Peer-Review 2: Communication Protocol

Roberto Scardia, Andrea Sgobbi, Luca Simei, Jonatan Sciaky

April 2023

1 Introduction

This document describes the network interactions between the server and the clients using TCP-IP socket connection and RMI API. The server allows the client to select the preferred communication protocol regardless of the choices of other clients. Efforts were made to ensure the sameness of the protocols; in fact, they share all of the data structures and most of the logic. The TCP-IP communication makes use of the default Java serialization procedure to send and receive objects over the socket.

2 Connecting to the Server

2.1 TCP-IP

The server exposes e TCP-IP socket using the Java standard net API. The default port is *1234*. After accepting the request the server will respond with an ***Integer*** object containing the ID number assigned to the client.

2.2 RMI

The server exposes a ***server*** object, the interface of which is provided in the APPENDIX A section of this document. Similarly, it's expected by the client to expose a ***client*** object with the interface also described in APPENDIX A. A client is registered to the server by calling the `void register(Client client)` method on the server object. The server already provides an RMI Registry on port *1099*.

3 Notifying Messages

As mentioned before, both TCP-IP and RMI protocols share all the data structures used to convey the MVC messages over the network. Their interfaces are defined in the APPENDIX B-C section of this document.

3.1 TCP-IP

In TCP-IP communication, the Server mimics the internal functioning of the RMI protocol operating on a ***ClientSkeleton*** object that implements the *Client* interface. The server implements a remote procedure call (RPC) interface. For each procedure, it expects to be provided a correct payload that implements the ***ViewMessage<>*** interface. For each method and request the correct payload types are listed in APPENDIX C. The server expects to receive objects only with the type described. Failing to do so will result in an error response. With every correct procedure call an *"OK"* Response object with *0* as status is sent. Every procedure could respond with a *"ServerError"* Response with *-2* as status if the server cannot satisfy the client's request for any reason. An ill-formed request or a request that will result in an error will yield no change to the internal state of the server. The server could also reply to a correct request with one or more objects that inherit from ***ModelMessage<>*** that represent a change in the game model. The *ModelMessage*<> objects are always guaranteed to be sent before the *Response* object. The server can also send unsolicited *ModelMessage*<> messages whenever there is a change in the server internal state not prompted by the client. The correct method dispatch inside the server is done by exploiting the *reflection* capability of the Java language operating on the dynamic type of the *ViewMessage*<> The following diagrams describe the communication sequence between a client and the server; the first with a generic, internally undefined client, the second with the actual client implemented in the project.

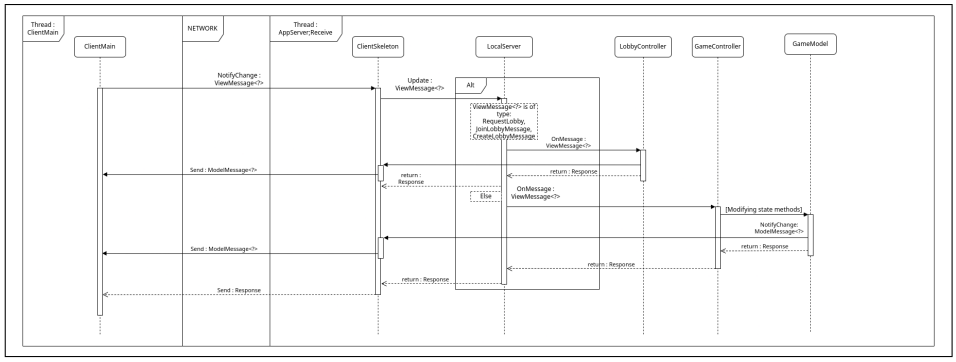

Figure 1: Client Agnostic Sequence Diagram

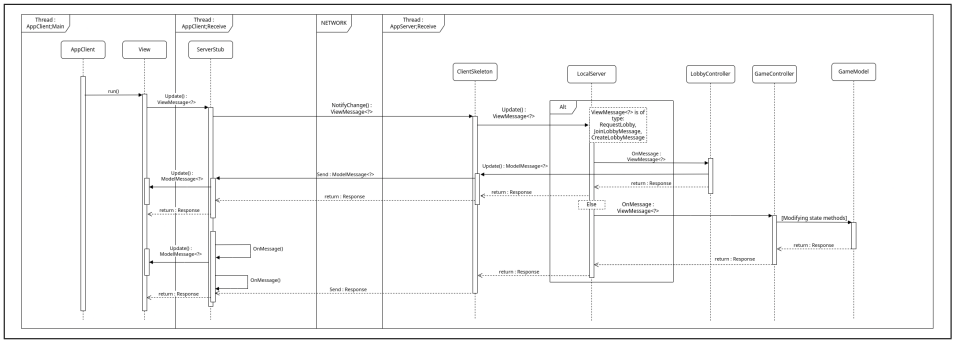

Figure 2: Implemented Client Sequence Diagram

3.2 RMI

The following sequence diagram describes the order of function calls that happen in the RMI protocol. As mentioned before, the server exposes a remote *server* object of *Server* type and the client needs to provide a ***Client*** object named "client" to the RMI registry.

