Distributed Systems

Project 2: Multiplayer Tetris

T6G01

David Reis *up201607927*
Francisco Pinho *up201303744*
José Gomes *up201308242*
Maria de Abreu *up201306229*

Table of Contents

Introduction

The application consists of a multiplayer tetris game. The users join a server and can create a lobby or join one created by other players, and when all the players signal themselves as ready the game will start. Each lobby can have up to 4 users and a minimum of 2. While in-game all players can see their opponents game, as well as their scores.

This report will explain its functioning and development, starting with a description of the solution's architecture, followed by a section on its implementation, a description of relevant issues found along the development and finally the conclusions the group took from this project.

Architecture

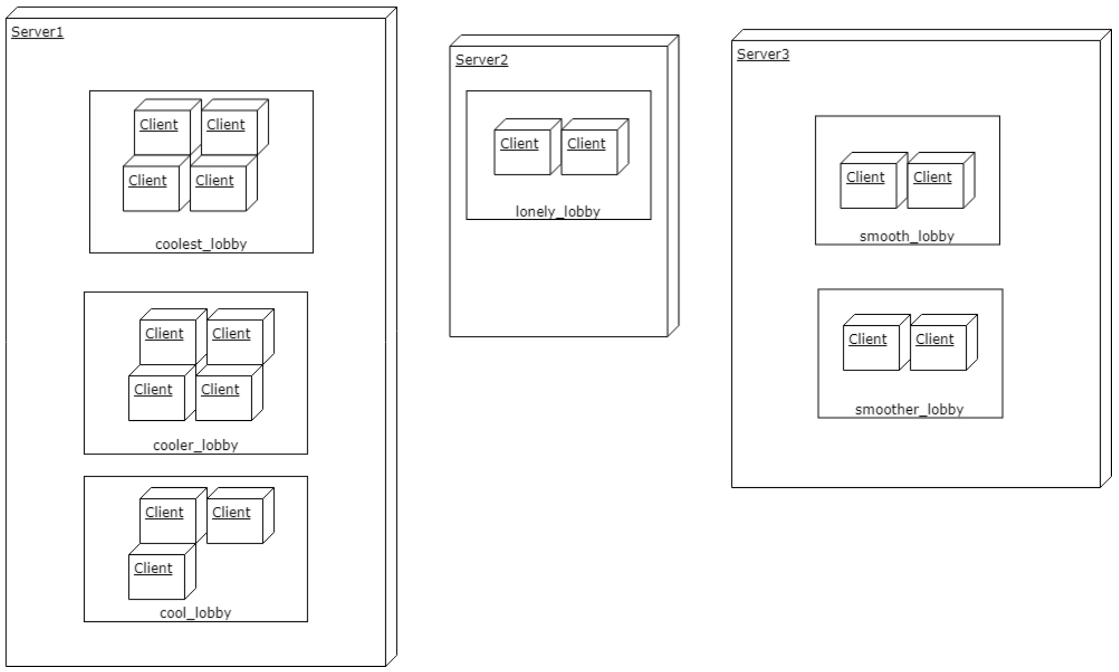

Figure 1: High level view of the architecture

Clients are connected through an application with LibGDX frontend, which is a game engine and custom network code developed for the purposes of this project. Servers were entirely developed by ourselves with the exception of the use of the javax.json library.

Clients can interact with servers in a short lived connections like polling for the list of lobbies or the list of players in the current lobby but once they join a lobby, logical constructs stored within the server itself they maintain a persistent connection with the lobby. Lobbies function as middlemen between clients, forwarding messages that come from a client to one or all clients.

Our application's protocol is based on several messages, such as:

- **CREATE** - This message is sent by the client to the server whenever the client wants to create a new lobby, the message consists of the keyword, lobby name, player name and ends with a CRLF
  ```
  CREATE <lobby> <player_name> CRLF
  ```

- **JOIN** - This message is sent by the client to the server whenever the client wants to join a lobby, the message consists of the keyword, lobby name, player name and ends with a CRLF
  ```
  CONNECT <lobby> <player_name> CRLF
  ```

- **READY** - The READY message consists on the keyword READY coupled with the username of the player, and it ends with a CRLF. It's used to signal which players are ready to start playing in a certain lobby. When all of the players (at least two are necessary) are ready, the game will begin.
  ```
  READY <player_username> CRLF
  ```

