
**Università di Pisa**
**Dipartimento di Informatica**

Laboratorio di Programmazione di Reti
corso B

# Word Quizzle

a project authored by

Francesco Iannelli

January 30, 2020

**Abstract**

This year networking course's final at Università di Pisa consisted of implementing a client-server application by the name of Word Quizzle. Word Quizzle is a competitive multiplayer game where the matches take place between two players whose objective is to correctly translate, from the italian language to the english language, the highest number of words submitted by the server in the given time. In this report I will describe my proposed solution consisting of what I think is a scalable server architecture obtained by pairing an iterative task dispatching with a concurrent task execution. Firstly, I will describe the problem in a more detailed way, particularly focusing on the various complications that I encountered and the corresponding solutions that I came up with. I will then describe in detail my version of Word Quizzle, spanning from the data structures that I found appropriate to use in each class implementation to the inter class communication and, last but not least, how I took care of the thread concurrency. Finally, I will conclude this report by making a list of all the things I could have done better and all the aspects where I think there's room for improvement.

## Problem Definition

Word Quizzle can be seen as a miniature social network, a user can create his own account providing the nickname and the password, log in with the freshly created account, make some friends and challenge them to a fierce translation match. In order to achieve all of this many things needs to be taken care of, the first of which is the **persistency** of the users data. Who wants to play a game for hours just to see all of his progress vanish when the server restarts? No one perhaps. To address this issue and to ensure that, among all the other important things to preserve such as nicknames and passwords, the hard earned score of the users remains unaltered between server's restarts I used the fasterXML backed library Jackson. I made this choice after checking out many others Java JSON libraries such as JSON.Simple and Gson, although not being the simplest of the three nor the most used it had the best performance while handling large files, such as databases, thus it was the best choice. Talking about performance, I will also say that I used a concurrent hashmap to store the users' data due to its high retrieval efficiency and thread safeness. To know more about that check the implementation of the QuizzleServer class and, more importantly, the concurrency section.

After persistency, the next thing that I had to take care of was the overall **scalability** of the architecture. I wanted to provide a performance close to that of a multi threaded system but with a much more lower cost in terms of cpu power and still obtaining a better performance than a single threaded server using a Selector. What I came up with is an architecture that uses a Selector to perform **task dispatching**, i.e. accepting clients requests, and then feeds them to a ThreadPoolExecutor which then performs **task execution**, I provided the threadpool with a core pool of four threads, as the majority of the mid-end and low-end cpus have two cores with four threads, and an unbound LinkedBlockingQueue in order to ensure the acceptance of all submitted tasks. This approach allows the thread on which the server's main method runs to solely handle the parsing of the clients requests thus delegating the real execution of the tasks to the threadpool. For every client request the server's selector has to read from the client's socket the parameters of the request, it then has to create the appropriate task and finally it has to send the task to the threadpool, forgetting about it, the threadpool then takes care of executing it and also to communicate the result to the client. The main problem that rose was the following: the match task takes much more time to complete than the other tasks, this posed the risk of saturating the threadpool. I solved the problem by allowing the threadpool's maximumPoolSize to be parsed from command line, allowing the server owner to set it **on his needs**, and by coding a reasonable small keepAliveTime in order to ensure the possibility for a thread to be recycled. Given the fact that, in the end, I adopted the selector-threadpool solution the next implementation's detail worth mentioning is how I handled the client-server communication. Due to the presence of the Selector I provided the server with **non blocking** sockets using Java NIO. Talking about the client though, I didn't see any particular reason at all to justify the use of non blocking sockets, on the contrary, it seemed to me a more correct client's behavior to block and wait for the server response so I provided the client with **blocking** sockets, I also liked more the idea of a *well behaved* client that waits for its current request to be executed before sending another one. Following the given requirements I ensured that most if not all of the client-server communication happens on the main TCP connection, which is the one associated with the socket that gets created upon accepting the incoming client's connection on the server's master socket, except for the client-server match communication, which instead takes place on two other TCP connections, set up by both clients with the server upon the acceptation of the match invitation by one of the two contestant clients. I would like to to conclude this section by stating that I was initially not satisfied with the fact that my application was only able to function on the loopback address so I came up with what I think is a pretty decent solution to make it work on the LAN of the host that runs the server program, in a way that allows the clients to dynamically discover the server's local IP address. To know more about the implementation check the client's description.

