Word Quizzle

Laboratorio di Reti, Università di Pisa

Matteo Montalbetti

A.A 2019/2020

Introduzione

WordQuizzle è un'applicazione client-server che permette ai propri utenti di sfidarsi in gare di traduzioni dall'italiano all'inglese. Inoltre, il sistema mantiene una rete sociale tra essi, in modo tale che gli iscritti al servizio possano condividere amicizie e punteggi. La realizzazione del progetto ha come obiettivo l'applicazione pratica di diverse tecnologie *Java* viste durante il corso di *Laboratorio di Reti*. Tra le tante, sono state utilizzate:

- Comunicazione tramite protocollo **TCP**;
- Comunicazione tramite protocollo **UDP**;
- Comunicazione tramite la tecnologia **RMI**;
- Richieste **HTTP** (API fetching);
- **Multithreading**, **sincronizzazione** e **concurrent collections**;
- Interazione con i **file**, Java NIO;
- **Serializzazione** e **Deserializzazione** (JSON);
- Interfaccia grafica con **Swing**;
- e altre...

Vedremo inizialmente l'aspetto architetturale del progetto: com'è stata organizzata la comunicazione tra il **Client** e il **Server**? Quali classi entrano in gioco nella realizzazione di tutte le funzionalità? Come sono collegate tra loro? Cercheremo quindi di costruirci un'idea quanto più solida di tutto il funzionamento dell'applicazione. Successivamente ci addentreremo nella visione dei vari thread e delle strutture dati utilizzate, cercando di mettere in luce l'aspetto concorrente del software. Seguirà poi un breve escursus di tutti

i moduli implementati, nonchè dei vari test definiti, per poi concludere con alcune linee guida utili alla corretta esecuzione dell'applicazione.

Durante l'implementazione dell'intero progetto ho cercato di applicare alcuni concetti appresi durante il percorso di studi (e non). Primo fra tutti, ho deciso di avere un approccio prettamente **modulare**, in modo da favorire la **riusabilità**, **manutenibilità** e **leggibilità** del codice (nonchè l'adozione del **single-responsibility principle**). Il progetto è stato in primo luogo diviso in tre **packages**, ognuno dei quali popolato da svariate classi specializzate:

- ***front_end_src***: contiene tutte le classi utilizzate dal **client**;
- ***back_end_src***: contiene tutte le classi utilizzate dal **server**;
- ***common_src***: contiene tutte le classi utilizzate sia dal client che dal server;

Ho pensato che una gestione di questo tipo possa essere comoda in un possibile caso di reale utilizzo del progetto: basterà *spostare* sul server fisico e sui vari client i packages adatti.

Inoltre, ho voluto applicare alcuni tra i *design patterns* visti durante il corso di *Ingegneria del Software*: il pattern **Singleton**, utilizzato per recuperare l'oggetto *UsersRegister* (dopo vedremo cos'è), e il pattern **Builder**, utilizzato per l'implementazione di alcuni componenti grafici custom.

Come d'accordo con il docente, il progetto verrà interamente presentato nella sua versione *"con client a linea di comando"*, concentrandoci inizialmente sulle funzionalità tecniche, di rete, e sull'architettura generale, piuttosto che sull'aspetto grafico. A fine relazione, l'ultimo capitolo sarà dedicato interamente a una sintetica presentazione della parte grafica, realizzata come ultimo step prima della consegna.

Prima di iniziare, ricordo, in caso di ogni dubbio, che tutto il codice è stato interamente commentato in modo quanto più esaustivo.

Contents

1 Architettura e funzionamento generale **5**

2 Threads, strutture dati e concorrenza **17**

3 La sfida **25**

4 Le classi implementate **29**

4.1 Common Package 29

4.1.1 CommonUtilities.java 29

4.1.2 UsersRegisterInterface 31

4.1.3 Exceptions 31

4.1.4 config.properties 31

4.2 Frontend Package 31

4.2.1 Client 32

4.2.2 ClientUtilities 32

4.2.3 UDPReceiver 32

4.2.4 Exceptions 33

4.3 Backend Package 33

4.3.1 Server 33

4.3.2 ServerUtilities 33

4.3.3 User 34

4.3.4 UsersRegister 34

4.3.5 RequestManager, APIFetcher, MatchManager, StructManager 36

4.3.6 Exceptions 36

4.3.7 config.properties 36

5 Compilazione, esecuzione e test **37**

6 L'interfaccia grafica **40**

Chapter 1

Architettura e funzionamento generale

Il cuore del sistema è la comunicazione tra la classe **Client** e la classe **Server**. La maggior parte di questa ha luogo su di una connessione **TCP**, che lega i due durante tutta la sessione di un utente. Quando si parla di architetture client-server la prima cosa da considerare, a mio avviso, è la strategia che il server dovrà attuare per gestire le molteplici connessioni verso i vari client. Le possibili strade sono essenzialmente due: utilizzare un approccio multithread o effettuare il **multiplexing dei canali** (tramite Java NIO); ho preferito utilizzare il secondo approccio, poichè migliore per **flessibilità**, **scalabilità** ed **efficienza**.