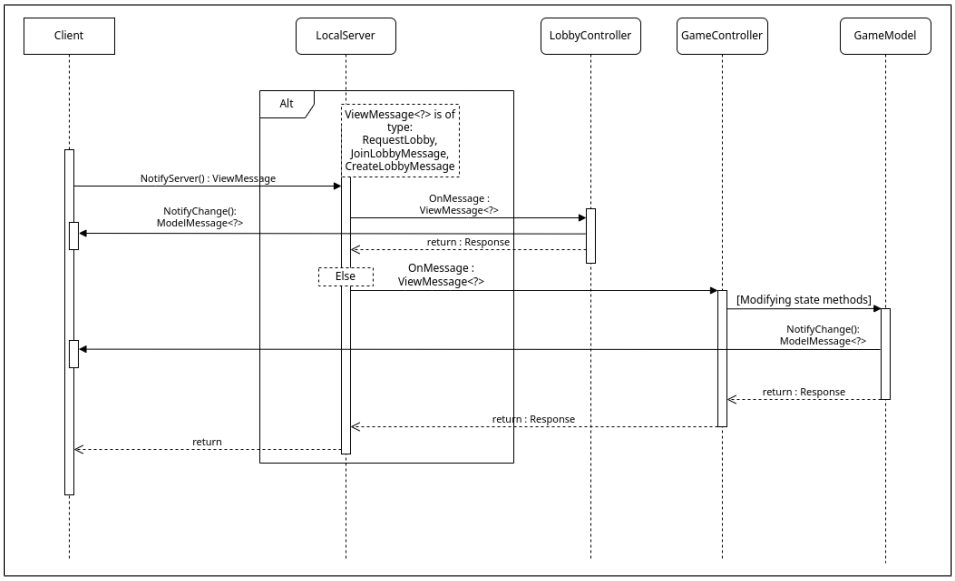

Figure 3: RMI Sequence Diagram

4 APPENDIX A

- Server interface.

```java
//Server.java
public interface Server extends Remote {
    /**
     * Register a client to the server
     * @param client the client to register
     */
    void register(Client client) throws RemoteException;

    /**
     * Notify the server that a client has made a choice
     * @param message the message sent by the view
     */

    Response update(ViewMessage<?> message) throws RemoteException;

}
```

- Client interface.

```java
//Client.java
public interface Client extends Remote {

    /**
     * Sets the internal client ID. Should only be called once
         inside a Server thread.
     * The ID is unique and is used by the server to identify each
         client.
     * @param clientID The ID number assigned by the server
     */
    void setClientID(int clientID) throws RemoteException;


    /**
     * Returns the ID client
     */
    int getClientID() throws RemoteException;

    /**
     * Notify the client of a model change
     * @param msg  The object that contains the model changes
         information
     */
    void update(ModelMessage<?> msg) throws RemoteException;
}
```

- Response class implementation.

```
    //Response.java
public record Response(int status, String msg, String Action)
     implements Serializable {
    public boolean isOk() { return status == 0; }
    @Override
    public String toString(){
        return status + ", " + msg;
    }
}
```

5 APPENDIX B

- Abstract ModelMessage data structure

```
abstract public class ModelMessage<Payload extends Serializable>
    implements Serializable {
    final private Payload payload;
    public Payload getPayload(){
        return payload;
    }
    public ModelMessage(Payload p){
        this.payload = p;
    }
}
```

- AvailableLobbyMessage implementation.

```
//AvailableLobbyMessage.java
public class AvailableLobbyMessage extends
    ModelMessage<LobbyPayload>{
    public AvailableLobbyMessage(List<LobbyController.LobbyView> p)
        {
        super(new LobbyPayload(p));
    }
}
//LobbyController.java->LobbyView
public record LobbyView(List<String> nicknames, int lobbySize, int
    lobbyID) implements Serializable {
    @Override
    public String toString() {
        StringBuilder nickames = new StringBuilder();
        for( String nickname : this.nicknames() ) {
            nickames.append("\t").append(nickname).append("\n");
        }
        return ("\n------LOBBY: " + this.lobbyID() + "------\n") +
                ("Occupancy: [" + this.nicknames().size() + "/" +
                    this.lobbySize() + "]\n")
                + nickames;
    }
}
```

