FACULTY OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Seminar Work

Fundamentals of Computer Networks. Multiplayer game "Reversi"

Volodymyr Pavlov

PILSEN, CZECH REPUBLIC

2025

FACULTY OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Seminar Work

Fundamentals of Computer Networks. Multiplayer game "Reversi"

Volodymyr Pavlov

PILSEN, CZECH REPUBLIC 2025

Citation in the bibliography/reference list:
PAVLOV, Volodymyr. *Fundamentals of Computer Networks. Multiplayer game "Reversi"*. Pilsen, Czech Republic, 2025. Seminar Work. University of West Bohemia, Faculty of Applied Sciences, Department of Computer Science and Engineering.

Contents

Task 1

Create a multiplayer game 'Reversi'. Solution should contain a server (written in C) with TCP scoket, which can process many players and at the same time can save game state. Client application (written in Kotlin), should support client login. After a login, client should continue with a last known state.

Implemented solution should be stable and be able to handle unexpected behavior (wrong input, client/server lost connection).

Game rules 2

Player color (white and black) is distributed randomly between two players. White player begins. Each turn, the player places one piece on the board with their color facing up.

Game starts with placing 4 pieces (2 of each color) in the middle of the board, so called *initial board state*, which can be seen in the Figure 2.1.

Each piece played must be laid adjacent to an opponent's piece so that the opponent's piece or a row of opponent's pieces is flanked by the new piece and another piece of the player's color. All of the opponent's pieces between these two pieces are 'captured' and turned over to match the player's color.

It can happen that a piece is played so that pieces or rows of pieces in more than one direction are trapped between the new piece played and other pieces of the same color. In this case, all the pieces in all viable directions are turned over.

The game is over when neither player has a legal move (i.e. a move that captures at least one opposing piece) or when the board is full.

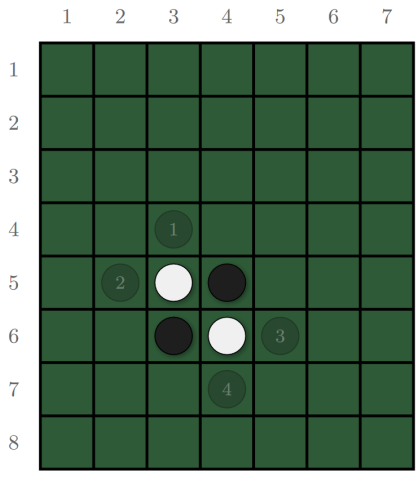


Figure 2.1: Possible game initial state.

Task Analysis 3

Due to a task restriction of using any existing data-interchange format (like *JavaScript Object Notation (JSON)* or *Extensible Markup Language (XML)*), a proprietary one should be created to be used in communication between server and client. Flexible data-interchange format should improve extensibility of requests and responses. The most challenging part will be to ensure multi-thread clients handing on server side, and proper error/unexpected behavior handing on both sides.

3.1 Key Components for Implementation

1. **Game engine** - this component will be response for all game-related processes.
    - **Game board** - create a data structure to store current game state.
    - **Move validation** - check if player move is correct and can be processed.
    - **Move processing** - process a correct move within all possible directions.
2. **Communication protocol**
    - **Representational State Transfer (REST)** - REST-like connection protocol will be used for communication between client[1] and server. It means, that whole communication will work in the way one client request will have one server response. Server will not send any non-client requested data.
    - **Message header** - Each message should have a header of a predefined length. The header should contain an identifier (to be sure that the socket is used by correct client), and a payload length (can be zero in case of empty payload).

[1] From now the *client* will be used in the meaning of a client application, connected to the server.

- **Message payload** - The payload is not mandatory. Payload should have a predefined structure to ensure correct composition, parsing and to make it easily extendable with additional values.
- **HTTP-like codes** - Each header should contain HTTP-like status code for easy flow and error handling.
- **Operation codes** - Each header should contain so-called *operation code* to differentiate message types handling.