- **BEGIN** - The BEGIN message contains the keyword BEGIN, the number of players that are ready to start playing and is ended with a CRLF. It's used to trigger the beginning of the game, signaling it has started.
  ```
  BEGIN <nr_players_ready> CRLF
  ```

- **GAMESTATE** - The GAMESTATE message carries the state of the game, including board and score of a certain player. It consists on the keyword GAMESTATE followed by the username of the player, the lobby and server where he is playing and his current score, followed by a CRLF. Then the board contents are included, ended with a CRLF.
  ```
  GAMESTATE <player_username> <lobby> <server> CRLF
  <board_contents> CRLF
  ```

- **GAMEOVER** - The GAMEOVER message signals the end of the game for a certain player, consisting on the keyword couple with the username and ended with a CRLF.
  ```
  GAMEOVER <player_username> CRLF
  ```

- **GAMEENDED -** The GAMEENDED is a message sent by the lobby to all clients when everyone has reached a gameover state, consisting on the keyword coupled with the final scores of all players and ended with a CRLF.
  ```
  GAMEENDED <final_player_scores> CRLF
  ```

- **SWAP** - The SWAP message consists on the keyword followed by the player's username, the lobby he's in, and the character correspondent to the pieces to swap, ended with a CRLF.
  ```
  SWAP <player_username> <lobby> <player_piece>
  <piece_to_swap> <opponent_username> CRLF
  ```

- **SWAPRESPONSE** - This message is sent as response to SWAP and is constituted by the keyword, the other player's username and his piece, ending also with a CRLF.
  ```
  SWAPRESPONSE <username> <piece> CRLF
  ```

- **LISTPLAYERS -** This message is sent by the client as a request for the list of the players currently connected to the lobby in the GUIWaitLobby screen and is constituted by the keyword and the name of the lobby, ending also with a CRLF.
  ```
  LISTPLAYERS <lobby> CRLF
  ```

- **ASKLIST** - This message is sent by the client as a request for the list of the lobbies in the chosen server and is constituted by the keyword and the name of the server, ending also with a CRLF.
  ```
  ASKLIST <server> CRLF
  ```

- **TESTCONNECTION -** This is a message sent by the client to a server as a way to check for server availability, consisting on the keyword coupled with a CRLF.
  ```
  TESTCONNECTION CRLF
  ```

- **ACKNOWLEDGED** - This is a message sent by the server to a client as a response to TESTCONNECTION, consisting on the keyword coupled with a CRLF.
  ```
  ACKNOWLEDGED CRLF
  ```

- **REPLICATE** - This is a message sent by a server to all other servers as a means to replicate lobbies where the game has already began, consisting on the keyword, server name and lobby name, ending with a CRLF.

```
REPLICATE <server> <lobby> CRLF
```

- **DELETE -** This is a message sent by the server to all other servers as a means of deleting lobbies from games that have already ended, consisting on the keyword, server of origin name and lobby name,ending with CRLF.
  ```
  DELETE <server> <lobby> CRLF
  ```

- **JOINED/CHANGEDSERVER/RECONNECTED -** This is a message sent by the server as a response to a request for a client to join a lobby, the message consists of the keyword which is irrelevant for the client and the lobby port that the client will need to establish a connection to the newly created socket in the lobby, the message ends with a CRLF.
  JOINED/CHANGEDSERVER/RECONNECTED `<lobby_port> CRLF`

