Nome: Michael **Cognome:** De Angelis **Matricola:** 560049 **Corso:** Informatica – B

Anno Accademico 2019 / 2020

Reti di Calcolatori e Laboratorio

Progetto "Word Quizzle"

"Università di Pisa – UNIPI"

Relazione del: 09.01.2020

Corso: Reti di Calcolatori e Laboratorio [LPRB1920]

Tipologia del lavoro svolto	Luogo di svolgimento delle mansioni	Gruppo di lavoro	Arco di tempo in cui si è svolto il progetto	Tempo impiegato per lo svolgimento delle mansioni
Realizzazione di un progetto assegnato e relativa relazione scritta	Propria abitazione	Singolo	Anno Accademico 2019 – 2020	Circa un mese

Sommario

Analisi e Progettazione 3
GUI: Analisi ed utilizzo 7
Esecuzione 12
Appendice A – Informazioni 12
Appendice B – Match 12

Analisi e Progettazione

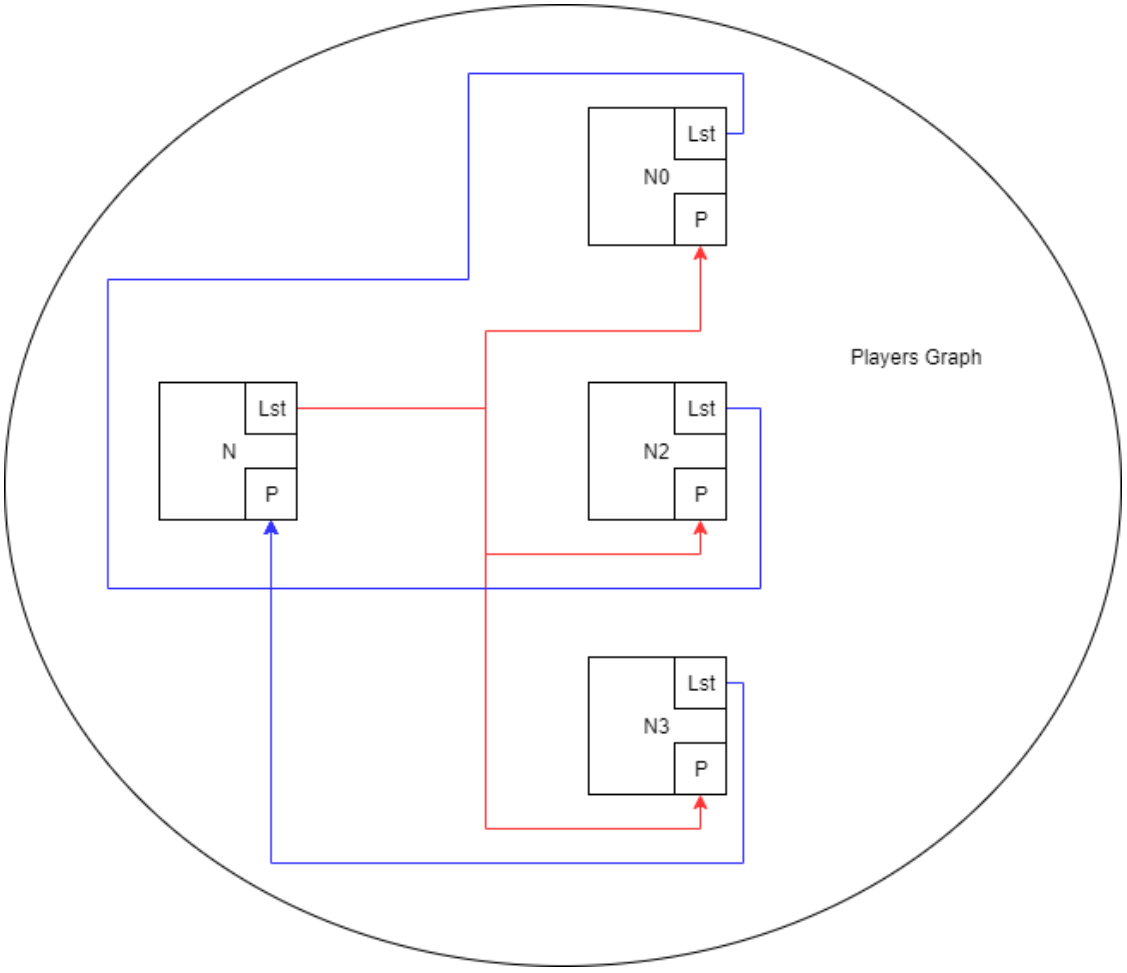
Nella seguente sezione ci occuperemo dell’analisi del testo proposto e delle scelte effettuate durante la fase di progettazione e realizzazione.

Di seguito viene riportato un frammento del testo del progetto:

> *“Il progetto consiste nell’implementazione di un sistema di sfide di traduzione italiano – inglese tra utenti registrati al servizio. Gli utenti registrati possono sfidare i propri amici ad un gara il cui scopo è quello di tradurre in inglese il maggior numero di parole italiane proposte dal servizio. Il sistema consente inoltre la gestione di una rete sociale tra gli utenti iscritti. L’applicazione è implementata secondo una architettura client server.”*

Come punto di partenza, analizziamo la rappresentazione degli utenti nel sistema e la loro organizzazione nella struttura dati utilizzata per tenere traccia delle informazioni e relazioni di questi ultimi. Ogni utente del sistema è infatti rappresentato da un’istanza della classe *Player;* suddetta classe è utilizzata per mantenere informazioni quali *username*, *password*, *status* (online, offline), *punteggio*, *porta UDP* (porta effimera assegnata dal Sistema Operativo, viene comunicata al server in fase di connessione) e *socket TCP* (utilizzato per la comunicazione). La struttura dati che invece si occupa dell’organizzazione degli utenti è un grafo non orientato implementato come una *Concurrent Hash Map*. L’hash map utilizzata effettua associazioni <*chiave, valore*> di tipo *Stringa -> Node*; la stringa rappresenta il nome dell’utente mentre Node rappresenta una classe che incapsula un’istanza della classe Player e contiene una lista di oggetti di tipo Player implementando così le relazioni di amicizia tra gli utenti del sistema.

