TETRIS BATTLE ROYALE

Lorenzo Rigoni
lorenzo.rigoni2@studio.unibo.it

Riccardo Moretti
riccardo.moretti6@studio.unibo.it

May 2025

Abstract

This report describes a project developed for the 2024-2025 Distributed Systems course at the University of Bologna called **Tetris Battle Royale**. This project is a multiplayer version of the classic game *Tetris*, where all players in a room play a standard game of *Tetris*. When a player completes a "Tetris" (defined as clearing four rows with a single piece), they send a broken row to all other players. One by one, all but one player will be defeated, and a winner will emerge.

AI Disclaimer

During the preparation of this work, the authors used GitHub Copilot to assist with code completion and documentation generation. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the final report/artifact.

1 Concept

Tetris Battle Royale is a multiplayer version of the classic game *Tetris*. Every game played by the users follows the original rules of the game. The game features three main screens:

- **Initial screen** where the player enters their name;
- **Search screen** where the player waits for a game;
- **Game screen** where the player plays the *Tetris* game.

When a player starts the search for a game, the system puts him in a room with a prefixed number of players. When the room reaches the number of players required, the game starts. During the game, in addition to their own screen, each player can see all other players' grids and their upcoming piece. When a player completes a Tetris, a new line with a missing space (hereafter referred to as a "broken row") will be added to the bottom of all other players' grids. The game ends when only one player remains (the winner).

1.1 Use Case Collection

Players interact with the game using their internet-connected computers. During gameplay, interactions with the system are frequent via keyboard commands. The data workload is distributed between clients and the server with frequent information exchanges.

2 Requirements

2.1 Functional Requirements

- When a player starts the game, an initial screen is shown where he enter his name, which will be used for all subsequent games.
- After entering their name, the player sees a "waiting room" screen until the room reaches the number of players required. When this number is reached, the game can start.
- *Fault tolerance*: Players send thir heartbeat to the system every 2 seconds. If a player isn't heard from in 5 seconds, the system removes him from the game.
- During gameplay, players interact in three ways:
  - They send/receive the current game state (*Tetris* grid state and current piece);
  - They send/receive a broken row after completing a Tetris;
  - They receive notifications about defeated players and game endings.
- If a player disconnects manually, they are removed from the game and appear defeated to others;
- If a player disconnects forcefully (by closing the game or network failure), the same occurs as with manual disconnection, triggered by heartbeat timeout;
- A player can pause the game locally (showing a "pause screen") while the game continues running;
- When eliminated, a player can spectate others until the game ends;
- The game ends when remains only one player alive, which is the winner.

2.2 Non-Functional Requirements

- The game must run on Windows, Unix and macOS;
- The code must be maintainable and extensible;
- The game must use resources efficiently to ensure smooth local and remote experiences.

3 Design

3.1 Architecture

The project follows MVC (Model-View-Controller) architecture for the game component. For the distributed component, it uses a **client-server pattern**. Specifically:

- The architecture is centralized with one always-active server;
- Clients connect to the active server;
- If the server is unavailable, a backup server starts and replace it.

Initially, we considered also the *peer-to-peer* pattern but we chose the *client-server* pattern for the following reasons:

- The server manages all the game states and events, the clients had only to perform the *Tetris* logics. In a P2P pattern, every peer had to syncronize to all other players, increasing the complexity.
- The client had only to communicate with the server. Otherwise, in P2P every peer had $N - 1$ connections to manage.
- The *client-server* pattern is easier to debug and mantain.

3.2 Infrastructure

- Each player is a client, with 5 players required per game;
- One server manages rooms;
- For development simplicity, clients and server run on the same machine, though clients can be distributed worldwide;
- No persistent data is stored - game state is volatile and replicated among clients;
- Game commands are processed locally to reduce server throughput;
- No authentication is performed;
- The clien notifies the server of game events;
- The game loop runs on each client;
- Rendering occurs client-side.

3.3 Modelling

The distributed component's entry point is the **Client** class, which handles:

- Player name and server-assigned ID;
- The primary and the backup server socket (IP and port);
- The player's MVC Controller.

Operations include:

- Connecting to the server;
- Sending/receiving game state;
- Sending/receiving broken rows;
- Handling defeat/game over;
- Graceful disconnection.

The server operations are handled by the **Server** class, which can broadcast messages to room players (managed by **GameRoom** class). In order to guarantee the correct switch from the primary server to the backup, the state of the server is managed by the **ServerState** class, which writes and reads the state of the server on a JSON file stored locally (the primary and the backup server will run on the same machine). The MVC Controller interacts only with the Client class, separating game logic from distributed components.