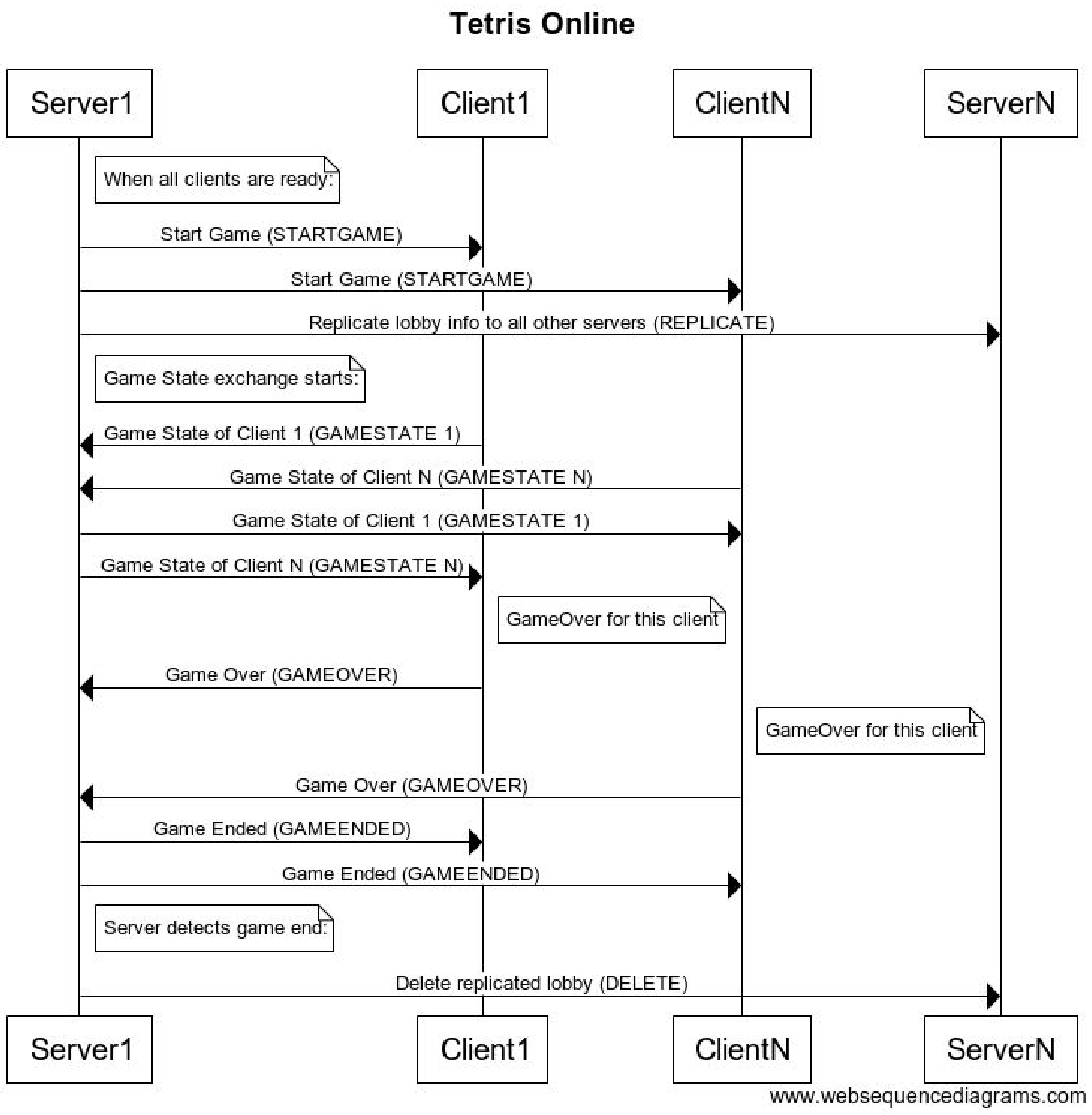
Tetris Online
Server1
Client1
ClientN
ServerN
When all clients are ready:
Start Game (STARTGAME)
Start Game (STARTGAME)
Replicate lobby info to all other servers (REPLICATE)
Game State exchange starts:
Game State of Client 1 (GAMESTATE 1)
Game State of Client N (GAMESTATE N)
Game State of Client 1 (GAMESTATE 1)
Game State of Client N (GAMESTATE N)
GameOver for this client
Game Over (GAMEOVER)
GameOver for this client
Game Over (GAMEOVER)
Game Ended (GAMEENDED)
Game Ended (GAMEENDED)
Server detects game end:
Delete replicated lobby (DELETE)
Server1
Client1
ClientN
ServerN
www.websequencediagrams.com

Figure 2: Sequence diagram of messages

Implementation

Concurrency

Server Concurrency

To handle multiple requests at once, the server has two main entry points. One entry point is for server-server communication to handle the replication of lobbies(triggered when a lobby enters the actual gameplay phase) and the deletion of said replicated lobbies once their respective gameplay session has ended. It is through a dedicated socket accepting connections from other servers on it's own port that this communication takes place, server-server communication is initiated in it's own thread through a thread pool executor on server startup through a Runnable called ServerReplicationService, this service accepts connections from the dedicated server-socket and handles the incoming requests on separate threads through an anonymous runnable that executes in it's own thread once again through the thread pool executor (TetrisServer class lines 63-71 and 216-232).

