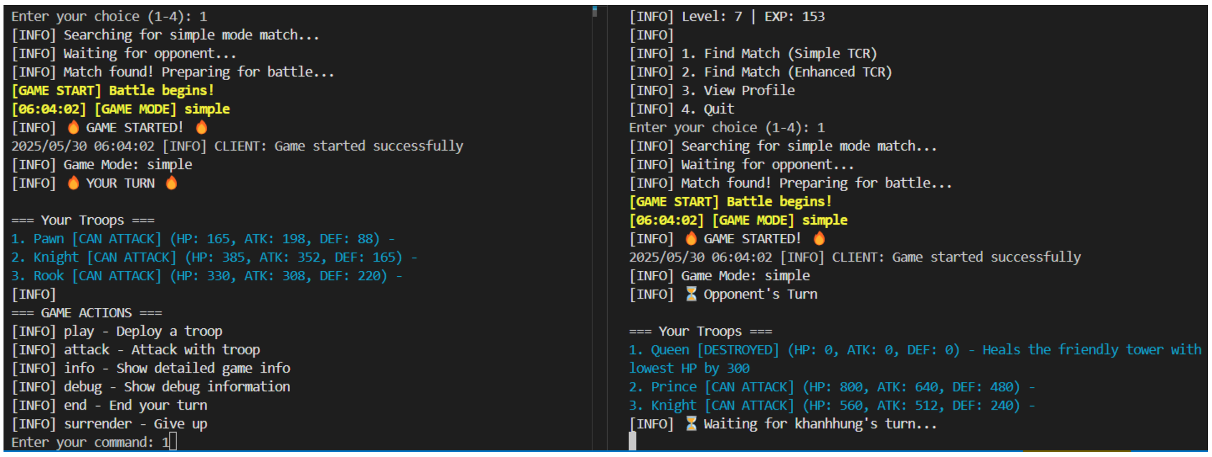
```
Enter your choice (1-3): 2
════════════════════════════════════════════════════════════
[INFO] 📝 REGISTRATION
Enter your username (3-20 characters): vohoaibao
Enter password (min 4 chars): 1234
Confirm password: 1234
[INFO] Registration successful
2025/05/30 06:00:58 [INFO] CLIENT: Authentication successful for vohoaibao
⠋ Authenticating...
✅ Authentication successful!
════════════════════════════════════════════════════════════
[INFO] 🎮 CLASH ROYALE TCR 🎮
[INFO] Welcome, vohoaibao!
[INFO] Level: 1 | EXP: 0
[INFO]
[INFO] 1. Find Match (Simple TCR)
[INFO] 2. Find Match (Enhanced TCR)
[INFO] 3. View Profile
[INFO] 4. Quit
Enter your choice (1-4):
```

**Figure 8.3**: Registration testing



```
[INFO] 📝 LOGIN
Enter your username (3-20 characters): khanhhhung
Enter password: 1234
[ERROR] player not found
2025/05/30 06:02:46 [ERROR] CLIENT: Error processing server message: authentication failed: player not found
⠋ Authenticating...
⠙ Authenticating...
⠹ Authenticating...
⠸ Authenticating...
⠼ Authenticating...
```

