Università di Pisa

Relazione progetto Laboratorio di Programmazione di Reti
a.a. 2019-20

Lorenzo Rasoini
Matricola 546045

Indice

1 Introduzione

Il progetto richiedeva l'implementazione del gioco **WordQuizzle** utilizzando il linguaggio Java.
Il gioco consiste in una sfida tra due utenti, dove il vincitore è il giocatore che è riuscito a tradurre correttamente il maggior numero di parole inglesi a lui presentate.
Si richiedeva che l'applicazione fosse implementata usando una architettura client-server in cui client e server comunicano tramite protocolli TCP, UDP e RMI.
Era inoltre richiesta l'implementazione di un database persistente per contenere le informazioni degli utenti usando il formato JSON.
Per quanto concerne il lato client era richiesta una implementazione di una CLI o di una GUI. All'interno del mio progetto sono presenti entrambe le implementazioni.

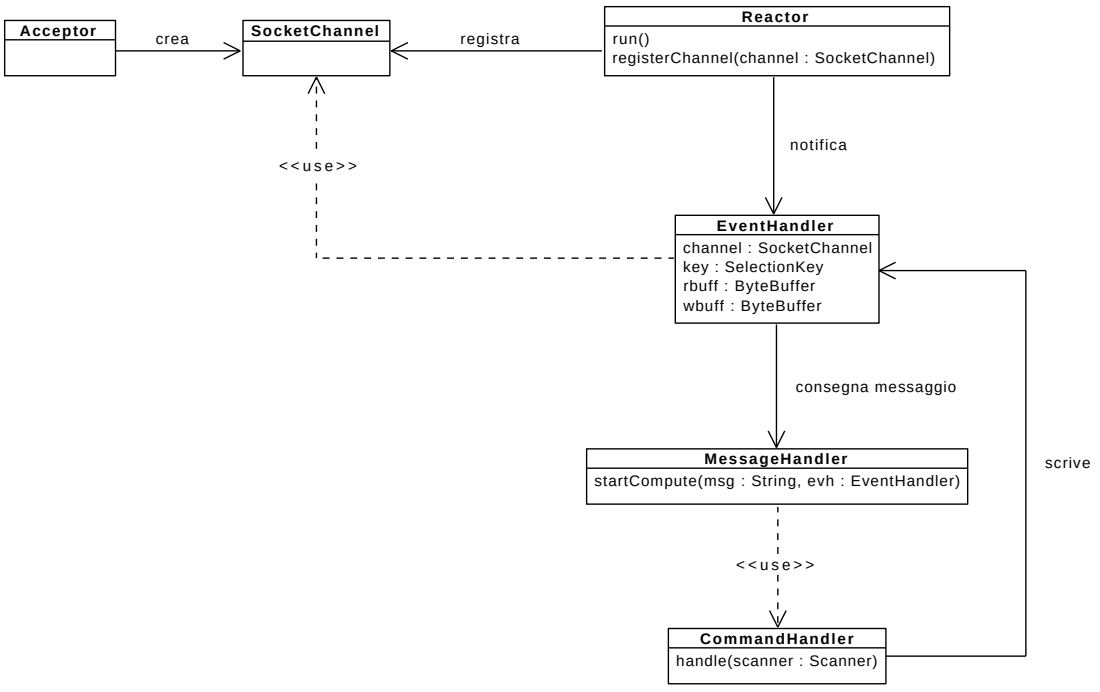
2 Architettura

Il progetto è diviso in due grandi package:

- `WQServer`, contenente tutte le classi necessarie al funzionamento del server.
- `WQClient`, composto da due package `cli` e `gui`, contenenti rispettivamente l'implementazione della CLI e della GUI.

Fuori da entrambi i package sono presenti le classi condivise da entrambi i package menzionati sopra (di particolare importanza la classe `Response` contenente praticamente la specifica del protocollo utilizzato all'interno del sistema).

2.1 Server



Architettura server

Il "nucleo" del server è costituito dalle classi `Reactor` e `EventHandler`. `Reactor` implementa un Reactor pattern semplificato, dove tramite la libreria NIO si effettua la selezione dei canali pronti per per la lettura e/o scrittura e in caso di responso positivo si passa il controllo al metodo del `EventHandler` associato al canale incaricato di gestire l'evento rilevato. Considerato che dopo aver lasciato il controllo al `EventHandler` il `Reactor` è momentaneamente non in grado di processare altre letture/scritture, ho cercato di minimizzare la durata delle operazioni lanciate da `EventHandler` il più possibile.
Una volta che `EventHandler` ha a sua disposizione un messaggio (sono gestite letture parziali e letture di più messaggi con la medesima `read`) viene invocato il factory method `MessageHandler.getHandler` che restituisce il `MessageHandler` appropriato per lo stato in cui si trova il client associato al `EventHandler.`

La parte di registrazione del canale è gestita dal metodo `Reactor.registerChannel` che si appoggia a una `BlockingQueue` contenente la coda dei canali in attesa di registrazione e una NIO pipe (usata anche per gestire le richieste di scrittura su un canale). Il meccanismo della pipe viene usato semplicemente per "svegliare" il Reactor e fargli rilevare la registrazione o la scrittura (una soluzione più semplice ma meno "divertente" da implementare sarebbe stato inserire un timeout nel metodo `select()`).