## Project Structure

While defining the structure of my project I tried to maintain a strict but cohesive modularity, isolating as much as possible each of the Word Quizzle application's features on which I then proceeded to shape the classes. I ended up shaping **eighteen classes** in total, the role of which will be carefully described in this section, however, for an even more detailed description of the classes logic it's highly suggested to check the **javadocs**.

### QuizzleServer

It's the server's class, here are declared and instantiated the server's selector and, server's threadpool and all of the class variables such as the match duration and the number of words to submit to the players per match.

### QuizzleClient and QuizzleClientGUI

Those classes contain all of the client's logic: they handle the communication with the server and can dynamically find the QuizzleServer's IP address by broadcasting probing UDP packets on the port passed by command line. A running QuizzleServer, if there's one on the LAN, will eventually respond with it's IP address and its port number. If the exploration takes too long the attempt is aborted. QuizzleClient can display a command list. It's also important to mention that the QuizzleServer can serve both those classes at the same time and two users using respectively QuizzleClient and QuizzleClientGUI can even match each other.

**QuizzleDatabase**

This class provides all the method to manipulate the users' database and it also handles the **serialization** and the **deserialization** of it. As mentioned in the previous section all the serialization is handled with the **Jackson** library, more precisely with **three** .jar files:

1. **jackson-annotations-2.10.2.jar**
2. **jackson-core-2.10.2.jar**
3. **jackson-databind-2.10.2.jar**

Every each file must be present in the `lib` folder, which must be included in the classpath. Also it's **very important**, to ensure the correct functioning of the application, that all of the above mentioned files match the **same version**.

**QuizzleUser**

It's the class that I used to represent a Word Quizzle **user**, it's provided with all the necessary fields to model a generic user such as the nickname, the password, the friend list and the score plus all the getters and setters methods of the case. QuizzleUser implements the Comparable interface in order to build a sorted scoreboard.

**MatchWords**

This class' purpose is to randomly **choose a list of $n$ words** from the italian dictionary text file, where $n$ is the number of match words set upon the server creation. It also takes care of the translation process, sending GET requests to the mymemory API. If the previously mentioned translation service is **not available** it alerts the MatchTask class to return an error message to both players stating that the service is currently unavailable and suggesting to rematch later. Also the match doesn't take place and the MatchTask is terminated. The class MatchWords it's only instantiated during matches, more precisely in the MatchTask.

**ItalianDictionary**

Although it is not a class, it's a file, it seemed opportune to me to mention it in this section due to its strict relationship with the MatchWords class. It's in fact the **text file where the MatchWords class randomly picks the words**.

**RegistrationRMI**

It's the remote interface that provides the user's registration method. The actual method implementation is in the QuizzleServer class. Among all the implementation details it stands out that each user's nickname must be **unique** and this uniqueness is ensured during the registration process.

**QuizzleMail**

It's the class that represent a **message** that must be delivered to the quizzle user in order to provide him with information about the result of his requested operation. The *address* of the message is the class' key field and the content of the *message* is the class' message field.

**Tasks**

The tasks are all **runnable classes**, they create an instance of a user-requested operation which is then executed by the server's threadpool. All of the tasks are constructed in the QuizzleServer's main thread.

**TaskInterface**: it's an interface all tasks implement, it extends runnable and provides utility methods to read and to write from/to a socket channel, to receive and send datagrams and to add messages in the server's post depot.

**Mailman**: it's the task that writes the responses in the sockets, it constantly checks if a message, if there are any in the quizzle server post depot, can be wrote to a client. It runs **as long as the server is up**.