Ipotizzando l’esistenza di un utente N avente tre amici, lo schema appena enunciato può essere rappresentato nel seguente modo:



La struttura incapsulata descritta poc'anzi è volta ed evitare di ricadere in una ricorsione infinita durante la scrittura del file *JSON*; Il file JSON richiesto dalla traccia del progetto, essenziale per mantenere persistenti le informazioni di registrazione, relazioni di amicizia e punteggio degli utenti, è infatti ottenuto a partire dal grafo attraverso la serializzazione di questo ultimo utilizzando i metodi forniti dalla libreria *Gson* (da questa operazione di salvataggio sono esclusi i campi status, porta UDP e socket TCP in quanto marcati come *transient*). Ad integrare quanto descritto vi sono le classi *NotExistingUsrException*, *WrongPswException* e *Status*; come intuibile dal nome, le prime definiscono due nuove eccezioni lanciate in caso di utente richiesto, per una data operazione, non esistente e in caso di password errata in fase di login per un determinato utente; Login definisce invece una semplice enumerazione che rappresenta gli stati (online o offline) dell'utente.

Il grafo degli utenti e le operazioni su di esso definite sono utilizzate dal server e dai Thread da esso generati; di conseguenza il grafo stesso è stato progettato in modo che tutte le funzioni effettuate su di esso siano Thread-safe. Tra i metodi definiti nella classe che ospita il grafo (*PlayerGraph*) solo due sono chiamati in modo concorrente: il metodo di aggiunta di nuovi nodi (istanza della classe Player e relativa lista di amici) e il metodo di aggiornamento del punteggio di un utente. Tale caratteristica è stata raggiunta effettuando il *multiplexing* dei canali attraverso NIO e quindi dedicando il thread principale del server ad assolvere alle seguenti richieste:

- Login;
- Logout;
- Aggiunta di un amico;
- Lista amici: si fa uso della classe *ToClientLst* in modo da comunicare al client una lista priva di informazioni sensibili (password, ecc.…);
- Mostra punteggio;
- Mostra classifica: anche in questo caso si fa uso della classe ToClientLst;
- Sfida;

L'operazione di registrazione viene effettuata, come da specifica, attraverso il servizio RMI (classe *WqSignUp* contenente l'interfaccia e classe *WqServer* contenente l'implementazione come classe privata) che si occuperà di chiamare la funzione di aggiunta di un nuovo nodo al grafo; dalla generazione di nuovi thread da parte del servizio RMI per gestire le richieste di registrazione si ha la concorrenza di cui discusso precedentemente. Per rendere di conseguenza Thread-safe il metodo è stata adottata la funzione *putIfAbsent* della classe Concurrent Hash Map in quanto essere garantita come Thread-safe.

Per quanto concerne invece il metodo di aggiornamento dei punteggi questo viene eseguito in modo concorrente dai Thread generati dal server una volta pervenuta una richiesta di sfida nel momento in cui questa ultima ha termine. Per garantire la sicurezza di tale funzione è stato utilizzato il metodo *computeIfPresent* (sempre della classe Concurrent Hash Map e anch'esso targato come Thread-safe) che prende come argomento una lambda function da applicare al nodo individuato.

La classe che specifica il comportamento del Thread gestore della sfida è denominata *MatchHandler* e istanze di suddetta classe sono generate dal metodo *handleMatch* della classe WqServer, ovvero la classe contenete l'implementazione del Server vero è proprio. I thread creati sono assegnati ad un Thread Pool.

I thread gestori della sfida si affidano alla classe *TranslationHandler* per ottenere le parole da fornire ai giocatori e le relative traduzioni contattando il servizio RESTful di traduzione.

I client, implementati nella classe *WqClientGUI*, che si connettono al server sono caratterizzati da un'interfaccia grafica realizzata utilizzando SWING; così come per il Server, anche nell'implementazione del client è stato utilizzato NIO ma, diversamente dal Server, in modalità bloccante non dovendo effettuare il

multiplexing dei canali. Il client, oltre ai thread generati automaticamente dai vari handler della GUI, genera esplicitamente due Thread:

- Un Thread che resta in attesa di ricevere messaggi UDP di sfida;
- Un Thread che, in caso di sfida, si occupa unicamente di leggere dal socket i messaggi pervenuti dal server e abilitare la scrittura verso il server delle traduzioni inserite dall’utente durate il gioco.

Dall’ultimo punto, si può dedurre che in lettura e scrittura su socket durante la partita sono eseguiti da due Thread diversi, più precisamente la scrittura è affidata al gestore dell’evento di click sul pulsante di invio nell’interfaccia di sfida. Per sincronizzare letture e scritture sono state utilizzate due variabili booleane, una variabile di condizione e ovviamente la relativa lock. L’unico Thread ad andare il wait è quello di scrittura in quanto si vuole fare in modo che il Thread di lettura sia sempre in ascolto dei messaggi che possono essere inviati dal Server (avversario che abbandona la partita, scadenza del timer della partita, errori di comunicazione).