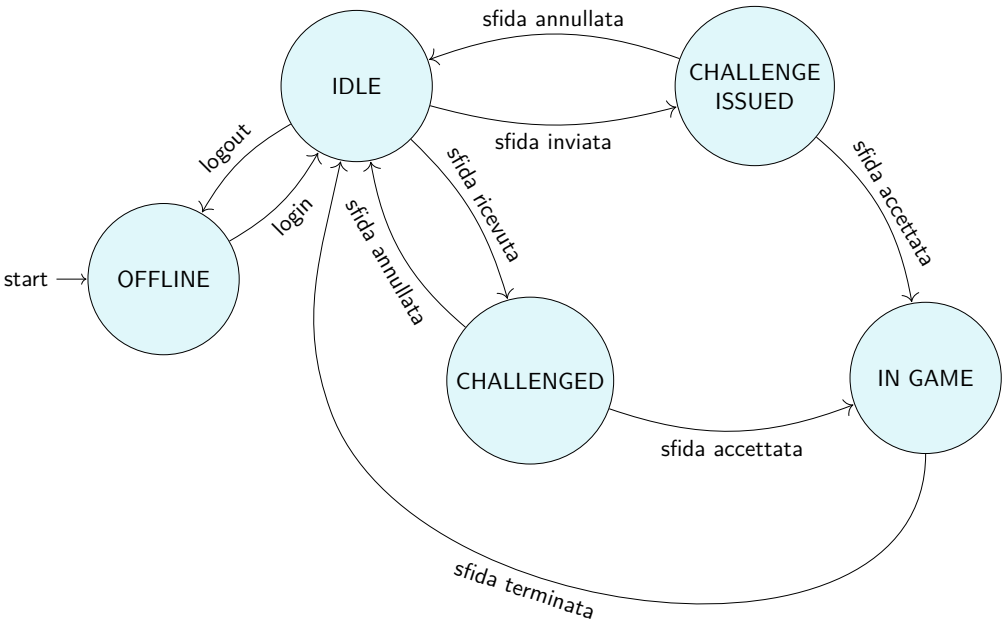
Questo approccio è stato da me scelto per la sua caretteristica di evitare completamente l'utilizzo di molteplici thread e l'overhead ad essi associato. Inoltre mi ha permesso di implementare un load balancing estremamente semplice (al momento della creazione di una connessione questa viene assegnata al thread `Reactor` che sta gestendo il numero più piccolo di connessioni).
Ritengo infine che un approccio del genere mi abbia permesso di essere più aderente al paradigma OOP e mi abbia permesso di ridurre il numero di metodi richiedenti la mutua esclusione all'osso (solamente il metodo `EventHandler.write` richiede l'acquisizione di una lock).

2.2 Client

Le scelte architetturali fatte all'interno dei client riflettono le scelte architetturali fatte per il server.
Anche nei client è presente una classe astratta `Reactor` che svolge sia il compito di ricevere e inviare messaggi che il compito di gestire i messaggi ricevuti e esporne gli effetti al resto del client. Proprio per questo motivo la classe `Reactor` viene estesa dalle classi `CLIReactor` e `GUIReactor` che implementano al loro interno la logica necessaria al funzionamento del rispettivo tipo di client (metodo `handleMsg`).

3 Altre componenti importanti

3.1 User



FSM che modella lo stato di un utente.[1]

.

La classe `User` modella un generico utente di **WordQuizzle**. La classe è provvista dei vari attributi e metodi necessari a gestire nome utente, password, punteggio e lista amici. Quest'ultima è implementata come una `ConcurrentHashMap<String, User>` per gestirne efficaciemente gli accessi concorrenti.
Funzionalità chiave esposta all'interno della classe è il metodo `setState` che permette la transizione di un utente da uno stato a un altro causando la stessa transizione anche lato client. All'interno della classe sono inoltre presenti il serializzatore e il deserializzatore JSON.

La gestione della concorrenza all'interno di questa classe (come nel resto del sistema) è stata implementata tramite monitor lock sui vari attributi della classe, anche dove apparentemente non sarebbero stati necessari, come sulle operazioni inerenti alla lista amici implementata come `ConcurrentHashMap<String, User>`. [2]

[1] Alcune transizioni non sono state rappresentante per non appesantire eccessivamente il grafo, è ad esempio possibile effettuare logout in qualsiasi stato diverso da OFFLINE ed è possibile abbandonare il gioco anche quando si è IN GAME

[2] "*Iterators are designed to be used by only one thread at a time. Bear in mind that the results of aggregate status methods including size, isEmpty, and containsValue are typically useful only when a map is not undergoing concurrent updates in other threads. Otherwise the results of these methods reflect transient states that may be adequate for monitoring or estimation purposes, but not for program control.*" da https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

3.2 Database

Il database di **WordQuizzle** è implementato come Singleton e gestisce le letture e le scritture al suo interno tramite una `ConcurrentHashMap<String, User>`. La soluzione qui adottata purtroppo non è in grado di scalare efficaciemente, in quanto avendo tutto il database memorizzato in unico file JSON siamo costretti a caricarlo tutto in memoria. D'altro canto però questa soluzione permette di avere un unico riferimento per ogni oggetto di tipo `User` e questo permette di non dover gestire la sincronizzazione degli utenti tra i vari thread.
Per gestire la concorrenza nell'accesso al Singleton ho usato un meccanismo di **double-checked locking**.