**LoginTask**: is the login operation.

**LogoutTask**: is the logout operation.

**AddFriendTask**: is the add friend operation.

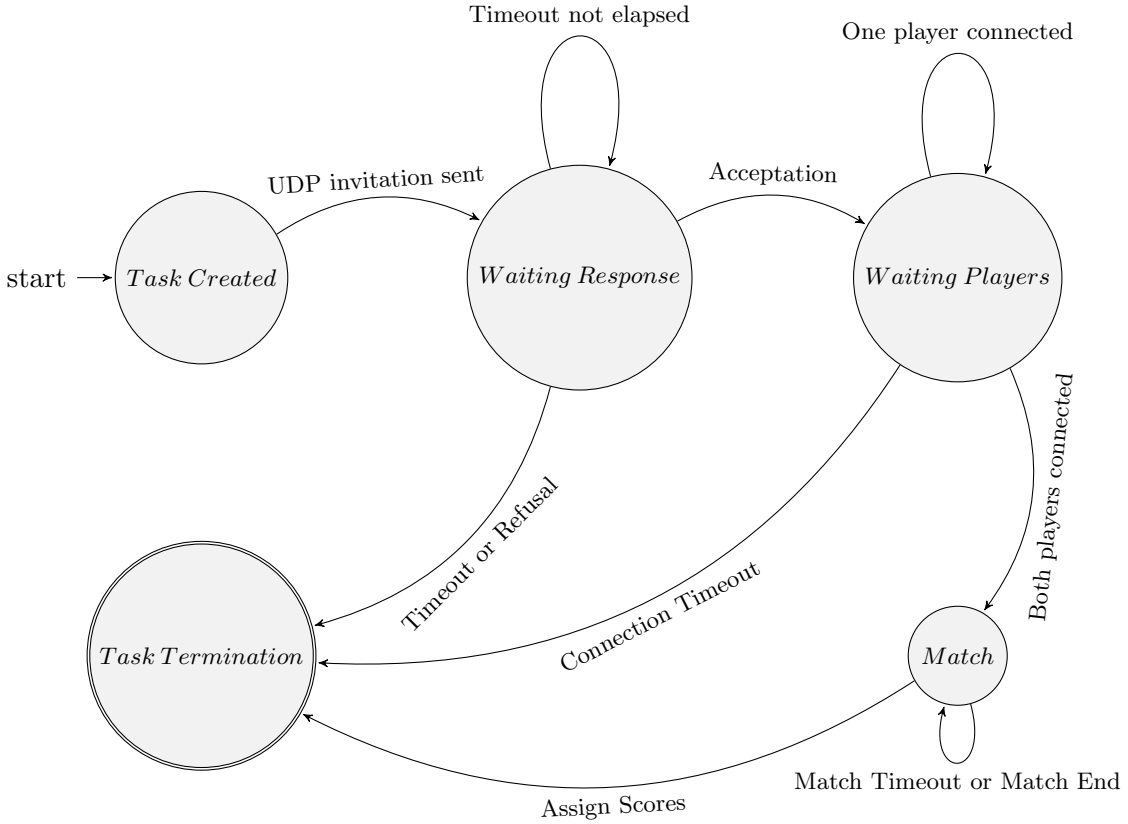
**GetFriendListTask**: prints the user's friend list.

