Computer Networks Lab, Word Quizzle project essay. UniPi, Computer Science dept.

Tommaso Colella
545625

January 14, 2020

Contents

1 Design choices

Hereafter are presented the design choices taken during the development of the Word Quizzle game.

1.1 General Architecture

The Initializer class is used to start the Server and provides it with the parametes used to control the duration of the match, the invitation timeout and the number of words to be extracted from the dictionary. It also instantiates the UserDB, the ConnectionHandler and the RegistrationHandler. The static Extractor.getInstance() method is used to get the instance of the Extractor Singleton.

The RegistrationHandler is responsible for the remote registration method, called by the client using the Java RMI technology.

I chose to implement the Word Quizzle server class using a Threadpool, assigning each new connection to the pool's threads. The Threadpool is created using the Executor static method `Executors.newCachedThreadPool()`, which provides a pool that improves the performance of programs that executing many short-lived asynchronous tasks, like our Word Quizzle (with the notable exception of the match, which takes a considerably longer amount of time to complete).

The Server class waits for connections on a ServerSocket, subsequently passing the freshly created sock as an argument to an instance of the ConnectionHandler class.

The ConnectionHandler parses each received message and takes the appropriate action in order to serve the client's request (e.g. it checks for each new request's permissions making good use of the UserDB class).

The UserDB is in charge of storing users' data. In order to do so, it employs a ConcurrentHashMap<String, User> which uses nicknames to map to every User instance. The ConcurrentHashMap is also very good at solving synchronization problems.

The UserDB also has a dedicated daemon thread used to serialize modifications on the filesystem, called SerializerDaemon.

I used the WordExtractor class in order to extract a specific number of words from a dictionary called `dictionary.txt` and ask for their translations (since there could possibly be more than one) to the MyMemory remote API. The WordExtractor is a Singleton and uses a HashMap<String, ArrayList String> in order to return the requested translations.

The Word Quizzle client implements a loop waiting for user input by means of a Console class instance, subsequently parsing it and sending the appropriate request to the remote server. The client blocks waiting for an answer. Each meaningful input uses a dedicated method that forges the request and writes it to the appropriate socket. A dedicated Thread is run by

the client in order to wait for incoming Datagrams bringing match requests with them.

The open source GSON library, mainly developed by Google, has been used for serialization and deserialization purposes.

1.2 Data structures

The `ConcurrentHashMap` data structure was used inside the UserDB. This choice was made in order to support concurrent retrievals of various kinds of data like users' scores, nicknames and friends.

Inside the WordExtractor, an ArrayList<String> has been used to store the dictionary line by line and an HashMap<String, ArrayList<String>> was chosen in order to return the word-translations mapping to the ConnectionHandler.

The ConcurrentHashMap (just like the simple HashMap) class permits fast O(1) retrievals provided we have a key to look up.

1.3 Activated Threads

The Client activates two threads:

- The main thread

  used to parse user input and forge the appropriate requests.

- The Listener thread

  used for the purpose of waiting for user challenges on a DatagramSocket opened by the main thread.

The UserDB activates the Serializer Daemon thread, clearly used for serialization purposes.

The Server runs a single thread to wait for users' connections and has a CachedThreadpool with a potentially unlimited maximum pool size for task completion purposes. This permits an arbitrary number of threads to be spawned in case of heavy load, but these threads despawn after a while as per the Executors.newCachedThreadPool() method's documentation.

1.4 Concurrency control

Concurrency control takes place inside the UserDB by using a ConcurrentHashMap to store Users.

Each modification of an User object is made thread safe by means of a synchronized code block. The synchronization takes place outside the User class, thus being an external synchronization employing the Java Monitor technology.

2 Classes

Hereafter the Word Quizzle classes are neatly presented. You can also refer to the javadocs (where the constructor parameters are thoroughly described) and to the commented code.

- `Initializer.java`

  Its purpose is to set up everything for the server-side execution of the Word Quizzle game. The code checks for command line arguments and possibly prints the help message, if appropriately requested. Consequently, instances of UserDB, Server and Registration classes are created. The Extractor's instance is retrieved by calling the static WordExtractor.getInstance() method. The appropriate Threads are started and finally the Initializer prints a welcome message.

- `UserDB.java`

  The UserDB preserves instances of the User classe inside its ConcurrentHashMap. At launchtime it deserializes the Database from the database.json file and creates it if absent. The UserDB has methods to get and put users inside of the ConcurrentHashMap. The GSON library is used to appropriately deserialize the database.

