WORTH Project Report

Teodorico Pacini, 578987

Contents

1 Premises **2**
1.1 System Architecture 3
1.2 Threads, data structures and concurrency 4

2 Classes Description **6**
2.1 Client 6
2.2 Server 6
2.3 Commons 7

3 Usage **8**

Chapter 1

Premises

The implementation of the client-server environment you will see in this project will be based on some key points:

- The clients are designed to be run from different machines, as is the server, which will have its own dedicated machine.
- The server will always remain active and may be shut down due to an anomaly or an unexpected error. To test data persistence, the server execution can be interrupted using any signal that terminates the thread (SIGINT, SIGKILL, ...).
- Users will only be able to use the client I created.
- The folder where data persistence will take place must be called *SavedState* and be located in the main directory of the project. The name and location of the directory can be modified through the *MAIN_PATH* variable found in the *DBMS.java* class.

Both the clients and the server have been tested on localhost.

1.1 System Architecture

The system is structured into two main components:

- The client (*ClientMain.java*), used by the user to interact with the system.
- The server (*ServerMain.java*) that handles client requests and responds with the outcome of the performed operation.

Clients and servers communicate through a TCP connection using sockets (*SocketChannel*). Given the high volume of requests the server may need to handle, choosing a lightweight and fast request management mechanism is essential. The WORTH's server provides channel multiplexing by leveraging a selector that ensures quick handling of client requests without generating excessive load on the CPU and RAM. The NIO multiplexed architecture is the ideal choice if you do not have high computational capacity available, as it allows managing multiple connections using a single thread. In contrast, in a multithreaded architecture, starting numerous threads requires greater computing power and increased memory usage which is directly proportional to the scale of the system. The NIO multiplexed approach, therefore, provides numerous advantages, and in cases where a single thread is not sufficient to handle all requests, additional threads can still be used.

The RMI mechanism is used during registration, the client and the server use a shared interface (RMIRegistrationInterface.java), while for the callback, two interfaces are used (one for the server (RMICallbackInterface.java) and one for the client(s) (ClientNotifyInterface.java)). When the user registers itself, its username is stored and all the clients are notified with the updated registered users' list via a callback. The clients registers to the callback on login, and unregister themselves on logout. Via the callback the system can also check if some clients are down due to unexpected errors.

The main operations are performed over the TCP connection that is established once the user requests to log in. To perform these operations, the user can use the commands provided by the CLI (which can be viewed with the "help" command). The client will display the server's responses on the screen, which can consist of error messages or success messages (e.g., 200 OK), followed by the result of the requested command.

There are some constrains on user input:

- The project name must not contain spaces, and if a string containing spaces is entered, it will be interpreted as two distinct strings.
- The same applies to the name of the cards.
- For everything else, spaces are allowed: the card description and messages sent in the chat can contain spaces.

The mechanism I chose to communicate the multicast address and port to the client is via an explicit request (*get_parameter*), similar to the other commands. However, unlike the other commands, it is not among the options available in the CLI.

Regarding persistence, I chose to write data to JSON files using the external library json-simple (version 1.1). This library provides an intuitive and fast

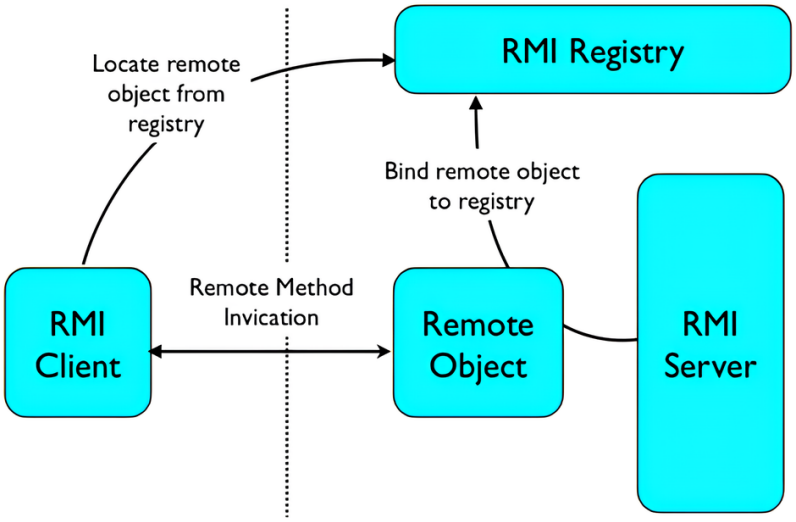

Figure 1.1: Java Remote Method Invocation

mechanism for writing and retrieving data from a JSON file, which is why I preferred it over GSON or Jackson, even though the latter two solve the issue mentioned below (non-generic parameters). I did not choose to serialize the data using the Serializable interface because I considered it to be a too resource-intensive approach.

