Attendance Registration App

Practical Assignment for Distributed Programming (1rst Goal)

Diogo Gomes - 2021137427

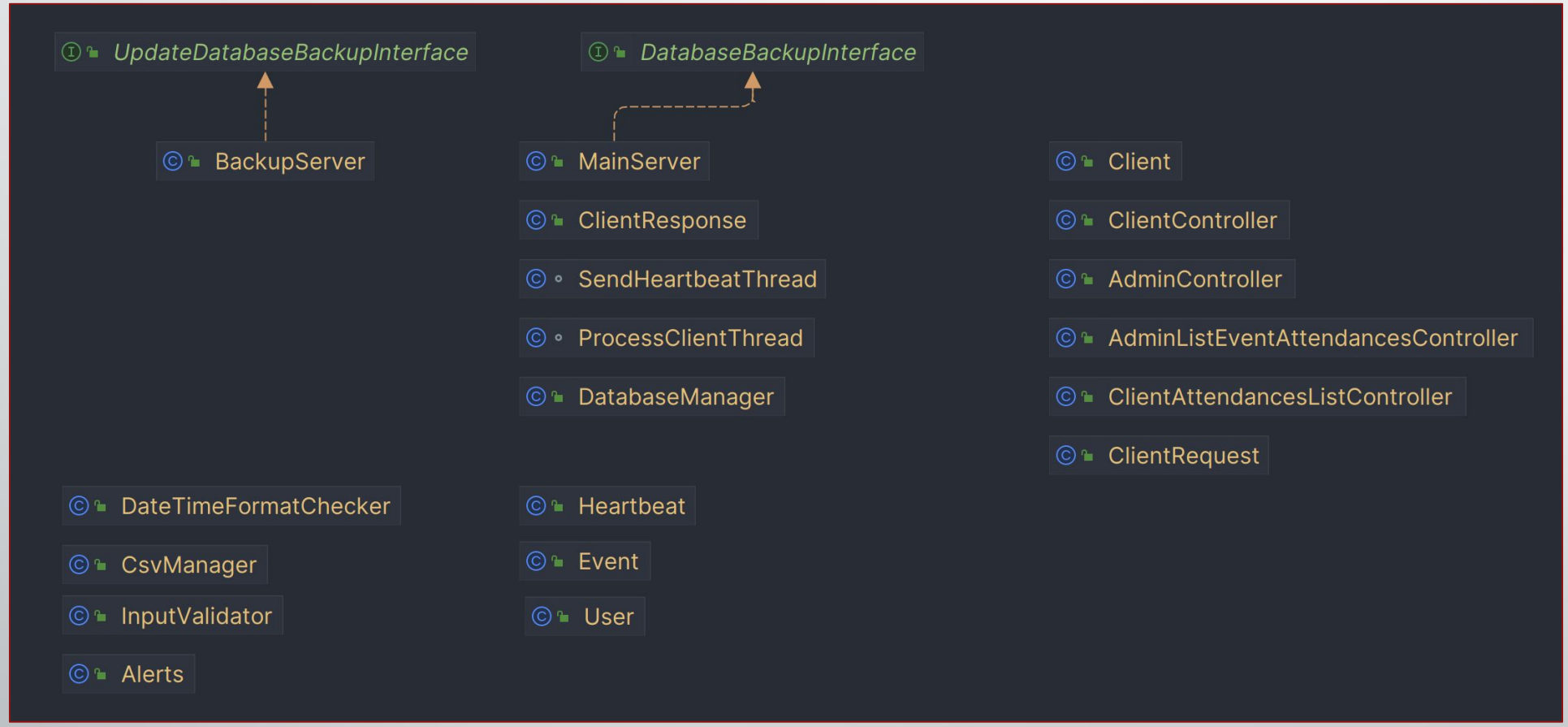
Joao Neves – 2021133564

Wojciech Larecki - 2023107712

Instituto Superior de Engenharia

Politécnico de Coimbra

Classes Diagram



MainServer

Main server's timeline of events:

1 – Starts by receiving the tcpServerPort, registryPort, rmiServiceName and databaseDirectory from the command line (+ validations)

2 – Creates the backupService associated to its database -> registers it so that the Backup Servers can locate it

3 – Starts a SendHeartbeatThread responsible for sending the heartbeats to multicast so that the Backup Servers can connect to the rmi service and keep receiving heartbeats every 10 seconds.

