UNIVERSITÀ DI PISA

LABORATORIO DI RETI

UNIVERSITY of PISA

WORDLE A WORD GAME 3.0

By

Neri Sansone - 621583

Department of Computer Science

UNIVERSITY of PISA

October 19, 2023

CONTENTS

1 Introduction **2**

2 Project Structure **3**

2.1 Introduction 3

2.1.1 Src 3

2.1.2 Lib 4

2.1.3 Bin 4

2.2 WordleClient 4

2.3 WordleServer 5

2.4 MenuHandler 5

3 Implementation choices **6**

3.1 User Experience 6

3.2 Edge Cases 7

4 Data Structures and Methods **8**

4.1 Data Structures 8

4.2 Synchronized methods 9

5 Thread scheme **10**

5.1 Server’s Threads 10

5.2 Client’s Thread 10

6 Compilation **11**

6.1 Javac 11

6.2 Jar 11

References **13**

1 INTRODUCTION

Wordle 3.0 is a simplified version of the already known game. It is based on a server-client mechanism which allows different users to play at the same time. After connecting to the server the users will be able to play trying to guess a ten letters long word within twelve tries using the given hints.

2 PROJECT STRUCTURE

2.1 Introduction

The general structure relies on a server - client paradigm. Every user can access and interact with the game via CLI (Command Line Interface). Once the client sends the requests to the server which will process and answer to every one of them. The projects is divided in three folders:

- src
- lib
- bin

Firstly I will give a short overview of every folder, then I will bring you to a more deep understanding of the core functions of the project.

2.1.1 Src

In the ***src*** folder we find the seven different *.java* files:

- **WordleClient.java**
- **WordleServer.java**
- **menuHandler.java**
- **User.java**
- **th_properties.java**
- **Mess_Receiver.java**
- **Change_Word.java**

2.1.2 Lib

This folder contains the only referenced library used in this project: *gson2.10.jar*

2.1.3 Bin

The ***bin*** folder contains all the *.class* files produced after having compiled, the *.properties* files used by the client and the server and the *users.json* file along with the file containing all the possible words.

2.2 WordleClient

The ***WordleClient*** class is responsible of interacting with the users via CLI (*Command Line Interface*). It immediately reads the *client_config.properties* file in order to connect to the server. After having opened the socket and the two streams for the *input* and *output* we get to the core of the program.
The client will print in the terminal the first menu in which it is possible choose between different options [1]:

- **REGISTER [1]** $\rightarrow$ if chosen will ask for an username and a password (*n.b* no empty username and/or password are allowed)
- **LOGIN [2]** $\rightarrow$ if chosen will lead to the login
- **EXIT [3]** $\rightarrow$ if chosen it will close the current session

Whether a registration is successful or not it will lead to the first menu again. If the user chooses to log in and such operation is successful the second menu will be printed. We are now able to choose between five options:

- **PLAY [1]** $\rightarrow$ if the word has not been played by the user it will start the game
- **STATISTICS [2]** $\rightarrow$ will show the statistics of the logged user
- **SHARE [3]** $\rightarrow$ will broadcast a message saying how many attempts the user needed to guess the word

[1] *n.b* every choice made is sent to the server

- **SHOW ME SHARING [4]** $\rightarrow$ will show the received messages
- **LOGOUT [5]** $\rightarrow$ will lead to the first menu while logging out

Concerning the client side the game is managed by the *WordleGame* function which is responsible for sending the entered guess to the server in order to receive back the possible hints and check if the user has won. This function returns the number of attempts used by the user to guess the word, this value will be used by the server for the sharing part.

2.3 WordleServer

The server is the very first class that we need to execute. Like the client it immediately reads the *server_config.properties*. It will then create a an ArrayList object containing all the user in the .json file if it exists, otherwise it will create a new users.json file. Afterwards it creates another ArrayList structure with all the given words along with a thread that is responsible of extracting the secret word that will be changed every two minutes (I choose two minutes in order to have a more loyal representation of the game without waiting for twenty-four hours). With the two ArrayList objects created the server also initialises the *th_properties* object (called *properties*), this object contains the user list, the word list and the secret word. Eventually with a try-catch block it creates a menuHandler thread for every connection accepted.

2.4 MenuHandler

The ***menuHandler*** class is responsible of managing every single request sent by the client and checking if the operations that the user is doing are legal (*i.e.* checking during the registration if the chosen username is available or not). It is also responsible: of adding the new users to the *users.json* file, of creating the hints for every guess received together with the update of the statistics. Eventually it is also responsible of sending the user' statistics as well as the broadcast message(s).

3 IMPLEMENTATION CHOICES

3.1 User Experience

As soon as a new user starts the client he or she will be required to register login afterwards. While registering the system will firstly ask to enter a valid username (*i.e.* no empty or already used username can be chosen) and secondly to enter a valid password (*i.e.* an empty password is **not** valid).