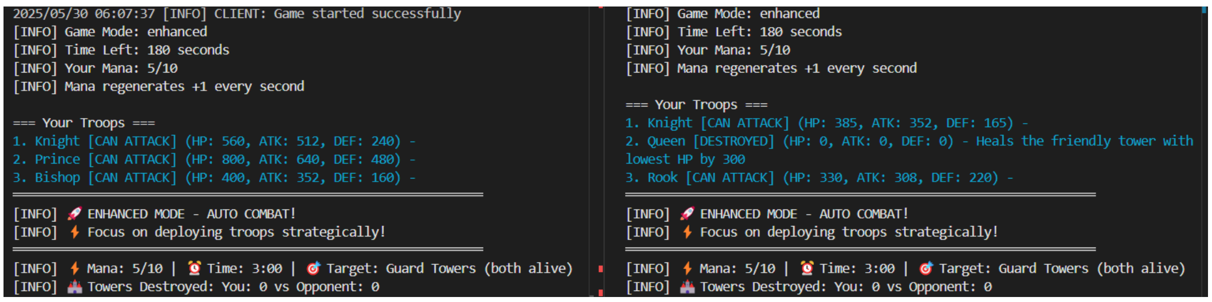
**Figure 8.4**: Login validation (FAILURE)



```
Enter your choice (1-4): 1
[INFO] Searching for simple mode match...
[INFO] Waiting for opponent...
[INFO] Match found! Preparing for battle...
[GAME START] Battle begins!
[06:04:02] [GAME MODE] simple
[INFO] 🔥 GAME STARTED! 🔥
2025/05/30 06:04:02 [INFO] CLIENT: Game started successfully
[INFO] Game Mode: simple
[INFO] 🔥 YOUR TURN 🔥

=== Your Troops ===
1. Pawn [CAN ATTACK] (HP: 165, ATK: 198, DEF: 88) -
2. Knight [CAN ATTACK] (HP: 385, ATK: 352, DEF: 165) -
3. Rook [CAN ATTACK] (HP: 330, ATK: 308, DEF: 220) -
[INFO]
=== GAME ACTIONS ===
[INFO] play - Deploy a troop
[INFO] attack - Attack with troop
[INFO] info - Show detailed game info
[INFO] debug - Show debug information
[INFO] end - End your turn
[INFO] surrender - Give up
Enter your command: 1
```

```
[INFO] Level: 7 | EXP: 153
[INFO]
[INFO] 1. Find Match (Simple TCR)
[INFO] 2. Find Match (Enhanced TCR)
[INFO] 3. View Profile
[INFO] 4. Quit
Enter your choice (1-4): 1
[INFO] Searching for simple mode match...
[INFO] Waiting for opponent...
[INFO] Match found! Preparing for battle...
[GAME START] Battle begins!
[06:04:02] [GAME MODE] simple
[INFO] 🔥 GAME STARTED! 🔥
2025/05/30 06:04:02 [INFO] CLIENT: Game started successfully
[INFO] Game Mode: simple
[INFO] ⌛ Opponent's Turn

=== Your Troops ===
1. Queen [DESTROYED] (HP: 0, ATK: 0, DEF: 0) - Heals the friendly tower with
lowest HP by 300
2. Prince [CAN ATTACK] (HP: 800, ATK: 640, DEF: 480) -
3. Knight [CAN ATTACK] (HP: 560, ATK: 512, DEF: 240) -
[INFO] ⌛ Waiting for khanhhung's turn...
```

