Final Report

Capture The Flag (CTF) Multiplayer Game

Kuang Yi

yi.kuang@studio.unibo.it

July 2025

1. Abstract

This project presents a real-time multiplayer web-based game named Capture the Flag. The primary goal of the project is to design and implement a fully functioning online game platform where users can connect from different devices, join game rooms, control their avatars, and compete to capture the opponent's flag. The system achieves real-time interaction using WebSocket technology and emphasizes scalability, fault-tolerance, and usability. The game has been extended to support multiple concurrent matches, room-based lobby management, and backup server redundancy for increased availability.

2. Concept

The project is a real-time multiplayer web application designed for competitive gaming. The product is a web-service supported by both a frontend browser-based GUI and a backend game management server.

Users interact with the game via standard web browsers on desktop or mobile devices. Each player enters a game room from a lobby and is automatically assigned a team. The game logic ensures each room operates independently. Players control their avatars using keyboard input to navigate a two-dimensional map and attempt to capture the enemy flag.

The system stores minimal user data, including a unique session ID and associated game state (e.g., team, position, score). The gameplay is structured for anonymous access, focusing on instant play rather than account management. Two major roles are defined: red team player and blue team player.

3. Requirements

Functional Requirements

The system must support real-time multiplayer gameplay.

The game must allow players to join specific game rooms.

The system must manage scores and determine game-over conditions.

Players must be able to move avatars using directional keys.

The game must support flag capture logic.

Players must be assigned to teams upon joining.

A lobby page must be provided to input and join rooms.

A reset button must allow manual game restart.

The client must receive game state updates via WebSocket.

Non-Functional Requirements

The system must support multiple concurrent games.

The system must recover gracefully if a server node fails.

The system should provide consistent and smooth real-time interaction.

The system must ensure at-most-once delivery for player actions.

Technological Choices

FastAPI: for asynchronous WebSocket support

Redis: for low-latency cross-server communication

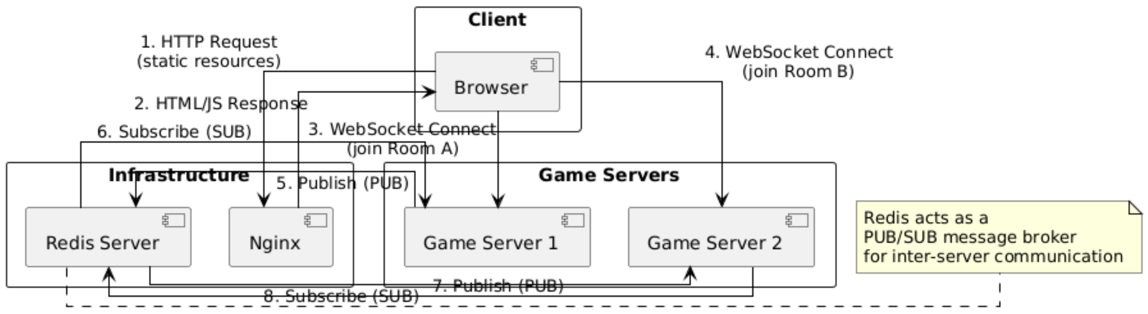
Docker: for containerized deployment

JavaScript: for cross-platform browser UI

These choices were made to ensure low latency, real-time interaction and cross-platform support. Redis helps to decouple game logic from synchronization.

4. Design

Below is a high-level component diagram illustrating the system architecture, including client, game servers, Redis message broker, and static file hosting via nginx:


Client
Browser
1. HTTP Request (static resources)
2. HTML/JS Response
3. WebSocket Connect (join Room A)
4. WebSocket Connect (join Room B)
5. Publish (PUB)
6. Subscribe (SUB)
7. Publish (PUB)
8. Subscribe (SUB)
Infrastructure
Redis Server
Nginx
Game Servers
Game Server 1
Game Server 2
Redis acts as a PUB/SUB message broker for inter-server communication

This component diagram gives a clear overview of the system structure and interconnection.