3. **Server components**

   - **Clients handling** - each client will have its own thread on server side, which will process every client request.
   - **Operations synchronization** - every data-related action should be synchronized to avoid concurrent resource access.
   - **Client timeouts** - if a client has not performed a request for the last **N** minutes (N is configurable), then the client should be disconnected and related socket should be closed.
   - **Invalid client operations** - server should correctly react on any client's invalid operation (malformed request, action which is not possible in the current client state).
   - **Client flow state** - each client can be in a one of the flow state: [*LOGIN, MENU, LOBBY, GAME*] [2]. The client flow state should be synchronized between server and client, and the server value will be the only source of truth [3].

4. **Client components** - Entire client application will be implemented using Kotlin multi-platform (KMP) for backend and frontend.

   - **User Interface (UI)** - Easy and user-friendly UI will be implemented using Compose multi-platform (CMP).
   - **State synchronization** - Client state should be server-driven and synchronized to avoid inconsistent client state.
   - **Continue on client login** - After login, client should synchronize state with server and continue from a state, where client was.

[2] Described more detailed in the [section ref]

[3] It means, that a client-side flow state for some reason does not match server-side value, then it should be adjusted on the client side.

Custom data-interchange format

4

For a data-interchange format for payload a string with semicolon delimiters was chosen. Supported data types can be found in the Table 4.1. A payload then can be composed into a string, which then can be transmitted via TCP socket. Also, the payload can be easily parsed from a string into objects format using regex expressions.

Data Type	**Syntax**	**Example**
Integer	key={int_value}	key=123
String	key="{string_value}"	key="123"
Boolean	key={bool_value}	key=true
Integer Array	key= [{int_value0},{int_value1}]	key=[1,2,3,4]
String Array	key= ["{string_value0}","{string_value1}"]	key=[“str1”, “str2”]
No Value	key=null	key=null

Table 4.1: Payload supported data types.

Communication Protocol

5

Communication protocol is inspired by REST style and Hypertext Transfer Protocol (HTTP). Any information transmitting is driven by a client (client sends a request to a server and waits for a response).

Each message (request or response) contains a header, which has predefined constant length, and a payload, which contains useful information (may be empty for some cases).

5.1 Header

Message *header* has length of 17 bytes and contains 5 sections described in Table 5.1. Header has always constant structure: `[Identificator,Length,Type,Subtype, Status]`.

Section	Length (bytes)	Data Type	Description
Identificator	4	String	Unique identificator for client-server connection
Length	7	Integer	Length of the message payload (max is 9999999)
Type	1	Integer	Message Type
Subtype	2	Integer	Message Subtype
Status	3	Integer	Response status

Table 5.1: Header sections

5.2 Payload

Message *payload* is optional. It uses Custom data-interchange format for data transmitting. *Payload* can have variable length up to 9999999 bytes. The message *payload* length is defined in the message *header*, in case of empty payload zero value in the *header* is set.

Client Flow 6

Each client has its own flow state, which is synchronized between server and client. It's used to show correct UI screen to a player, to perform correct ping/get-state requests from a client, restrict or permit some user actions and to make a client able to continue from a state it was after reconnection (in case if the client is stored in server's cache). A client can be in four states: **No State**, **Menu**, **Login** and **Game**. Detailed description of client's states can be found in Table 6.1

No State	If a client has no state, it means that login was not performed and a client state could not be fetched from a server. In this case a user is asked to login.
Menu	This state means, that the client is logged in, but has no lobby or game assigned. In this case open lobbies list is displayed to a user. Player can create a new lobby, or connect to am existing one.
Lobby	In this state client is assigned to a lobby, but game was not started. Lobby screen is displayed to a user. Player can leave the lobby, or start the game in case if he/she is a lobby host and there are enough players.
Game	In this state game is already started and client is participating it. User is allowed to make game moves or leave the game. If one of the game players leaves, the game is terminated for all players.

Table 6.1: Client flow states.

Some player actions (e.g. creating a new lobby or leaving current lobby) can affect current client flow state, which will lead to UI changes. If the client state is changed on server side (e.g. game is terminated), then client-side flow state value is adjusted to ensure consistency with a server. Entire client flow overview can be seen in Figure 6.1.