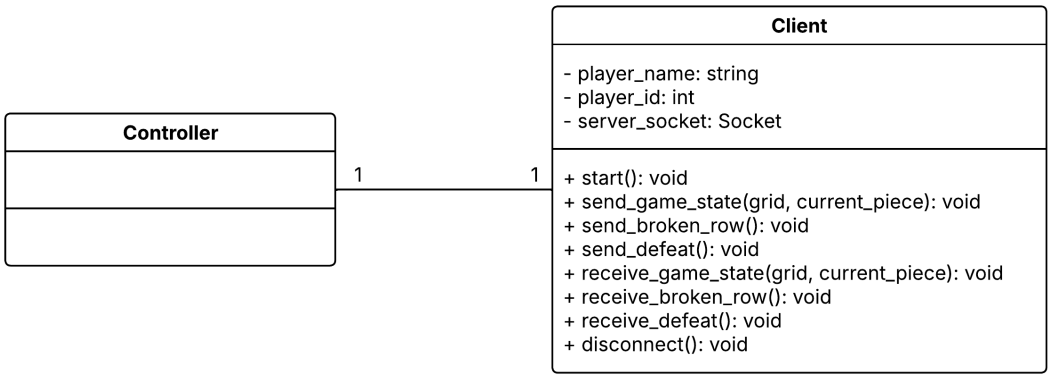

Figure 1: Position of Client in MVC architecture

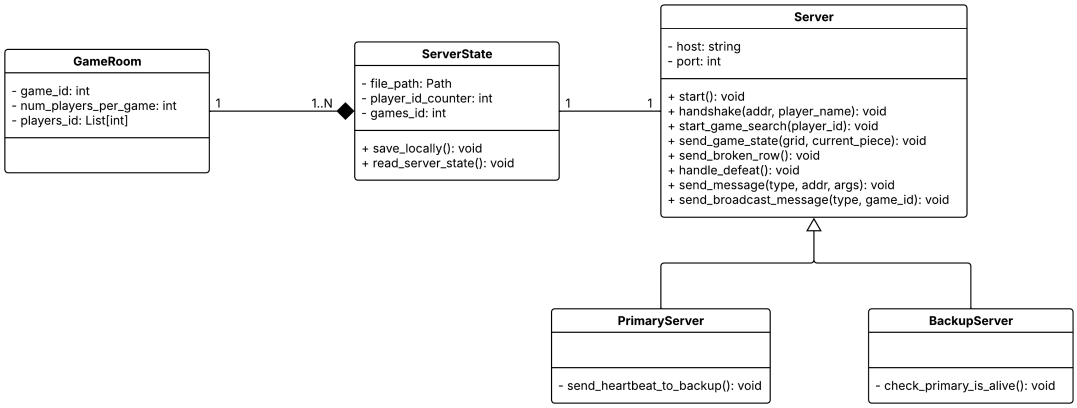

Figure 2: Server and his classes

3.3.1 Events

Main events include:

- Player searches for a game;
- Room reaches the required number of players;
- Game starts;
- Player places a piece;
- Player clears a row;
- Player loses;
- Player wins.

3.4 Interaction

The system uses an *authoritative server* pattern, with the server managing room events while clients handle local game commands.

3.4.1 Handshake

When a new client connects:

1. He sends a handshake message;
2. Server assigns and sends a client ID.

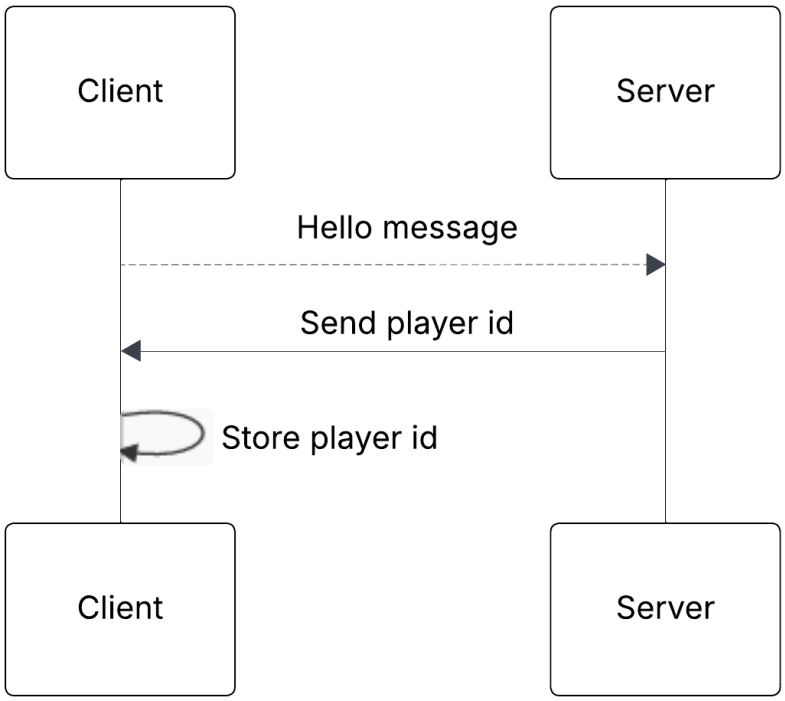

Figure 3: Sequence diagram of client-server handshake

3.4.2 Game Search

After receiving an ID, the client searches for a game. The server finds an available room or creates one, then notifies clients when the game starts.

