

# CMPT 371 Final Project

Spring 2025, Prof. Shervin Shirmohammadi

GitHub Repository: <https://github.com/jnn4/CMPT-371-FP>

Game Demo: <https://youtu.be/evRVT6tB6cI>

*Group 19*

Bianca Dimaano 301550989

Jaycie Say 301584508

Jennifer Huang 301584324

Quang Anh Pham 301576174

# Overview

## Game Concept

Onigiri Wars is a multiplayer game where players control a character that can move across a grid of squares. Each player has a unique color and ID, and they compete by claiming squares on the grid. The player who controls the most squares at the end of the game wins. The game features a client-server architecture, where the server manages the game state and handles communication with connected clients. Each client is connected to the server through a socket, and communication is handled via messages that specify player actions, game updates, and other relevant information.

## Game Flow

- *Initialization*: Players connect to the server and are assigned unique IDs and starting positions on the grid.
- *Lobby*: Players join a lobby after connecting to the server. When all players are ready, they are then loaded into the actual game.
- *Movement*: Players move across the grid by sending movement commands to the server. The server processes the movements and updates the grid.
- *State Updates*: The server broadcasts state updates: when a player joins, when a player is ready to start, player movements throughout the game, etc.
- *End Game*: The game ends when all the squares are claimed by the players, and the player with the most filled-in squares wins. Each player's score will be displayed in the final scoreboard at the end of the game.

## Design Details

The code follows the MVC design pattern, where we have separated the logic between the Model, View, and Controller. We have also implemented the Observer design pattern, where multiple ClientHandler objects (acting as observers) register with the GameServer to receive real-time updates across each connected client. This ensures decoupling between all our components for easier scalability and flexibility.

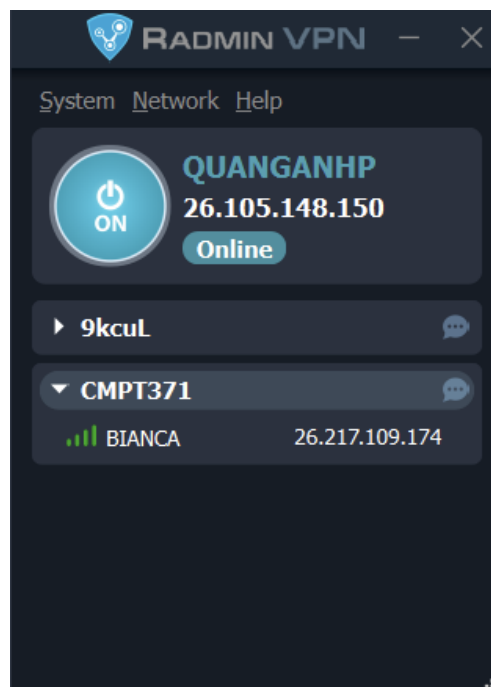
The Model represents the core logic of the game. It is organized into distinct classes that each handle a specific aspect of the game's functionality, such as managing the grid, tracking players, and controlling shared resources. The classes Grid, Player, and Square are responsible for maintaining the game state and defining the rules of interaction. These classes encapsulate the logic that controls gameplay and manages the state of all players connected to the game.

Following this, the View is implemented through the GameGUI class, which renders the game user interface and allows players to interact with the game. The GUI listens for user inputs (e.g., directional keyboard presses) and communicates these actions to the GameClient.

Finally, the Controller is distributed across the client and server sides. On the client side, the GameClient class acts as the main coordinator between the user interface and the network; It sends player actions to the server, and updates the GUI accordingly. Next, the GameServer class maintains the overall game state, where it synchronizes the game actions between the client and server. Each client is managed by a ClientHandler class, which listens for inputs from its respective client and relays updates back to the server and other connected players.

## Game Demonstration: Starting the Server

Due to physical restraints, we were unable to get multiple players onto the same LAN network. To circumvent this, we employed the use of RadminVPN.



*Fig. 1. RadminVPN*

One player acts as the “server”, hosting a virtual private network (“CMPT371”) for other players to join in so that they can play the game as if everyone is on the same LAN. Afterward, one of the players can be chosen as the one hosting the GameServer, and others can replace the server’s IP address with that player’s address to be able to join in the game.

In the demo video, 26.105.148.150 was designated as the “server”, and another player connected to the GameServer by connecting to socket 26.105.148.150:12345. Messages (“System.out[...]”) in the terminals are also left in for demonstration purposes of the locks.

