Multiplayer ASCII Space Invaders: Client-Server Architecture

Student:
André Joaquín Pacheco Taboada
National University of Engineering, Faculty of Sciences,
e-mail: andre.pacheco.t@uni.pe

Student:
Sergio Sebastian Pezo Jimenez
National University of Engineering, Faculty of Sciences,
e-mail: sergio.pezo.j@@uni.pe

Student:
Arbués Enrique Pérez Villegas
National University of Engineering, Faculty of Sciences,
e-mail: arbues.perez.v@uni.pe

May 6, 2025

Abstract

This report presents the design and implementation of a multiplayer ASCII Space Invaders game using a client-server architecture in Java. The project features a networked system allowing multiple players to connect and play simultaneously, with a centralized server managing game state and synchronization. Key components include the client-side rendering using Java Swing, server-side game logic, thread management for concurrent connections, and a robust network protocol using Java's object serialization. The report covers the system architecture, communication protocols, threading models, and key implementation details.

Keywords: client-server architecture, Java networking, multithreading, game development, object serialization.

Contents

1 Introduction

Space Invaders is a classic arcade game where players control a ship to shoot descending aliens. This project extends the traditional single-player experience to a networked, multiplayer version rendered in ASCII characters. The implementation features a robust client-server architecture that manages multiple concurrent players, game state synchronization, and network communication.

The key objectives of this project were:

- Implement a stable client-server architecture for real-time multiplayer gaming
- Develop thread-safe game state management with proper synchronization
- Create network protocols for efficient state updates and action transmission
- Design a user-friendly interface using ASCII art characters for rendering
- Support multiple players joining and playing simultaneously

This report details the project's architecture, technologies used, communication protocols, threading models, and the implementation details of both client and server components.

2 Project Overview

2.1 Theoretical Framework

2.1.1 Client-Server Model

The client-server model is a distributed application structure that partitions tasks between providers of a resource or service (servers) and service requesters (clients). In our implementation, the server is responsible for:

- Managing the authoritative game state (player positions, invaders, projectiles)
- Processing player inputs and updating the game world
- Handling collision detection and game logic
- Broadcasting state updates to all connected clients

The clients are responsible for:

- Rendering the game state received from the server
- Capturing player input and sending it to the server
- Maintaining a local representation of the game state for rendering

This separation of concerns allows for a centralized authority that prevents cheating and ensures all players experience a consistent game world.

2.1.2 Concurrent Programming and Threading

The project heavily relies on concurrent programming techniques to handle multiple clients and game logic simultaneously. Key threading concepts used include:

- Thread pools for managing client connections efficiently
- Dedicated threads for non-blocking network I/O
- Thread-safe data structures for state sharing between threads
- Proper synchronization to avoid race conditions

2.1.3 Network Communication

The network communication is built on TCP/IP sockets, which provide reliable, ordered data transmission. Java's object serialization is used to transmit complex data structures between client and server, allowing for simplified message passing of game states and player actions.

3 Project Structure

The project is organized into three main packages:

- **client**: Contains classes for the client application (UI, rendering, input handling)
- **server**: Contains classes for the server application (connection handling, game state management)
- **common**: Contains shared classes used by both client and server (data structures, constants)

3.1 Key Components and Their Purpose

Table 1: Key Components and Their Purpose

Component	**Purpose**
`GamePanel.java`	The main client UI component responsible for rendering the game, handling user input, and communicating with the server.
`GameServer.java`	The central server component managing client connections, game logic, and state broadcasting.
`ClientHandler.java`	Handles communication with a single connected client, processing incoming actions and sending state updates.
`GameStateUpdate.java`	Serializable object encapsulating the entire game state for network transmission.
`PlayerAction.java`	Serializable object representing player actions (movement, shooting) sent from client to server.
`PlayerState.java`	Represents a player's current state (position, score, lives).
`Constants.java`	Shared configuration values and constants used by both client and server.

3.2 Architecture Diagrams

3.2.1 Component Diagram

The component diagram illustrates the high-level components and their dependencies within the system.