- SendHeartbeatThread calls the method sendHeartBeat() every 10 seconds

- sendHeartBeat() creates and sends a heartbeat with all of the required attributes to the multicast group "230.44.44.44" with port "4444" . The heartbeat is sent as a DatagramPacket

4 – Creates a ServerSocket with the desired tcpServerPort and starts accepting clients, creating a ProcessClientThread for each one of them to attend their requests and send them the correspondent responses

- ProcessClientThread gets the ObjectInputStream and ObjectOutputStream of the client socket, receives the request, checks the request type and calls the correspondent DatabaseManager method to deliver a response with the return of the method.

- If the database was changed, it updates the backup servers and sends a new heartbeat

5 – Waits for all the threads to join

```
Connection established with success!
Database Version: 4
DatabaseBackupService created and in execution...
Service database-backup registered in registry with port 1099
Server is running fine at address 0.0.0.0 and port 5001
[Heartbeat sent]
[Heartbeat sent]
```

Note: Some client requests do not need to invoke DatabaseManager methods but most of them do.

Client

Client's timeline of events:

1 – Starts by receiving the listeningTCPPort and serverAddress from the command line (+ validations)

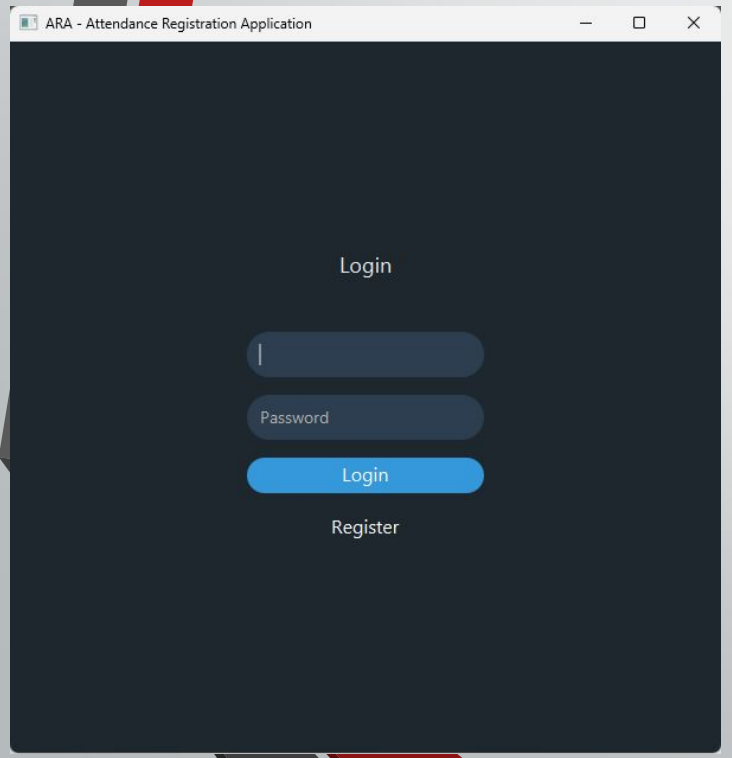
2 – Connects to the server socket

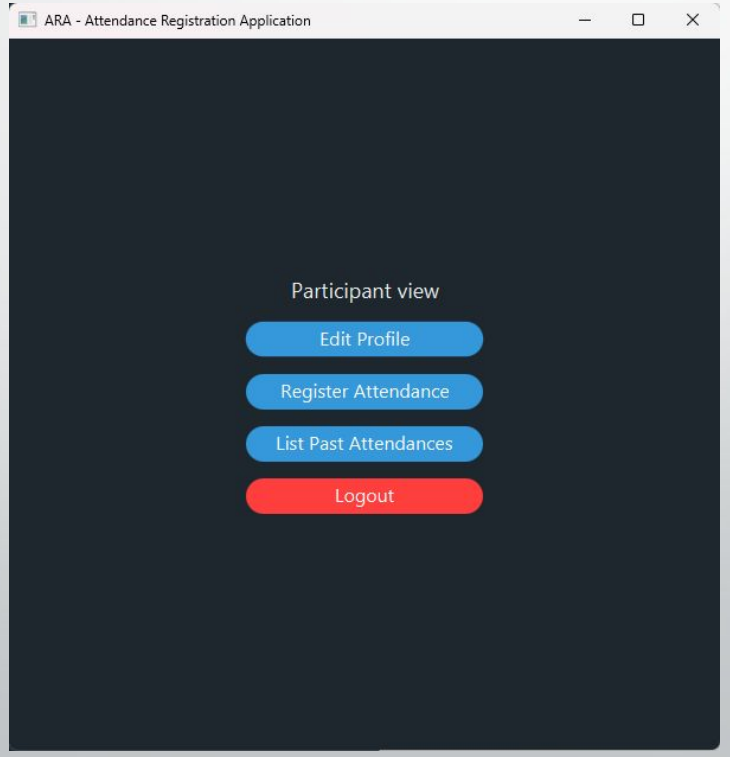
3 – Launches the Application with UI

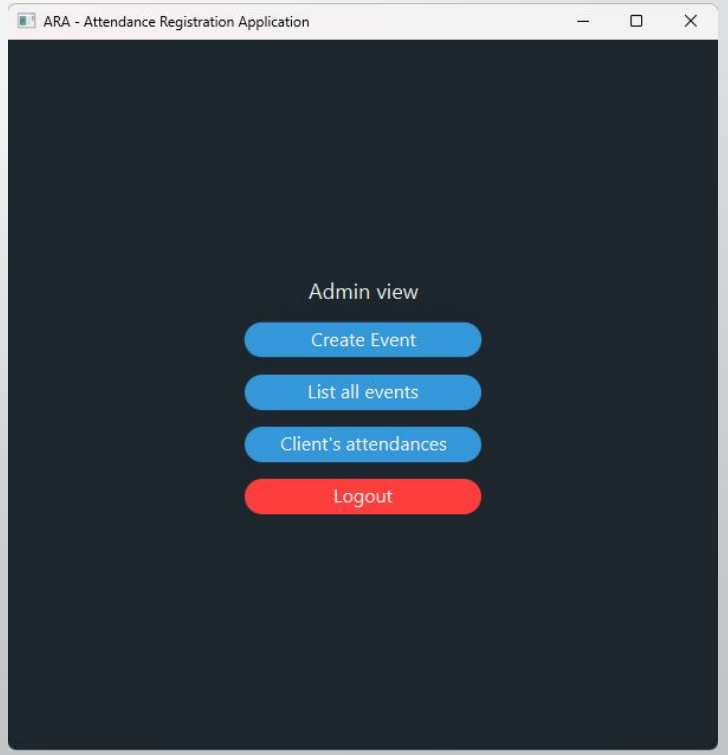
4 – Responds to the user's inputs sending requests to the server when needed and presenting feedback for the responses gotten

5 – After the login process, a set of UI's is chosen depending on the user type (participant or admin)