**Figure 8.5**: Match found & Start the Game (Simple mode)



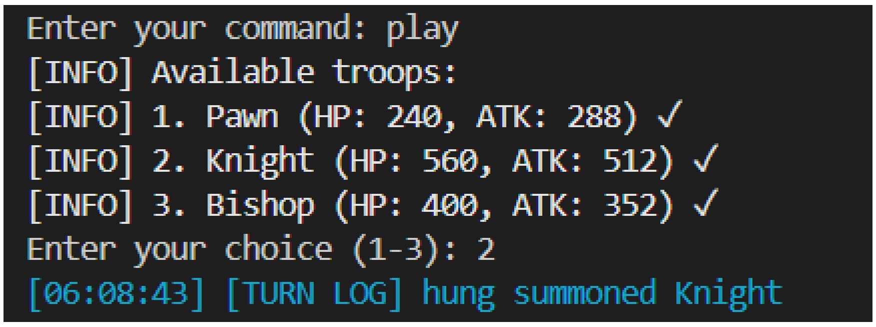
```
2025/05/30 06:07:37 [INFO] CLIENT: Game started successfully
[INFO] Game Mode: enhanced
[INFO] Time Left: 180 seconds
[INFO] Your Mana: 5/10
[INFO] Mana regenerates +1 every second

=== Your Troops ===
1. Knight [CAN ATTACK] (HP: 560, ATK: 512, DEF: 240) -
2. Prince [CAN ATTACK] (HP: 800, ATK: 640, DEF: 480) -
3. Bishop [CAN ATTACK] (HP: 400, ATK: 352, DEF: 160) -
════════════════════════════════════════════════════════════════
[INFO] 🚀 ENHANCED MODE - AUTO COMBAT!
[INFO] ⚡ Focus on deploying troops strategically!
════════════════════════════════════════════════════════════════
[INFO] ⚡ Mana: 5/10 | ⏰ Time: 3:00 | 🎯 Target: Guard Towers (both alive)
[INFO] 🏰 Towers Destroyed: You: 0 vs Opponent: 0
```

```
[INFO] Game Mode: enhanced
[INFO] Time Left: 180 seconds
[INFO] Your Mana: 5/10
[INFO] Mana regenerates +1 every second

=== Your Troops ===
1. Knight [CAN ATTACK] (HP: 385, ATK: 352, DEF: 165) -
2. Queen [DESTROYED] (HP: 0, ATK: 0, DEF: 0) - Heals the friendly tower with
lowest HP by 300
3. Rook [CAN ATTACK] (HP: 330, ATK: 308, DEF: 220) -
════════════════════════════════════════════════════════════════
[INFO] 🚀 ENHANCED MODE - AUTO COMBAT!
[INFO] ⚡ Focus on deploying troops strategically!
════════════════════════════════════════════════════════════════
[INFO] ⚡ Mana: 5/10 | ⏰ Time: 3:00 | 🎯 Target: Guard Towers (both alive)
[INFO] 🏰 Towers Destroyed: You: 0 vs Opponent: 0
```

**Figure 8.6**: Match found & Start the Game (Enhanced mode)

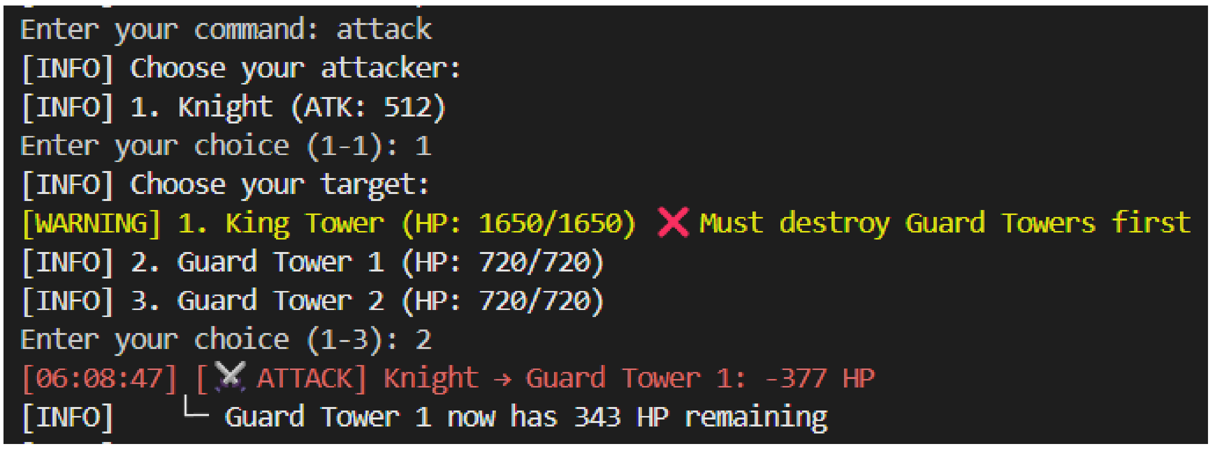
Damage calculations were verified under various troop and tower stat configurations. These tests confirmed that the combat mechanics accurately apply the intended damage formulas across different scenarios.

Both the Simple turn-based mode and the Enhanced continuous mode were simulated extensively. This allowed validation of the gameplay flow, rule enforcement, and overall system behavior in each mode.