**GetScoreTask**: prints the user's score.

**GetScoreBoardTask**: prints the user's scoreboard.

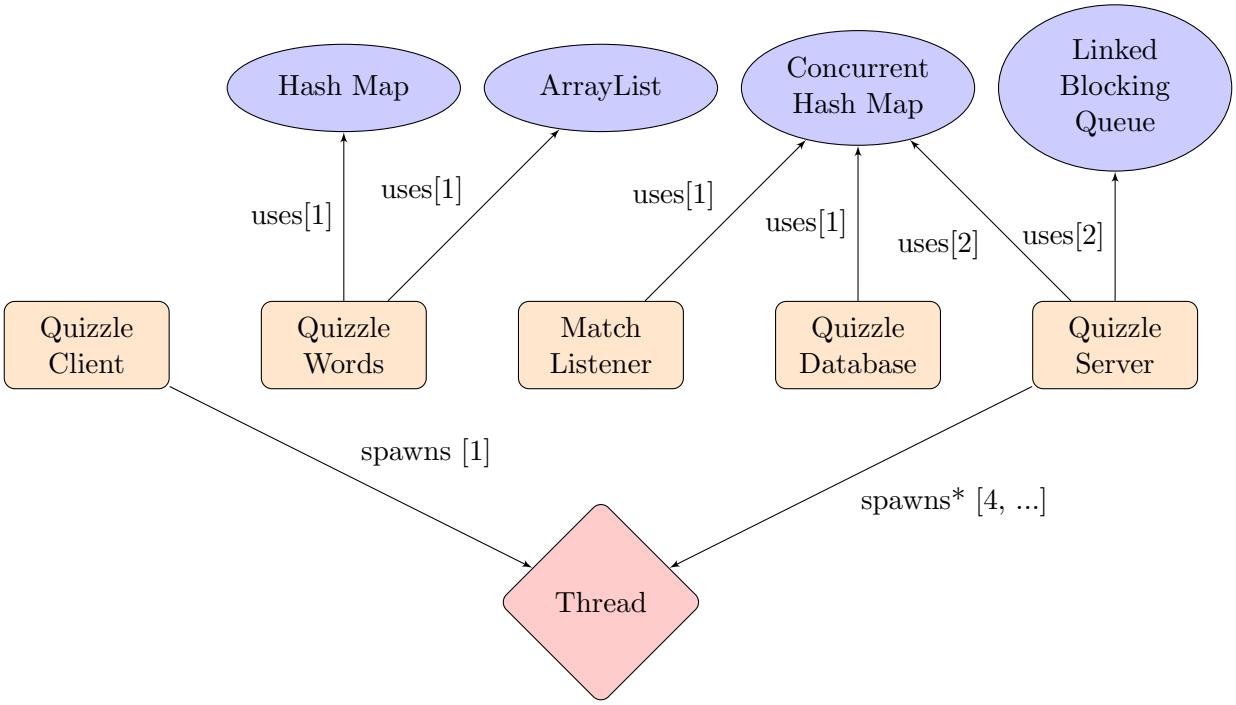
**MatchTask**: sets a translation match between two users. This is the most worthy task to pay attention to. I will explain its functioning with the help of a finite state machine in the next page. Also checking the **javadocs** is highly recommended.

**Match Task Finite State Machine**




**Concurrency and Data Structures**

I think it's a good idea to group the dissertations of both thread concurrency and data structures used because the two topics are strictly correlated. Firstly I will provide a brief explanation with some diagrams. The ellipses represent the data structures, the rectangles with rounded corners represent the project's classes, and the diamond shape represent a thread instance. The labels on the relations edges indicate the arity of the relation.



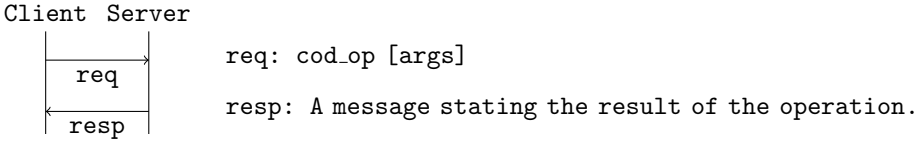
As can be easily seen in the diagram above, the QuizzleClient class spawns **two** threads: one is the thread on which the class main method runs and the other one executes the MatchListener class `run()` method, constantly listening for match invitations, which upon arrival are inserted in a ConcurrentHashMap, where the keys are the users' nicknames and the values are the UDP datagram invitations. QuizzleServer also spawns some threads though indirectly, the number varies from **four** to the threadpool's **maxPoolSize**, which is parsed by command line. The class also uses two ConcurrentHashMap: one for keeping trace of online users while the other is used as a book address to send match invitations. Another class that makes good use of the ConcurrentHashMap is QuizzleDatabase, in fact, the database itself is nothing more than that. Finally let's talk about MatchWords, i.e. the class that provides the words and their translations, it uses an HashMap whose keys are the words and whose values are an ArrayList containing all of the translations. For a more detailed description it's highly suggested to check the **javadocs**. Let's dive now in the concurrency discussion, I would like to begin by making a list of all the possible race conditions that I spotted:

1. The insertion of a user in the database.
2. The adding of a friend by a user.
3. The serialization of the database.
4. The updating of a user's score.
5. The insertion of a user among the online users and his UDP match address in the book address upon the completion of the login operation.
6. The removal of a user from the online users and his UDP match address from the book address upon the completion of the logout operation.

