Game Project Report

CIVADE Thomas, ERGIN Seçkin Yağmur
and GROUSSON Dylan

November 28, 2025

https://github.com/Luminosaa/IDS-ERGIN-CIVADE

1 Deployment

To deploy the game, RabbitMQ needs to be installed and running. Additionally, Maven must be installed. Once RabbitMQ is up and Maven is set up, the game must be compiled by running the command: *mvn package*.
To start the game server, a chief server or a normal server needs to be launched:
Chief server: *java -jar game/target/Server-jar-with-dependencies.jar chief*
Normal server: *java -jar game/target/Server-jar-with-dependencies.jar chief*
A client can only join the game if a chief server is active, the client application can be started by executing the following command:
Graphic client: *java -jar game/target/GraphicalClient-jar-with-dependencies.jar [player name]*
Normal client: *java -jar game/target/Client-jar-with-dependencies.jar [player name]*

2 Architecture

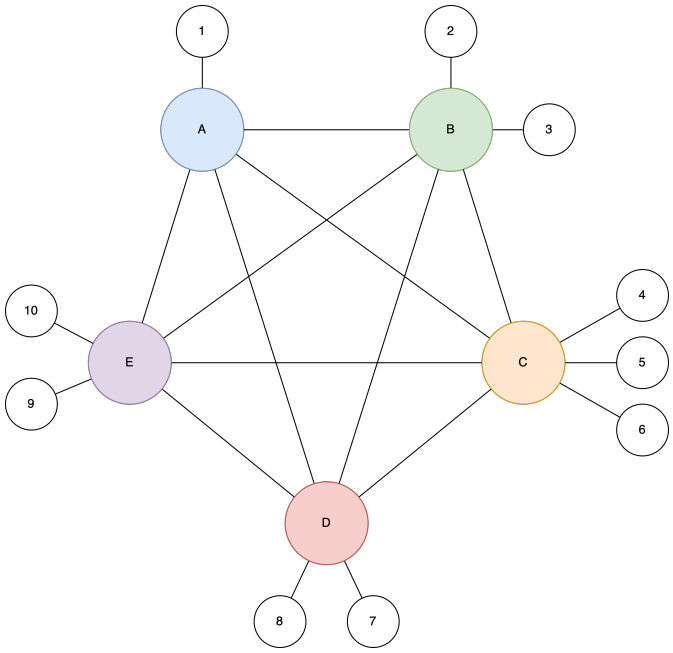

Figure 1: Our game system architecture

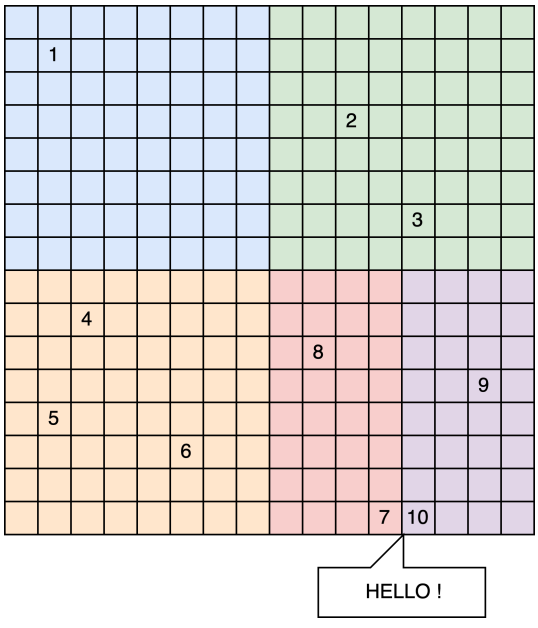

Figure 2: How we represent it

2.1 Server

Servers communicate with each other using RabbitMQ message queues. Each server is responsible for managing a specific section of the game grid which is represented as a *Map<String, Set<Point>>* where the key is the server's queue name, and the value is the coordinates managed by the server. Players are tracked in a *Map<String, Point>*, where the key is the player's unique identifier (queue name), and the value is the player's current position on the grid. One server acts as the chief server and manages the architecture, handling server start/stop events and maintaining the list of active servers (serverArchitecture).