The dedicated socket for client-server communication is accepting connections on the main thread of the server but each incoming request is dispatched into it's own thread through the thread pool executor and handled in the Runnable called ClientConnectionHandler (TetrisServer class lines 54-61 and 74-88). Connections between client-server are very transient, the server responds to requests like listings, creation and joining of lobbies, however once a client is connected in a lobby the connection remains persistent but independent of the server (with the exception of any sort of fault in the client or the server where a server, same server or not, will have to reconnect the client to a lobby). Each lobby contains its own thread pool executor where up to four client listeners execute (TetrisServer class 134-155, 182-201 and TetrisLobby class 40-58).

In short, server handles requests to/from servers and clients but once clients join their intended lobbies, the lobby construct is responsible for handling persistent communication with its respective client.

Client Concurrency

Whenever clients need to send a request/message to the server or lobby, these actions are always executed through runnables that are either scheduled or executed through a scheduled or regular thread pool executor.

During gameplay there is a thread that listens for any messages sent by the lobby and updates the UI and the player's board accordingly (gui.GUIMultiGame class 81-97, network.TetrisClient class 223-329).

Message Dispatching

List Lobbies

After choosing the server, the user will be presented with a number of lobbies listed in the menu, if there are any created lobbies in that server. This list is retrieved by a message that requests the list of lobbies that are still open(less than 4 players and game hasn't started yet). The client polls the server periodically to keep display an updated list, this is done through a scheduled thread pool executor. The code for this feature can be found in gui.GUIMultiPlayer in lines 102-108,68-88 and TetrisClient class in lines 35 to 51, on the server side TetrisServer.java in lines 156 to 166.

Create Lobby

If the user wants to create his own lobby instead of joining one, he can go to the create lobby user interface, where he can write the name of the lobby that he wishes to create. The client then sends a CREATE message, which is described above, to the chosen server.

When the server receives the CREATE message from the client, it checks if there is already a lobby with the same name. If not, it creates an object from the class TetrisLobby and automatically connects the client to the newly created lobby, using the join lobby procedure which is described in the next paragraph. It also answers to the client with an CREATED message which gives it the port to which the client should connect to join the lobby.

Join Lobby

Instead of creating a lobby the user can join a lobby which has already been created by another player, and is waiting for more players to join. A lobby will be on this state until every other player is marked as ready. To join a lobby, the client sends a CONNECT message, followed by the name of the lobby it wishes to join.

When the server receives the message request to join an existing lobby, the server will add the user to the players list and set the player as "waiting", and return a message JOINED/CHANGEDSERVER/RECONNECTED back to the client which contains the TCP port through which the client should connect in order to connect to the lobby.

List Players

When a user joins a lobby or create a lobby the application will display an updated list of players currently in the lobby and their state(waiting or ready), similarly to list lobbies we use a scheduled thread pool executor to periodically poll the server. We can find the code for this in class gui.GUIWaitLobby lines 75-81, TetrisClient in lines 74-107 and server side TetrisServer in the lines 167-181.

Game Start Process

Whenever each player in a lobby is ready to start the game, he can click a button to signal his readiness in the UI itself, this sends over a "READY" message to the lobby and upon receiving that message the lobby updates a hashmap that contains the name of each player and a boolean that represents their "ready" state. Once all players are detected as ready by the lobby a "BEGIN" message is sent to all the clients so that they can transition into the gameplay screen. We can find this feature in the com.sdis.tetris.gui.GUIWaitLobby class in lines 112-126, ...network.TetrisClient in lines 178-179 and 349-363, on the server side TetrisLobby class in lines 135 to 138 and in the lines 128 to 130.

Gamestate

In order to let players see each other's games/boards on their screens, their gamestates must be sent over to other players. As such, GAMESTATE messages are sent periodically, every 5 frames of game to be precise, with the respective player's username and score so the boards can be updated on everyone's screens seamlessly. The board contents are first converted into a JSON string and sent in this form, and when they're received they're converted back into an object. Code for this can be found in gui.GUIMultigame class in lines 435-446, network.TetrisClient in lines 181-185 and in the server side at network.TetrisLobby in lines 161-178.

Swap Pieces

Whenever a player clears 3 or more lines in a row during the game of multiplayer tetris, they can use the keyboard keys "1", "2" or "3" to swap pieces with one of their opponents. By sending a "SWAP" message through the server with correct destination (opponent name) the client that triggered the swapping will send over the type of the tetronimo that is currently falling in their board and receive the type of tetronimo from the opposing player in a "SWAPRESPONSE" message and by parsing these messages each of the clients will change the pieces that are falling on their respective boards. The code for this feature can be found in gui.GUIMultiGame in lines 153-188, network.TetrisClient lines 187-216 and lines 275-312(reception of the swap messages) and TetrisLobby lines 139-160.