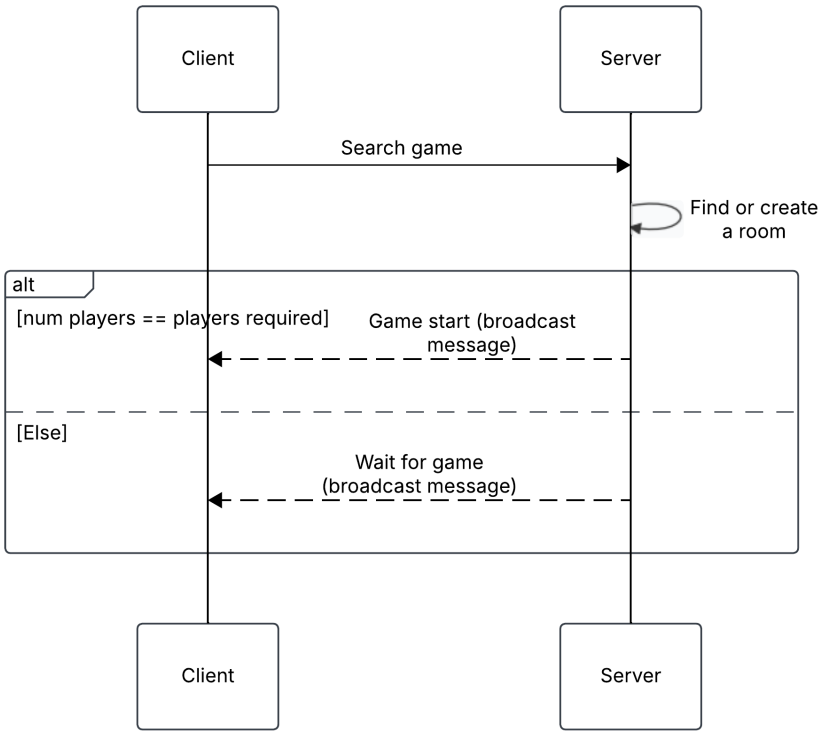

Figure 4: Sequence diagram of game search

3.4.3 Room Events

During gameplay:

- Game state changes are sent broadcast;
- Broken rows are sent to all players;
- Defeats are announced.

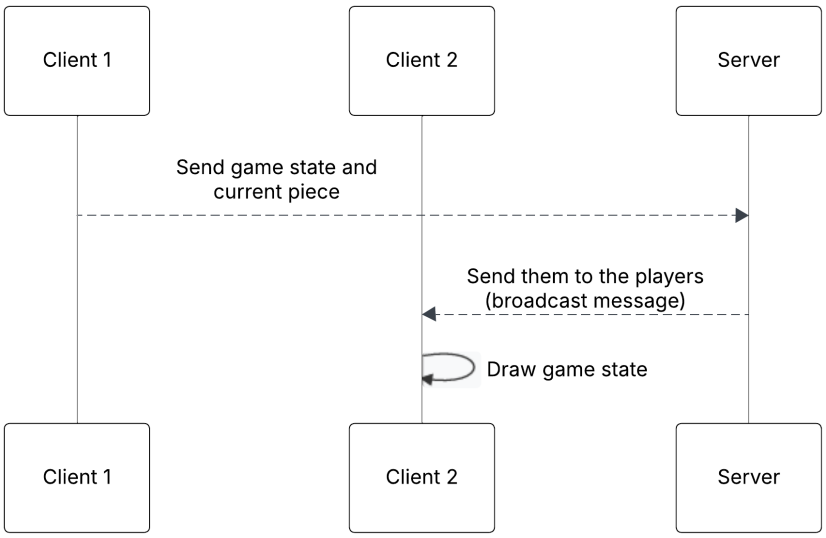

Figure 5: Sequence diagram of game state messages

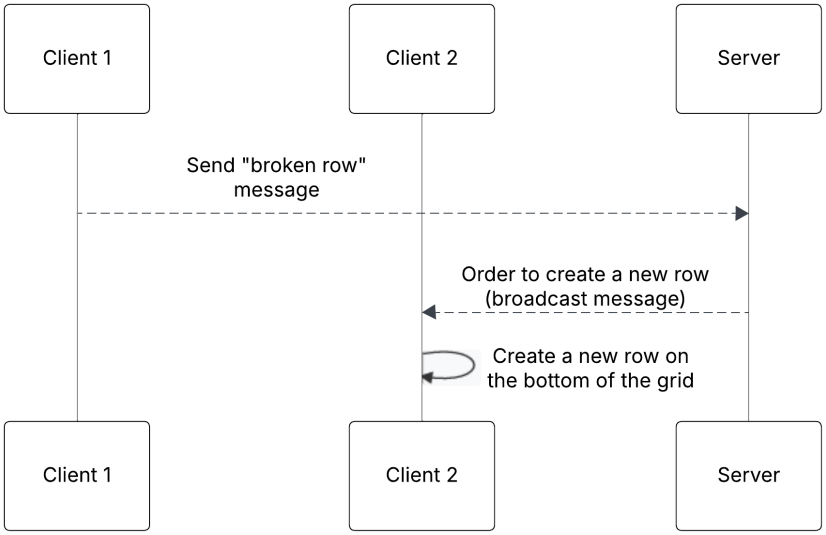

Figure 6: Sequence diagram of broken row messages

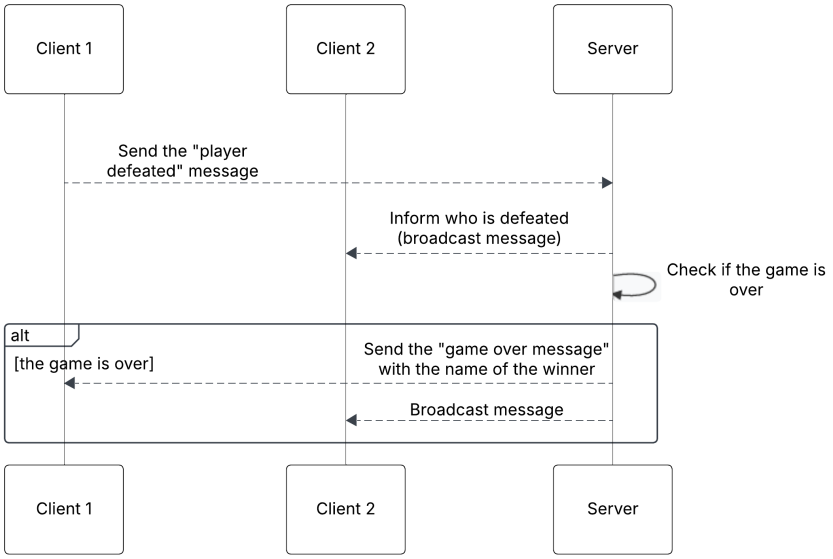

Figure 7: Sequence diagram of player defeat

3.5 Behavior

Client-side is stateful:

- Starts with name entry;
- Waits in lobby until enough players;
- Continues until defeat or victory;
- Allows spectating after defeat;
- Can exit anytime.