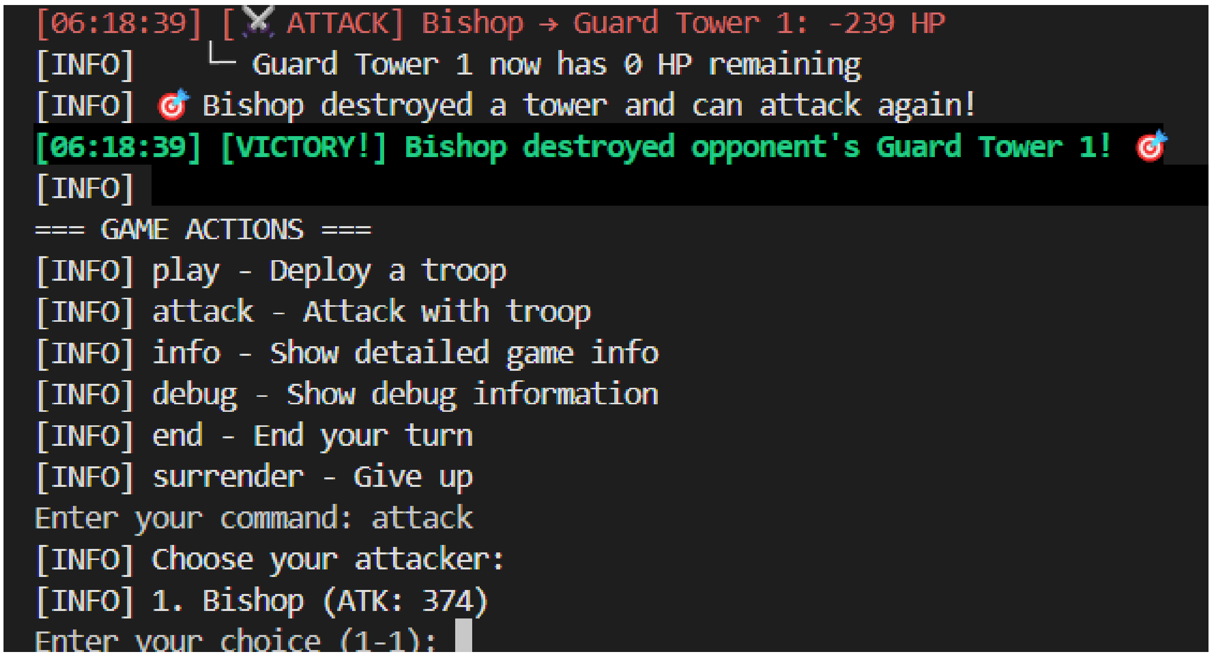
```
Enter your command: play
[INFO] Available troops:
[INFO] 1. Pawn (HP: 240, ATK: 288) ✓
[INFO] 2. Knight (HP: 560, ATK: 512) ✓
[INFO] 3. Bishop (HP: 400, ATK: 352) ✓
Enter your choice (1-3): 2
[06:08:43] [TURN LOG] hung summoned Knight
```