- BoardMessage implementation, signals a change in the game board.

```
//BoardMessage.java
public class BoardMessage extends ModelMessage<Board> {
    public BoardMessage(GameModel p) {
        super(p.getBoard());
    }
```

```
}
```

- CommonGoalMessage implementation, notifies the update of the current common goal top score.

```
\\CommonGoalMessage.java
public class CommonGoalMessage extends
    ModelMessage<CommonGoalPayload>{
    public CommonGoalMessage(Integer CommonGoalId,
        CommonGoalPayload.Type type, Integer topScoreAvailable) {
        super(new CommonGoalPayload(CommonGoalId, type,
            topScoreAvailable));
    }
}
\\CommonGoalPayload.java
public record CommonGoalPayload(Type type, int availableTopScore)
    implements Serializable {
    public enum Type{
        X, Y
    }
}
```

- CurrentPlayerMessage implementation, notify the change of the current player whose nickname is the payload.

```
public class CurrentPlayerMessage extends ModelMessage<String>{
    public CurrentPlayerMessage(String p) {
        super(p);
    }
}
```

- EndGameMessage implementation, signals the end of the game and contains the winner player and a leader-board with all the players' points.

```
//EndGameMessage.java
public class EndGameMessage extends ModelMessage<EndGamePayload>{
    public EndGameMessage(EndGamePayload p) {
        super(p);
    }
}
//EndGamePayload.java
public record EndGamePayload(String winner, Map<String, Integer>
    points) implements Serializable {}
```

- IncomingChatMessage implementation, notifies the arrival of a new chat message. Contains the nickname of the sender.

```java
//IncomingChat.java
public class IncomingChatMessage extends ModelMessage<String>{
    private final String sender;
    public IncomingChatMessage(String p, String sender) {
        super(p);
        this.sender = sender;
    }

    public String getSender() {
        return sender;
    }
}
```

- ShelfMessage implementation, signals the change of a player's shelf. Contains the player's nickname.

```java
\\ShelfMessage.java
public class ShelfMessage extends ModelMessage<Shelf> {
    private final String player;
    public ShelfMessage(Shelf p, String player) {
        super(p);
        this.player = player;
    }
    public String getPlayer(){
        return this.player;
    }
}
```

- StartGameMessage implementation, signals the start of the game and contains the list of nickanames of all players, the common goals IDs and for each player its personal goal ID.

```java
//StartGameMessage.java
public class StartGameMessage extends
     ModelMessage<StartGamePayload> {
    public StartGameMessage(List<String> p, Integer personalGoalId,
         Integer XCGnumber, Integer YCGnumber) {
        super(new StartGamePayload(p, personalGoalId, XCGnumber,
            YCGnumber));
    }
}
//StartGamePayload.java
public record StartGamePayload(List<String> nicknames, int
     personalGoalId, int XCGnumber, int YCGnumber) implements
     Serializable {}
```

- UpdateScoreMessage implementation, notifies that the player *player*

```
//UpdateScoreMessage.java
public class UpdateScoreMessage extends
    ModelMessage<UpdateScorePayload> {
    public UpdateScoreMessage(Integer score,
        UpdateScorePayload.Type type, String nickname) {
        super(new UpdateScorePayload(type, score, nickname));
    }
}
//UpdateScorePayload.java
public record UpdateScorePayload(Type type, int score, String
    player) implements Serializable {
    public enum Type {
        CommonGoal, PersonalGoal, Adiajency, Bonus
    }
}
```

6 APPENDIX C

- Abstract ViewMessage data structure.

```
//ViewMessage.java
public abstract class ViewMessage<Payload extends Serializable>
    implements Serializable {

    private final Payload payload;
    private final String nickname;

    private final int clientId;
    public ViewMessage(Payload payload, String playerNick, int
        clientId){
        this.payload = payload;
        this.nickname = playerNick;
        this.clientId = clientId;
    }
    public Payload getPayload(){
        return this.payload;
    }
    public abstract Class<?> getMessageType();
    public String getPlayerNickname(){
        return this.nickname;
    }
    public int getClientId(){
        return this.clientId;
    }
}
```