Players

Each server is responsible for managing the players in its assigned section. When a player logs in, if there is available space, the chief server assigns them to a random server. otherwise, it rejects the login. On logout, the server removes the player and notifies all other servers to update their records. The servers also track the players existance and positions, updates them as they move within the grid.

Movement

For each move, servers calculate the new coordinates and check for collision between players by comparing their coordinates. If a player moves within the same server's area and there's no collision, the server updates and shares the new position with all players and other servers. If there's a collision, the move is rejected. For moves across server, the player is moved to the new server and the information about the player and the new position is sent to the new server.

Hello

When a player moves next to another player, the server notifies both players. If a move is made to the boundary of a server, it checks with the neighboring server if there is a player on its boundary next to it's player. If there is, the neighboring server notifiese its player and notifies the other server, which leads to other server notifying its player.

2.2 Client

The client establishes a RabbitMQ connection and channel to communicate with servers, it listens for messages on its receive_queue which is the client's unique queue name using a DeliverCallback. To keep a track of the players, the client maintains a map where the key is the player's unique identifier (queue name), and the value is the player's current position on the grid. The client also keeps a local copy of the server architecture, represented as a map where each key is a server's queue name, and the value is the set of points it manages on the grid.

Login/Logout

To join the game, the client contacts the chief server, which responds with success or failure. When client disconnects, it sends a logout message and closes its connection.

Movement

The client reads movement commands (UP, DOWN, LEFT, RIGHT), sends them to the server and waits for the server's message. Based on the message, it updates the player's position, handles server switches, or shows an error if the move is invalid.

2.3 Dynamic Server Management

In the game, servers can become unavailable at any time. When this happens, the server sends a message to notify the client of the change. The client then updates its server map and adjusts the player's position accordingly. Additionally, the client listens for validation messages, confirming that the player has successfully moved to the new server. Throughout this process, both the client and server manage the player's movement and update their respective maps to reflect the new server state.

2.4 Message

Message is a class used to encapsulate the data and topics for communication between clients and servers, where topics represent the type of message and data represents the payload, which can be any object. For the data to be sent over the RabbitMQ queues, the message class includes methods to convert itself to and from a byte array.

3 Implementation

You will see here how messages are treated by the both client and server nodes, we invite you to see the Appendix **??** for more details about the implementation (formalized as discussed in the courses).

3.1 Server

- *LOGIN*: When a player tries to log in, if a free position is found, the server assigns it and replies with *LOGIN_SUCCES* and *UPDATE_SERVER_ARCHITECTURE*. It also broadcasts *ASK_FOR_PLAYERS* and *UPDATE_PLAYERS* to servers and players. Otherwise, it replies with *LOGIN_ERROR*.
- *MOVE*: If the player requests a movement to a valid position within the current server's area, the server updates the player's position and sends *HELLO* to nearby players. If the target is in another server, it sends *CROSS_SERVER_MOVEMENT*. If the move is invalid (occupied or out of bounds), it replies with *INVALID_MOVEMENT*.
- *HELLO*: Sent between nearby players to signal mutual visibility or proximity. This message is broadcast to all players involved.
- *LOGOUT*: When a player logs out, it is removed from the local server list, and *NOTIFY_LOGOUT* is broadcast to both players and servers to keep states synchronized.
- *NOTIFY_LOGOUT*: On receiving this, other servers remove the player and notify their own clients with *NOTIFY_LOGOUT*.
- *ASK_FOR_PLAYERS*: Another server requests the local player list. The current server responds with *UPDATE_PLAYERS* containing its player list.
- *UPDATE_PLAYERS*: Carries an updated list of players and is broadcast to all local players to synchronize their views.
- *CROSS_SERVER_MOVEMENT*: Used when a player moves from one server's area to another. If the destination is free and valid, the player is added, and both origin and player receive *VALID_CROSS_SERVER_MOVEMENT*. Otherwise, the origin receives *INVALID_CROSS_SERVER_MOVEMENT*.
- *CROSS_SERVER_HELLO*: Sent when a player's movement reveals neighbors in another server. Triggers mutual *HELLO* messages between cross-server players.
- *VALID_CROSS_SERVER_MOVEMENT*: Confirms that a player has successfully moved into the destination server. The origin server then removes the player from its local list.