**Figure 8.7**: Deploying the troop



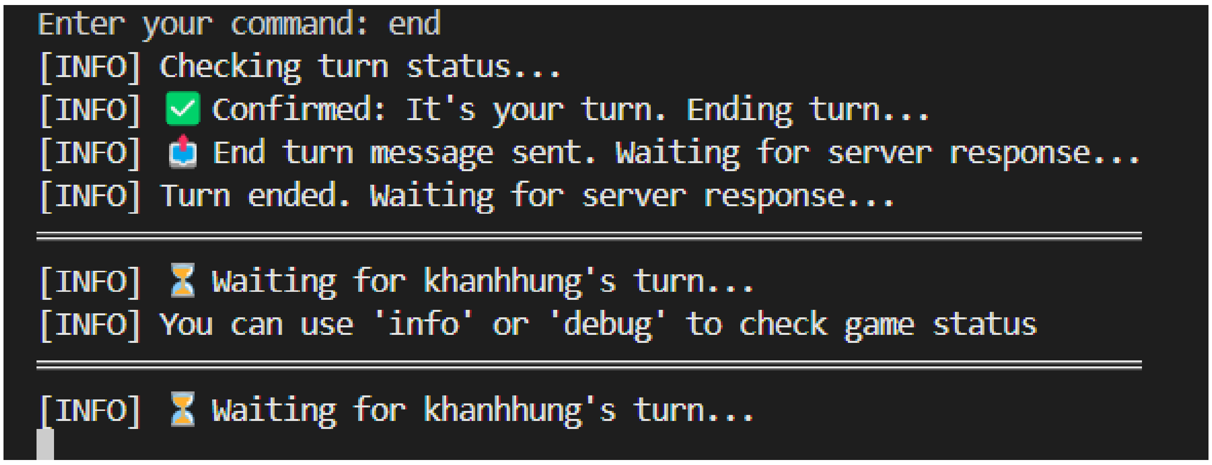
```
Enter your command: attack
[INFO] Choose your attacker:
[INFO] 1. Knight (ATK: 512)
Enter your choice (1-1): 1
[INFO] Choose your target:
[WARNING] 1. King Tower (HP: 1650/1650) ❌ Must destroy Guard Towers first
[INFO] 2. Guard Tower 1 (HP: 720/720)
[INFO] 3. Guard Tower 2 (HP: 720/720)
Enter your choice (1-3): 2
[06:08:47] [⚔️ ATTACK] Knight → Guard Tower 1: -377 HP
[INFO]    └ Guard Tower 1 now has 343 HP remaining
```

**Figure 8.8**: Combat in game



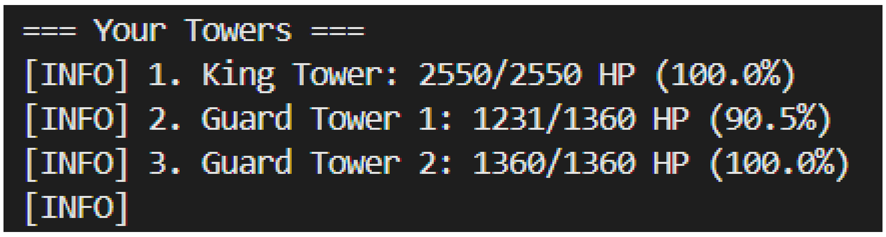
```
[06:18:39] [⚔ ATTACK] Bishop → Guard Tower 1: -239 HP
[INFO]     └ Guard Tower 1 now has 0 HP remaining
[INFO] 🎯 Bishop destroyed a tower and can attack again!
[06:18:39] [VICTORY!] Bishop destroyed opponent's Guard Tower 1! 🎯
[INFO]
=== GAME ACTIONS ===
[INFO] play - Deploy a troop
[INFO] attack - Attack with troop
[INFO] info - Show detailed game info
[INFO] debug - Show debug information
[INFO] end - End your turn
[INFO] surrender - Give up
Enter your command: attack
[INFO] Choose your attacker:
[INFO] 1. Bishop (ATK: 374)
Enter your choice (1-1):
```

