Multiplayer ASCII Space Invaders: Client-Server Architecture

Andre Pacheco Sergio Pezo Arbués Pérez

Universidad Nacional de Ingeniería, Faculty of Sciences

May 6, 2025

Table of Contents

1. Introduction
2. Architecture
3. Client-Side Architecture
4. Server-Side Architecture
5. Network Communication
6. Threading Models
7. Game Logic Flow
8. Performance and Challenges
9. Conclusion

Introduction

Project Overview

- A networked multiplayer version of the classic Space Invaders game
- Rendered using ASCII characters in Java Swing
- Client-server architecture
- Multiple players can join a central server
- Real-time state synchronization

Key Objectives

Implement stable client-server architecture

Develop thread-safe game state management

Create efficient network communication protocols

Support multiple concurrent players

Architecture

Component Diagram

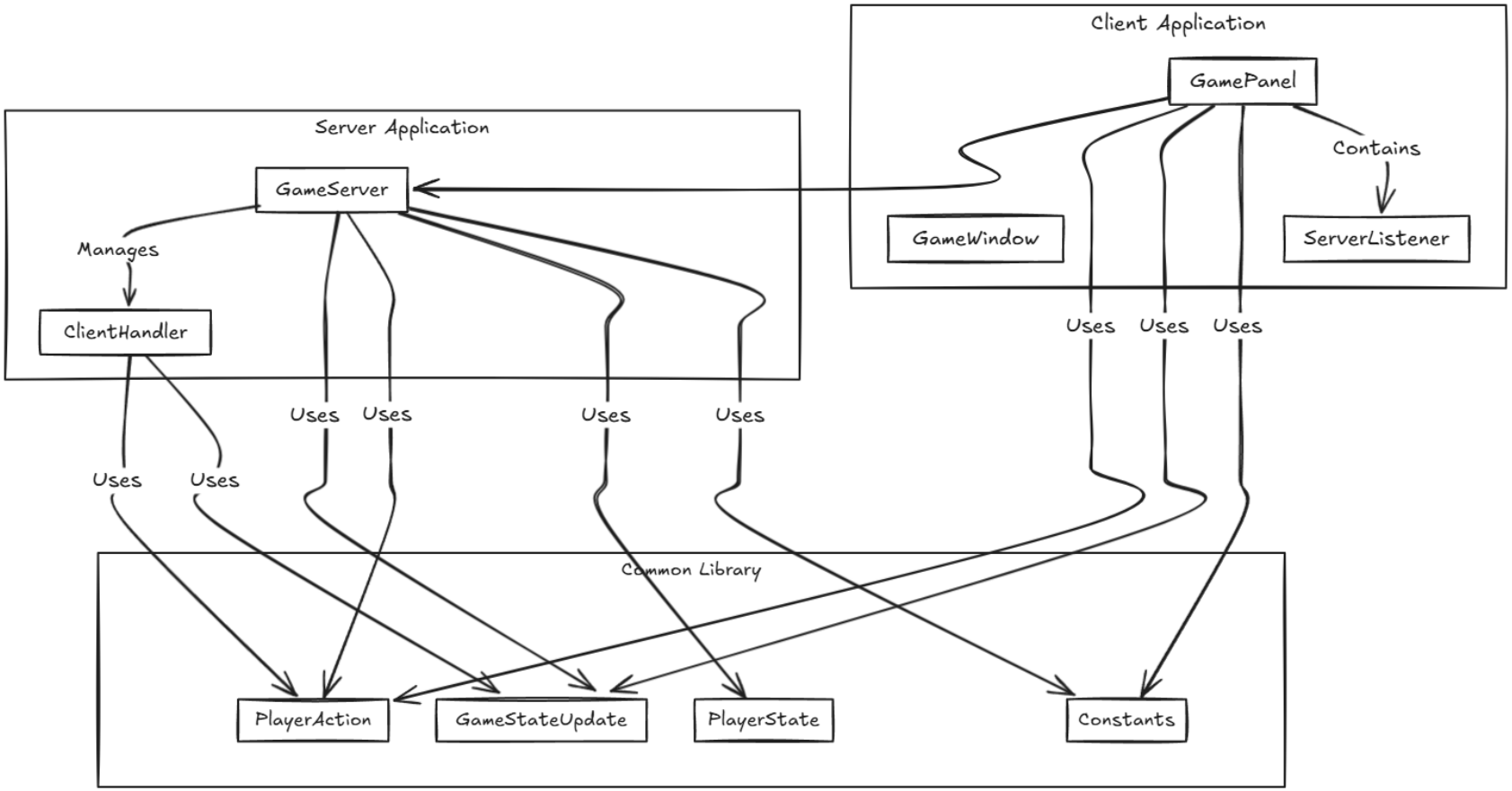

Figure: Component Diagram showing the relationships between major system components