- `SerializerDaemon.java`

  The SerializerDaemon is used to periodically serialize the database on the filesystem. The daemon simply waits 15 seconds and then serializes the database using the GSON library.

- `User.java`

  The User class is used to store each user's nickname, password hash, score, friendlist, last known UDP port and session ID. There are simple getters and setters to access the fields. The User class also has a boolean indicating whether a user is considered to be logged in or not. The session ID is used to check for operations' legality inside the ConnectionHandler. The class implements the Comparable<T> interface and the compareTo method for scoreboard ordering.

- `WordExtractor.java`

  The class is used to extract words from the dictionary and to get their translations from the MyMemory API. It uses Random to create indexes in order to access an ArrayList where each dictionary word is stored, then it asks for translations sending a GET request to the MyMemory API. In order to do so, it employs the Java URL class. To parse MyMemory answers, we use the GSON library. The class has methods to return the words and their translations.

- `RegistrationInterface.java`

  Remote Interface for the user registration RMI. The actual implementation of the class is inside the RegistrationHandler, which is the exported remote object.

- `RegistrationHandler.java`

  The RegistrationHandler is responsible for calling the register() remote method. Since RegistrationHandler extends the UnicastRemoteObject class, explicit object exportation is unnecessary. The init() method creates a Registry and binds to it the RegistrationHandler object. The register() method calls addUser on the ConcurrentHashMap<String, User> db, then it prints the outcome of the operation.

- `Server.java`

  The Server class is one of the application's core classes. It waits for users' connections on a ServerSocket, subsequently passing the freshly created Sockets to a ConnectionHandler task.

- `ConnectionHandler.java`

  Used to parse client messages and to take appropriate action, the ConnectionHandler class has methods to read and write sockets and to correctly answer to each request. In order to serve the clients, the UserDB is often queried for data, and, in case of an accepted match request, two more threads are opened to serve each client's game. The class bears some similarities to Client.java in its readMsg, writeMsg and parseInput methods.

- `MatchHandler.java`

  The MatchHandler is a Runnable class used to serve words to and receive translations back from the clients. The ConnectionHandler waits for its termination in order to calculate scores.

- `Client.java`

  The Client class is a simple request-response blocking client. It connects to the server specified by a command line hostname and is then used to access Word Quizzle's server functionalities. It also calls a remote method on the server for user registration purposes and has a dedicated thread to wait for user challenges.

Appendix A How to compile and execute

A.1 Compile instructions

Just use
`javac -cp ./:lib/gson-2.8.6.jar *.java`
in order to quickly compile all classes to Java bytecode.