Architecture

The system uses a client-server architecture with additional support for fault tolerance via a backup server. Clients communicate with the server via WebSocket connections. Redis is used as a pub/sub bus to synchronize game states across multiple server nodes.

Infrastructure

Frontend: HTML + JS client served by nginx

Backend: FastAPI WebSocket server

Redis: Message bus for state synchronization

Docker: Containerization of all components

nginx: Static file serving

Each game room is managed independently. Clients communicate with a server (either primary or backup) which updates game state and broadcasts it to others. Component discovery is managed statically

Modelling

Domain Entities:

Player: includes ID, position, team

Flag: team-assigned flag coordinates

GameRoom: holds players, scores, flags, and states

Domain Events:

Player joins room

Player moves

Flag captured

Game ends

Messages:

Client → Server: { action: move | reset, direction }

Server → Client: updated game state

System State:

Active players per room

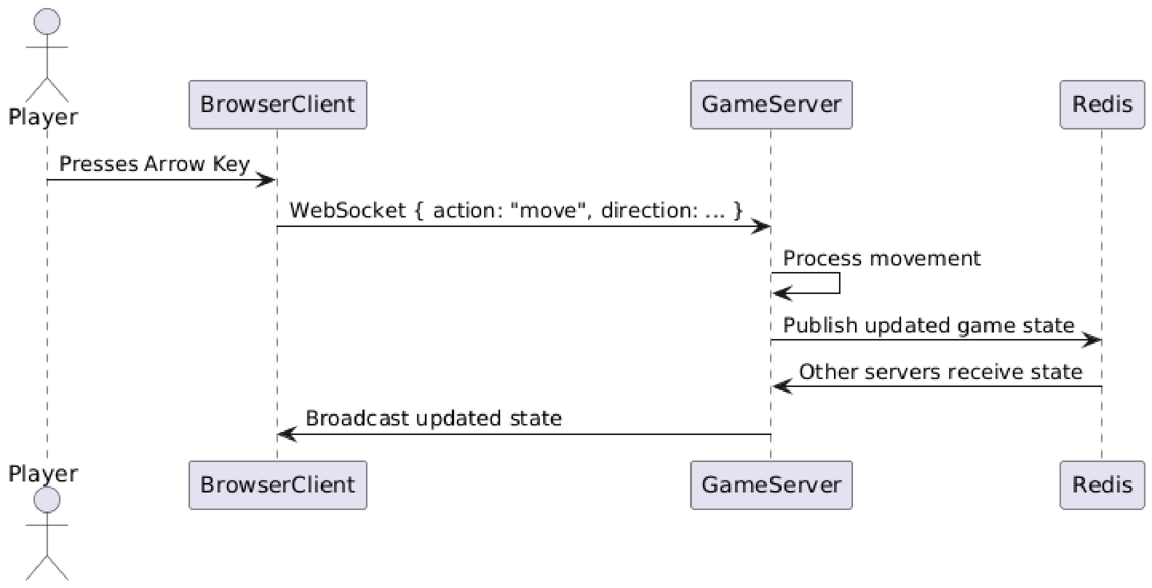
Flags’ positions

Scores and game over condition

Interaction

Clients initiate WebSocket connections upon joining a room. The server responds to player actions by updating the game state and broadcasting updates.