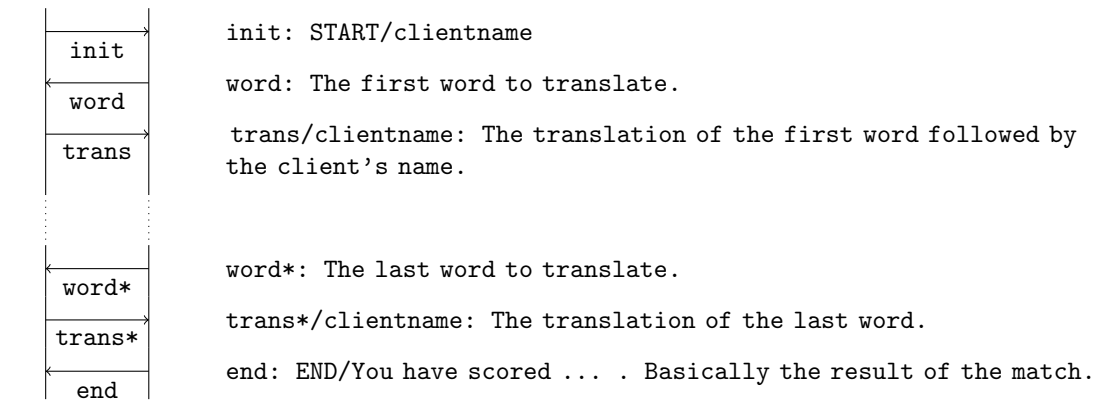
All of the race conditions above have been taken care of by using a ConcurrentHashMap and by declaring the method that perform the serialization of the database as synchronized, as the database must be serialized upon each modification. I conducted extensive tests and never had a problem related to concurrency. To my knowledge there are not any unchecked race conditions.

### Client-Server Communication Protocol

In this section I will briefly illustrate all of the possible client-server interactions with the aid of an interaction diagram to provide an overall description of the **communication protocol**'s functioning.




E.g. for the `login` operation the format of **req** would be: `0 username password UDP_port_number`. I'll provide the same scheme to illustrate the protocol for the client-server **match** communication.




Those protocols apply to **both** the command line client program and the graphic user interface one. Thus, as previously mentioned, the QuizzleServer application doesn't perceive **any difference** between the two implementations when providing the service and no modification to its code is needed.

### Compilation and Execution

To compile the project on **Linux distributions** simply run the following commands in the project folder:

```
javac -cp "./:lib/*" QuizzleClient.java
javac -cp "./:lib/*" QuizzleServer.java
```

To run the server on **Linux distributions** simply run the following command:

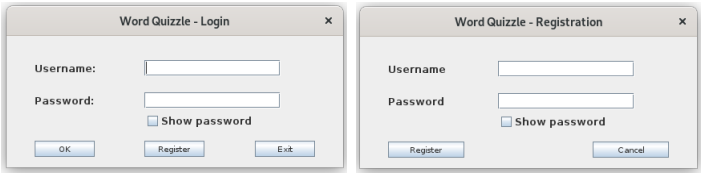
```
java -cp "./:lib/*" QuizzleServer.java <TCP_port> <probe_port> <match_timer>
                    <invitation_timer> <num_words>
```

To run both the clients versions on **Linux distributions** simply run the following commands:

```
java -cp "./:lib/*" QuizzleClient.java <probe_port>
java -cp "./:lib/*" QuizzleClientGUI.java <probe_port>
```

`QuizzleClient` and `QuizzleServer` take `--help` as argument in order to provide the user with an useful command list. To **stop** the client use the `quit` command and to stop the server feel free to use `ctrl + c`.

### Brief Graphic User Interface Guide




From right to left: the login window and the registration window. Those are pretty self explanatory.


This is the main window of the application, to match a friend or accept his invitation just select him from the scoreboard and press the match button. Invitations are notified with a pop-up.

### Self-critique

I am overall very satisfied with what I achieved, nevertheless I do not fear to admit that there are certain things that I could have done better, among which:

- I could have made the threadpool fixed size, thus emphasizing the crux of my architecture, by keeping the status of the clients' requests in a data structure, such a ConcurrentHashMap where the keys are the nicknames and the values are the operations, and by implementing only three tasks: a read task, a write task, and a cleanup tasks, for the tasks that need a timeout. Although it sounds very exciting it's also very mind bending.
- A thing I'm not satisfied at all with is the server termination. There is plentiful of fancier ways to stop a while cycle than `ctrl + c`. Still this is a very minor rant.
- I could have used a factory class to create the tasks, although it would have been only a style exercise, in the end, I would have liked more my work.