Each project is stored on disk using a separate directory, each directory contains data about cards, members and all the information related to that specific project.

Note: the *@SuppressWarnings("unchecked")* annotations that you may find in the methods using this external library are due to the use of "non-generic parameters" by the library itself. From what I have tested, however, I have not encountered any errors during its use.

1.2 Threads, data structures and concurrency

When the program is launched, two threads are scheduled: one for the server ('ServerMain.java') and one for each client ('ClientMain.java'). The server uses three data structures within its class:

- *ArrayList<Project>projects*: list of projects created by various users.
- *HashMap<String, String>users*: list of registered users, with each user associated with their current status (online or offline).
- *HashMap<String, String>credentials*: list of registered users, with each user associated with their password.

None of these data structures implement any concurrency mechanisms, nor do the methods within the *ServerMain.java* class use the ***synchronized*** keyword. This is because the use of a selector typically does not lead to concurrent data access situations. On the server side, the only control I considered necessary is in the DBMS class. In this case, the methods to check the existence of a username and the one to register a user, can be invoked by multiple threads, potentially leading to data inconsistency. For this reason, I used ***synchronized*** on these methods.

Regarding the client side, no concurrency occurs, each client has its own remote interface.

Chapter 2

Classes Description

2.1 Client

- **ClientMain.java**: client's main class, implements the user requests (login, listProjects, addMember...), the registration process for the callback, the request sender method, and the parser for user input. This class handles the user request through the CLI and forwards it to the server using the TCP socket. Before performing any operation, the user should be logged in.
- **ClientNotifyImpl.java**: stores the implementation of the RMI objects. It stores inside an HashMap the users list.
- **ClientNotifyInterface.java**: RMI objects' interface which will be registered in the RMI Registry. It is used by the client to update the local users list.
- **MulticastInfos.java**: stores port, address and multicast socket used by the client to communicate with the project's chat.

2.2 Server

- **ServerMain.java**: server's main class, implements methods for logging users in and out, executing their requests (moveCard, cancelProject, showCard, ...), restoring and persisting projects and cards to disk, and auxiliary methods that can handle the configuration of TCP communication, initialization, and export of RMI objects. At startup, the server restores the data from the disk, creates and configures the two RMI services, open the connection socket and create the channel selector.
- **AdvKey.java**: contains nickname, request, and response. This allows the server to create an instance of this class and associate it with a key using the attach method.
- **Card.java**: represents the cards that can be part of a project. It contains three strings (name, description, and list) that identify the name, description of the card, and the list in which it is currently located, respectively,

and two functions to keep track of the lists in which the card belongs (ordered) and to handle card movement from a list to another.

- **Project.java**: contains methods and data structures necessary for project management. It provides methods to add new members or cards, view all cards or all members, view a specific card, and move it from one list to another. Additionally, it has a method called *isDone* that tells the server if all the cards are in the "done" list (allowing the project to be deleted).
- **DBMS.java**: registers new users to the "service" and ensure persistence of registrations to disk. The class implements a *setLocal_ref* method to obtain the reference to the RMI callback, so that with each registration, the DBMS can add the new user to the list of registered users.

2.3 Commons

- **RMICallbackImpl.java**: manages the RMI Callback mechanism. It has two methods invoked by clients to register or unregister from callbacks.
- **RMICallbackInterface.java**: interface of the RMI object used for callbacks. It has two fields, *PORT* and *REMOTE_OBJECT_NAME*, which are used to bind the remote object to the registry and to locate it in the registry.
- **RMIRegistrationImpl.java**: manages the RMI mechanism for user registrations. It implements the register method, which checks the validity of the username and password, then performs checks by invoking the DBMS methods and returns the response to the user based on the outcome.
- **RMIRegistrationInterface.java**: interface of the RMI object used for user registrations.

Chapter 3

Usage

The project employs only one external dependency, ***json-simple***, which is stored in the root directory of the project. To compile the code I created two bash scripts:

- **clean.sh**: remove all auxiliary files created during past compiling. It is executed before compiling the project.
- **compile.sh**: run clean.sh and compile the entire project. If some errors occur, messages tagged with *"[ERROR]"* will be displayed.

Note: before compiling or running the project, you must navigate to the "src" directory.

To compile the entire project, execute the following commands in the terminal (from the "src" directory):

1. *chmod +x compile.sh*
2. *./compile.sh*

If no error messages are displayed, the compilation was successful.
To run the client and the server:

- Client: java Client.ClientMain localhost 4382
- Server: java -cp :../json-simple-1.1.jar Server.ServerMain 4382

where *localhost* is the server's IP address, and *4382* is the port exposed by the server.

Note: the software was tested with OpenJDK 17.