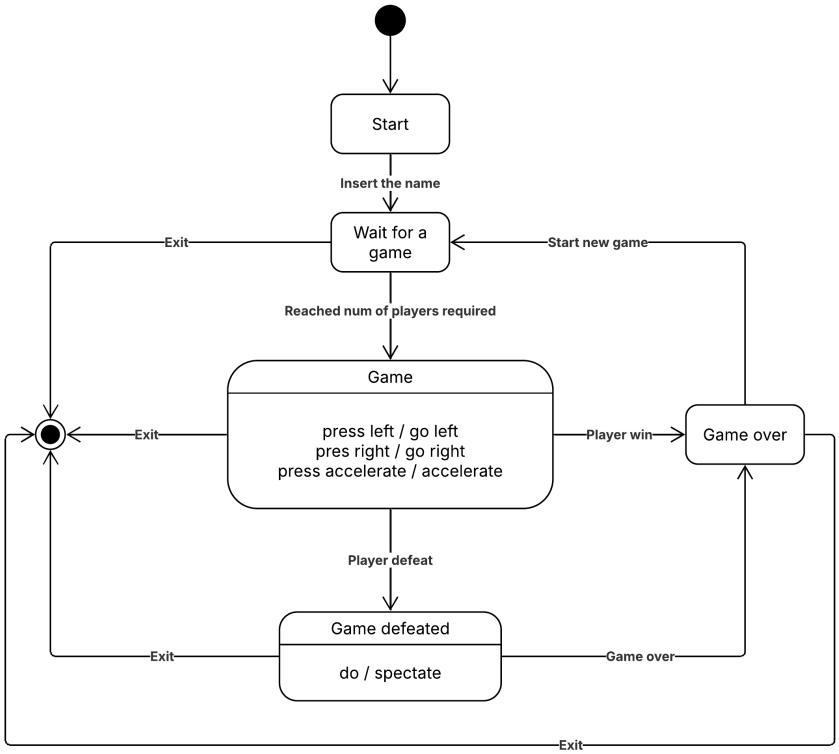

Figure 8: State diagram of client-side system

The server is stateless, acting as the authoritative source for room events.

3.6 Data and Consistency Issues

No persistent data is stored. The authoritative server ensures consistency by managing all updates.

3.7 Fault-Tolerance

- Game state is managed client-side for availability;
- Heartbeat monitoring detects disconnections (for clients but also for server);
- Backup server activates if primary fails;
- If a client quits the game or crashes, he will considered defeat and he will be removed from the game.

3.8 Availability

In order to increase the availability of the system, a backup server is also started with the primary one. It's behaviour is described as it follows:

- The client sends a *ping* packet to the primary server every 2 seconds and waits for a reply;
- If the primary server don't reply in 3 seconds, the client sends a message to the backup server;
- The backup server uses a *heartbeat* logics with the primary server;
- When the backup server receives a message from a client, he checks the last heartbeat received from the primary server. If the time gap is greater than 3 seconds, the backup server replaces the primary and reads the previous server state.

In this way, the problems concerning the network partitioning are resolved:

- The backup server is activated only after a double check on primary server;
- The server state is managed by the ServerState class, separatly from the server logics and saved on a JSON file.

4 Implementation

For the communications between the client and the server, the system uses the **UDP protocol**. We choose this protocol because it is good for his low-latency, which is important in a real-time application. To represent data during the transport, the system uses the *BSON* (*Binary JSON*) format in order to avoid latency bandwith waste. Every packet is formed as follows:

- The type of the package;
- The data to transfer.

```
{
  "type": "ping"
  "data": {}
}
```

The types of package and the methods for encoding and decoding data are implemented in the class *Package*.

```
from bson import BSON
class Package:
  """Manage the content and the delivery of packages"""
  HAND_SHAKE = "hand_shake"
  PING = "ping"
  HEARTBEAT = "heartbeat"
  START_SEARCH = "start_search"
  WAIT_FOR_GAME = "wait_for_game"
  PLAYER_LEFT = "player_left"
  GAME_START = "game_start"
  SEND_ROW = "send_row"
  ROW_RECEIVED = "row_received"
  GAME_STATE = "update_state"
  PLAYER_DEFEATED = "player_defeated"
  GAME_OVER = "game_over"
  ERROR = "error"

  @staticmethod
  def encode(packet_type, **kwargs):
    """Create the packet"""
    packet = {"type": packet_type, "data": kwargs}
    return BSON.encode(packet)

  @staticmethod
  def decode(data):
    """Decode the recived packet"""
    packet = BSON(data).decode()
    return packet["type"], packet["data"]
```

The following points describe how the iteractions between the client and the server are implemented.