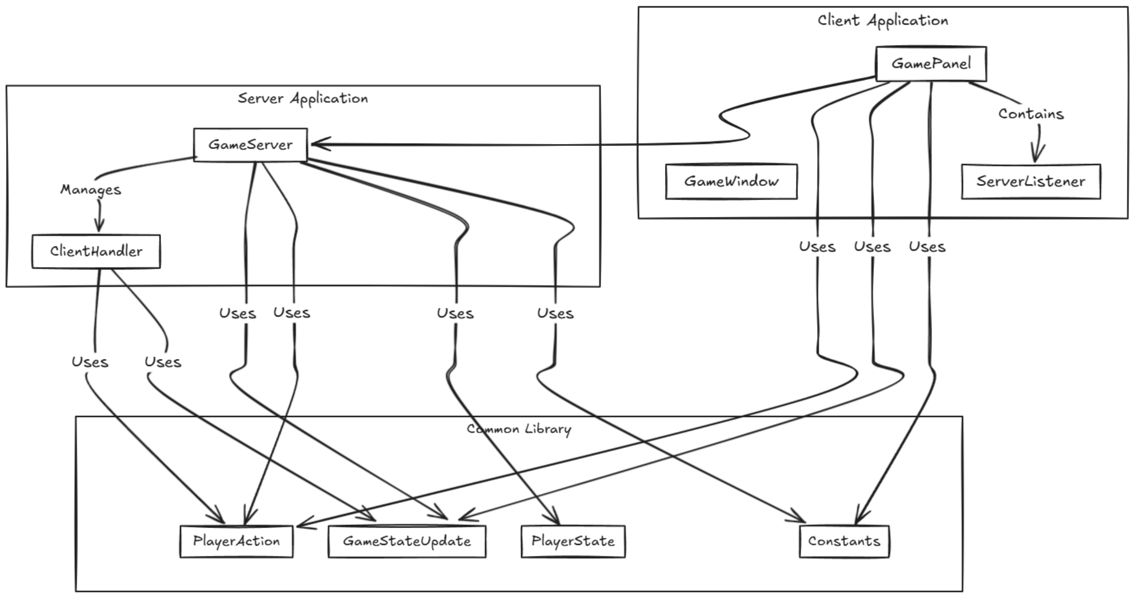

Figure 1: Component Diagram showing the relationships between major system components

3.2.2 Simplified Class Diagram

The class diagram outlines the key classes and their relationships.

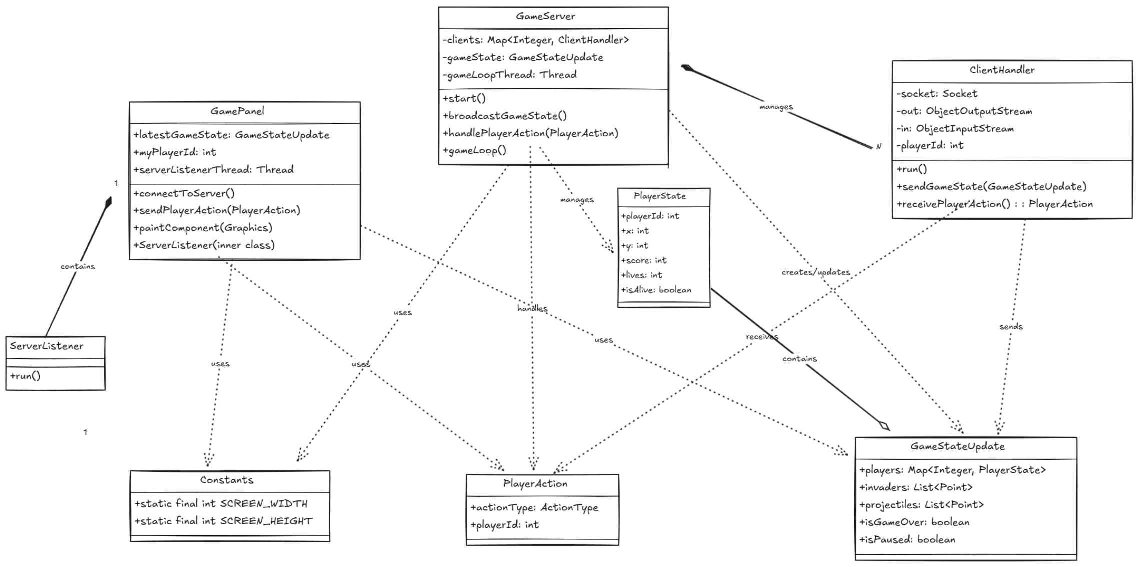

Figure 2: Simplified Class Diagram showing the key classes and their relationships

3.2.3 Client-Server Interaction Sequence Diagram

This diagram illustrates the typical communication flow between a client and the server.