## Application Layer Messaging Scheme

Communication between the server and clients is established using raw sockets. The custom application-layer messaging scheme is built using string-based messages passed between the server and clients. Messages are formatted as comma-separated values, with each part representing specific information (e.g. player ID, coordinates, color, ...).

For example:

```
Server: ASSIGN_PLAYER,P1,0,0,#f0adc6
Server: PLAYER_JOINED,P1,0,0,#f0adc6
Added Player: P1 at (0,0)
Player is ready
Server: LOBBY_STATE,P1,READY;
```

- "PLAYER\_JOINED, P1, 0, 0, #FF0000": Player has joined with Player ID P1 at starting position (0,0) and their unique color is red.

```
Server: COUNTDOWN,3
Server: COUNTDOWN,2
Server: COUNTDOWN,1
Server: GAME_STARTED
Server: PLAYER_MOVED,P1,0,0,#f0adc6
```

- "GAME\_STARTED: The game has started.
- "MOVE, P1, 2, 3": Player P1 moves to coordinates (2,3).

These messages can be sent by either the server or the clients to update the game status and ensure synchronization (see Fig. 2.1 for the client’s side of processing messages from the server). For the sake of clarity, messages from the server and messages from the clients have different names, even if they serve similar functionalities.

The game is set up by the run method on the server-side, in ClientHandler (see Fig. 2.2). After a player is added, they are assigned an ID, color, and positions, before being added into the game. This event is also broadcast to all clients, so that their GUI can load in this new player in the lobby, and later on in the game board.

```

/**
 * Processes a single message received from the server.
 * @param message The raw message from the server
 */
private void processServerMessage(String message) {
    String[] parts = message.split(",");
    String command = parts[0];

    // Handle different server commands
    switch (command) {
        case "ASSIGN_PLAYER":
            handlePlayerAssignment(parts);
            break;

        case "GAME_STARTED":
            SwingUtilities.invokeLater(() -> {
                if (gui != null && playerId == null) {
                    System.err.println("Game started but player ID not
assigned!");
                    return;
                }
                gui.startGame();
            });
            break;

        // ...

        case "MOVE_CONFIRMED":
            if (gui != null && parts.length >= 4) {
                String confirmedPlayerId = parts[1];
                int newX = Integer.parseInt(parts[2]);
                int newY = Integer.parseInt(parts[3]);
                gui.onMoveConfirmed(confirmedPlayerId, newX, newY);
            }
            break;

        // ... Other code omitted ... See repository for more.
    }
}

```

*Fig. 2.1. processServerMessage method in GameClient.java*

```

/**
 * The main execution method for the ClientHandler thread.
 * Listens for incoming messages from the client, processes them, and responds accordingly.
 * Handles client connection, player assignment, game state updates, and message processing.
 */
@Override
public void run() {
    try {
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream(), true);

        // Check if the server has space for more players
        if (gameServer.getPlayerCount() > gameServer.getMaxPlayers()) {
            sendMessage("SERVER_FULL");
            socket.close();
            return;
        }

        // Assign a player ID and position, then initialize the player
        String playerId = "P" + gameServer.getNextPlayerId();
        int[] startPos = getCornerPosition(playerId);
        String playerColor = getCornerColor(playerId);
        this.player = new Player(playerId, startPos[0], startPos[1], playerColor);

        // Add player to the game and lock the initial position
        gameServer.addPlayer(player);
        gameServer.getGrid().getSquare(startPos[0], startPos[1]).tryLock(player);

        // Send player assignment and broadcast player join
        sendMessage("ASSIGN_PLAYER," + playerId + "," + startPos[0] + "," + startPos[1] + "," +
playerColor);
        gameServer.broadcast("PLAYER_JOINED," + playerId + "," + startPos[0] + "," + startPos[1] +
"," + playerColor);

        // ... Other code omitted ... See repository for more.
    }
}

```

*Fig. 2.2. run method in ClientHandler.java*

# Client-Server Communication & Synchronization

## Opening Sockets

In the server, we initialize a `ServerSocket` (Fig. 3) with a unique port number to listen for incoming client connections. When a client connects, the server accepts the connection and creates a `ClientHandler` thread to handle the communication.

```
/**
 * Starts the server and listens for incoming client connections.
 * Once a client is connected, a new ClientHandler thread is created for communication.
 *
 * @param args command-line arguments (not used).
 */
public static void main(String[] args) {
    try (ServerSocket serverSocket = new ServerSocket(PORT)) {
        System.out.println("Maze Game Server started on port " + PORT);

        // Instantiate GameServer
        GameServer gameServer = new GameServer();

        // Continuously accept new client connections
        while (true) {
            Socket clientSocket = serverSocket.accept();
            // Pass the GameServer instance as an observer
            ClientHandler clientHandler = new ClientHandler(gameServer, clientSocket);

            synchronized (clients) {
                clients.add(clientHandler);
            }

            new Thread(clientHandler).start();
        }
    } catch (BindException e) {
        System.err.println("Error: Port " + PORT + " is already in use. Please use a different port.");
        System.exit(1);
    } catch (IOException e) {
        System.err.println("Error: An unexpected I/O error occurred.");
        e.printStackTrace();
        System.exit(1);
    }
}
```