**Figure 8.9**: Troop can attack once more time when destroying a tower



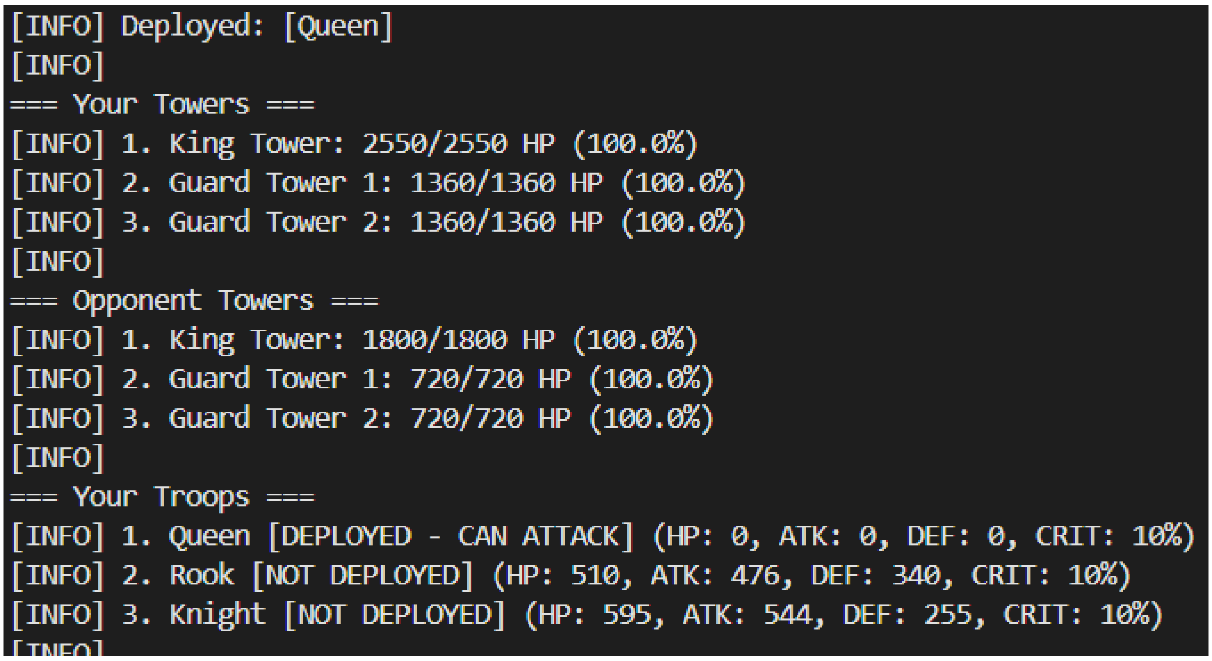
```
Enter your command: end
[INFO] Checking turn status...
[INFO] ✅ Confirmed: It's your turn. Ending turn...
[INFO] 📤 End turn message sent. Waiting for server response...
[INFO] Turn ended. Waiting for server response...
══════════════════════════════════════════════════
[INFO] ⌛ Waiting for khanhhung's turn...
[INFO] You can use 'info' or 'debug' to check game status
══════════════════════════════════════════════════
[INFO] ⌛ Waiting for khanhhung's turn...
```

**Figure 8.10**: End turn action



```
=== Your Towers ===
[INFO] 1. King Tower: 2550/2550 HP (100.0%)
[INFO] 2. Guard Tower 1: 1231/1360 HP (90.5%)
[INFO] 3. Guard Tower 2: 1360/1360 HP (100.0%)
[INFO]
```

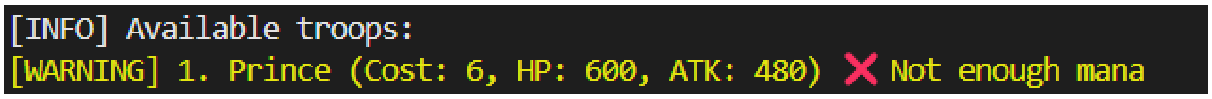
**Figure 8.11**: Case tower be attacked



```
[INFO] Deployed: [Queen]
[INFO]
=== Your Towers ===
[INFO] 1. King Tower: 2550/2550 HP (100.0%)
[INFO] 2. Guard Tower 1: 1360/1360 HP (100.0%)
[INFO] 3. Guard Tower 2: 1360/1360 HP (100.0%)
[INFO]
=== Opponent Towers ===
[INFO] 1. King Tower: 1800/1800 HP (100.0%)
[INFO] 2. Guard Tower 1: 720/720 HP (100.0%)
[INFO] 3. Guard Tower 2: 720/720 HP (100.0%)
[INFO]
=== Your Troops ===
[INFO] 1. Queen [DEPLOYED - CAN ATTACK] (HP: 0, ATK: 0, DEF: 0, CRIT: 10%)
[INFO] 2. Rook [NOT DEPLOYED] (HP: 510, ATK: 476, DEF: 340, CRIT: 10%)
[INFO] 3. Knight [NOT DEPLOYED] (HP: 595, ATK: 544, DEF: 255, CRIT: 10%)
```