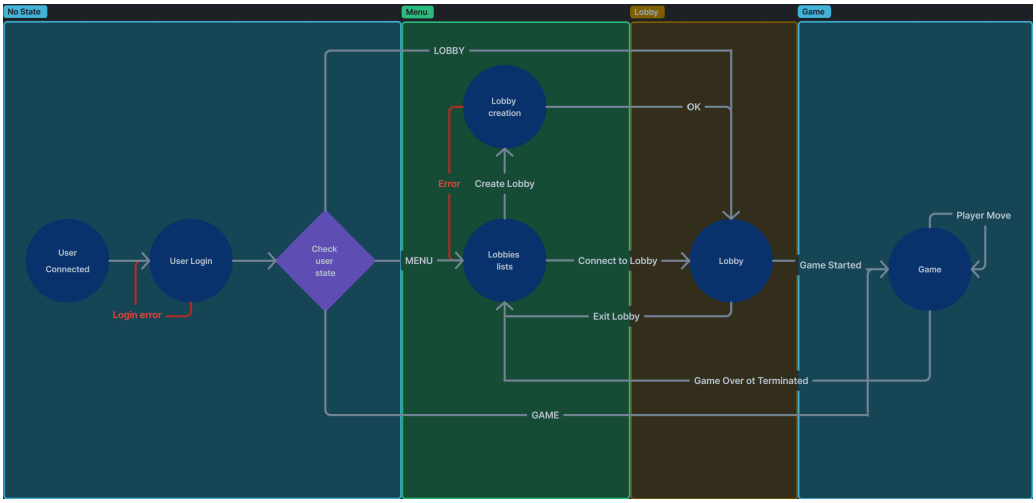


Figure 6.1: Client flow overview.

6.1 Edge Cases

- **Connection lost** - Loading screen with connection message is shown. Client attempts to reconnect to the server. After a successful reconnection, client performs auto-login with previously entered username a fetches user state from the server.
- **Game: opponent disconnected** - The message about game termination is shown. Client goes back to a menu state.
- **Game: opponent is offline** - The message about opponent's offline status is shown. A player can wait or press a quit button. In case of quit, game is terminated.

Requests and Responses

7

In this chapter used client requests and server responses will be described. On some request types there can be more than one response type, HTTP-like response codes are used to differentiate response processing.

Each request has its own type (see Table 7.1) and subtype (see Table 7.2), which indicates how it should be processed on server. They are a mandatory part os a message header.

Type	Code	Description
GET	1	Is used to retrieve information from a server.
POST	2	Is used to change/post some information on a server.

Table 7.1: Request types.

Subtype	Code	Description
PING	01	Is used to ping a server.
LOGIN	02	Is used to perform client login.
CREATE_GAME	03	Is used to perform creating a new game.
LOBBY_EXIT	04	Is used to exit a lobby or game.
LOGOUT	05	Is used to perform client logout.
START_GAME	06	Is used to start a game.
LOBBIES_LIST	07	Is used to get open lobbies list.
LOBBY_CNCT	08	Is used to perform client connection to a lobby.
GAME_STATE	09	Is used to retrieve current game state.
GAME_MOVE	10	Is used to perform a player game move.
LOBBY_STATE	11	Is used to retrieve current lobby state.
HANDSHAKE	12	Is used to perform a client-server handshake.

Table 7.2: Request subtypes.

7.1 Common

These requests and responses are used when a client is not logged in or for performing actions with its login.

7.1.1 Ping

A simple ping request-response which is used to check if the server is online. Client request and server response structures can be seen in Table 7.3.

Section	Value	Description
Type	1	GET
Subtype	01	PING

Request header.

Request payload is empty.

Section	Value	Description
Type	1	GET
Subtype	01	PING
Status	200	OK

Response header.

Response payload is empty.

Table 7.3: Ping request/response.

7.1.2 Handshake

It is used to verify new client connection. If handshake was not performed, any other requests from client are aborted. Client request and server response structures can be seen in Table 7.4.

Section	Value	Description
Type	2	POST
Subtype	12	HANDSHAKE

Request header.

Request payload is empty.

Section	Value	Description
Type	2	POST
Subtype	12	HANDSHAKE
Status	200	OK

Response header.

Response payload is empty.

Table 7.4: Handshake request/response.

7.1.3 Login

This request i subtype is used to perform client login (see Table 7.5). If login is accepted by server, OK response is returned (see Table 7.6). Otherwise, if username is already in use, CONFLICT response is sent (see Table 7.7).

Section	Value	Description
Type	2	POST
Subtype	02	LOGIN

Request header.

Field name	Type	Description
username	string	Username for login.