- The first iteraction is the handshake. The new client sends an handshake tipe packet to the server without any data.

```
    self.send(Package.HAND_SHAKE)

    def send(self, packet_type, **kwargs):
      '''Send a packet to the server'''
      data = Package.encode(packet_type, **kwargs)
      self.client_socket.send(data)
```

  The server calculates the client identifier and sends it to him. After that, starts the game search.

```
    def hand_shake(self, addr):
      '''When client connects to the server for the first time
         , the server sends the id of the player associated
         to the client.'''
      actual_counter = self.player_id_counter
      self.player_id_counter += 1
      self.player_and_addr[actual_counter] = addr
      self.addr_and_player[addr] = actual_counter
      self.player_and_game[actual_counter] = self.
         check_availables_games()
      self.send_message(Package.HAND_SHAKE, addr, player_id =
         actual_counter)
      self.start_game_search(actual_counter,self.
         player_and_game[actual_counter])
```

- With the aim to control if a player is still connected, a logic of heart-beat packet has been implmented. The client every 2 seconds sends to the server an *heartbeat* type packet.

```
    def send_heartbeat(self):
      '''Send the client heartbeat to the server'''
      while self.running:
        self.send(Package.HEARTBEAT)
        time.sleep(2)
```

  The server, every 2 seconds, checks all the clients connected. If a client havent't send the heartbeat in five seconds, he will kicked out.

```
    def timeout_monitor(self):
      '''Check if all the clients are still connected'''
      while True:
        time.sleep(2)
        current_time = time.time()
```

```
        for addr, p_time in list(self.last_seen.items()):
          if current_time - p_time > 5:
          self.handle_disconnection(self.addr_and_player[addr
             ])
```

- When the player start the search of a game, the server checks if there is a room available. If there isn't, he creates a new one. After that, the server checks if that room have a number of players enough to start a game. If it is, sends to all the player of the room the packet with the command to start the game, however the server sends a packet of type *wait for game* with the actual number of players. In both case, the packet is send in a broadcast way.

```
    def start_game_search(self, player_id, game_id):
      '''Start the search of a game'''
      if len(self.games) == 0 or game_id == -1:
        game_id = self.games_id
        self.games.append(GameManager(self.games_id))
        self.games_id += 1

      if self.games[game_id].add_player_to_game(player_id):
        self.send_broadcast_message(game_id, None, Package.
           GAME_START)
      else:
        self.send_broadcast_message(game_id, None, Package.
           WAIT_FOR_GAME, number_of_players = len(self.games[
           game_id].players_id))
```

- During the game, the client and the server exchange the packets on the events of the game:
  - Client

```
      while self.running:
        try:
          package = self.client_socket.recv(10240)
          type, data = Package.decode(package)
          self.handle_received_packet(type, data)
        except Exception as e:
          print(f"Client error: {e}")
          traceback.print_exc()
          self.running = False

      def handle_received_packet(self, type, data):
        '''Handle the data received from the server'''
        if type == Package.HAND_SHAKE:
          self.player_id = int(data["player_id"])
        elif type == Package.WAIT_FOR_GAME:
          self.wait_for_game(int(data["number_of_players"
             ]))
```

```
        elif type == Package.GAME_START:
          self.start_game()
        elif type == Package.GAME_STATE:
          self.receive_game_state(int(data["sender"]),
              data["grid"], data["current_piece"])
        elif type == Package.ROW_RECEIVED:
          self.receive_broken_row()
        elif type == Package.PLAYER_DEFEATED:
          self.receive_defeat(int(data["player_id"]))
        elif type == Package.GAME_OVER:
          self.receive_game_over(int(data["winner"]))
```

– Server

```
      while True:
        try:
          data, addr = self.socket.recvfrom(10240)
          p_type, p_data = Package.decode(data)
          self.handle_received_packet(addr, p_type, p_data
              )
        except Exception as e:
          print(f"Error in the receivment of a packet on
              the server: {e}")
          traceback.print_exc()

      def handle_received_packet(self, addr, p_type,
          p_data):
        '''Handle the client requests'''
        if p_type == Package.HAND_SHAKE:
          self.hand_shake(addr)
        elif p_type == Package.HEARTBEAT:
          self.last_seen[addr] = time.time()
        elif p_type == Package.START_SEARCH:
          self.start_game_search(int(p_data["player_id"]))
        elif p_type == Package.GAME_STATE:
          self.send_game_state(int(p_data["player_id"]),
              p_data["grid"], p_data["current_piece"])
        elif p_type == Package.SEND_ROW:
          self.send_broken_row(int(p_data["target"]))
        elif p_type == Package.PLAYER_DEFEATED:
          self.handle_defeat(int(p_data["player_id"]))
        elif p_type == Package.PLAYER_LEFT:
          self.handle_disconnection(int(p_data["player_id"
              ]))
```

4.1 Technological details

The project is developed in Python. For the distributed part, the system uses the following libraries of Python:

- socket;
- threading;
- time;
- BSON.

For the game part, it uses *PyGame* library.

5 Validation

5.1 Automatic Testing

Unit tests using PyTest cover:

- Game mechanics;
- Room management;
- Client-server communication.

To run the tests, if you have Poetry, use the command `poetry install` and, then, use `poetry run pytest`.

5.2 Acceptance Testing

Manual testing verified event triggering and gameplay on Windows and Linux systems.

6 Release

The project has two main packages:

- `game` - Tetris logic and view, following the MVC pattern;
- `remote` - this package collects all the distributed comonents (primary and backup server, client, server state, room management and package).

Then, the project release is composed by three executables:

- `start-primary`: the executable used to start the primary server;
- `start-backup`: the executable used to start the backup server;
- `main`: the executable used to start the game client-side;

The release is available on the repository releases tab.

7 Deployment

1. Download the executables from repository releases tab;
2. Run primary server executable;
3. Run backup server executable;
4. Run main executable;
5. Enter name to begin.

8 User Guide

First of all, you have to launch the server by the exacutable downloaded. After that, you can execute the game. The first screen is the *initial screen* where you can put your name.