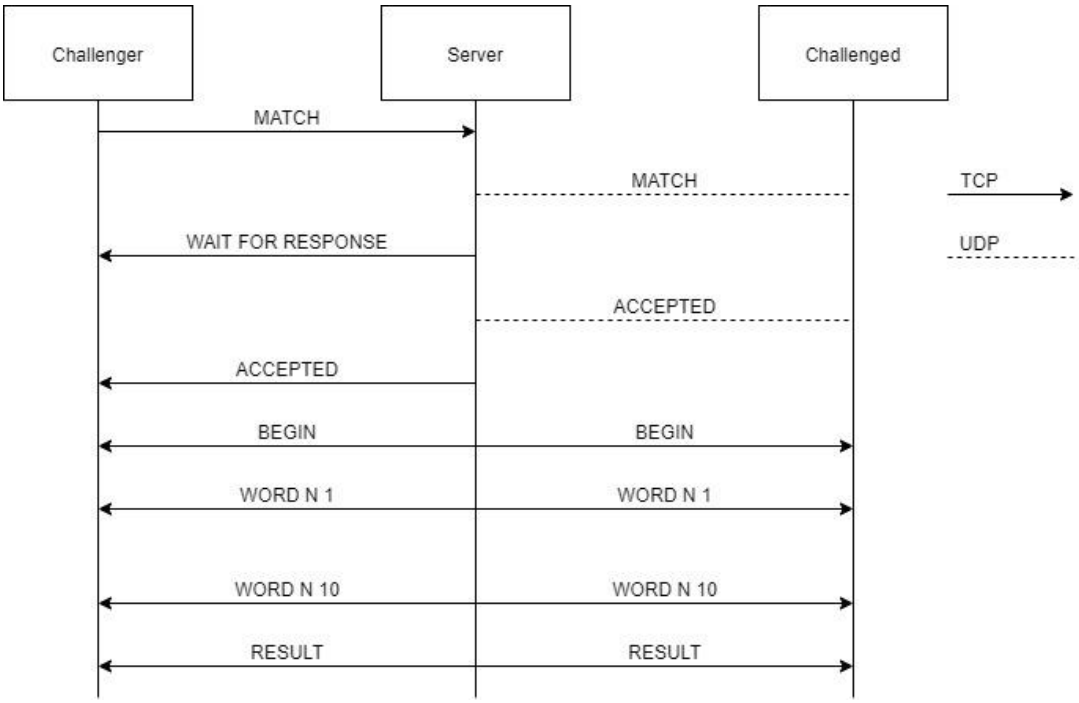
Cercando di seguire una linea di condotta più RESTful possibile, le comunicazioni tra Client e Server sono effettuate non attraverso lo scambio di semplici stringhe ma bensì di stringhe JSON ottenute da oggetti di tipo *Collections.singletonMap* e definendo di conseguenza (e utilizzando come chiavi) vari codici di successo / errore / comunicazione e relativi messaggi di interazione. Tali definizioni sono raccolte nella classe *MyUtilities* a cui tutte la classi precedentemente definite fanno riferimento. La formazione delle stringhe JSON, come la serializzazione del grafo di cui discusso precedentemente è affidata ai metodi descritti nella classe *JsonHandler*; in quest’ultima, per evitare problemi di concorrenza nel momento in cui il server utilizza la stessa istanza della classe, i metodi sono marcati come *synchronized* sfruttando quindi il monitor intrinseco dell’oggetto.

L’ultima classe di cui occorre fare menzione è *ErrorMacro*, essa contiene infatti tutte le operazioni che vengono effettuate in caso di errore durante tutto il tempo di vita dei servizi in esecuzione in modo che questi, anche grazie ad una funzione di *shutDownHook* appositamente definita, cercano di terminare le esecuzioni in modo pulito e “graceful”.

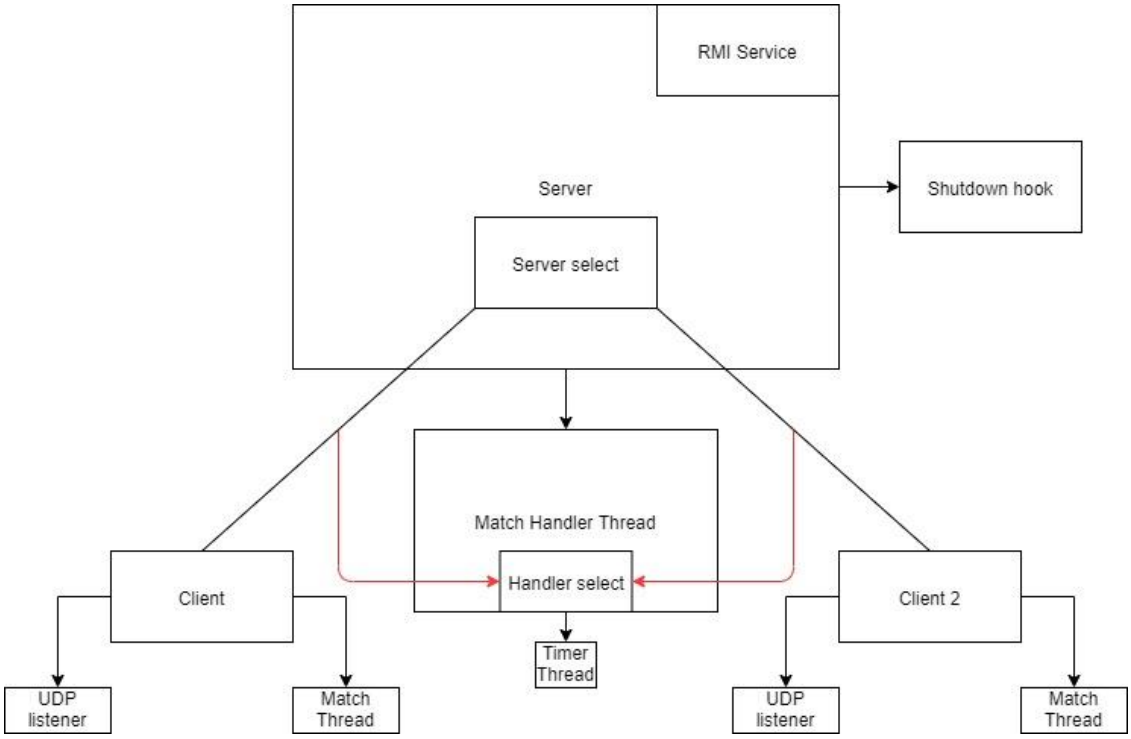
Continuando a discutere dei Thread attivati, l’ultimo di cui occorre fare menzione è il Thread che si occupa di simulare il timer della partita di due giocatori; questo viene infatti attivato ad inizio partita ed esegue una *sleep* di due minuti, passato tale arco di tempo, se non è stato interrotto prima a causa della fine della partita stessa, esegue la funzione di *wakeup* sul selettore che si occupa di gestire i due socket dei giocatori nel Thread dedicato al match, svegliandolo e permettendo da uscire dal ciclo di lettura / scrittura ed inviare i punteggi ai due partner della sfida.