Below is a detailed sequence diagram illustrating the full flow of a player's movement request and its propagation across the system:

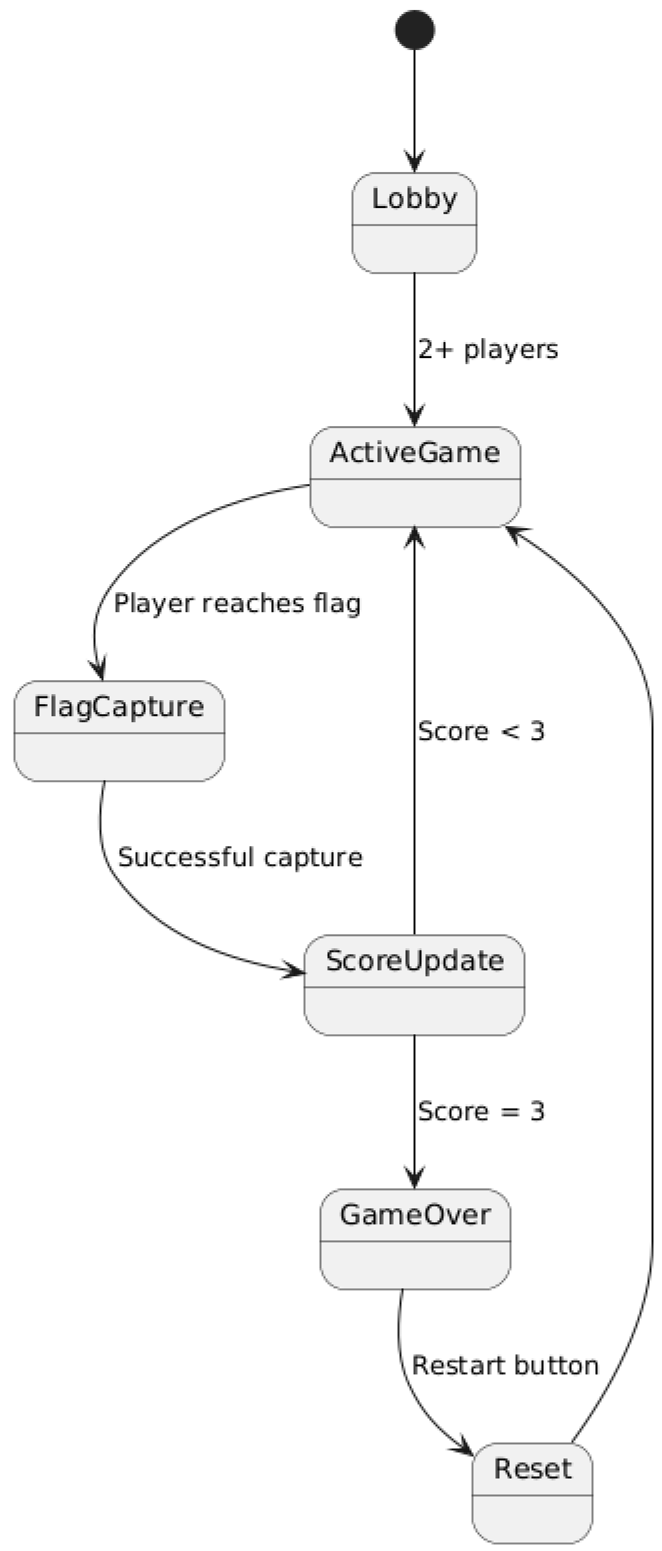


This interaction ensures that all clients connected to a room stay synchronized, even if they are connected to different backend nodes. initiate WebSocket connections upon joining a room. The server responds to player actions by updating the game state and broadcasting updates.

Behaviour

Servers process incoming actions, update game state and check capture conditions. If the game ends, no further scoring is processed unless a manual reset is triggered.

Below is a state diagram illustrating the lifecycle of a game session.