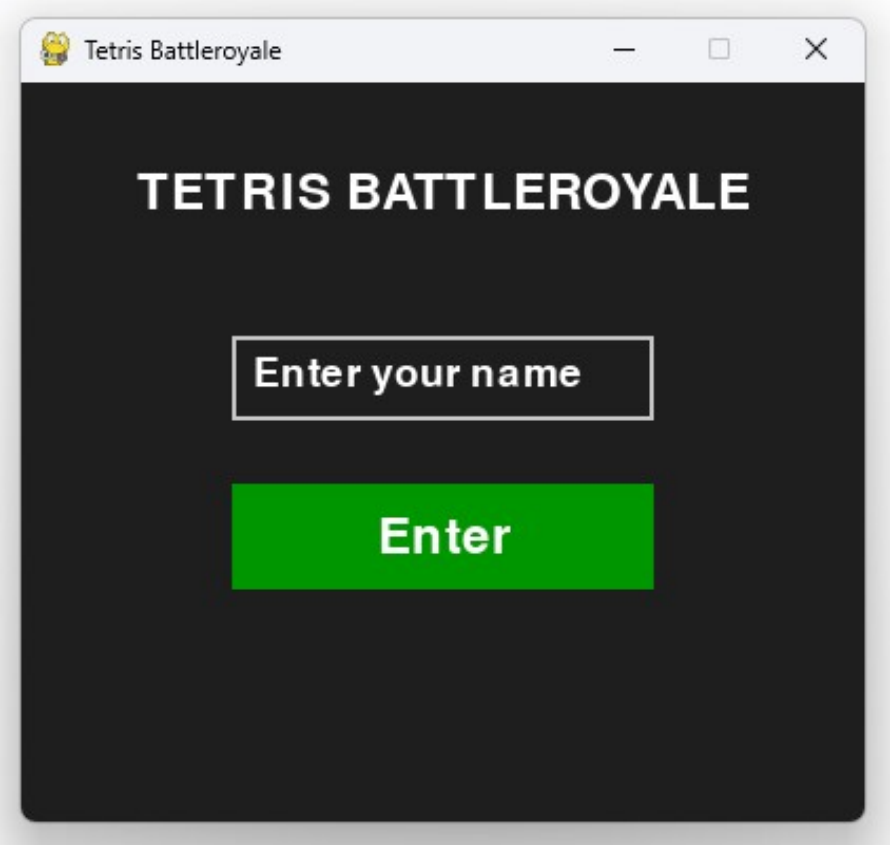

Figure 9: The initial screen of the game

Once you enterend your name, the *search screen* will appear with the actual number of players in the waiting room.

When the prefixed number of players will be reached, the *game screen* will be shown and the *Tetris* game will start. The commands are the following:

- ↓: move the piece down;
- **Hold** ↓: move the piece down faster;
- ←: shfit the piece to the left;
- →: shfit the piece to the right.
- ↑: rotates the piece.

- **Space**: drops the piece to the bottom.

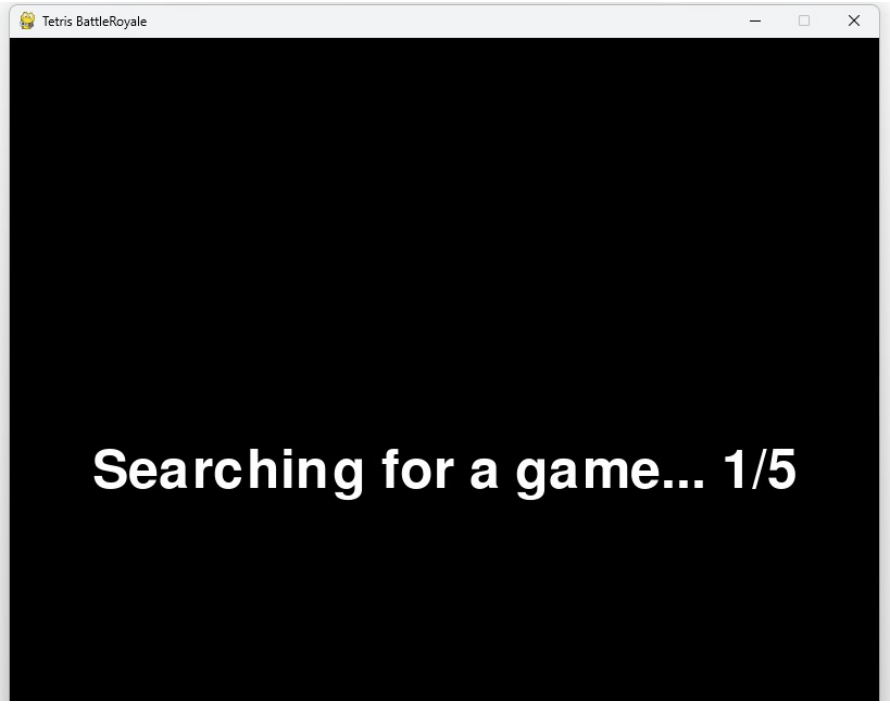

Figure 10: The search screen

In the game screen, the four screens on the left are the *opposites screens* with the name of the owner.

When a player is defeated, a red cross will appear in the player screen. When the game finishes, the name of the winner appears on the screen.