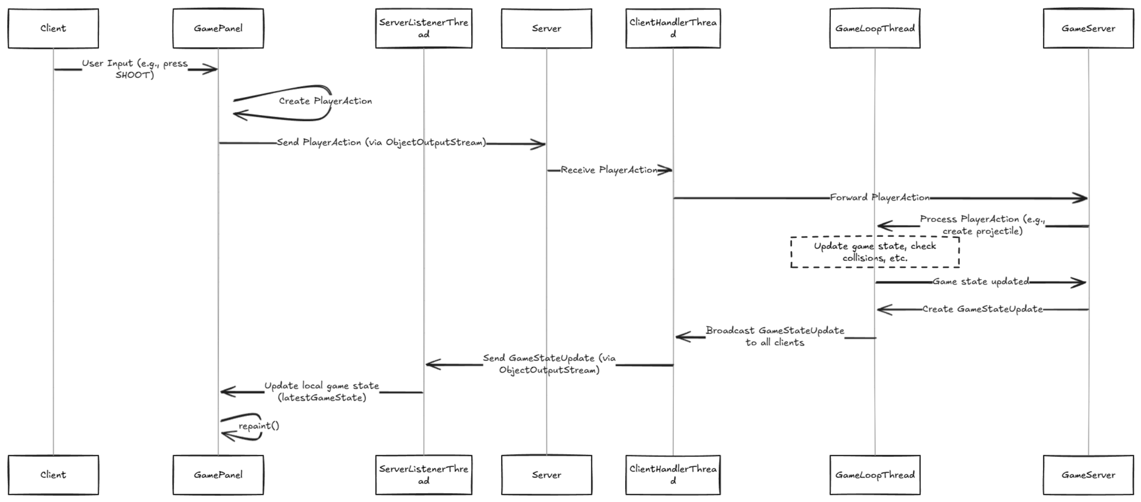
Client
GamePanel
ServerListenerThread
Server
ClientHandlerThread
GameLoopThread
GameServer
User Input (e.g., press SHOOT)
Create PlayerAction
Send PlayerAction (via ObjectOutputStream)
Receive PlayerAction
Forward PlayerAction
Process PlayerAction (e.g., create projectile)
Update game state, check collisions, etc.
Game state updated
Create GameStateUpdate
Broadcast GameStateUpdate to all clients
Send GameStateUpdate (via ObjectOutputStream)
Update local game state (latestGameState)
repaint()

Figure 3: Client-Server Interaction Sequence Diagram showing the communication flow

4 Client-Side Architecture

4.1 GamePanel Class

The `GamePanel` class is the heart of the client application, extending `JPanel` to provide a graphical interface for the game. It has multiple responsibilities:

4.1.1 Rendering

The panel renders the game state using ASCII characters. It handles drawing:

- Player ships
- Invaders (aliens)
- Projectiles (both player and invader)
- Barriers
- Score and status information

```java
private void drawAsciiArt(Graphics g, String[] art, int x, int y) {
    if (art == null) return; // Avoid NPE if sprite is null
    for (int i = 0; i < art.length; i++) {
        g.drawString(art[i], x, y + i * charHeight);
    }
}
```

Listing 1: ASCII Art Rendering in GamePanel

4.1.2 User Input

The panel captures keyboard events through a `KeyAdapter` inner class, which translates user input into `PlayerAction` objects that are sent to the server. The class maintains state flags (`movingLeft`, `movingRight`) to avoid sending redundant movement start/stop actions.

```java
@Override
public void keyPressed(KeyEvent e) {
    // Game input handling
    if (!isConnected || myPlayerId == -1) return;

    int key = e.getKeyCode();

    if (key == KeyEvent.VK_LEFT || key == KeyEvent.VK_A) {
        if (!movingLeft) {
            sendAction(new PlayerAction(PlayerAction.ActionType.MOVE_LEFT_START, myPlayerId))
    ;
            movingLeft = true;
            if(movingRight)
                sendAction(new PlayerAction(PlayerAction.ActionType.MOVE_RIGHT_STOP,
    myPlayerId));
            movingRight = false;
        }
    }
    // Other key handling...
}
```