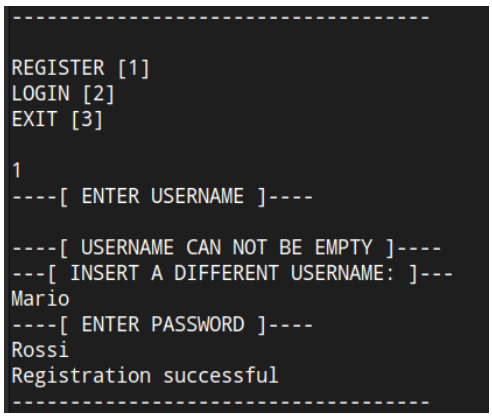

Figure 3.1: Registration

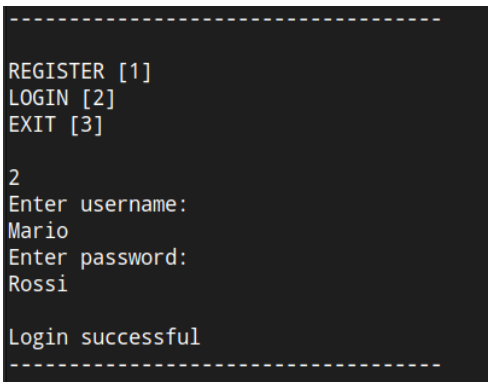

Figure 3.2: Log in page

An user is **not** able to log in simultaneously onto two different clients in fact there is a variable inside the *User* class that checks if an user is already logged in. After logging the user will be able to choose one of the options listed in section 2.2. If the user has not guessed the current secret word yet, he will be able to play [1]. After playing he will be allowed to share the number of attempts he needed to guess the word. In order to implement such feature I added a variable that shows if the user has already played the word or not.

[1] If the guessed word contains three *C*s but the secret word contains two *C*s then the hints will show three *"?"*, which means that the hints system I built does **not** count the occurrences of a letter.

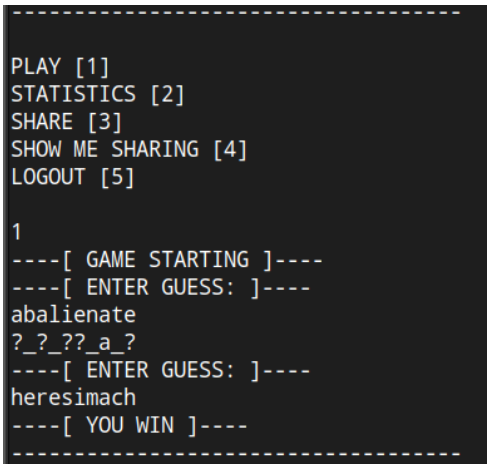

Figure 3.3: Game play

It is always possible to see the statistics and the received message(s) through the *SHOW ME SHARING* option. The sharing side relies on a UDP connection and with the multicast sends the message to all the connected clients.

3.2 Edge Cases

- If it happens that the client crashes unexpectedly in whatsoever moment of the session, no problems: the user will just need to start again a new client.
- If the server crashes unexpectedly, then all the connected clients, as soon as they try to communicate with the server, will be closed and it will be necessary to launch again the server.

4 DATA STRUCTURES AND METHODS

4.1 Data Structures

- **words_list** → it is an ArrayList<String> object that contains all the words provided in the words.txt file.
- **users_list** → it is an ArrayList<User> object that contains all the users that are inside *users.json*
- **User** → it collects all the relevant details of an user such as the *username*, the *password* and a HashMap<Integer,Integer> that shows how many words have been guessed at a certain try.

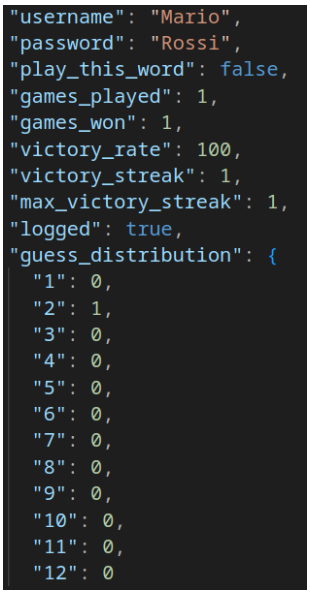

Figure 4.1: Sample user's data structure

- **repository** → it is an ArrayList<String> object that saves every message received via UDP.

I choose to use ArrayList structures and not built-in array because because an ArrayList is easier to manipulate (its size can be modified without any problem).

4.2 Synchronized methods

All the methods I used to read and modify the several ArrayList objects are synchronized:

- **checkUser** $\rightarrow$ used to check if an entered username while registering is valid or not.
- **update_users_list** $\rightarrow$ used to update the users list.
- **findUserByUsername** $\rightarrow$ used to get to all the account's details of a specified username.

5 THREAD SCHEME

There are three different type of threads, two created by the server and one by the client.

5.1 Server’s Threads

The very first thread that is created by the server is *Change_Word* which is responsible for randomly choosing a new secret word whenever the timer expires. Then the server creates the *menuHandler* thread every time that a new client connects, these threads are also the real core of the whole project since they answer to the clients’ requests.

5.2 Client’s Thread

The only thread created by the client is *Mess_Receiver* which is responsible of add the client to the multicast group and of saving and printing the received messages through a busy waiting mechanism.

6 COMPILATION

6.1 Javac

As required it is possible to compile the project through **javac**, we just need to add a command in order to include the *gson-2.10.jar* library. In order to compile we need to:

1. Open a terminal
2. Move ourselves to the **src** folder
3. Execute: javac -cp "./../libs/gson-2.10.jar" ./*.java -d ./../bin

After the *-cp* we had the library, then the files we want to compile and eventually with the *-d* we specify the destination for the *.class* files we are creating.
Once we want to execute the code we just need to:

1. Move to the **bin** folder
2. Execute[1]:
    - java -cp ".:./../libs/gson-2.10.jar" WordleServer for Ubuntu/Linux
    - java -cp ".;./../libs/gson-2.10.jar" WordleServer for Windows

6.2 Jar

In order to create the *jar* files I added a *manifest* file for each jar file in which I put the main classes.

- jar cfm server.jar manifestWordleServer.txt *.class
- jar cfm client.jar manifestWordleClient.txt *.class

[1] n.b. in order to launch a client we need to change WordleServer with WordleClient

In order to execute them we need to execute:

- java –jar server.jar
- java –jar client.jar

BIBLIOGRAPHY