6 – All of the requests are done in this way: User input -> ClientController calls the correspondent client.method() -> Client sends the correspondent request to the MainServer







BackupServer

Backup server's timeline of events:

1 – Starts by receiving the databaseDirectory from the command line (+ validations)

2 – Connects to the multicast in order to receive the heartbeats

3 – With the first heartbeat's info, it downloads the database a first time through the registry given in the first heartbeat

4 – It adds itself to the list of backup servers on the MainServer so he can be notified of all the changes via callback

5 – Enters a while loop that:

- Calls the receiveHeartBeat() method:
  - Sets a timeout of 30 seconds -> receives the datagram packet in form of a serialized Heartbeat object
  - Prints the heartbeat received
- If the heartbeat's database version is different from the local database version -> exit

- Every time the database is updated, the MainServer updates the backup servers so they can download a new copy of the original database (this was implemented via callback)

Client <–> MainServer

The communication between client and server was made through a TCP connection (with parameters listening port, server address provided in the command line) where the client sends a *ClientRequest* (with a designated type, user and other necessary arguments) as a serialized object which is then read by the server.

Example (client side):

```
public boolean register(String email, int user_id, String password) {
    clientRequest = new ClientRequest(new User(email, user_id, password), ClientRequest.Type.REGISTER);
    sendClientRequest(clientRequest);

    return receiveClientResponse();
}
```

The response gotten from the main server is then sent back as a serialized object of type ClientResponse (with the user and the type (ERROR, SUCCESS, etc.)).