*Fig. 3. main method from GameServer.java*


The `ServerSocket` listens for incoming connections on a specified port. When a client connects, a new `ClientHandler` is instantiated, and a new thread is started to manage the client's communication.

## Handling Shared Object

The game state, including the grid, players and game rules, is shared by the server to all clients. The GameServer class manages this shared state and uses synchronization to ensure thread safety between multiple clients interacting with it.

Here we have the movePlayer function that attempt to move player onto a square. If that square is unclaimed (this is checked inside the tryLock(player) method from Fig. 5), the player would claim that square before being moved onto it. However, if that square is already claimed, an error message would be printed out in the terminal, and the player would remain where they were instead of on the new square.

By doing the check before the player can officially move, the game makes sure that there can be no illegal moves from anyone at any time.



```
/**
 * Moves a player to a new position on the grid.
 *
 * This method attempts to move a player to the specified coordinates on the grid.
 * If the move is successful, the player's position is updated, and the grid square is locked.
 * The game state is updated accordingly, and observers are notified of the change.
 *
 * @param playerId the ID of the player.
 * @param newX the new X-coordinate on the grid.
 * @param newY the new Y-coordinate on the grid.
 * @return true if the move is successful, false otherwise.
 */
@Override
public synchronized boolean movePlayer(String playerId, int newX, int newY) {
    Player player = players.get(playerId);
    if (player == null) {
        return false;
    }

    Square square = grid.getSquare(newX, newY);

    if (square.tryLock(player)) {
        player.setX(newX);
        player.setY(newY);
    }

    return player.move(newX, newY, grid);
}
```

*Fig. 4. movePlayer method from GameServer.java*



```

/**
 * Attempts to lock the square for the given player.
 *
 * @param player the player attempting to claim the square
 * @return true if the lock was acquired, false otherwise
 */
public boolean tryLock(Player player) {
    if (player == null || isWall) return false;

    try {
        if (lock.tryLock(LOCK_TIMEOUT_MS, TimeUnit.MILLISECONDS)) {
            owner = player;
            updateLabelForLock(player);
            System.out.printf("Square at [%d,%d] locked by %s\n",
                              getX(), getY(), player.getId());
            return true;
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }

    System.out.printf("Failed to lock square at [%d,%d] (locked by %s)\n",
                      getX(), getY(), (owner != null ? owner.getId() : "unknown"));
    return false;
}

```

*Fig. 5. tryLock from Square.java*

```

/**
 * Broadcasts a message to all connected clients.
 *
 * This method sends a message to all registered observers, typically used to notify
 * players about events like player movements, game status updates, or other game events.
 *
 * This method is synchronized to ensure thread-safety when broadcasting messages to multiple
 * clients simultaneously.
 *
 * @param message The message to be broadcast to all observers.
 */
@Override
public synchronized void broadcast(String message) {
    synchronized (clients) {
        for (ClientHandler client : clients) {
            client.sendMessage(message);
        }
    }
}

```

*Fig. 6. Broadcasting Game State Updates from GameServer.java*

The broadcast method ensures all connected clients receive real-time updates about the game state from the server. Any methods from the server that require communication with the clients call this method. This is essential for maintaining synchronization between the server and all clients.

## **Conclusion**

Onigiri Wars fulfills the game and technical requirements by implementing a fully functional client-server architecture using raw sockets and custom application-layer messaging. Each player runs as a client connecting to a central server, which can be started by any player. The server manages the core game state and handles real-time updates using the Observer design pattern to broadcast changes to all connected clients. Shared resources, in our case, grid squares, are synchronized using locking mechanisms to ensure only one player interacts with them at a time. This setup enables consistent gameplay, proper concurrency control, and a scalable foundation for multiplayer interaction.

## **Group Contributions**

Bianca Dimaano: 25%

Jaycie Say: 25%

Jennifer Huang: 25%

Quang Anh Pham: 25%