Class Diagram

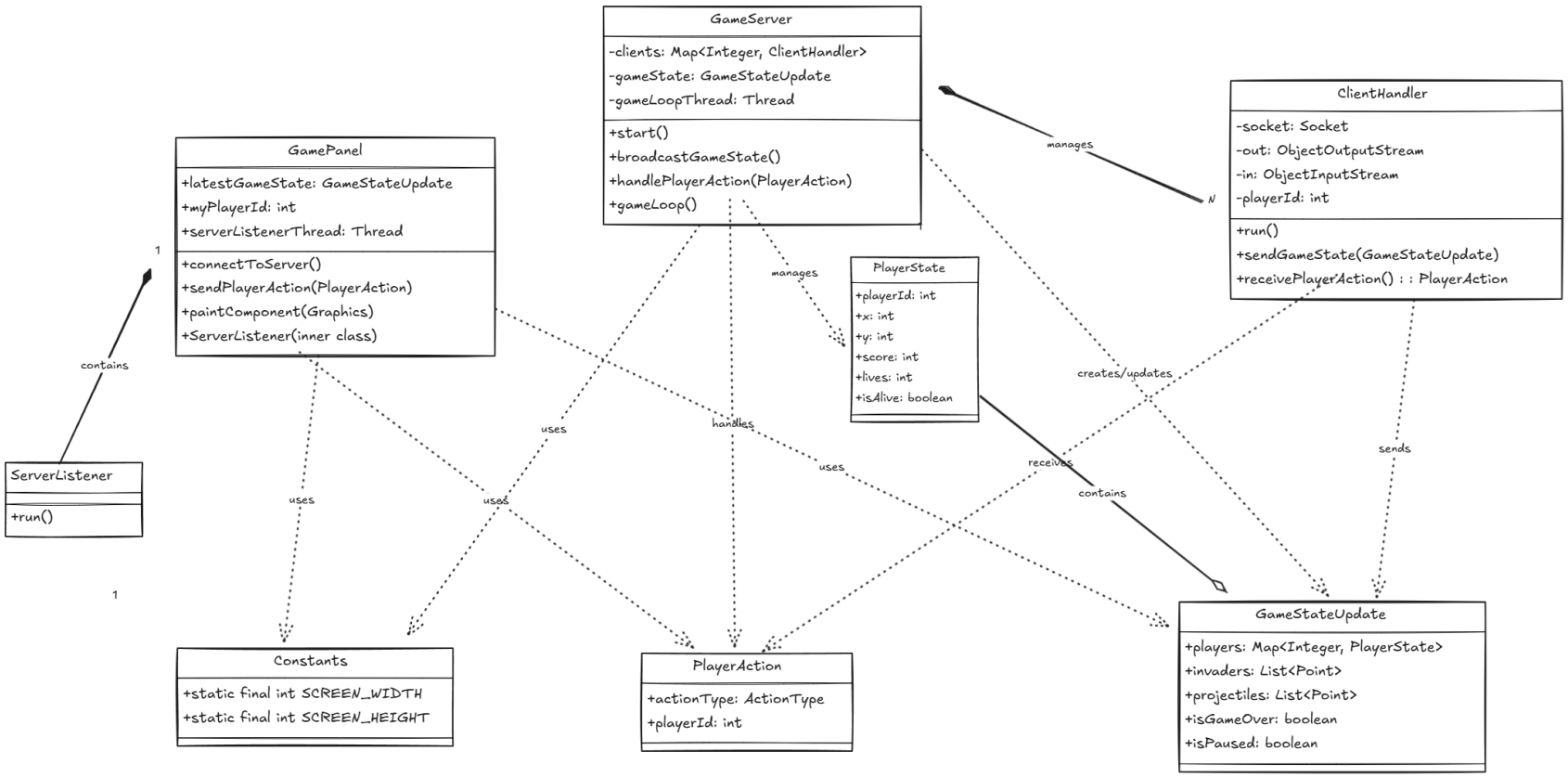

Figure: Simplified Class Diagram showing the key classes and their relationships

Client-Server Interaction

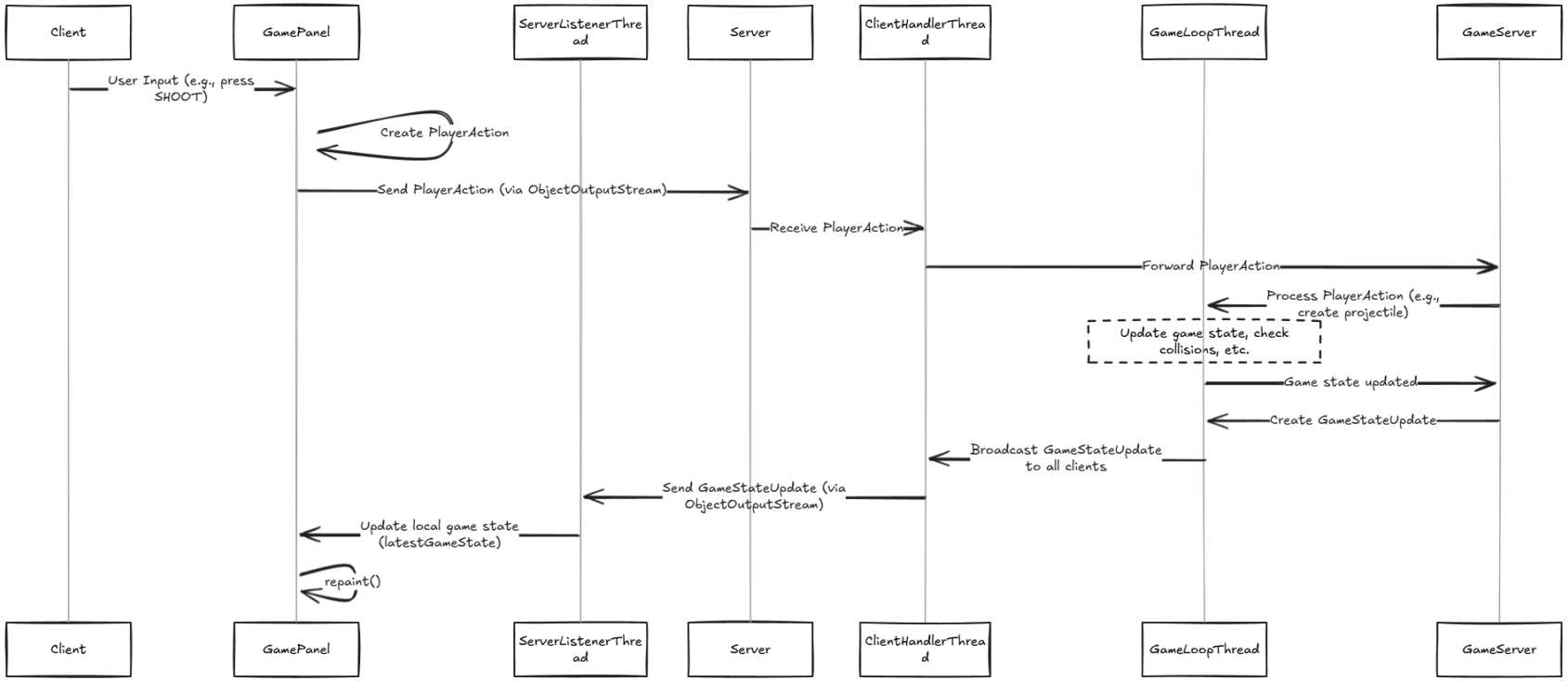

Figure: Client-Server Interaction Sequence Diagram showing the communication flow

Project Structure

Three Main Packages

- **client**: UI, rendering, input handling
- **server**: Connection handling, game state management
- **common**: Shared data structures, constants