- *INVALID_CROSS_SERVER_MOVEMENT*: The movement failed due to conflicts (e.g., occupied position), so the origin informs the player using *INVALID_MOVEMENT*.
- *UPDATE_SERVER_ARCHITECTURE*: Updates the server's view of which server handles which positions. Players no longer within the local server's region are moved using *ADD_PLAYER* and notified via *SWITCH_SERVER*.
- *ADD_PLAYER*: Inserts a player at a given position into the receiving server's player list after migration due to architecture change.
- *SWITCH_CHIEF*: Changes the leader server (chief), assigning it management of specific players or regions, typically for fault tolerance or dynamic rebalancing.

3.2 Client

- *LOGIN_ERROR* Triggered when login fails. The client prints an error message and immediately exits the application.
- *LOGIN_SUCCESS* The login was successful. The client stores the queue (i.e., communication interface) of the server it connected to.
- *UPDATE_PLAYERS* Received from the server to synchronize the local list of visible players. The client updates its map view accordingly with *displayMap()*.
- *HELLO* Informs the client that two players are close to each other. The message is printed in a human-readable format showing which player greets the other.
- *VALID_CROSS_SERVER_MOVEMENT* Confirms a successful server-to-server movement. The client updates its server queue and position map, then refreshes the displayed game state.
- *NOTIFY_LOGOUT* Tells the client that a player has disconnected. The player is removed from the local view and the map is refreshed.
- *INVALID_MOVEMENT* Indicates that a movement request was invalid (e.g., blocked or out-of-bounds). The client simply prints an error message.
- *SWITCH_SERVER* Instructs the client to update its target server queue, typically following a region change or server reassignment.
- *UPDATE_ARCHITECTURE* Updates the client's view of the global server architecture. This new information is visualized via *displayMap()*.

4 Conculsion

To conclude, this project was very fun for all of us since it gave us the opportunity to exploit all our knowledge in the field of distributed systems. It was still quite challenging when dealing with dying server nodes that need to be handled properly by other nodes (regain node area and players).

A Some Conventions

The name for our queues are the same as the name of the entity, for example: the server "server1" can receive messages by sending messages on the queue named "server1" as well.

B Server Implementation

B.1 Declarations

```
CommItf in;
Topic_t LOGIN, MOVE, LOGOUT, NOTIFY_LOGOUT, ASK_FOR_PLAYERS, UPDATE_PLAYERS,
    INVALID_MOVEMENT, CROSS_SERVER_MOVEMENT, VALID_CROSS_SERVER_MOVEMENT,
    INVALID_CROSS_SERVER_MOVEMENT, INVALID_CROSS_SERVER_ARCHITECTURE,
    ADD_PLAYER, LOGIN_ERROR, LOGIN_SUCCES SWITCH_SERVER, SWITCH_CHIEF;
Data_t ByteArray
Message_t <Topic_t, Data_t>;
Tuple_t (x, y);

// The list of the players currently handled in the server's node
players = <String, Point>[];
// The current server architecture, mapping the server node names with the
// positions that it handle
serverArch = <String, Point[]>[];
```

B.2 Rules

B.2.1 *LOGIN* topic

```
in ? <LOGIN, d> =>
    player = d,
    pos = randomFreePos(),
    if (pos):
        players.set(<player, pos>),
        player ! <LOGIN_SUCCES, in>,
        player ! <UPDATE_SERVER_ARCHITECTURE, serverArch>,
        broadcastServers(<ASK_FOR_PLAYERS, in>),
        broadcastServers(<UPDATE_PLAYERS, players>),
        broadcastPlayers(<UPDATE_PLAYERS, players>);
    else:
        player ! <LOGIN_ERROR, null>;
```