MainServer side:

```
case REGISTER → {
    User temp = databaseManager.insertUser(
            receivedRequest.getUser().getEmail(),
            receivedRequest.getUser().getPassword(),
             user_type: "participant",
            receivedRequest.getUser().getUser_id());
    response = new ClientResponse(temp);

    if (response.user == null) {
        response.type = ClientResponse.Type.ERROR;
    } else {
        response.type = ClientResponse.Type.SUCCESS;
    }
}
```

MainServer <–> DatabaseManager

The communication between MainServer and Database was made through a DatabaseManager class, responsible for altering the database attending to the Server's requests.

Example:

MainServer

```
case REGISTER → {
    User temp = databaseManager.insertUser(
            receivedRequest.getUser().getEmail(),
            receivedRequest.getUser().getPassword(),
             user_type: "participant",
            receivedRequest.getUser().getUser_id());
    response = new ClientResponse(temp);

    if (response.user == null) {
        response.type = ClientResponse.Type.ERROR;
    } else {
        response.type = ClientResponse.Type.SUCCESS;
    }
}
```

DatabaseManager

```
public User insertUser(String email, String password, String user_type, int user_id) {
    String sqlVerifyEmailInDB = "SELECT COUNT(*) as TOTAL from users where email= ?";
    int numUsersWithSameEmail;
    String sql = "INSERT INTO users(email,password,user_type,user_id) VALUES(?,?,?,?)";
    int affected;
    try (PreparedStatement stmt = conn.prepareStatement(sqlVerifyEmailInDB)) {
        stmt.setString( parameterIndex: 1, email);
        ResultSet result = stmt.executeQuery();
        numUsersWithSameEmail = result.getInt( columnLabel: "TOTAL");
        if (numUsersWithSameEmail ⩾ 1) {
            System.out.println("Invalid email");
            return null;
        }
    } catch (SQLException e) {
        System.out.println("SQL Error: " + e);
        return null;
    }
    try (PreparedStatement pstmt = conn.prepareStatement(sql)) {
        pstmt.setString( parameterIndex: 1, email);
        pstmt.setString( parameterIndex: 2, password);
        pstmt.setString( parameterIndex: 3, user_type);
        pstmt.setInt( parameterIndex: 4, user_id);
        affected = pstmt.executeUpdate();
    } catch (Exception e) {
        return null;
    }
    if (affected == 0)
        return null;
    alterDbVersion();
    return login(email, password);
}
```

MainServer <–> BackupServer

The communication between MainServer and Backup servers was made through:

- callback mechanism so that the backup servers can update their databases and versions
    - Backup server is notified everytime the main server's database is updated and downloads the latest version
- multicast in order to send/receive heartbeat
    - checks if the heartbeat version is the same as the current database version (so that it can know if it has to stop or if he didn't lose track of the database)