Lato client la questione è invece molto più semplice: all'avvio si tenta di instaurare una connessione con il server aprendo una socket verso esso da utilizzare durante tutta la sessione (in realtà una *SocketChannel*, per essere precisi. Per semplicità però, la chiameremo in questo capitolo semplicemente *socket* oppure, ancora meglio, *channel*). Attenzione però! È **necessario** loggarsi (eventualmente dopo essersi registrati) per poter utilizzare le funzionalità messe a disposizione dal server.

Una volta *gettato* lo scheletro, capire l'architettura e il funzionamento del sistema richiede anzitutto porsi una semplice, ma essenziale, domanda: che cosa deve fare il software? Quali funzionalità devono essere implementate? Essenzialmente sono due gli aspetti da realizzare: anzitutto *WordQuizzle* è un sistema di sfide di traduzione; allo stesso tempo però, deve permettere la

gestione di una rete sociale tra gli utenti, i quali si registreranno al servizio, scambieranno amicizie, condivideranno classifiche, ecc...

Credo che sia utile per adesso separare questi due scenari, analizzando in primo luogo l'aspetto sociale del software e la sua realizzazione, poichè più semplice. Il fulcro di un qualsiasi social network (grande o piccolo che esso sia) è sicuramente il **database**: per mantenere e gestire una rete sociale tra utenti è necessario avere una struttura che permetta il salvataggio e la gestione dei dati di tutti coloro che usufruiranno del servizio e ne faranno quindi in qualche modo parte. Nel mio progetto, questa struttura è astratta da un oggetto di tipo **UsersRegister** il quale, come unico attributo, incapsula una **ConcurrentHashMap**: sarà questa la struttura dati che realmente conterrà le informazioni (non mi soffermo sulle motivazioni di questa scelta, ne parleremo nel prossimo capitolo). Manca però il protagonista di tutto ciò: l'utente! Ho deciso di creare una classe che lo rappresentasse, chiamata banalmente **User**; a questo punto è chiaro che la mappa prima citata conterrà tanti oggetti di tipo *User* (in realtà, essendo una mappa, è più corretto indicare il formato di una entry, ovvero una coppia chiave-valore, nel nostro caso composta da *String id - User user*). Ma perchè non ho utilizzato direttamente la *ConcurrentHashMap*, piuttosto che incapsularla in un oggetto? I motivi sono vari, primo tra tutti la possibilità di avere in un'unica classe la struttura e tutti i metodi utili alla gestione della stessa. Inoltre, tramite questo approccio, riesco ad implementare il pattern **Singleton**, facendo così in modo che ci sia un'unica istanza della classe (e quindi anche del *database*) per tutto il codice del server, recuperabile da tutti i thread (in maniera **sincronizzata**) nel momento del bisogno.

All'avvio, il *Server* procederà ad istanziare per la prima volta l'oggetto *UsersRegister*, cosa che non sempre comporta una **new**: se il server è stato già aperto in precedenza, sarà presente sul file system un file **.json**, che verrà deserializzato. Difatti, ad ogni modifica della struttura dati, questa viene serializzata e salvata all'interno del file appena citato, così da riuscire a persistere i dati (sebbene fino ad ora io abbia utilizzato il termine *database*, ciò a cui riferivo era in realtà un semplice file *.json*). Per il meccanismo di serializzazione ho utilizzato la libreria esterna **Gson** e i **FileChannel**. Ho scelto la libreria di Google per la sua semplicità ed intuitività, e i *FileChannel* per la loro efficienza, nonchè comodità in termini di **sincronizzazione** (sono difatti thread-safe per loro natura, ma, come già detto, aspetti di questa