Game Over Process

Similarly to the game start process whenever a client reaches a game over they send over a "GAMEOVER" message to the lobby, once the lobby detects that all clients have finished their game a “GAMEENDED” message will be sent to all clients so that they can all transition into the high scores screen. We can find code for this in gui.GUIMultiGame in lines 85-91, TetrisClient in lines 313-322 and server sode TetrisLobby in the lines 180 to 198.

Lobby Replication

Whenever a game begins, the replication of a lobby is triggered by the lobby itself by accessing the replication service of the server, this can be found in TetrisLobby lines 68-76, TetrisServer lines 234-254. A packet consisting of the header and JSON representation of the lobby is sent through an SSLSocket to the other servers’ server_socket, the socket responsible for accepting incoming messages from other servers. The reception of the packet can be found in class TetrisServer lines 216-232 and lines 293-298.

Whenever a game ends, the server sends a “delete” message to other servers in order for them to delete the replication of lobbies that are no longer needed. This can be found in TetrisLobby lines 78-80 and TetrisServer lines 216-232 and lines 299-302.

JSON payloads

Messages that have a payload (game states and lobby replications) make use of the javax.json library to convert objects into a json string representation. The conversion to and from JSON of these objects can be found in the classes ColorJSON, a wrapper class for the state of the board and TetrisLobbyJSON, a wrapper class for a lobby. These classes can be found in the **com.sdis.tetris.network** package.

Relevant issues

Security

We are using SSL to secure every connection that we make (Server -> Server, in replication of lobbies and fault tolerance, and Server->Client during normal execution of the program). An example of our implementation of secure connections can be seen in the constructor of the class TetrisServer.java (lines 36-72) where a SSL socket factory is created and used to create SSL sockets for the client and server connections.

Fault Tolerance

Whenever a game is initiated a lobby is always replicated from its original servers to the other servers. This means that our application has the capability of resisting both client side and server side failures.

Process of recovering from failures: Server Perspective

Replicated lobbies are kept in a different data structure than lobbies that were originated in the server itself, if the server receives a request from a client to join a certain lobby it will first check if the lobby exists in its own storage of lobbies and if it doesn't it will look in the replicated lobbies hashmap, if there is a match, the server will create a new lobby based on the replication(The replication is simply an instance of a wrapper class for data necessary to resume a lobby) and thus the game can resume on this new server and all the clients will be connected to this new lobby based on the replication. (TetrisServer lines 182-208).

A lobby where the game has already started will not be listed as possible lobbies for connection for clients but when a server receives a join request from a client and that client had already been connected to that lobby but disconnected, the server will accept this reconnection.(TetrisServer lines 186-189).

Process of recovering from failures: Client Perspective

1. A game is initiated on server1 some_lobby
2. Server1 shuts down, all the clients get socket exceptions on trying to send their game state to the lobby(com.sdis.tetris.gui.GUIMultiGame class lines 438-442)
3. Each client runs the canReachAnyServer function where a backup server is chosen in case of server failure (com.sdis.tetris.gui.GUIMultiGame class lines lines 100-112, com.sdis.tetris.network.TetrisClient class lines 378-413).

   If the client can't reach any server nothing will happen until the client tries to send their game state again.
4. If the client can reach any server it will firstly try to reconnect the original lobby on the original server:

   5.1. It succeeds, fault was in the client and game keeps going(GUIMultiGame line 105,TetrisClient lines 423-454)

   5.2. It fails, server failure(GUIMultiGame line 107, TetrisClient lines 415-420)

6. Clients reconnects to a replication of the lobby on the chosen backup server(all the clients choose the same backup server) and start sending their game states and listening for messages on said lobby.

The end result is that whatever faults might occur the game keeps on going seamlessly.

Conclusion

We managed to achieve the main objectives of this project by developing a working solution that allows for multiple games of up to 4 players of Tetris over the network with communication over secure SSL channels and robust fault tolerance .

With more time we could have implemented more interaction between the players besides the swapping of pieces power-up.

We could have also implemented some kind of anti cheating system within the server itself that would keep track of the scores and states of the clients to check for impossible transitions between states or scores by using timestamps for example, to simulate possible cheating we would need to implement cheats or falsified requests in the client to test the possibilities.