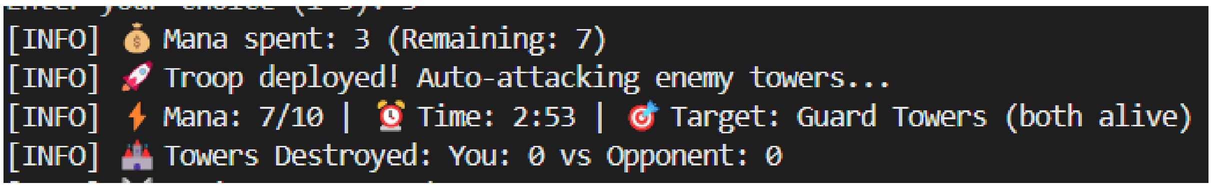
**Figure 8.12**: Queen heals the tower by 300 HP

The mana regeneration system and experience point (EXP) leveling logic were confirmed through targeted tests. These validations ensured resource accumulation and player progression operate as designed.



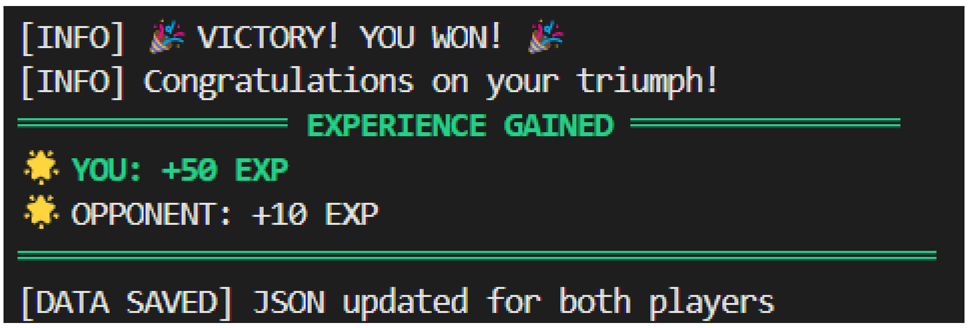
```
[INFO] Available troops:
[WARNING] 1. Prince (Cost: 6, HP: 600, ATK: 480) ❌ Not enough mana
```

**Figure 8.13**: Case enough mana for deploying the troop



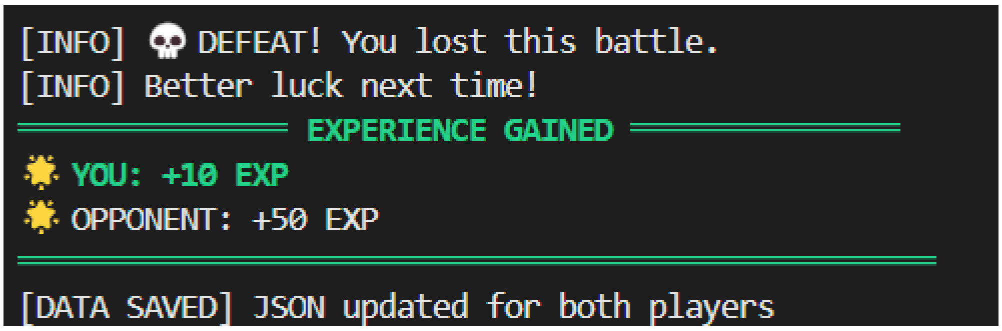
```
[INFO] 💰 Mana spent: 3 (Remaining: 7)
[INFO] 🚀 Troop deployed! Auto-attacking enemy towers...
[INFO] ⚡ Mana: 7/10 | ⏰ Time: 2:53 | 🎯 Target: Guard Towers (both alive)
[INFO] 🏰 Towers Destroyed: You: 0 vs Opponent: 0
```