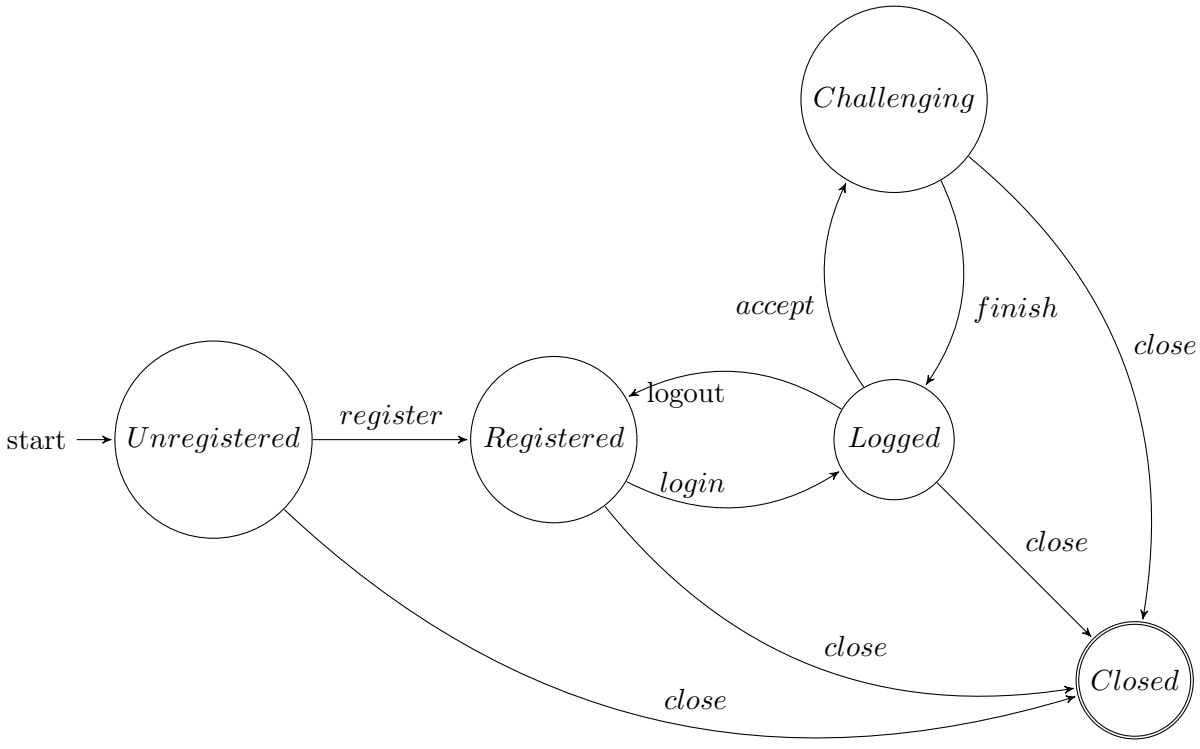
A.2 Execution instructions

To run Word Quizzle locally use:
`java -cp ./:lib/gson-2.8.6.jar Initializer [numWords] [challengeTimer] [matchDuration]`
`java Client 127.0.0.1`

Always include an argument to the Client's execution: either the server's hostname or `"-help"` if you want an explanation on the different client commands available

Appendix B Cool and pretty useless graphs

B.1 Client Finite State Automaton



B.2 Match UML Sequence Diagram

Below you can see a simplified (and ugly) UML Sequence Diagram (Figure 1) modeling the match sequence from the viewpoint of the match sender. The slanted arrows represent UDP messages.

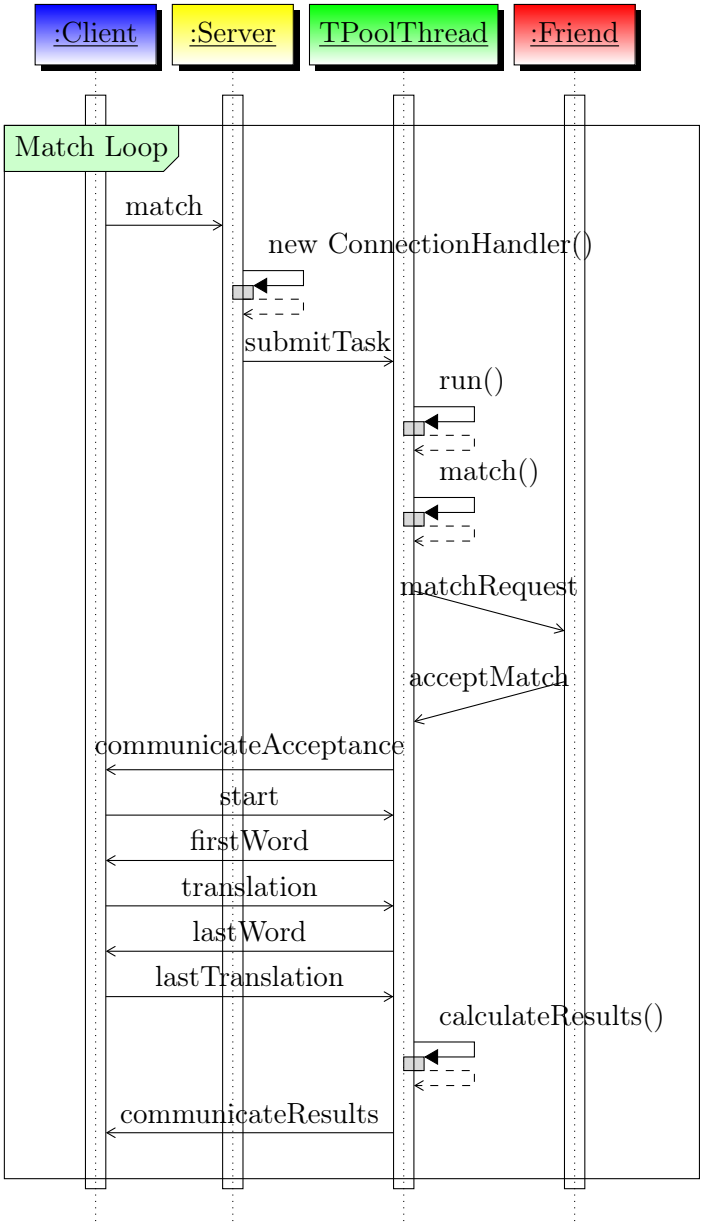


Figure 1: This UML diagram represents the match sequence from the client's viewpoint.