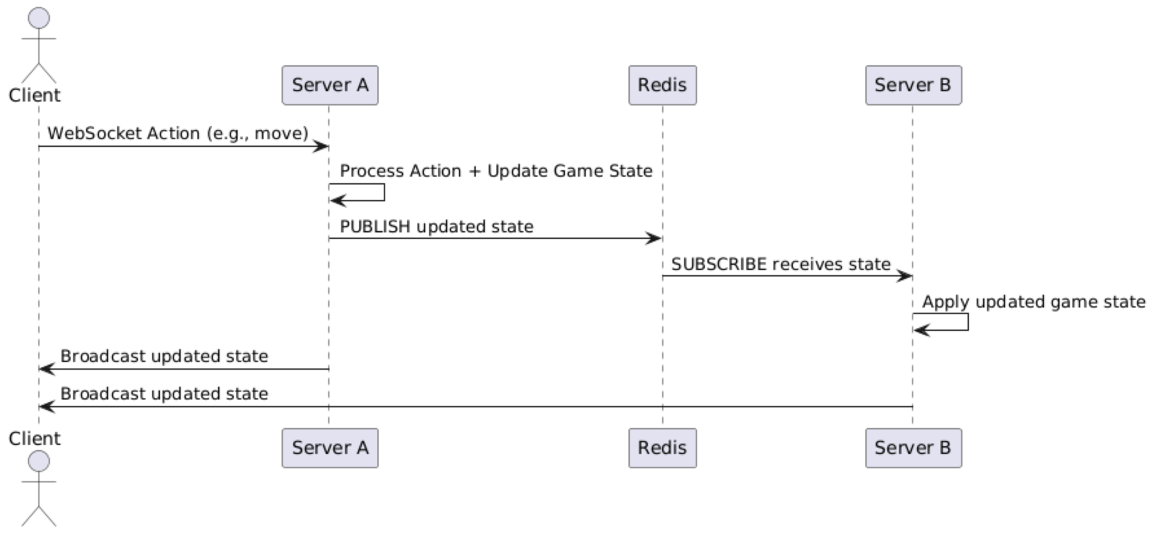


Data and Consistency

Game state is maintained in memory per room. Redis ensures cross-server consistency. No persistent storage is used beyond in-session memory.

To synchronize state across servers, Redis pub/sub is used. Whenever a game server processes a player action (e.g., movement or flag capture), it publishes the updated game state to a Redis channel. All servers subscribe to this channel and update their local copy accordingly. This ensures consistency even in multi-node deployments.

Below is a diagram illustrating the Redis synchronization flow:



Fault-Tolerance

Redis pub/sub ensures that multiple nodes stay synchronized. If a server fails, clients can connect to the backup. The system tolerates network partitions with degraded availability but no data loss.

Availability

No caching is implemented; load is managed by container orchestration. WebSocket connections are resilient to temporary disconnects.

Security

No authentication is implemented, but player identity is derived from persistent local storage. No sensitive data is transmitted.

5. Implementation

Technical Implementation

Network Protocol: WebSocket

Data Format: JSON for all messages

Language: Python (FastAPI) + JS

Synchronization: Redis pub/sub

Containerized with Docker

Key Code Snippets

Room-Based Game Isolation:

The snippet demonstrates how each WebSocket connection is tied to a specific room, enabling room-based isolation of gameplay. When a user connects, the server uses the room parameter to access or create a GameManager instance corresponding to that room. As a result, each room maintains an independent game state, player list, and score, which enables the platform to support multiple concurrent matches running in parallel.

```
@app.websocket("/ws/{room}/{player_id}")
async def websocket_endpoint(websocket: WebSocket, room: str, player_id: str):
    await websocket.accept()
    if room not in games:
        games[room] = GameManager()
    if room not in clients:
        clients[room] = {}
    game = games[room]
    clients[room][player_id] = websocket
    player = games[room].add_player(player_id)
```

Broadcasting Game State to Clients:

Whenever the game state is modified (e.g., a player moves, scores, or the game ends), the server invokes this function to publish the new state to Redis and send updates to all connected clients in the same room. This mechanism ensures real-time synchronization between clients and across server nodes, which is essential for smooth gameplay in a distributed system.

```
async def broadcast_state(room: str):
    state = games[room].get_state()
    state["_source"] = SERVER_ID
    state["room"] = room
    state["online"] = list(clients[room].keys())
   # Post to the Redis "game" channel (be careful to avoid recursive loops, you can add
the server logo to the message)
    await publish("game:{room}", state)
    # Broadcast to all clients on this server
    for ws in clients[room].values():
        try:
            await ws.send_text(json.dumps(state))
        except Exception:
            pass
```