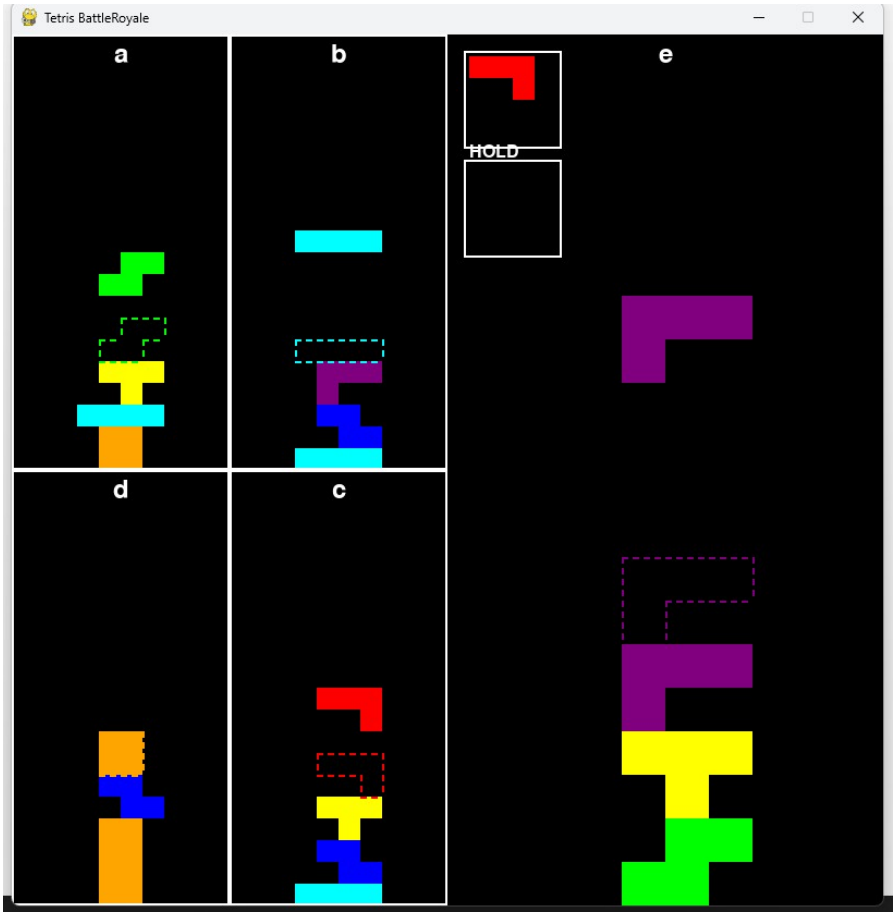

Figure 11: The game screen

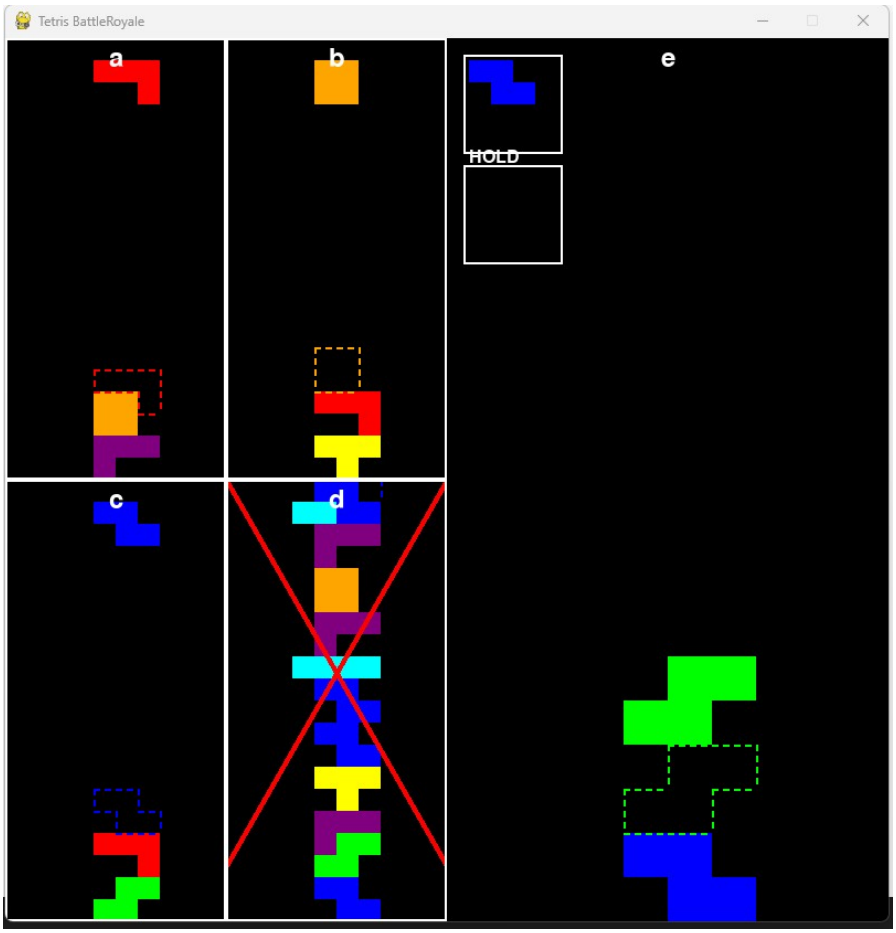

Figure 12: The defeat of player ”d”