3.3 CommandHandler

Questa classe contiene tutte le classi utilizzate da `EventHandler` per realizzare la logica necessaria a rispondere a un messaggio ricevuto da un client.
L'handler più interessante tra quelli presenti nella classe è `IssueChallengeHandler` che ha come incarico la creazione e notifica di una sfida fra due utenti e include al suo interno la procedura di invio del pacchetto UDP di notifica sfida tramite la classe `DatagramSocket`.

3.4 Acceptor

Questa classe implementa l'accettore di connessioni TCP usando la libreria NIO. Una volta accettata la connessione viene richiesto al `Reactor` meno "impegnato" dei quattro creati all'avvio di registrare la connessione al suo interno.

3.5 RegisterServerHandler

Questa classe implementa la registrazione a **WordQuizzle** tramite RMI. Vengono controllate al suo interno le varie condizioni che devono essere soddisfatte per essere registrati, quali unicità dello username scelto e presenza di soli caratteri validi nello username e nella password.

3.6 Challenge

La classe `Challenge` implementa al suo interno tutta la logica inerente una sfida tra due giocatori, a partire dalla creazione dei timer di timeout fino alla richiesta delle traduzioni al servizio di traduzione offerto dalla API REST di translated.net
Questa classe è l'unica del sistema (oltre a `Acceptor` e `Reactor`) che richiede l'esecuzione all'interno di un thread separato, scelta motivata dalla presenza della richiesta HTTP GET che può bloccare per un tempo abbastanza lungo (circa 1s sulla macchina su cui ho sviluppato e testato il sistema)
All'interno della classe sono implementate tutte le procedure relative allo svolgimento della sfida, quali:

- Definizione delle variabili di gioco K, T1, T2, X, Y e Z.
- Gestione dei timer per il timeout sia della richiesta di sfida che della sfida avviata.
- Caricamento del dizionario e selezione delle parole su cui si giocatore.
- Memorizzazione delle "statistiche" della partita all'interno della classe `GameData`.

- Assegnazione dei punti e scelta del vincitore.
- Meccanismo di terminazione eccezionale della sfida.

3.6.1 Chiamata alla API REST

La chiamata alla API REST di translated.net avviene tramite le classi `URL` e `URLConnection`: una volta ricevuta la risposta si procede a leggerne il contenuto e, dopo aver verificato che la risposta HTTP sia di tipo `200 OK`, si procede a leggere il JSON ricevuto e a inserire le traduzioni all'interno di una `HashMap<String, ArrayList<String>>` avente la parola in italiano come chiave e la lista delle possibili traduzioni come valore. Questa struttura dati verra consultata ogni volta un giocatore invia al server una possibile traduzione.

4 Istruzioni per compilazione e uso

4.1 Compilazione

Per compilare il progetto è necessario portarsi all'interno della cartella `wordquizzle` ed eseguire da shell:

```
javac -d bin -cp src:lib/gson-2.8.6.jar src/wordquizzle/wqserver/*
javac -d bin -cp src:lib/gson-2.8.6.jar src/wordquizzle/wqclient/cli/*
javac -d bin -cp src:lib/gson-2.8.6.jar src/wordquizzle/wqclient/gui/*
```

4.2 Uso

Per eseguire il server è necessario portarsi all'interno della cartella `wordquizzle/bin` ed eseguire da shell:

```
java -cp ../lib/gson-2.8.6.jar:. wordquizzle.wqserver.WQServer <porta>
```

Per eseguire il client è necessario portarsi all'interno della cartella `wordquizzle/bin` ed eseguire da shell:

```
java -cp ../lib/gson-2.8.6.jar:. wordquizzle.wqclient.cli.WQClient
<ip_server> <porta_server>
```

Per usare il client GUI è sufficiente sostituire `gui` a `cli` nel comando sopra.
Le istruzioni per il client CLI si possono ottenere con il comando:

```
java -cp ../lib/gson-2.8.6.jar:. wordquizzle.wqclient.cli.WQClient --help
```

5 Considerazioni finali

Sono complessivamente rimasto soddisfatto del lavoro svolto e del risultato ottenuto e ritengo che il progetto sia stato di grande aiuto nella mia comprensione del linguaggio Java e del paradigma object-oriented (oltre alla programmazione di rete ovviamente).
Nonostante ciò mi sento in dovere di indicare alcune caratteristiche del sistema che avrei potuto implementare più efficientemente, come il database, che avrei voluto distribuire su più file JSON per ottenere una scalabilità un poco maggiore rispetto a quella ottenibile dall'usare un singolo file, o come la classe `User`, per la quale sarebbe stato rivedere alcune operazioni e ridistribuirle su classi differenti.