Request payload.

Table 7.5: Login client request.

Section	Value	Description
Type	2	POST
Subtype	02	LOGIN
Status	200	OK

Response header.

Field name	Type	Description
state	string	Client flow state.

Response payload.

Table 7.6: Login server response - OK.

Section	Value	Description
Type	2	POST
Subtype	02	LOGIN
Status	409	CONFLICT

Response header.

Field name	Type	Description
msg	string	Error message.
state	string	Client flow state.

Response payload.

Table 7.7: Login server response - NOK - username declined.

7.1.4 Logout

This request i subtype is used to perform client logout. If client is not logged in, no actions are performed and OK response is returned. Client request and server response can be seen in Table 7.8

Section	Value	Description
Type	2	POST
Subtype	05	LOGOUT

Request header.

Request payload is empty.

Section	Value	Description
Type	2	POST
Subtype	05	LOGOUT
Status	200	OK

Response header.

Response payload is empty.

Table 7.8: Logout request/response.

7.2 Menu and Lobby

These subtypes of requests are used to retrieve information about current lobbies state and to interact with a lobby.

7.2.1 Retrieve Open Lobbies

This request is used to get open lobbies, to which a client can connect (see Table 7.9 and Table 7.10).

Section	Value	Description
Type	1	GET
Subtype	07	LOBBIES_LIST

Request header.

Request payload is empty.

Table 7.9: Get lobbies client request.

Section	Value	Description
Type	1	GET
Subtype	07	LOBBIES_LIST
Status	200	OK

Response header.

Field name	Type	Description
state	string	Client flow state for synchronization.
lobbies	string array	A list of open lobbies names.
lobby_hosts	string array	A list of open lobbies host usernames.

Response payload.

Table 7.10: Get lobbies server response.

7.2.2 Create Lobby

This request is used to process new lobby creation, triggered by a user. Client sends a request with a lobby name (see Table 7.11). Server respond with OK if a lobby is created successfully (see Table 7.12), and with status CONFLICT if a lobby with requested name is already exists (see Table 7.13).

Section	Value	Description
Type	2	POST
Subtype	03	CREATE_GAME

Request header.

Field name	Type	Description
name	string	Lobby name.

Request payload.

Table 7.11: Create lobby client request.

Section	Value	Description
Type	2	POST
Subtype	03	CREATE_GAME
Status	200	OK

Response header.

Field name	Type	Description
state	string	Client flow state for synchronization.
name	string	Lobby name.
user	string	Client name.

Response payload.

Table 7.12: Create lobby server response - OK.

Section	Value	Description
Type	2	POST
Subtype	03	CREATE_GAME
Status	409	CONFLICT

Response header.

Field name	Type	Description
msg	string	Error message.
state	string	Client flow state for synchronization.

Response payload.

Table 7.13: Create lobby server response - CONFLICT.

7.2.3 Connect to Lobby

This request subtype is used to perform client connection to an existing open lobby (see Table 7.14). If connection was successful, OK response is returned (see Table 7.15), otherwise, if connection was not performed, NOT FOUND response is returned (see Table 7.16).

Section	Value	Description
Type	2	POST
Subtype	08	LOBBY_CNCT

Request header.

Field name	Type	Description
lobby	string	Lobby name.

Request payload.

Table 7.14: Connect to lobby client request.

Section	Value	Description
Type	2	POST
Subtype	08	LOBBY_CNCT
Status	200	OK

Response header.

Field name	Type	Description
state	string	Client flow state for synchronization.
lobby	string	Lobby name.
host	string	Host name.
players	string array	Players names list.

Response payload.

Table 7.15: Connect to lobby server response - OK.

Section	Value	Description
Type	2	POST
Subtype	08	LOBBY_CNCT
Status	404	NOT FOUND

Response header.

Field name	Type	Description
msg	string	Error message.
state	string	Client flow state for synchronization.
lobby	string	Lobby name.

Response payload.

Table 7.16: Connect to lobby server response - NOT FOUND.

7.2.4 Get Lobby State

This request is used to get current state of the lobby, in which client is (see Table 7.17). If state is retrieved successfully, then OK response is returned (see Table 7.18). In case when the game is started, REDIRECT response is used (see Table 7.19).