Listing 2: Key Event Handling in GamePanel

4.1.3 Network Communication

The panel handles all network communication with the server:

- Establishing the socket connection
- Sending player actions via `ObjectOutputStream`
- Receiving game state updates via `ObjectInputStream`
- Managing the network lifecycle (connection, disconnection)

```
// Send action to server
private synchronized void sendAction(PlayerAction action) {
    if (!isConnected || out == null) return;
    try {
        out.writeObject(action);
        out.flush();
    } catch (IOException e) {
        System.err.println("Error sending action ("+action.type+"): " + e.getMessage());
        disconnect();
    }
}
```

Listing 3: Network Communication in GamePanel

4.1.4 ServerListener Thread

To prevent blocking the UI thread, a dedicated `ServerListener` thread continuously listens for incoming objects from the server. Upon receiving a game state update, it updates the local representation of the game state used for rendering.

```
private class ServerListener implements Runnable {
    @Override
    public void run() {
        try {
            Object initialMsg = in.readObject();
            if (initialMsg instanceof Integer) {
                myPlayerId = (Integer) initialMsg;
                // Player ID received processing...
            }

            while (isConnected && !Thread.currentThread().isInterrupted()) {
                Object receivedObject = in.readObject();
                if (receivedObject instanceof GameStateUpdate) {
                    latestGameState = (GameStateUpdate) receivedObject;
                    // Update local state copies...
                }
            }
        } catch (IOException | ClassNotFoundException e) {
            // Error handling and disconnection...
        }
    }
}
```

Listing 4: ServerListener Implementation

5 Server-Side Architecture

5.1 GameServer Class

The `GameServer` class is the central authority in the game system, responsible for:

5.1.1 Client Connection Management

The server listens for incoming connections on a dedicated port, assigns a unique player ID to each new client, and creates a `ClientHandler` for each connection.

```java
try {
    Socket clientSocket = serverSocket.accept();
    int playerId = nextPlayerId.getAndIncrement();
    System.out.println("Client connected: " + clientSocket.getInetAddress()
                      + " assigned ID: " + playerId);
    PlayerState newPlayerState = new PlayerState(playerId,
                                Constants.PLAYER_START_X,
                                Constants.PLAYER_START_Y,
                                0, Constants.PLAYER_LIVES);
    ServerPlayer newServerPlayer = new ServerPlayer(newPlayerState);
    players.put(playerId, newServerPlayer);

    ClientHandler clientHandler = new ClientHandler(clientSocket, this, playerId);
    clients.add(clientHandler);
    clientExecutor.submit(clientHandler);
} catch (IOException e) {
    // Error handling...
}
```

Listing 5: Connection Handling in GameServer

5.1.2 Game State Management

The server maintains the authoritative version of the game state, including:

- Player positions and states
- Invader positions
- Projectile tracking
- Barrier states
- Game status (running, paused, game over)

5.1.3 Game Logic Loop

A dedicated thread runs the main game loop, which:

- Updates all game objects based on elapsed time
- Checks for collisions between game objects
- Processes player actions
- Broadcasts the updated game state to all clients

```java
private void gameLoop() {
    long lastUpdateTime = System.nanoTime();
    while (running) {
        long now = System.nanoTime();
        double deltaTime = (now - lastUpdateTime) / 1_000_000_000.0;
        lastUpdateTime = now;

        GameStateUpdate currentState = null;

        // Lock state during update/check phases
        synchronized (this) {
            switch (serverState) {
                case RUNNING:
                    updateServerGameState(deltaTime);
                    checkServerCollisions();
                    checkGameConditions();
                    currentState = createGameStateUpdate();
                    break;
                // Other state handling...
            }
        }

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

        // Sleep logic...
    }
}
```

Listing 6: Game Logic Loop in GameServer

5.1.4 Server State Machine

The server implements a state machine with five distinct states:

- `WAITING`: Waiting for the first player to connect
- `LOBBY`: Players connected, waiting for game start
- `RUNNING`: Game in progress, logic active
- `PAUSED`: Game temporarily paused by a player
- `GAME_OVER`: Game finished, waiting for restart

5.2 ClientHandler Class

Each `ClientHandler` instance runs in its own thread and is responsible for all communication with a single client:

```java
@Override
public void run() {
    try {
        while (running) {
            // Read actions from the client
            PlayerAction action = (PlayerAction) in.readObject();
            if (action != null) {
                // Ensure action has correct player ID
                action.playerId = this.playerId;
                server.handlePlayerAction(action);
            }
        }
    } catch (IOException | ClassNotFoundException e) {
        // Error handling...
    } finally {
        server.removeClient(this);
        closeConnection();
    }
}

// Method to send game state to client
public void sendGameState(GameStateUpdate gameState) {
    if (!running || out == null) return;
    try {
        out.writeObject(gameState);
        out.reset(); // Important for mutable objects
        out.flush();
    } catch (IOException e) {
        // Error handling...
    }
}
```

Listing 7: ClientHandler Implementation

6 Network Communication

6.1 Socket Type

The project uses TCP/IP sockets for reliable, ordered communication:

- `java.net.ServerSocket` on the server to listen for and accept connections
- `java.net.Socket` on both client and server for connection endpoints

6.2 Data Serialization

Java Object Serialization is used to transmit complex Java objects over the network:

- Classes implementing `java.io.Serializable` can be sent over the network
- `ObjectOutputStream` serializes objects into a byte stream
- `ObjectInputStream` deserializes the byte stream back into objects

6.3 Common Data Structures

Three main serializable classes facilitate communication:

- `GameStateUpdate`: Encapsulates the entire game state for rendering
- `PlayerAction`: Represents a command sent from client to server
- `PlayerState`: Holds state information for a single player

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

    // Using simpler structures for serialization
    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 for positions and states...
}
```

Listing 8: GameStateUpdate Class

7 Threading Models

7.1 Server-Side Threading

The server employs a multi-threaded design for scalability and responsiveness:

- **Main Server Thread**: Accepts client connections and initializes resources
- **Game Logic Thread**: Runs the main game loop for state updates
- **Client Handler Threads**: One thread per client for network I/O

Thread-safe collections are used to manage shared state:

- `ConcurrentHashMap` for player state
- `CopyOnWriteArrayList` for game objects
- `synchronized` blocks for critical sections

7.2 Client-Side Threading

The client uses a simpler threading model:

- **Event Dispatch Thread (EDT)**: Handles UI rendering and user input
- **ServerListener Thread**: Dedicated to receiving server messages

This separation prevents UI freezing during network operations.

8 Game Logic Flow

8.1 Connection and Initialization

1. Server starts and begins listening for connections
2. Client connects and receives a unique player ID
3. Client sends a START_GAME action when ready
4. Server initializes the game and transitions to RUNNING state

8.2 Gameplay Loop

1. Server updates game state (positions, collisions)
2. Server broadcasts GameStateUpdate to all clients
3. Clients render the received state
4. Clients send PlayerAction objects based on user input
5. Server processes actions and updates game state

8.3 Algorithms

8.3.1 Server-Side Collision Detection

8.3.2 Game State Update Broadcasting

9 Performance Analysis

9.1 Threading Considerations

The project's use of multiple threads provides several performance benefits:

- Dedicated game loop thread ensures consistent updates regardless of client count
- Client handler threads isolate network I/O for each connection
- Thread pool prevents resource exhaustion with many clients
- Non-blocking network operations keep the UI responsive

Algorithm 1 Collision Detection Algorithm

```
1: procedure CheckServerCollisions
2:     Initialize collections to track objects for removal
3:     for each player projectile do
4:         Create collision bounds for projectile
5:         hitSomething ← false
6:         for each invader do
7:             if invader is alive AND invader bounds intersect projectile bounds then
8:                 invader.alive ← false
9:                 Add invader to removal list
10:                Add projectile to removal list
11:                proj.active ← false
12:                hitSomething ← true
13:                Award points to player who fired projectile
14:                break
15:            end if
16:        end for
17:        if NOT hitSomething then
18:            for each barrier do
19:                if barrier is alive AND barrier bounds intersect projectile bounds then
20:                    barrier.hit()
21:                    Add projectile to removal list
22:                    proj.active ← false
23:                    hitSomething ← true
24:                    break
25:                end if
26:            end for
27:        end if
28:    end for
                 ▷ Similar checks for invader projectiles vs players/barriers          ▷ And invaders vs barriers
29:    Remove all objects in removal lists from their collections
30: end procedure
```

Algorithm 2 Game State Update and Broadcasting

