**GoMoKu (Connect-5) Game Design Document**

**1. Introduction**

GoMoKu (Connect-5) is an online chess-like strategy game played on a 10×10 grid. The objective is to align five stones vertically, horizontally, or diagonally. This document outlines the design and implementation of the GoMoKu game, focusing on system architecture, client-server communication, support for multiple concurrent players, and key design considerations.

**2. System Architecture**

The GoMoKu game is built on a **client-server model** to facilitate real-time multiplayer gameplay. The architecture comprises:

* **Clients:** Implemented as Java applets, clients serve as the graphical user interface (GUI) for players. They handle user interactions, validate moves, and display the game state.
* **Server:** A Java-based server manages game sessions, player matchmaking, move synchronization, and game state validation. It ensures efficient communication between clients and oversees the overall game logic.

**Thin Server and Fat Client Paradigm**

* **Server Responsibilities:**
  + **Matchmaking:** Assigns players to game sessions.
  + **Message Transmission:** Facilitates communication between clients.
  + **Game State Management:** Maintains and updates the game state.
  + **Concurrency Handling:** Supports multiple game sessions simultaneously.
* **Client Responsibilities:**
  + **User Interface:** Renders the game board, player names, and results.
  + **Move Validation:** Checks for legal moves before sending them to the server.
  + **Win Detection:** Determines if a player has achieved five stones in a row.

A screenshot of a computer

Description automatically generated

Figure 2: Graphical User Interface (GUI) of the GoMoKu game, displaying player information and the 10x10 game board.

**3. Client-Server Communication**

Communication between the client and server is established via **Stream Sockets**. The protocol involves sending and receiving predefined message types to synchronize game states and handle user actions.

**Message Types**

* **Messages Received by Server:**
  + START: Initiates a new game session.
  + WIN: Indicates a player has won the game.
  + REGRET\_REQUEST: A player requests to undo a move.
  + REGRET\_OK: Confirmation to proceed with undoing a move.
  + TURN\_OVER: Indicates the end of a player's turn.
  + DOG\_FALL: Represents a disconnection or game termination.
* **Messages Sent by Server:**
  + END\_GAME: Notifies clients that the game has concluded.
  + YOUR\_TURN: Signals a client that it's their turn to make a move.
  + CAN\_REGRET: Grants permission to undo the last move.
  + REJECT\_REGRET: Denies a regret request.
  + REQUEST\_START: Prompts clients to begin a new game.

**Communication Flow**

1. **Connection Establishment:** Clients connect to the server using the server's IP address and designated port (8888).
2. **Game Initialization:** Upon connection, clients send a START message to join a game session.
3. **Gameplay Synchronization:** Clients send move data to the server, which validates and broadcasts the move to the opponent.
4. **Win Detection:** The server evaluates the game state after each move to determine if a player has won.
5. **Game Termination:** The server sends an END\_GAME message upon game conclusion, notifying both clients.

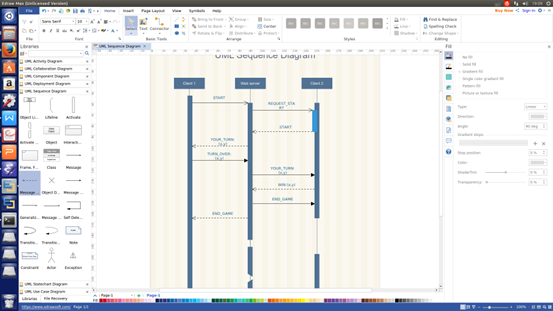


Figure 1: UML Sequence Diagram illustrating the interaction between Client 1, Web Server, and Client 2 during a GoMoKu game session.

**4. Support for Multiple Concurrent Players**

The server is designed to handle multiple game sessions concurrently, ensuring scalability and efficient resource management.

**Data Structures**

* **Match List:** Maintains active game sessions, each pairing two clients.
* **Vacancies List for Waiting:** Keeps track of clients waiting to be matched.
* **Vacancies List is Idle:** Manages idle resources and available slots for new game sessions.

**Concurrency Handling**

* **Thread Management:** Each game session is managed in a separate thread to allow simultaneous gameplay without interference.
* **Synchronization:** Critical sections of code are synchronized to prevent race conditions and ensure data integrity.

**5. Design and Implementation Issues**

**User Identification**

* **IP Address + User ID:** Each user is uniquely identified by their IP address combined with a unique User ID, ensuring distinct sessions and personalized gameplay.

**Platform Identification**

* **Platform Codes:**
  + **WEB = 1**
  + **Android = 2**
  + **iOS = 3**

These identifiers facilitate platform-specific optimizations and support for potential future expansions.

**Data Structures**

* **Board Structure:**

public class Board {

private boolean[][] position = new boolean[10][10];

private String opponentID;

private int boardID;

public boolean isLegalStep(int x, int y) { /\* Implementation \*/ }

public boolean setPosition(int x, int y) { /\* Implementation \*/ }

}

The Board class maintains the state of the game, including stone placements and opponent information.

**Error Handling**

* **Invalid Moves:** The client validates moves before sending them to the server, reducing unnecessary server load.
* **Disconnections:** The server monitors client connections and handles unexpected disconnections gracefully, notifying the remaining player and updating match lists accordingly.

**Scalability**

* **Efficient Resource Utilization:** The server employs efficient data structures and thread management to handle a large number of concurrent game sessions without performance degradation.