Section	Value	Description
Type	1	GET
Subtype	11	LOBBY_STATE

Request header.

Request payload is empty.

Table 7.17: Get lobby state client request.

Section	Value	Description
Type	1	GET
Subtype	11	LOBBY_STATE
Status	200	OK

Response header.

Field name	Type	Description
state	string	Client flow state for synchronization.
lobby	string	Lobby name.
host	string	Host name.
players	string array	Players names list.

Response payload.

Table 7.18: Get lobby state server response - OK.

Section	Value	Description
Type	1	GET
Subtype	11	LOBBY_STATE
Status	301	MOVED

Response header.

Field name	Type	Description
state	string	Client flow state for synchronization.
lobby	string	Lobby name.
host	string	Host name.
players	string array	Players names list.

Response payload.

Table 7.19: Get lobby state server response - REDIRECT.

7.2.5 Start the Game

This request is used to start the game from the current lobby (see Table 7.20). If game is started successfully, then OK response is returned (see Table 7.21). If the lobby has not enough players to start the game, NOT ALLOWED response is returned (see Table 7.22). If the current player has no rights to start the game (e.g. not the host), UNAUTHORIZED response is used (see Table 7.23).

Section	Value	Description
Type	2	POST
Subtype	06	START_GAME

Request header.

Request payload is empty.

Table 7.20: Start the game client request.

Section	Value	Description
Type	2	POST
Subtype	06	START_GAME
Status	200	OK

Response header.

Field name	Type	Description
state	string	Client flow state for synchronization.

Response payload.

Table 7.21: Start the game server response - OK.

Section	Value	Description
Type	2	POST
Subtype	06	START_GAME
Status	405	NOT_ALLOWED

Response header.

Field name	Type	Description
msg	string	Error message.
state	string	Client flow state for synchronization.
players	string array	Players names list.

Response payload.

Table 7.22: Start the game server response - NOT ALLOWED.

Section	Value	Description
Type	2	POST
Subtype	06	START_GAME
Status	401	UNAUTHORIZED

Response header.

Field name	Type	Description
msg	string	Error message.
state	string	Client flow state for synchronization.
players	string array	Players names list.

Response payload.

Table 7.23: Start the game server response - NOT ALLOWED.

7.2.6 Exit the Lobby

This request subtype is used to perform a player exiting the lobby or game (see Table 7.24). The only response is OK (see Table **??**), which indicates that operation was successful.

Section	Value	Description
Type	2	POST
Subtype	04	LOBBY_EXIT

Request header. *Request payload is empty.*

Table 7.24: Exit the lobby client request.

Section	Value	Description
Type	2	POST
Subtype	04	EXIT_LOBBY
Status	200	OK

Response header.

Field name	Type	Description
state	string	Client flow state for synchronization.

Response payload.

Table 7.25: Exit the lobby server response - OK.

7.3 Game

These requests are used to retrieve information about the current game state and to perform player actions with it.

7.3.1 Get Game State

This request is used to retrieve the current game state (see Table 7.26). If the game state can be get, OK response is returned with game state information (see Table 7.27). In case if the game is over, RESET response is returned with information about a winner (see Table 7.28). If the game is terminated (one of the players exited the game), then MOVED PERMANENTLY response is returned (see Table 7.29). If a player is trying to get a state of a game which is not started yet, NOT FOUND response is returned (see Table 7.30).

Section	Value	Description
Type	2	POST
Subtype	09	GAME_STATE

Request header.

Request payload is empty.

Table 7.26: Get game state client request.

Section	Value	Description
Type	2	POST
Subtype	09	GAME_STATE
Status	200	OK

Response header.

Field name	Type	Description
state	string	Client flow state for synchronization.
players	string array	Players names list.
player_codes	string array	Player codes list.
opponent_connected	bool	Shows if opponent has a connection to the server.
current_player	string	Name of the player who is on turn.
board	integer array	Board cells representation.
board_size	integer array	board size [cols, rows].

Response payload.

Table 7.27: Get game state server response - OK.

Section	Value	Description
Type	2	POST
Subtype	09	GAME_STATE
Status	205	RESET

Response header.

Field name	Type	Description
state	string	Client flow state for synchronization.
players	string array	Players names list.
winner	string	Name of the game winner.

Response payload.