Key Components

- `GamePanel.java`: Client UI, rendering, input, network
- `GameServer.java`: Server logic, state management
- `ClientHandler.java`: Per-client communication
- `GameStateUpdate.java`: Serializable game state
- `PlayerAction.java`: Client-to-server commands

Client-Side Architecture

GamePanel Class

- Heart of the client application
- Extends JPanel for graphical interface
- Multiple responsibilities:
  - Rendering game state using ASCII characters
  - Processing user input via KeyAdapter
  - Network communication with server
  - Managing local game state

ServerListener Thread

Dedicated thread that continuously listens for objects sent by the server:

- First receives player ID assignment
- Then receives GameStateUpdate objects
- Updates local state representation
- Prevents UI thread from blocking on network I/O

ASCII Rendering and Input Handling

ASCII Rendering:

```
1 private void drawAsciiArt(Graphics g,
2                           String[] art,
3                           int x, int y) {
4     if (art == null) return;
5     for (int i = 0; i < art.length; i++) {
6         g.drawString(art[i], x,
7                      y + i * charHeight);
8     }
9 }
```

Input Handling:

```
1  public void keyPressed(KeyEvent e) {
2      if (!isConnected) return;
3
4      int key = e.getKeyCode();
5
6      if (key == KeyEvent.VK_LEFT) {
7          if (!movingLeft) {
8              sendAction(new PlayerAction(
9                  PlayerAction.ActionType
10                 .MOVE_LEFT_START,
11                 myPlayerId));
12             movingLeft = true;
13         }
14     }
15     // Other key handling...
16 }
```

Server-Side Architecture

GameServer Class

- Central authority for the game system
- Responsibilities:
  - Accepting client connections
  - Assigning unique player IDs
  - Managing authoritative game state
  - Running game logic updates
  - Broadcasting state to clients

Server State Machine

- `WAITING`: Waiting for first player
- `LOBBY`: Players connected, awaiting start
- `RUNNING`: Game in progress
- `PAUSED`: Game temporarily paused
- `GAME_OVER`: Game ended, waiting for restart

ClientHandler Class

- One instance per connected client
- Each runs in its own thread
- Responsibilities:
  - Reading PlayerAction objects from client
  - Forwarding actions to GameServer
  - Sending GameStateUpdate objects to client
  - Managing connection lifecycle

Thread Safety

Thread pool (ExecutorService) manages all ClientHandler threads to scale efficiently with increasing client connections.

Game Loop and ClientHandler Implementation

Game Loop:

```
private void gameLoop() {
    long lastUpdateTime =
        System.nanoTime();
    while (running) {
        // Calculate delta time

        synchronized (this) {
            switch (serverState) {
                case RUNNING:
                    updateServerGameState(
     deltaTime);
                    checkServerCollisions
     ();
                    checkGameConditions();
                    currentState =

     createGameStateUpdate();
                    break;
                // Other states...
            }
        }

        if (currentState != null) {
            broadcastGameState(
     currentState);
        }

        // Sleep to maintain frame rate
    }
}
```

ClientHandler:

```
@Override
public void run() {
    try {
        while (running) {
            // Read actions from client
            PlayerAction action =
                (PlayerAction) in.
     readObject();
            if (action != null) {
                action.playerId = this.
     playerId;
                server.handlePlayerAction(
     action);
            }
        }
    } catch (Exception e) {
        // Handle disconnection
    } finally {
        server.removeClient(this);
        closeConnection();
    }
}
```

Network Communication

Network Protocol

Socket Type

- TCP/IP sockets for reliable, ordered communication
- `ServerSocket` for accepting connections
- `Socket` for client-server endpoints

Data Serialization

- Java Object Serialization for complex objects
- Classes implement `Serializable` interface
- `ObjectOutputStream` for sending
- `ObjectInputStream` for receiving

Common Protocol Classes

- `GameStateUpdate`: Server to client (full game state)
- `PlayerAction`: Client to server (player commands)
- `PlayerState`: Player data (position, score, lives)

Network Protocol Classes

GameStateUpdate:

```
public class GameStateUpdate
    implements Serializable {
    private static final long
        serialVersionUID = 2L;

    public List<SimplePosition> invaders;
    public List<SimplePosition>
        playerProjectiles;
    public List<SimplePosition>
        invaderProjectiles;
    public Map<Integer, PlayerState>
        players;
    public List<SimpleBarrierState>
        barriers;
    public int currentLevel;
    public boolean isGameOver = false;
    public boolean isPaused = false;

    // Inner classes...
}
```