Per quanto concerne invece la fase di sfida è stato stabilito un protocollo per effettuare una sorta di *handshaking* tra i due giocatori prima di iniziare il match vero e proprio. Inoltre, si è scelto di inviare la risposta alla richiesta di sfida utilizzando il protocollo di trasporto UDP in quanto così è stato possibile mostrare come le informazioni del mittente del datagramma UDP siano contenute nell’intestazione di quest’ultimo e come possono essere sfruttate. Ovviamente, facendo una scelta di questo genere è doveroso prendere in considerazione il caso in cui la risposta alla sfida venga persa a causa nella natura del protocollo UDP; per compensare questo errore si è quindi stabilito un timer sulla lettura dell’*advertisement* di *begin* che viene inviato dal server una volta confermata la partita. Si rende inoltre noto che, utilizzando NIO non è possibile utilizzare la funzione *setSoTimeout* disponibile alla libreria di Java.net e quindi per ovviare a questa mancanza è stato usato direttamente il socket incapsulato dall’oggetto *socketChannel*; alternativa valida a questo procedimento è invece l’utilizzo della funzione *select(timeout)*, tale opzione è stata però scartata in quanto l’operazione di attesa dell’advertisement di begin viene fatta client-side dove l’utilizzo di un selettore risulta poco adatto sfruttando socket bloccante e collegandosi ad un unico Server.

Si invita il lettore a osservare lo schema della pagina successiva in modo da comprendere le fasi del match.



Possiamo infine sintetizzare quanto detto attraverso il seguente schema rappresentante la situazione in cui due utenti decidono di sfidarsi (si fa inoltre notare che i socket, all’handler del match, sono ovviamente passati dal server stesso):



Per quando riguarda invece le parole utilizzate per le traduzioni, si utilizza un file di circa mille termini che viene caricato all’avvio del server in una lista; per ogni sfida si selezionano in modo casuale dieci parole;

I punteggi sono assegnati nel seguente modo:

- Risposta corretta: +2;
- Risposta errata: -1;
- Vittoria: +3

Si fa inoltre presente che, causa utilizzo della libreria Gson, il progetto è stato inizializzato come progetto Maven indicando nel file *pom.xml* la dipendenza dal JAR Gson 2.6.2; non è quindi necessario scaricare manualmente la versione corretta della libreria.

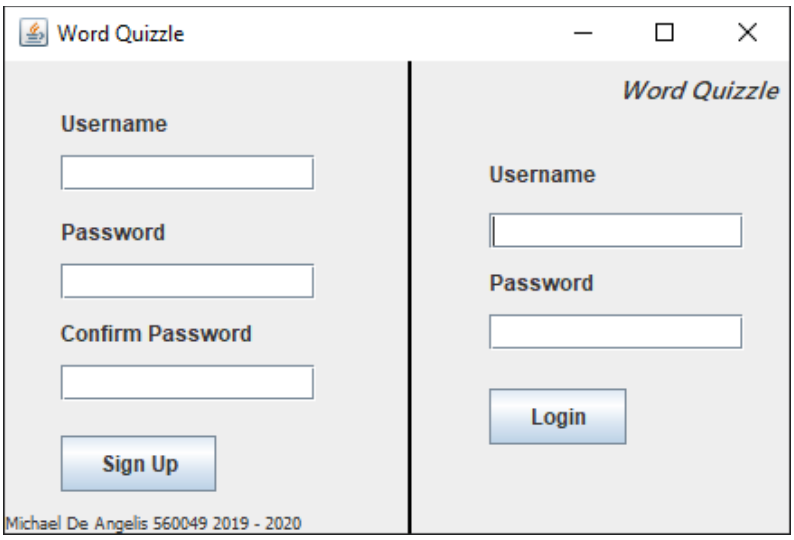
GUI: Analisi ed utilizzo

Come precedentemente esplicitato, per il client è stata realizzata un’interfaccia grafica; tale interfaccia fa utilizzo del *Card Layout* per passare da una schermata all’altra durante le varie fasi di utilizzo. Le schermate visualizzabili sono tre:

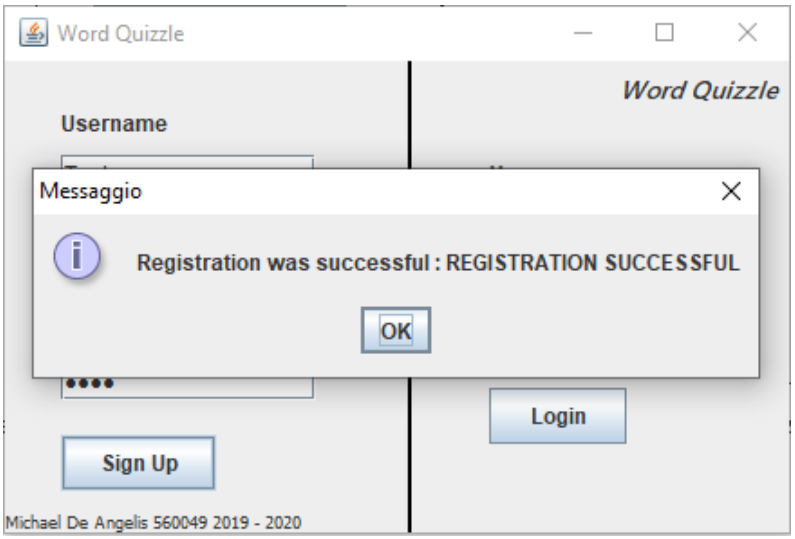
- Registrazione e accesso: abilita la registrazione di un utente e il successivo login;
- Comandi al server: interfaccia in cui inviare i vari comandi al server cliccando i relativi pulsanti;
- Sfida: interfaccia in cui si ricevono le parole da tradurre e si possono inviare le traduzioni scritte dall’utente.

Osserviamo quindi nel dettaglio ognuna della sopracitate interfacce attraverso un semplice esempio di utilizzo.

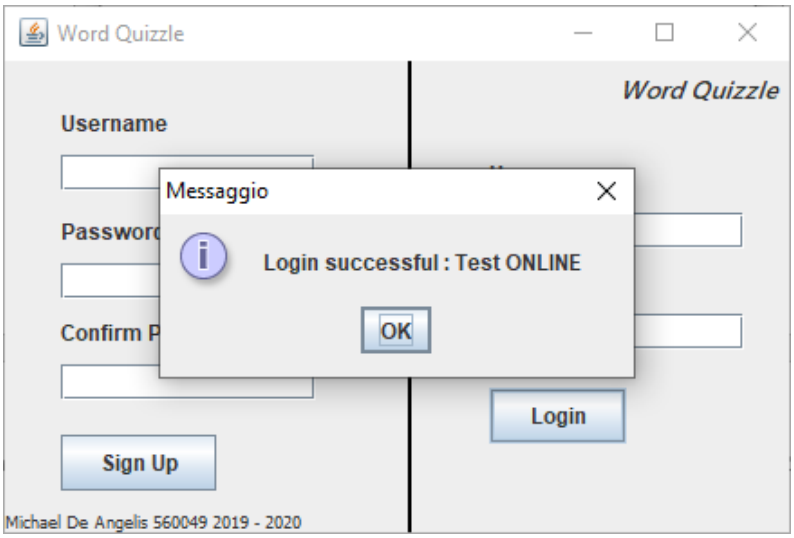
Appena avviato il client avrà il seguente aspetto:



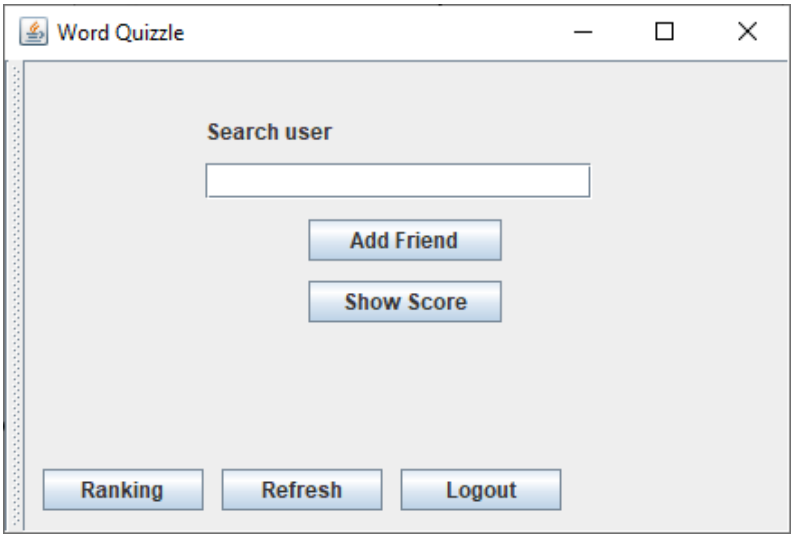
Proviamo a creare un nuovo utente “Test” con password “test” e cliccare di conseguenza sul pulsante Sign Up:



Effettuiamo quindi il Login con l’utente appena creato:



Ci ritroveremo quindi nella seguente interfaccia:

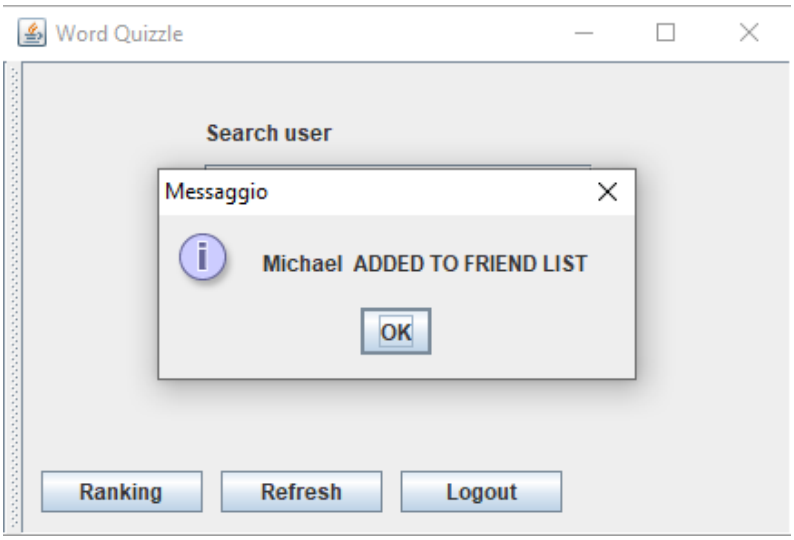


Dalla seguente interfaccia sono possibili le seguenti azioni:

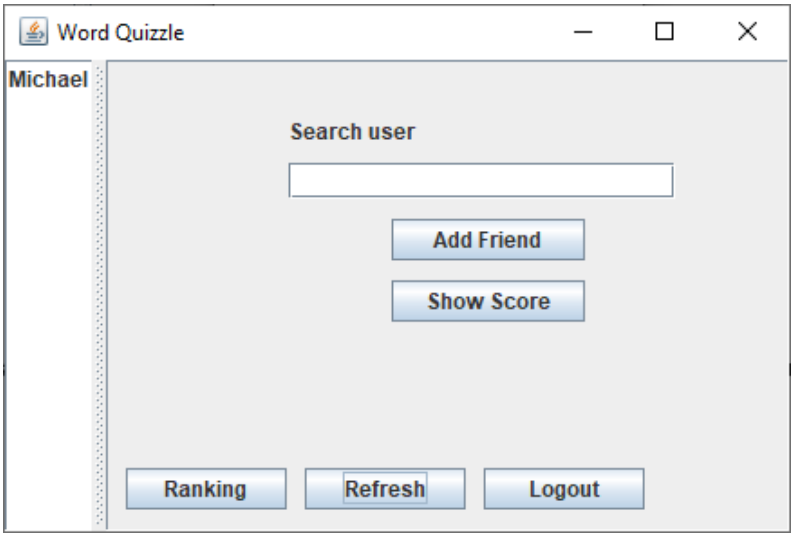
- Aggiunta di un utente alla lista amici digitando il relativo username;
- Mostrare il punteggio di un determinato utente digitando il relativo username;
- Mostrare la classifica, basata sui punteggi, tra l’utente connesso e i suoi amici;
- Effettuare l’aggiornamento della schermata per ottenere la lista degli utenti amici;
- Effettuare il logout.

L’aggiunta del pulsante di refresh è stata presa per mostrare in modo esplicito l’invio della richiesta della lista amici, tale pulsante potrebbe essere facilmente eliminato facendo aggiornare la lista amici (visibile sulla sinistra quando vi sono amici) ogni volta che viene aggiunto un nuovo utente alla propria cerchia di amici.

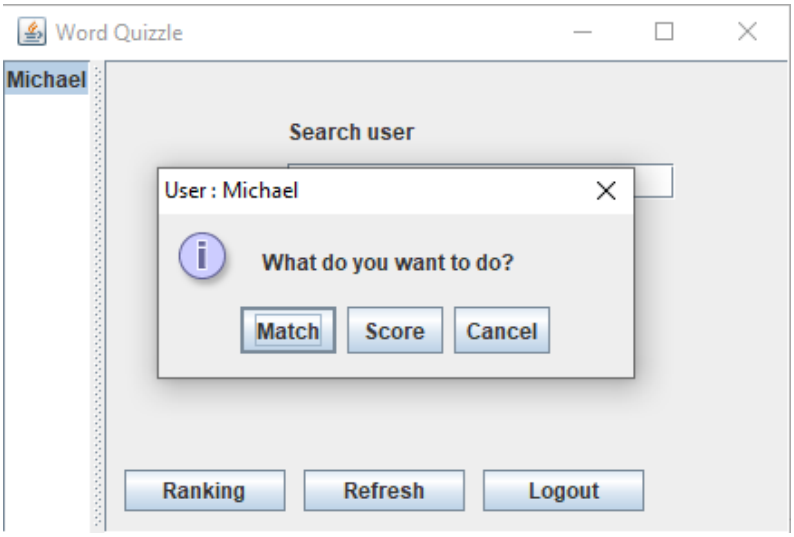
Aggiungiamo quindi un utente (precedentemente creato seguendo la procedura sopra esplicata):



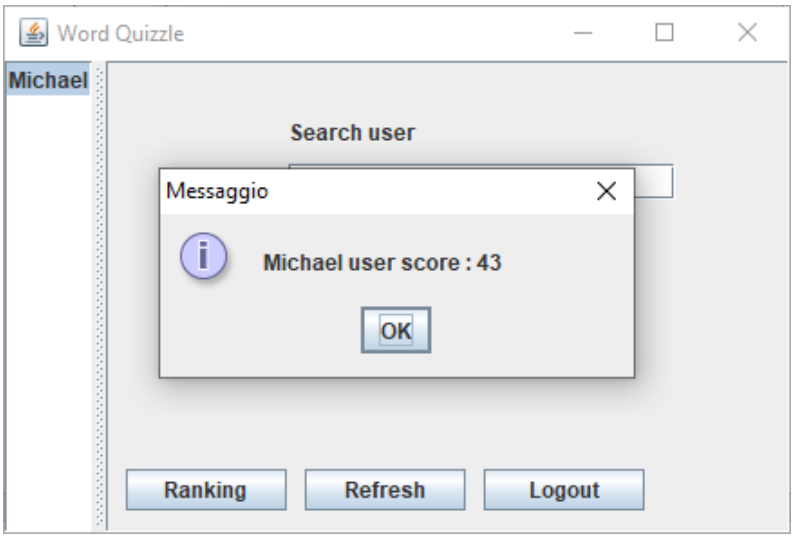
Chiuso il messaggio di successo, clicchiamo su refresh ed espandiamo la tendina sulla sinistra per mostrare la lista amici:



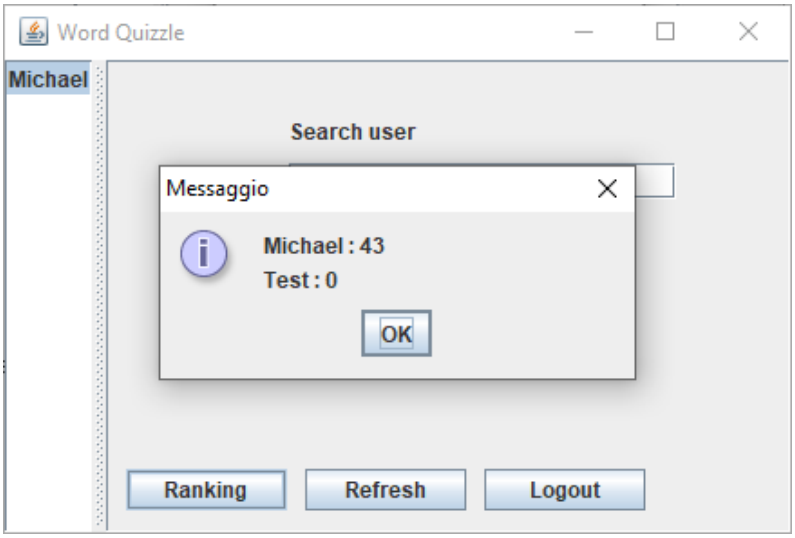
È possibili visualizzare il punteggio degli utenti in due modi: quello precedentemente illustrato, oppure, se l’utente è vostro amico, cliccando direttamente su di esso:



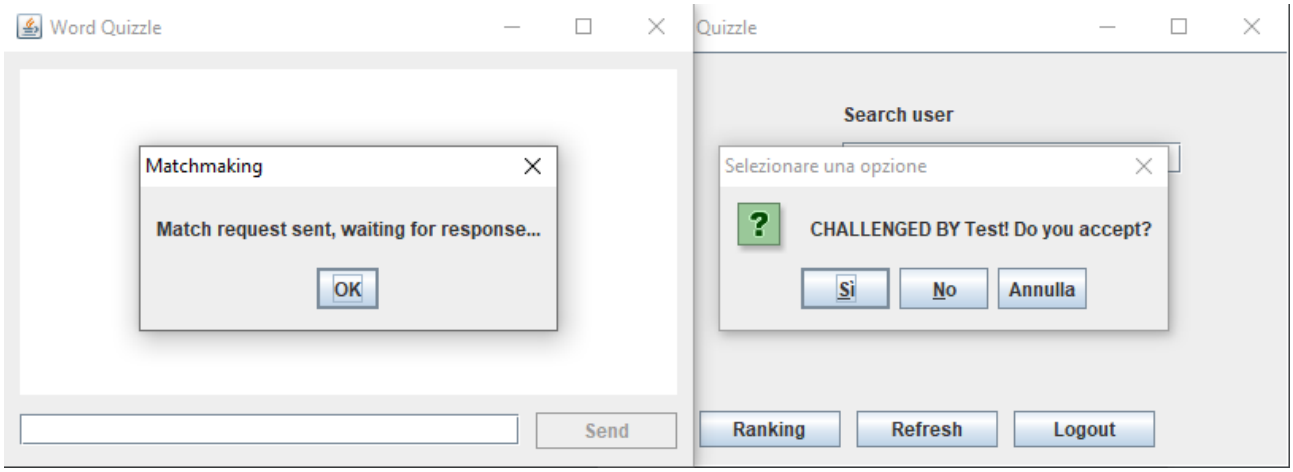
Il punteggio dell’utente viene visualizzato nel seguente modo:



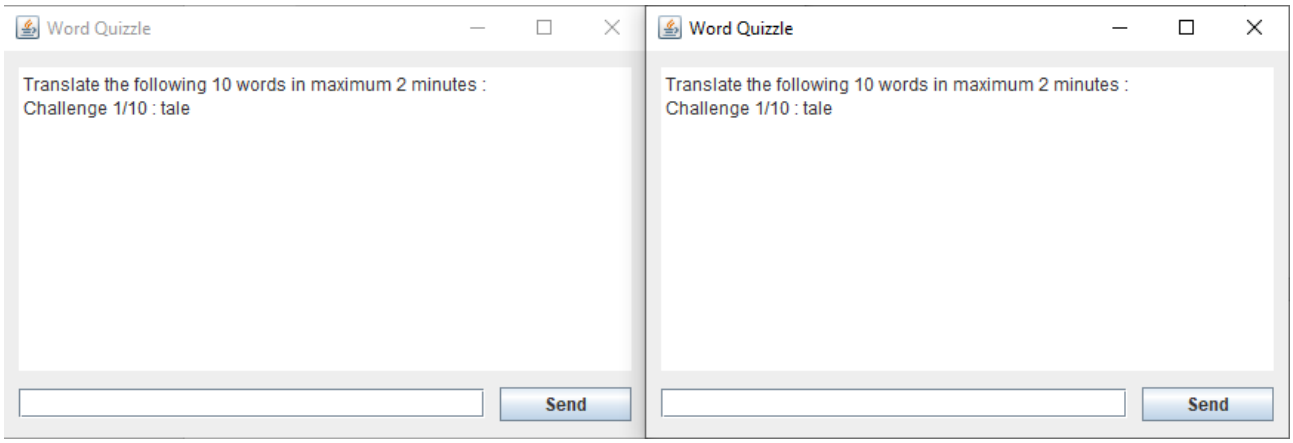
Proviamo adesso a visualizzare la classifica cliccando sul tasto Ranking:



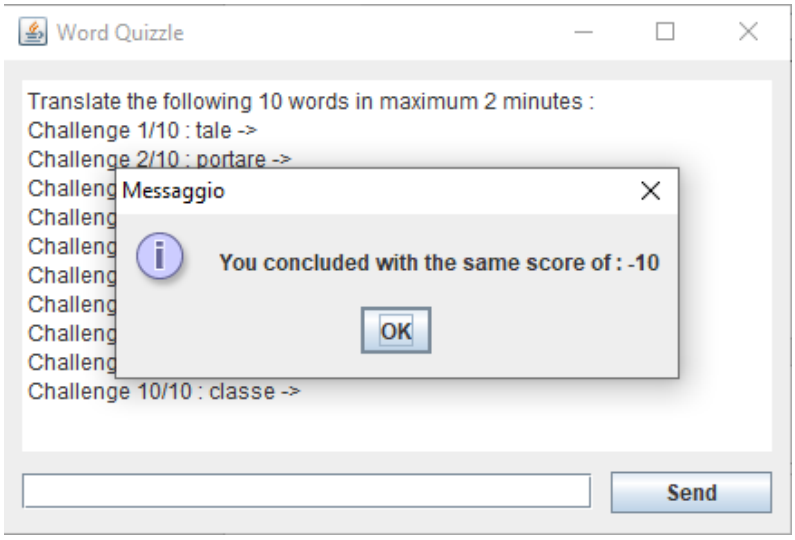
Proviamo a sfidare l’utente cliccando su di esso e scegliendo l’opziona *Match*:



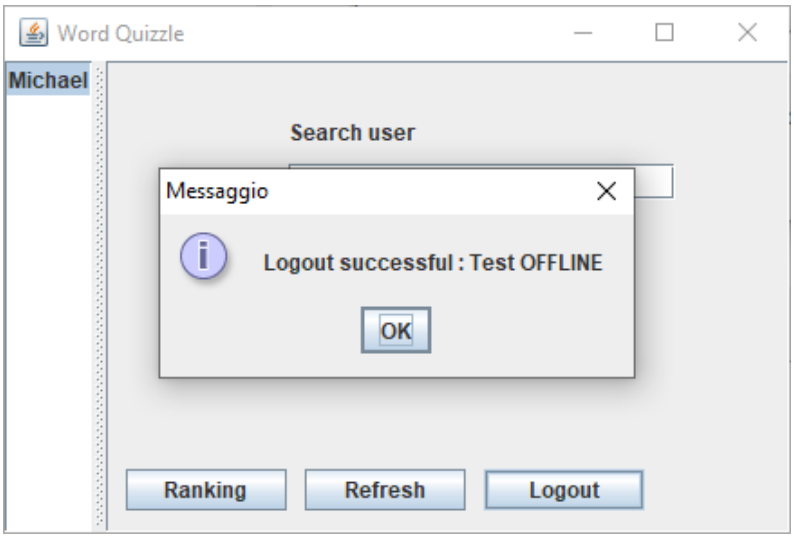
Accettando la sfida ognuno dei due utenti si troverà nell’interfaccia dedicata alla sfida:



Terminata la sfida, gli utenti ricevono il risultato della sfida e vengono riportati all’interfaccia di invio comandi al server:



Effettuiamo quindi il logout; a seguito dell’operazione verremo riportati alla schermata iniziale di registrazione e accesso.



Esecuzione

Per rendere operativa l’architettura sviluppata è necessario eseguire innanzitutto la componente Server, per farlo è sufficiente eseguire *WqServer.java* senza specificare alcun argomento. La conseguenza di tale esecuzione è l’avvio dei server che resterà in ascosto di nuove richieste di registrazione (RMI) sulla porta *8080* e sulla porta *9999* in attesa di connessioni TCP degli utenti. Successivamente all’avvio del Server sarà possibile connettersi al servizio eseguendo quante istanze si vuole di *WqClientGUI.java*.

Appendice A – Informazioni

Vengono di seguito riportate varie informazioni di cui prendere visione:

- Sistema operativo di riferimento: Windows 10 64 bit;
- Ambiente di sviluppo: Eclipse, inherited encoding Cp1252;
- Numero di parole possibili: 1160
- Encoding del file di parole: Cp1252;

Un punto su cui fare particolare attenzione è infatti l’encoding del file; Per visualizzare correttamente a schermo le parole accentate è necessario che l’encoding del file sia lo stesso utilizzato dal terminale / ide utilizzato per la visualizzazione. In *Eclipse* è possibile impostare l’encoding nel seguente modo:

Run -> Run configuration -> Common -> Encoding -> Cp1252

Tale encoding è impostato di default su Windows.

Se il file di parole dovesse essere erroneamente cancellato, all’avvio del Server, il file stesso verrà automaticamente scaricato da una repository su *Git Hub*; anche in questo caso l’encoding sarà Cp1252.

Appendice B – Match

Se un giocatore A è in partita con un giocatore B e contemporaneamente un giocatore C sfida uno di questi due; il giocatore C dovrà attendere la scadenza della richiesta di sfida.

Ulteriore ragione per cui è stata scelta la risposta al match utilizzando il protocollo UDP è permettere all’thread gestore della sfida di inserire il canale del giocatore sfidato nel proprio selettore unicamente nel momento in cui la sfida viene accettata e non alla creazione dell’handler stesso.