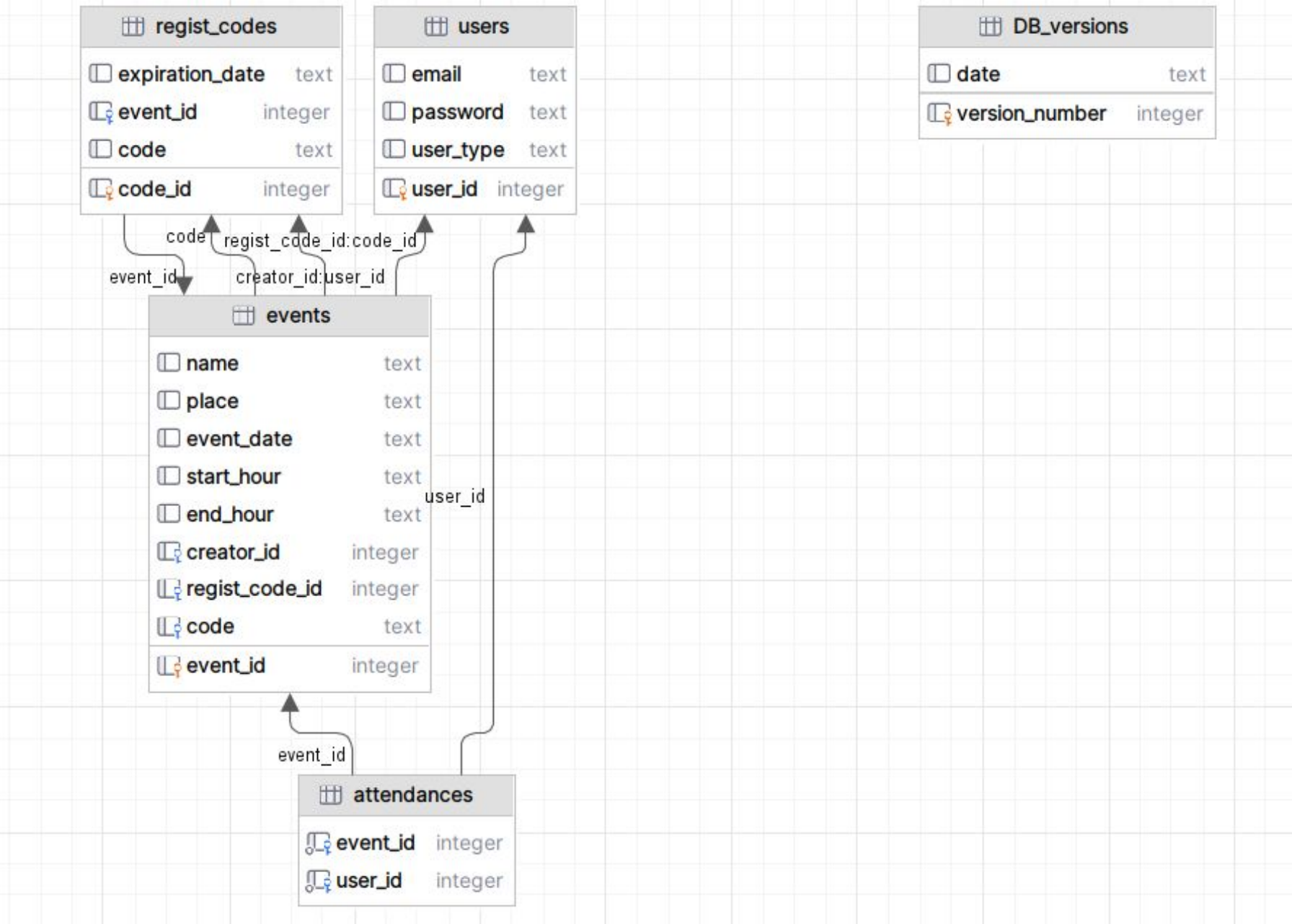
```
public interface UpdateDatabaseBackupInterface extends Remote {
    1 usage  1 implementation  Wojciech
    void update() throws RemoteException;
}
```

```
public interface DatabaseBackupInterface extends Remote {
    8 usages
    final ArrayList<UpdateDatabaseBackupInterface> OBSERVERS = new ArrayList<>();
    1 usage  1 implementation  João Neves
    byte[] getDatabaseChunk(long offset) throws IOException;
    1 usage  1 implementation  João Neves
    void addBackupServer(UpdateDatabaseBackupInterface observer) throws RemoteException; // adds observer, register to main server

    1 usage  João Neves
    static void removeBackupServer(UpdateDatabaseBackupInterface observer) throws RemoteException // removes observer
    {...}

    1 usage  João Neves
    static void updateBackupServers() throws RemoteException {...}
}
```

Database Model



Quick User Guide - General

- login:
  - Fill the Text Fields -> "Login" Button
- register:
  - "Register" Button/Label -> Fill the form-> "Confirm registering" Button
- logout:
  - "Logout" Button

Quick User Guide - Participant

Participant:

- edit profile:
  - "Edit Profile" Button -> Fill the Text Fields -> "Confirm Edit" Button
- register an attendance at a certain event:
  - "Register Attendance" Button -> Fill the Code -> "Submit attendance" Button
- list past attendances:
  - "List Past Attendances" Button
- get csv file for past attendances:
  - "List Past Attendances" Button -> "Get csv file" -> Choose where to save -> "Ok" button on popup

Quick User Guide - Admin

Admin:

- creating event:
  - "Create Event" Button -> fill the required forms -> click the "Confirm edit" button
- delete user attendance from event:
  - "Client's attendances" Button -> select the event -> select the user -> "Delete attendance" Button
- add user attendance to event:
  - "Client's attendances" Button -> select the event -> write the email on the designated field-> add attendance
- list all events:
  - "List all events" Button
- edit an event:
  - "List all events" Button -> select the desired event -> "Edit selected event" Button -> fill the required fields/"Generate new code" Button -> Confirm edit
- delete an event:
  - "List all events" Button -> select the desired event -> "Delete" Button
- list event's attendances:
  - "Client's attendances" Button -> select the desired event
- get csv file for event attendances
  - "Client's attendances" Button -> select event -> "Get csv file" Button

O Fim

The End

Koniec