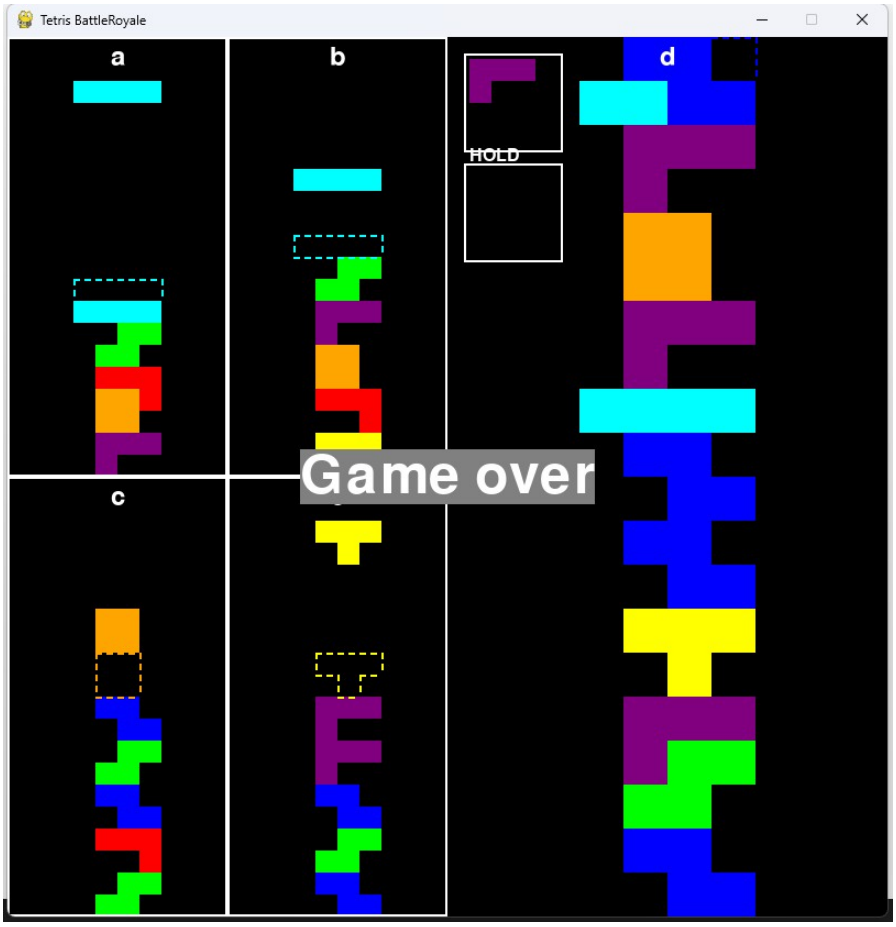

Figure 13: The defeat of a player

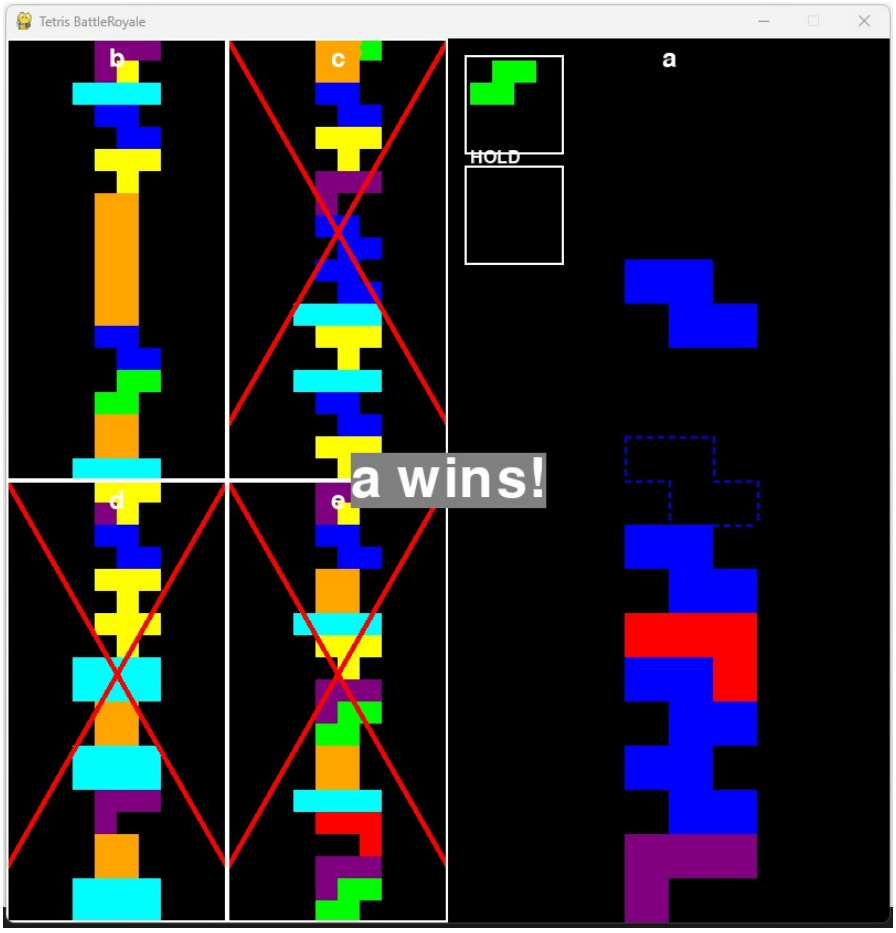

Figure 14: The game over screen with the name of the winner

9 Self-Evaluation

9.1 Lorenzo Rigoni

I really enjoyed working on this project, as it challenged me with a new way to program systems. I dedicated significant time and effort to ensure the codebase was modular and easy to extend, and I'm proud of the final result. One of the main strengths of the project is the simplicity witch players can connect and start a game. A weakness is the fact that only one server manages all the game room, so with a large number of players this could be a problem.

9.2 Riccardo Moretti

Working on this project has been a challenging but rewarding experience, our goal was to provide a polished and seamless experience for the end user and, to that end, I believe my efforts, our efforts, have been well spent.

I believe the multiplayer feels very "alive" in the sense that despite being a game that is played remotely the players can really appreciate how close the enemies feel thanks to their opponents game state being always on screen and updated to their latest move.