Table 7.28: Exit the lobby server response - RESET.

Section	Value	Description
Type	2	POST
Subtype	09	GAME_STATE
Status	301	MOVED_PERMANENTLY

Response header.

Field name	Type	Description
state	string	Client flow state for synchronization.
msg	string	User message.

Response payload.

Table 7.29: Exit the lobby server response - MOVED.

Section	Value	Description
Type	2	POST
Subtype	09	GAME_STATE
Status	404	NOT_FOUND

Response header.

Field name	Type	Description
state	string	Client flow state for synchronization.
msg	string	User message.

Response payload.

Table 7.30: Exit the lobby server response - NOT FOUND.

7.3.2 Player Move

This request subtype is used to perform player move in a game (see Table 7.31). If the move was successful, then OK response is used (see Table 7.32). If the player is not on turn, CONFLICT response is returned (see Table 7.33). In case when the move is invalid, NOT ALLOWED response is used (see Table 7.34).

Section	Value	Description
Type	2	POST
Subtype	10	GAME_MOVE

Request header.

Field name	Type	Description
x	integer	X coordinate of the move.
y	integer	Y coordinate of the move.

Request payload.

Table 7.31: Player move client request.

Section	Value	Description
Type	2	POST
Subtype	09	GAME_STATE
Status	200	OK

Response header.

Field name	Type	Description
x	integer	X coordinate of the move.
y	integer	Y coordinate of the move.

Response payload.

Table 7.32: Player server response - OK.

Section	Value	Description
Type	2	POST
Subtype	09	GAME_STATE
Status	409	CONFLICT

Response header.

Field name	Type	Description
state	string	Client flow state for synchronization.
msg	string	User message.

Response payload.

Table 7.33: Player server response - CONFLICT.

Section	Value	Description
Type	2	POST
Subtype	09	GAME_STATE
Status	405	NOT_ALLOWED

Response header.

Field name	Type	Description
state	string	Client flow state for synchronization.
msg	string	User message.

Response payload.

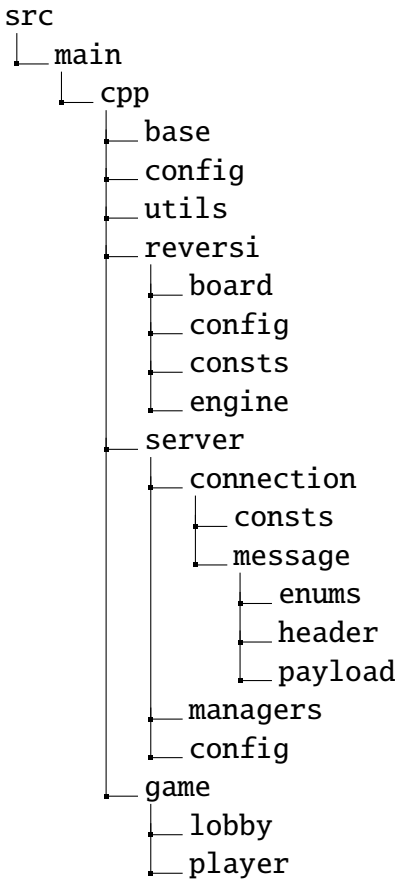
Table 7.34: Player server response - NOT ALLOWED.

Server 8

Server is written in C++ 17 along with `CMake` build system and utility packages (`log4cxx`, `fmt` and `SimpleIni`). These tools were chosen to simplify memory management (utilizing `std::shared_ptr`) and to enable working in the object-oriented programming paradigm.

8.1 Project Structure

The implementation of the program consists of several modules to achieve greater flexibility in supporting or modifying the code. The following briefly shows project structure:



8.2 Key Implementations

8.2.1 Main Function

Program entry point - `main.cpp` file. Main function takes care of loading all needed configuration from `.ini` files, initializes logging and server. Starts the server.

8.2.2 Server

Server class implementation encapsulates the key functionalities required to manage server operations in a networked application. The server is initialized with configuration details (`server_config`) and a message manager (`message_manager`) and verifies that all required configurations are in place before starting.