B.2.2 *MOVE* topic

```
in ? <MOVE, d> =>
    move = d,
    player = move.getPlayer(),
    currPos = players[player],
    newPos = computePos(currPos, move.getDir()),
    currServerArea = serverArch[in],
    if (newPos in currServerArea):
        // movement handled by the current node
        if (newPoint in players.getValues()):
            // already a player on the desired position
            player ! <INVALID_MOVEMENT, null>;
        else:
            // the desired position is available
            players.set(<player, newPos>),
```

```
            aroundPos = getArroundPlayerPos(player),
            for pos in aroundPos:
                if pos in currServerArea:
                    // near position in current node
                    if pos in players:
                        // there is a player arround the moving one
                        aroundPlayer = findPlayerAtPos(pos),
                        player ! <HELLO, (player, aroundPlayer)>,
                        aroundPlayer ! <HELLO, (player, aroundPlayer)>,
                else:
                    nextNode = getNodeFromPos(pos),
                    if (nextNode):
                        // pos is handled by another node
                        crossServerMove = crossServerMovement(in, nextNode,
                            player, pos),
                        nextNode ! <CROSS_SERVER_HELLO, crossServerMove>,
            broadcastServers(<UPDATE_PLAYERS, players>),
            broadcastPlayers(<UPDATE_PLAYERS, players>);
    else:
        // movement getting the player out of the node scope
        nextNode = getNodeFromPos(newPos),
        if (newServer):
            // player movement getting in another server
            crossServerMove = crossServerMovement(in, nextNode, player, newPos)
    ,
            nextNode ! <CROSS_SERVER_MOVEMENT, crossServerMove>;
        else:
            // player movement getting out of bounds
            player ! <INVALID_MOVEMENT, null>;
```

B.2.3 *HELLO* topic

```
in ? <HELLO, d> =>
    broadcastPlayers(d);
```

B.2.4 *LOGOUT* topic

```
in ? <LOGOUT, d> =>
    player = d,
    players.remove(player),
    broadcastPlayers(<NOTIFY_LOGOUT, player>),
    broadcastServers(<NOTIFY_LOGOUT, player>);
```

B.2.5 *NOTIFY_LOGOUT* topic

```
in ? <NOTIFY_LOGOUT, d> =>
    player = d,
    players.remove(player),
    broadcastPlayers(<NOTIFY_LOGOUT, d>);
```

B.2.6 *ASK_FOR_PLAYERS* topic

```
in ? <ASK_FOR_PLAYERS, d> =>
    queue = d,
    queue ! <UPDATE_PLAYERS, playerList()>;
```

B.2.7 *UPDATE_PLAYERS* topic

```
in ? <UPDATE_PLAYERS, d> =>
    broadcastPlayers(<UPATE_PLAYERS, d>);
```

B.2.8 *CROSS_SERVER_MOVEMENT* topic

```
in ? <CROSS_SERVER_MOVEMENT, d> =>
    crossServerMove = d,
    currServerArea = serverArch[in],
    newPos = crossServerMove.newPos,
    fromServer = crossServerMove.fromServer,
    toServer = crossServerMove.toServer,
    if newPos in currServerArea:
        if newPos in players:
            // the wanted position is already taken by another player
            fromServer ! <INVALID_CROSS_SERVER_MOVEMENT, crossServerMove>;
        else:
            // the wanted position is available
            players.set(<player, newPos>),
            aroundPos = getArroundPlayerPos(player),
            for pos in aroundPos:
                if pos in players:
                    // near position handle a player
                    aroundPlayer = findPlayerAtPos(pos),
                    toServer ! <HELLO, (player, aroundPlayer)>,
                    aroundPlayer ! <HELLO, (player, aroundPlayer)>,
            fromServer ! <VALID_CROSS_SERVER_MOVEMENT, crossServerMove>,
            player ! <VALID_CROSS_SERVER_MOVEMENT, crossServerMove>,
            broadcastServers(<UPDATE_PLAYERS, players>),
            broadcastPlayers(<UPDATE_PLAYERS, players>);
    else:
        // the node doesn't handle the desired position
        fromServer ! <INVALID_CROSS_SERVER_MOVEMENT, crossServerMove>;
```