```
procedure GameLoop
    lastUpdateTime $\leftarrow$ System.nanoTime()
    while running do
        now $\leftarrow$ System.nanoTime()
        deltaTime $\leftarrow$ (now - lastUpdateTime) / 1,000,000,000.0
        lastUpdateTime $\leftarrow$ now
        currentState $\leftarrow$ null
        synchronized (this) do
        if serverState = RUNNING then
            UpdateServerGameState(deltaTime)
            CheckServerCollisions()
            CheckGameConditions()
            currentState $\leftarrow$ CreateGameStateUpdate()
        else if serverState $\in$ {WAITING, LOBBY, GAME_OVER, PAUSED} then
            currentState $\leftarrow$ CreateGameStateUpdate()
        end if

        if currentState $\neq$ null then
            BroadcastGameState(currentState)
        end if
        Sleep to maintain frame rate
    end while
end procedure
```

9.2 Optimization Techniques

Several optimization techniques are employed:

- Thread-safe collections minimize synchronization overhead
- Object pools could be used to reduce garbage collection pressure
- The server only sends the minimum necessary state information
- Client-side prediction could be implemented to reduce perceived latency

10 Challenges and Solutions

10.1 Concurrency Management

Challenge: Managing access to shared game state from multiple threads.
Solution: Using thread-safe collections and synchronized blocks to prevent race conditions. The server employs `ConcurrentHashMap` for players and `CopyOnWriteArrayList` for game objects.

10.2 Network Synchronization

Challenge: Keeping client and server game states synchronized despite network latency.
Solution: The server maintains the authoritative state, and clients render based on the latest received state. The server sends frequent updates (up to 60 times per second).

10.3 Connection Handling

Challenge: Handling client connections and disconnections gracefully.
Solution: Using a dedicated `ClientHandler` per connection with proper error handling and resource cleanup. The server detects disconnections via exceptions and removes clients cleanly.

11 Conclusion and Future Work

11.1 Accomplishments

The project successfully implements a multiplayer version of Space Invaders with:

- A robust client-server architecture
- Thread-safe game state management
- Efficient network communication
- Support for multiple concurrent players
- Graceful handling of connections/disconnections

11.2 Limitations

Current limitations include:

- Limited player differentiation (color only)
- No player-to-player interaction
- Single game instance per server
- Basic ASCII graphics

11.3 Future Work

Potential improvements include:

- Implementing player-vs-player modes
- Adding support for multiple game rooms
- Enhancing the graphics and UI
- Adding persistent leaderboards
- Implementing client-side prediction for smoother gameplay
- Adding more game features (power-ups, different enemy types)

12 Appendix: Key Code Snippets

12.1 Server Game State Update Creation

```java
private GameStateUpdate createGameStateUpdate() {
    GameStateUpdate update = new GameStateUpdate();

    // Players (send copies of PlayerState)
    Map<Integer, PlayerState> playerStates = new ConcurrentHashMap<>();
    for(Map.Entry<Integer, ServerPlayer> entry : players.entrySet()){
         playerStates.put(entry.getKey(), entry.getValue().state);
    }
    update.players = playerStates;

    // Invaders
    List<GameStateUpdate.SimplePosition> invaderPositions = new ArrayList<>();
    for (ServerInvader inv : invaders) {
        invaderPositions.add(new GameStateUpdate.SimplePosition(inv.x, inv.y));
    }
    update.invaders = invaderPositions;

    // Add projectiles, barriers, and other state...

    // Other state
    update.currentLevel = this.currentLevel;
    update.isGameOver = (serverState == ServerState.GAME_OVER);
    update.isPaused = (serverState == ServerState.PAUSED);

    return update;
}
```

Listing 9: Creating GameStateUpdate in Server