Server creates a socket to listen for incoming connections. Binds the socket to the specified address and port and listens for client connections and manages the connection queue size. Accepts incoming client connections and manages individual client threads. Processes client messages and handles the exchange of requests and responses. Closes client connections when necessary, including during timeouts or errors. Handles scenarios where clients disconnect or time out, ensuring the server resources are released properly. Monitors client activity and disconnects idle or unresponsive clients. Integrates with client and lobby managers to manage game-specific or application-specific states.

8.2.3 Client Manager

Client Manager class provides core functionalities to manage client connections and their states on the server. It takes care of storing a new client connection in a connection map and removing client connections safely when they are no longer active. Provides functionality for clients login and logout. Ensures thread-safe access to client connections and client data using shared and unique locks for read and write operations. The **Client Manager** class plays a critical role in maintaining the integrity and thread-safety of client-related data and ensuring seamless interaction between the server and its clients.

8.2.4 Lobby Manager

The **Lobby Manager** class is responsible for managing lobbies and their interactions with clients in a multiplayer environment. It's responsible for creating a new lobby with a specified name and ensuring that the provided lobby name is unique. It allows players to exit lobbies. If the host leaves, it updates the host or removes the lobby if it becomes empty. Also it updates the player's state to reflect their disconnection from the lobby.

8.2.5 Message Manager

The **Message Manager** class is a core component responsible for processing and managing communication between clients and the server. It handles incoming messages from clients and routes them based on their type and subtype. Ensures handshake and login are performed before processing certain requests. Validates message identifiers to ensure requests are from authorized sources. Generates appropriate error responses for invalid requests, unauthorized actions, and resource conflicts.

8.3 How to Build and Run

8.3.1 Requirements

To build the server executable file next components are required:

- **`Linux OS`** – the server can be build only under `GNU/Linux` operation system.
- **`C++ 17+`** – Installed and configured `C++` of version 17 or higher is required.
- **`CMake 3.25+`** – Installed and configured `CMake` of version 3.25 or higher is required.
- **`log4cxx`** – Installed `liblog4cxx-dev` package is required.

8.3.2 Build and Run

The project can be build using common way of building CMake projects and run by executing `server` executable file after the build. The process is described in the listing 8.1.

Source code 8.1: Example server build and run

```
# Navigate to the project directory
cd project

# Create a build directory
mkdir build && cd build

# Configure the project
cmake ..

# Build the project
make

# Run the executable
./server
```

8.3.3 Configuration

Server specific parameters such as port, client and handshake timeouts etc. can be configured using `config.ini` file located in the `config` folder in the project root.

Client 9

The client application implementation is developed in `Kotlin` language together with `Kotlin Multiplatform` and `Compose Multiplatform` frameworks. Also, the implementation relies on `koin` dependency injection library. Such development stack was chosen because it's easy to maintain and extend with a new features. Mentioned frameworks provides a mechanism for fast project migration to other platforms such as `Android`, `IOS` and `WEB` while sharing common UI and business logic implementation. Also, `Compose Multiplatform` allows developers to use modern `Android-style` UI implementation with a variety of pre-made features.

9.1 Project Structure

The project implementation is divided into two main parts:

- **`commonMain`** – contains implementation of common business logic and UI which can be used on different platforms.
- **`desktopMain`** – contains desktop `Linux` specific implementation such as running the `Compose Application`.

This structure ensures easy maintain of the project and its flexibility from platform and new features perspective.

9.2 Key Implementations

The whole project is written in Model-View-ViewModel (MVVM) paradigm to provide a clear separation of concerns and facilitates testability, maintainability, and scalability.

9.2.1 Modules

The **Modules.kt** file contains `koin` modules declaration and provides dependency injection support to the project. It ensures correct `koin` initialization with pro-

vided parameters which allows adaptation of used dependencies without any code changes.

9.2.2 State Services

State services are used to encapsulate a state, to provide a unified interface to interact with the state and its thread-safety. The following state services are present in the project:

- **ClientStateService** – encapsulates the client state such as login state, username, current flow state, available lobbies list and the current lobby.
- **ConnectionStateService** – encapsulates the server connection state and stores information if the connection is alive, last successful server ping timestamp, socket, and if a handshake was performed.
- **GameStateService** – contains information about the current game state: players list, if the opponent is online, current player turn, and board cells data.
- **UserMessageStateService** – it is a utility state service which is used to manage which message (if required) is displayed to a user. It ensures that all fired messages will be shown.