Redis-Based Server Synchronization:

Each server node subscribes to a Redis pub/sub channel to receive game state updates from other nodes. The _source field is used to filter out self-originated messages, preventing infinite update loops. This setup ensures all server instances share a consistent view of the game state, making redundancy and failover possible.

```
async def consume_game_updates():
    async for msg in subscribe():
        room = msg.get("room")
        if not room or msg.get("_source") == SERVER_ID:
```

```
        continue
    if room not in games:
        games[room] = GameManager()
        clients[room] = {}
```

Persistent Identity via Local Storage:

To ensure continuity across page reloads or reconnections, the front-end stores the room name and player ID in localStorage. This approach allows the system to preserve player identity, ensuring the user rejoins the same team and avatar, which is especially important in fault-tolerant, session-based gameplay.

```
const room = localStorage.getItem("room");
const playerId = localStorage.getItem("playerId") || generateId();
```

Technologies Used

Server:	Python 3.11, FastAPI, Uvicorn
Client:	JavaScript, HTML5 Canvas
Infrastructure:	Docker, Docker Compose
Messaging:	Redis (pub/sub)

6. Validation

Acceptance Testing

Manual testing was performed with multiple players across different browsers and rooms.

Verified team assignment and movement

Verified flag capture and score update

Verified server failover via backup server

Verified reconnection with session persistence

Automated testing was planned but removed due to real-time interaction complexity.

Testing Note:

To simulate multiple distinct players, it is recommended to use different browsers (e.g., Chrome, Firefox, Edge) or open incognito/private windows. This ensures each client has a unique `playerId` stored in their local storage.

If a player closes and reopens the game in the same browser window, the system will retrieve the previous `playerId` from `localStorage` and automatically rejoin the game as the same player. Therefore, using multiple separate browsers is important to avoid identity conflicts during multiplayer testing.

7. Release

All components are bundled into Docker containers. The system is deployed using docker-compose. Services include client (nginx), server (FastAPI), and Redis.

8. Deployment

Installation Instructions

Install Docker and Docker Compose

Clone repository: git clone https://github.com/leahkuang/Capture-the-Flag.git

Start services: docker-compose up --build

Access game at: http://localhost:3000

Expected Outcome:

Redis container running

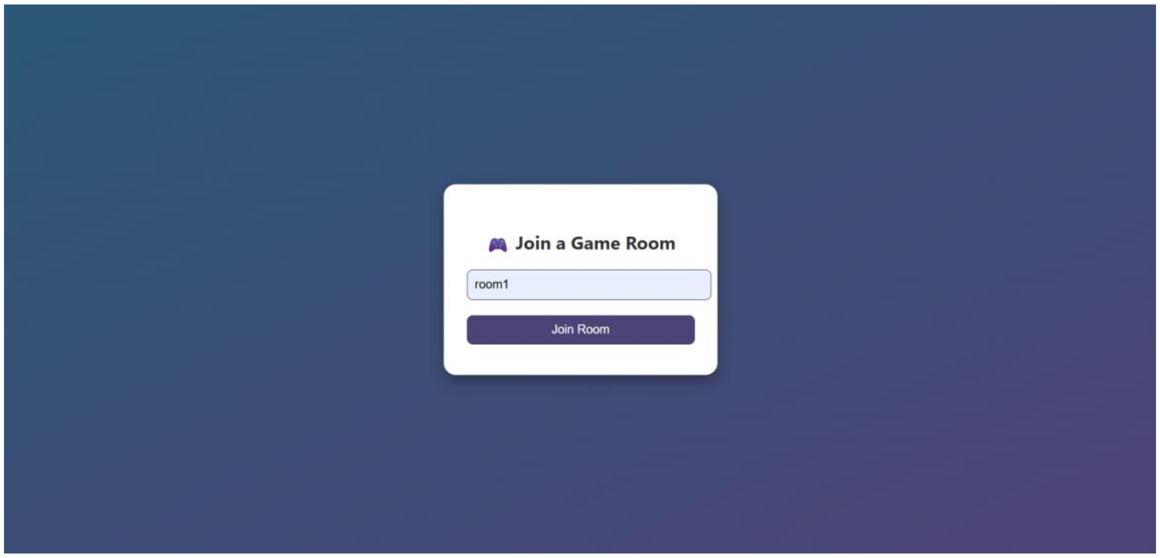
FastAPI server on port 8000

Client accessible on port 3000

9. User Guide

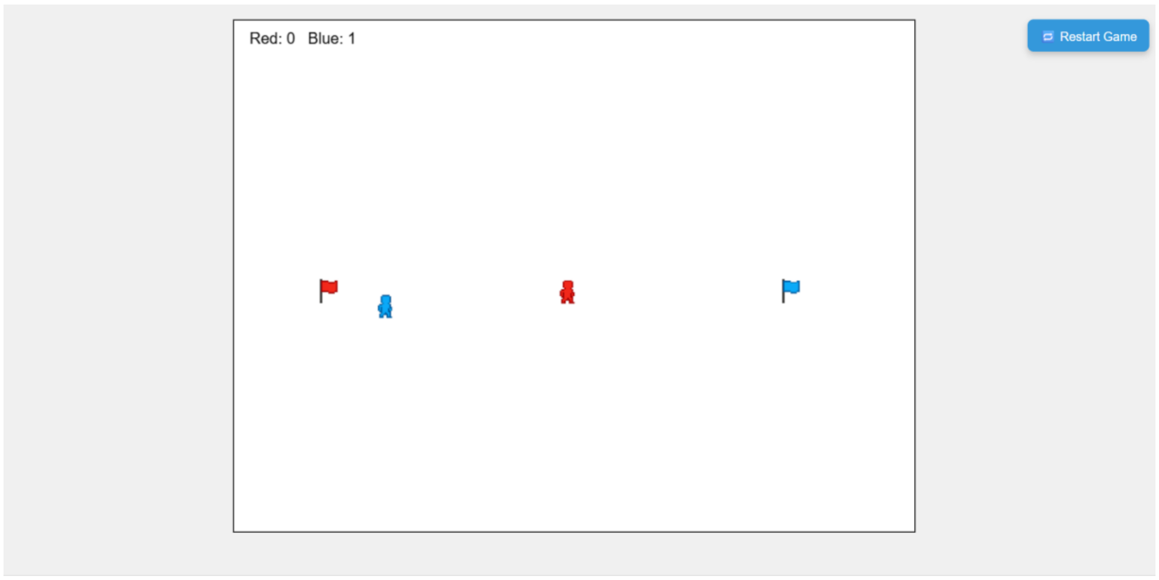
Open browser to http://localhost:3000

Input a room name and click join:



Use arrow keys to move

Capture opponent's flag to score:



Click "Reset" to manually restart

Player Identity Behavior

The system uses the browser's `localStorage` to persist the `room` name and the `playerId`. This allows seamless reconnection in case of accidental refresh or disconnect.

If the same browser is used to rejoin the game, the system will automatically restore the previous session using the stored `playerId`. The player will be placed back in the same team and position as before.

To test the game with multiple players, it is essential to:

- Use different browsers, or

- Open multiple incognito/private windows

This ensures that each player instance has a unique identity and does not overwrite or interfere with others.

Troubleshooting

Issue	**Solution**
Connection failed	Check Docker containers
Movement not working	Refresh browser
No other players	Check multiple connections
Game not resetting	Verify server logs