**Figure 8.14**: Mana regeneration system



```
[INFO] 🎉 VICTORY! YOU WON! 🎉
[INFO] Congratulations on your triumph!
═══════════════ EXPERIENCE GAINED ═══════════════
🌟 YOU: +50 EXP
🌟 OPPONENT: +10 EXP
═════════════════════════════════════════════════

[DATA SAVED] JSON updated for both players
```

**Figure 8.15**: EXP reward after match (WIN)



```
[INFO] 💀 DEFEAT! You lost this battle.
[INFO] Better luck next time!
══════════════ EXPERIENCE GAINED ══════════════
🌟 YOU: +10 EXP
🌟 OPPONENT: +50 EXP
═══════════════════════════════════════════════

[DATA SAVED] JSON updated for both players
```

**Figure 8.16**: EXP reward after match (LOSE)

## 9. Challenges & Improvements

Synchronizing real-time game data using UDP presented significant challenges, primarily due to the inherent unreliability and potential for packet loss associated with this protocol. To address these issues, retransmission request mechanisms were implemented, allowing clients and the server to detect lost packets and request their re-sending. This approach improved data consistency and ensured smoother real-time synchronization despite the unreliable nature of UDP.

Balancing troop statistics to achieve fair and engaging gameplay required extensive trial and error. Adjustments to attributes such as health, attack power, defense, and mana costs were iteratively tested to fine-tune the relative strengths and weaknesses of various troops. This process was essential to maintain competitive balance, preventing any single unit from dominating the game while preserving strategic diversity.

Looking forward, several enhancements could further enrich the game experience. Adding a wider variety of troops would increase strategic depth and player choice. Developing a graphical user interface (GUI) would improve accessibility and visual appeal beyond the current text-based display. Additionally, integrating AI-controlled opponents could provide practice opportunities and single-player gameplay options, expanding the game's versatility and user engagement.

## 10. Conclusion

This project offered a valuable hands-on experience in applying networking protocols within the context of a real-time multiplayer game. By integrating both TCP and UDP communication alongside concurrent programming techniques in Golang, the development process reinforced a practical understanding of net-centric programming principles. The implementation of the experience (EXP) and leveling system introduced persistent player progression, enhancing player engagement and replayability. Overall, this project successfully combined network communication, game logic, and state management to deliver a functional and extensible multiplayer game framework.

# Appendix

### Tower Stats

| Type | HP | ATK | DEF | CRIT | EXP |
|---|---|---|---|---|---|
| King Tower | 2000 | 500 | 300 | 10% | 200 |
| Guard Tower | 1000 | 300 | 100 | 5% | 100 |

### Troop Stats

| Name | HP | ATK | DEF | MANA | EXP | Special |
|---|---|---|---|---|---|---|
| Pawn | 50 | 150 | 100 | 3 | 5 | |
| Bishop | 100 | 200 | 150 | 4 | 10 | |
| Rook | 250 | 200 | 200 | 5 | 25 | |
| Knight | 200 | 300 | 150 | 5 | 25 | |
| Prince | 500 | 400 | 300 | 6 | 50 | |
| Queen | N/A | N/A | N/A | 5 | 30 | Heals friendly tower 300 |

# References

1. Go Documentation. (2024). The Go Programming Language. https://go.dev/doc/
2. Go Wiki. (2024). Concurrency Patterns. https://github.com/golang/go/wiki/Concurrency
3. JSON.org. (2024). Introducing JSON. https://www.json.org/
4. TechTarget. (2025). PDU Layer Implementation. https://www.techtarget.com/searchnetworking/definition/protocol-data-unit-PDU
5. Go Modules. (2024). https://go.dev/doc/modules/managing-dependencies
6. Go Networking. (2024). https://go.dev/doc/effective_go#networking
7. Supercell. (2024). Clash Royale - Epic Real-Time Card Battles. https://supercell.com/en/games/clashroyale/
8. Linode. (2025). Developing UDP and TCP Clients and Servers in Go. https://www.linode.com/docs/guides/developing-udp-and-tcp-clients-and-servers-in-go/