9.2.3 ConnectionService

The **ConnectionService** is used to take care of the server connection and messages sending and receiving. It provides a method to establish a connection to the server with provided parameters, and a method to exchange a message with the server (send the given message and read the response message).

9.2.4 Message Processors

The **Message Processors** are used to process a message on the business logic layer - construct a message using provided parameters and dedicated message type and subtype, send it to the server, retrieve a response message and process received data using `ConnectionService`. Here is a list of implemented message processors:

- **CommonProcessor** – is a base implementation for all message processors which performs client-server interaction. It ensures that every exception thrown during the message sending and receiving is properly handled and propagated to UI. It uses coroutines to provide asynchronous execution and to avoid main thread blocking.

- **CommonClientProcessor** – is a dedicated implementation which extends `CommonProcessor` with a client state update method.
- **HandshakeProcessor** – processing a handshake with the server. Rise a fatal error in case of failure.
- **LoginProcessor** – processing a client login with provided username.
- **LogoutProcessor** – processing a client logout.
- **PingProcessor** – processing a simple ping to the server.
- **GetLobbiesProcessor** – processing a get of currently available lobbies.
- **CreateLobbyProcessor** – processing a lobby creation with provided lobby name.
- **ConnectToLobbyProcessor** – processing a connection to a lobby with a specified name.
- **ExitLobbyProcessor** – processing an exiting from the current lobby.
- **GetLobbyStateProcessor** – processing getting a state of the current lobby.
- **StartGameProcessor** – processing a game start.
- **GetGameStateProcessor** – processing getting a state of the current game.
- **GameMoveProcessor** – processing a game move using given coordinates.

9.2.5 PingService

The **PingService** takes care of the regular ping to the server and updates the current client state with the given data from the server. Type of the used message processor depends on the client flow state:

- **no state** – `PingProcessor` is used to check server connection state.
- **MENU** – `GetLobbiesProcessor` is used to update the list of the available lobbies.
- **LOBBY** – `GetLobbyStateProcessor` is used to update the current lobby state.
- **GAME** – `GetGameStateProcessor` is used to update the current game state.

9.3 How to Build and Run

9.3.1 Requirements

- **`Windows`** or **`Linux`** operation system.
- **`Java 17+`** – installed and configured `Java` of version 17+.
- **`Gradle 8.7+`** – installed and configured `Gradle` of version 8.7+.

9.3.2 Build and Run

In the project root folder there are two prepared files: `gradlew` and `gradlew-linux`. These files can be used to build and run the project under `Windows` and `Linux` operation systems. The example of building the executable file on `Linux` can be seen in the listing **??**

Source code 9.1: Example of client build on Linux

```
# Navigate to the project directory
cd project

./gradlew-linux createDistributable
```

After the executable file is build, it can be directly run.

9.3.3 Configuration

Client connection properties such as server IP address, server port and ping intervals can be found in the build directory in the `config/connection.properties` file.

Conclusion 10

This semester project, explored the fundamentals of computer networks by implementing a client-server architecture for an interactive multiplayer game. The work demonstrates the practical application of theoretical concepts such as communication protocols, data interchange formats, and state synchronization in a real-world scenario.

Through the development of the server (using C++) and the client (using Kotlin), this project showcased the importance of modular design, efficient communication, and robustness in handling concurrent user interactions. The use of custom protocols and proprietary data formats further emphasized flexibility and adaptability while meeting the challenges posed by restrictions on standard interchange formats.

The project also underscored the significance of error handling, threading, and state management in ensuring a stable multiplayer experience. The implementation successfully addressed edge cases like disconnections and unexpected inputs, reinforcing the importance of creating resilient systems in network programming.

In summary, this work not only fulfilled the technical requirements but also provided valuable insights into software design principles, cross-platform development, and the intricacies of computer networking. This experience has contributed significantly to the understanding of modern development paradigms, preparing the foundation for tackling more complex and large-scale systems in the future.

List of Abbreviations

UI User Interface

REST Representational State Transfer

KMP Kotlin multi-platform

CMP Compose multi-platform

JSON JavaScript Object Notation

XML Extensible Markup Language

HTTP Hypertext Transfer Protocol

MVVM Model-View-ViewModel

List of Figures

List of Tables

List of Listings