PlayerAction:

```
public class PlayerAction
    implements Serializable {
    private static final long
        serialVersionUID = 5L;

    public enum ActionType {
        MOVE_LEFT_START,
        MOVE_LEFT_STOP,
        MOVE_RIGHT_START,
        MOVE_RIGHT_STOP,
        SHOOT,
        CONNECT,
        DISCONNECT,
        START_GAME,
        TOGGLE_PAUSE
    }

    public ActionType type;
    public int playerId;

    public PlayerAction(ActionType type,
                        int playerId) {
        this.type = type;
        this.playerId = playerId;
    }
}
```

Threading Models

Server-Side Threading

- Multi-threaded design for scalability
- Three main thread types:
  - **Main Server Thread**: Accepts connections
  - **Game Logic Thread**: Updates game state
  - **Client Handler Threads**: One per connected client
- Thread-safe collections for shared state:
  - `ConcurrentHashMap` for players
  - `CopyOnWriteArrayList` for game objects
  - `synchronized` blocks for critical sections

Thread Pool

`ExecutorService` manages client handler threads efficiently, scaling with the number of connected clients.

Client-Side Threading

- Simpler threading model with two threads:
  - **Event Dispatch Thread (EDT)**:
    - Handles UI rendering
    - Processes user input
    - Manages UI component state
  - **ServerListener Thread**:
    - Receives messages from server
    - Updates local game state
    - Prevents UI freezing during network operations

Thread Interaction

EDT reads from local state that ServerListener updates, creating a producer-consumer pattern with minimal synchronization.

Game Logic Flow

Connection and Game Flow

Connection Process

1. Server starts listening for connections
2. Client connects and receives unique player ID
3. Player enters lobby state on server
4. Client sends START_GAME when ready
5. Server initializes game and transitions to RUNNING

Gameplay Loop

1. Server updates game state (positions, collisions)
2. Server broadcasts state to all clients
3. Clients render received state
4. Clients send player actions based on input
5. Server processes actions to update next frame

Collision Detection Algorithm

Algorithm 1 Server-Side Collision Detection (Simplified)

```
1:  procedure CheckServerCollisions
2:      Initialize removal lists
3:      for each player projectile do
4:          for each invader do
5:              if collision detected then
6:                  Mark invader as destroyed
7:                  Mark projectile as destroyed
8:                  Award points to player
9:                  break
10:             end if
11:         end for
12:         for each barrier do
13:             if collision detected then
14:                 Reduce barrier health
15:                 Mark projectile as destroyed
```

Performance and Challenges

Threading Performance

- Multi-threaded design provides several benefits:
  - Dedicated game loop ensures consistent updates
  - Isolated client handlers prevent blocking
  - Thread pool efficiently manages resources
  - UI remains responsive during network operations
- Thread-safe collections minimize synchronization overhead
- Object reset() method prevents serialization caching issues

Optimization Potential

- Object pooling to reduce garbage collection
- Delta compression for network packets
- Client-side prediction to reduce perceived latency

Challenges and Solutions

Concurrency Management

- **Challenge**: Managing access to shared game state
- **Solution**: Thread-safe collections and synchronized blocks

Network Synchronization

- **Challenge**: Maintaining consistent game state
- **Solution**: Server as authoritative source with frequent updates

Connection Handling

- **Challenge**: Graceful connection/disconnection
- **Solution**: Robust error handling and resource cleanup

Game State Updates

- **Challenge**: Efficient transmission of state
- **Solution**: Minimal serializable classes with essential data

Conclusion

Accomplishments & Future Work

Accomplishments

- Implemented robust client-server architecture
- Created thread-safe game state management
- Developed efficient network protocol
- Supported multiple concurrent players
- Handled connections/disconnections gracefully

Future Work

- Player-vs-player modes
- Multiple game rooms per server
- Enhanced graphics and user interface
- Persistent leaderboards
- Client-side prediction for smoother gameplay
- Additional game features (power-ups, enemy types)

Questions?

Thank You!

Any Questions?