- Create a new game Lobby with *size* number of player
  - Error Reply:
    - If size < 2 or size > 4 Response :
      `[status=-1, msg="InvalidLobbySize"]`
    - If a player with the same nickname is already present:
      `[status=-1, msg="Nickname Taken"]`
  - No Update Reply.

```
//CreateLobbyMessage.java
public class CreateLobbyMessage extends ViewMessage<Integer> {
    public CreateLobbyMessage(Integer size, String playerNick, int
        clientId) {
        super(size, playerNick, clientId);
    }

    @Override
    public Class<?> getMessageType() {
```

```
        return this.getClass();
    }

}
```

- Request the information of all the lobbies with *size* size
  - No particular Error Replay
  - Update Reply
    - `AvaliableLobbyMessage`

```
//RequestLobbyMessage.java
public class RequestLobbyMessage extends ViewMessage<Integer> {
    public RequestLobbyMessage(Integer size, String playerNick, int
         clientId) {
        super(size, playerNick, clientId);
    }
    @Override
    public Class<?> getMessageType() {
        return this.getClass();
    }
}
```

- Request to join the lobby with *lobbyId* id.
  - Error Replay
    - If the lobby is full or an invalid id was sent:
      `[status=-1, msg="Lobby Unavailable]`
    - If the player's nickname is already taken :
      `[status=-1, msg="Nickname Taken"]`
  - Update Replay
    - `GameStartMessage`

```
//JoinLobbyMessage.java
public class JoinLobbyMessage extends ViewMessage<Integer> {
    public JoinLobbyMessage(Integer lobbyId, String playerNick, int
         clientId) {
        super(lobbyId, playerNick, clientId);
    }
    @Override
    public Class<?> getMessageType() {
        return this.getClass();
    }
}
```

- Send player move
  - Update Reply
    - `BoardMessage`
    - `ShelfMessage`
    - `CurrentPlayerMessage`
    - `CommonGoalMessage`
    - `UpdateScoreMessage`
    - `EndGameMessage`
  - Error Replay
    - If the player is not the current player:
      `[status=-1, msg="Not the current player : event will be ignored"]`
    - If the move is ill-formed or illegal:
      `[status=-1, msg="Illegal move : ignoring player action"]`

```
//MoveMessage.java
public class MoveMessage extends ViewMessage<Move> {
    public MoveMessage(Move move, String playerNick, int clientId) {
        super(move, playerNick, clientId);
    }
    @Override
    public Class<?> getMessageType() {
        return this.getClass();
    }
}
//Move.java
public record Move(List<Coordinate> selection, List<Tile> tiles,
     int column) implements Serializable {}
```

- Send chat message to another player in the lobby with the nickname *playerDestination*. If *playerDestination* is not provided the server will send the message to all players in the lobby.
  - Update Reply
    - `IncomingChatMessage`
  - No particular Error Reply

```
public class ChatMessage extends ViewMessage<String> {
    private final String dest;
    public ChatMessage(String s, String playerNick, int clientId) {
        super(s, playerNick, clientId);
        dest = "BROADCAST";
    }
    @Override
    public Class<?> getMessageType() {
```

```
        return this.getClass();
    }
    public ChatMessage(String s, String PlayerNick, String
         playerDestination, int clientId){
        super(s, PlayerNick, clientId);
        dest = playerDestination;
    }
    public String getDestination(){
        return this.dest;
    }
}
```

7 APPENDIX D

Here is a list with the name and descriptions of all the common goals with their ID number

0. SixGroupTwoTileGoal : "Six groups, each containing at least 2 tiles of the same type."
1. FourGroupFourTileGoal : "Four groups, each containing at least 4 tiles of the same type."
2. FourCornersGoal : "Four tiles of the same type in the four corners of the shelf."
3. TwoGroupSquareGoal : "Two 2x2 squares of tiles of the same type."
4. ThreeColumnSixTileGoal : "Three columns, each formed by 6 tiles of maximum 3 different types."
5. EightUniqueGoal : "Eight tiles of the same type, in any position."
6. DiagonalFiveTileGoal : "Five tiles of the same type forming a diagonal."
7. FourRowFiveTileGoal : "Four rows, each formed by 5 tiles of maximum 3 different types."
8. TwoColumnDistinctGoal : "Two columns each formed by 6 distinct types of tiles."
9. TwoRowDistinctGoal : "Two rows each formed by 5 distinct types of tiles."
10. CrossGoal : "Five tiles of the same type forming an X shape."
11. DecreasingColumnsGoal : "Five columns of increasing or decreasing height, with tiles of any type."