B.2.9 *CROSS_SERVER_HELLO* topic

```
in ? <CROSS_SERVER_HELLO, d> =>
    crossServerMove = d,
    newPos = crossServerMove.newPos,
    fromServer = crossServerMove.fromServer,
    if newPos in players:
        player = crossServerMove.player,
        aroundPlayer = findPlayerAtPos(newPos),
        fromServer ! <HELLO, (player, aroundPlayer)>,
        aroundPlayer ! <HELLO, (player, aroundPlayer)>;
```

B.2.10 *VALID_CROSS_SERVER_MOVEMENT* topic

```
in ? <VALID_CROSS_SERVER_MOVEMENT, d> =>
    crossServerMove= d,
    removePlayer(crossServerMove.player;
```

B.2.11 *INVALID_CROSS_SERVER_MOVEMENT* topic

```
in ? <INVALID_CROSS_SERVER_MOVEMENT, d> =>
    crossServerMove = d,
    crossServerMove.player ! <INVALID_MOVEMENT, null>;
```

B.2.12 *UPDATE_SERVER_ARCHITECTURE* topic

```
in ? <UPDATE_SERVER_ARCHITECTURE, d> =>
    serverArch = d,
    currServerArea = serverArch[in],
    broadcastPlayers(d),
    playersToRemove = getPlayersToRemove(),
    for p in playersToRemove:
        pos = players[p],
        nextNode = getNodeFromPos(pos),
        nextNode ! <ADD_PLAYER, (p, pos)>,
        p ! <SWITCH_SERVER, nextNode>,
        players.remove(p);
```

B.2.13 *ADD_PLAYER* topic

```
in ? <ADD_PLAYER, d> =>
    player, position = d
    player.set(<player, position);
```

B.2.14 *SWITCH_CHIEF* topic

```
in ? <SWITCH_CHIEF, d> =>
    - getting the current chief server cells/players
    - becomming the chief node until it dies;
```

C Client Implementation

C.1 Declarations

```
CommItf in, serverQueue;
Topic_t NOTIFY_LOGOUT, UPDATE_PLAYERS, INVALID_MOVEMENT,
    VALID_CROSS_SERVER_MOVEMENT, LOGIN_ERROR, LOGIN_SUCCESS, SWITCH_SERVER,
    UPDATE_ARCHITECTURE;
Data_t ByteArray
Message_t <Topic_t, Data>;

// The list of the players currently handled in the server's node
players = <String, Point>[];
// The current server architecture, mapping the server node names with the
// positions that it handle
serverArch = <String, Point[]>[];
```

C.2 Rules

C.2.1 *LOGIN_ERROR* topic

```
in ? <LOGIN_ERROR, d> =>
    print("Login error"),
    exit(1);
```

C.2.2 *LOGIN_SUCCES* topic

```
in ? <LOGIN_SUCCESS, d> =>
    serverQueue = d;
```

C.2.3 *UPDATE_PLAYERS* topic

```
in ? <UPDATE_PLAYERS, d> =>
    serverPlayers = d,
    players = serverPlayers,
    displayMap();
```

C.2.4 *HELLO* topic

```
in ? <HELLO, d> =>
    (player1, player2) = d,
    print(player1 + "says hello to " + player2);
```

C.2.5 *VALID_CROSS_SERVER_MOVEMENT* topic

```
in ? <VALID_CROSS_SERVER_MOVEMENT, d> =>
    crossServerMove = d,
    serverQueue = crossServerMove.toServer,
    players.set(crossServerMove.player, crossServerMove.newPos)
    displayMap();
```

C.2.6 *NOTIFY_LOGOUT* topic

```
in ? <NOTIFY_LOGOUT, d> =>
    client = d,
    removePlayer(client),
    displayMap();
```

C.2.7 *INVALID_MOVEMENT* topic

```
in ? <INVALID_MOVEMENT, d> =>
    print("Invalid movement");
```

C.2.8 *SWITCH_SERVER* topic

```
in ? <SWITCH_SERVER, d> =>
    nextNode = d,
    serverQueue = nextNode;
```

C.2.9 *UPDATE_ARCHITECTURE* topic

```
in ? <UPDATE_ARCHITECTURE, d> =>
    serverArch = d,
    displayMap();
```