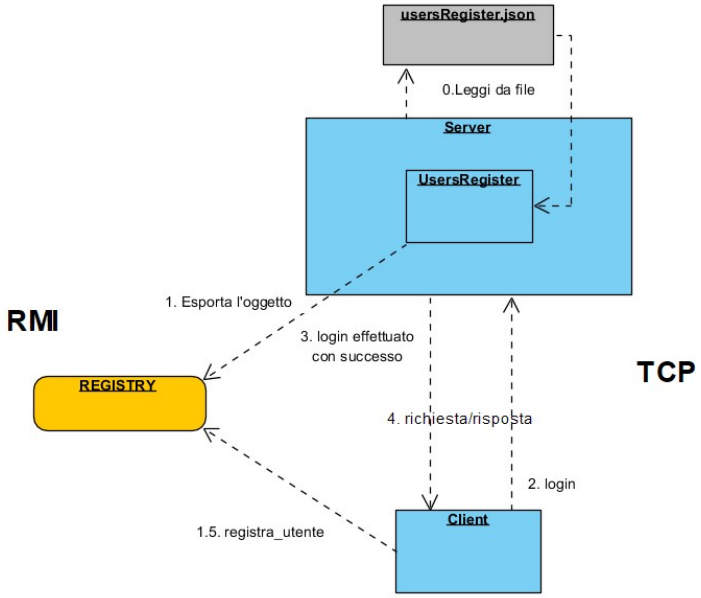
natura sono discussi nel prossimo capitolo).

Bene, a questo punto la rete sociale può prendere vita: il server riceverà i vari comandi dai client, modificando, comunicando, aggiornando e salvando le informazioni degli utenti su essi loggati.

Sebbene tutta la comunicazione (o quasi) abbia luogo sulla socket *TCP* nominata precedentemente, l'operazione di registrazione è stata implementata tramite tecnologia **RMI**. Quando il server viene messo in esecuzione, dopo aver instanziato l'oggetto *UsersRegister*, procede all'**esportazione** di esso sul **registro**, così da renderlo accessibile da remoto. I client, nel momento in cui vorranno eseguire una registrazione, utilizzeranno lo **stub** locale, recuperato dal medesimo registro, per invocare il metodo **registerNewUser**. L'invocazione di esso sarà poi ricevuta dallo **skeleton** del server, e arriverà infine all'oggetto vero e proprio. Tengo a precisare una scelta progettuale fatta, lato client: sebbene il recupero dello stub possa essere fatto un'unica volta ad inizio esecuzione, ho preferito ripetere tutta l'operazione di accesso all'oggetto remoto ogni qual volta il client chiede di registrare un nuovo utente. Il motivo è semplice: l'operazione di registrazione è solitamente richiesta poche volte, soprattutto se da uno stesso client. Molto spesso è richiesta la prima volta che l'utente apre quel client, dopo di che l'operazione principale diventa la *login*. Pertanto, in questo modo, evito inutili connessioni verso il server in caso l'utente non abbia intenzione di registrarsi (sebbene l'obiettivo dell'*RMI* sia proprio quello di astrarre tutta la parte di rete, per far sì che si possa continuare ad usufruire del paradigma ad oggetti, la gestione della connesione sottostante non è affatto banale).

Per quanto riguarda la gestione della rete sociale, tutto qua! Una volta eseguito il login, client e server continueranno a comunicare fin quando vorranno tramite un semplicissimo protocollo richiesta/risposta (per quanto riguarda la comunicazione a livello tecnico, quindi a livello di codice, si leggano, al capitolo 4, le classi *CommonUtilities* e *Server*: ho deciso di implementare un meccanismo di messaggistica che prevede l'invio della dimensione del messaggio prima del messaggio stesso, così da evitare inutili allocazioni di memoria per i buffer di lettura).

Uno schema di ciò che abbiamo appena discusso:



Occupiamoci adesso dell'altro aspetto da realizzare: il sistema di sfide. Difatti, tra i vari comandi disponibili, la richiesta di sfidare un altro utente dà vita ad un meccanismo ben più complicato di quello visto fino ad ora. Ci concentreremo su uno scenario ben preciso, in cui un utente, *idA*, sfida un suo amico, *idB*. Fino ad ora, le uniche componenti in gioco erano il **Server** e il **Client** (e **UsersRegister**, se vogliamo); è ora necessario introdurne di nuove, che saranno di supporto all'implementazione della funzionalità di sfida. Partiamo dal client: la specifica del progetto richiede che il server inoltri una richiesta di sfida ad un dato utente tramite la comunicazione **UDP**. È allora necessario che ogni client abbia a sè associato un componente che riceva, in maniera continuativa, questi messaggi, i quali possono arrivare in un qualsiasi momento. Il componente in questione prende il nome di **UDPReceiver**, ed è un **runnable** lanciato dal client al momento del suo avvio.

Spostiamoci adesso sul server. Prima di continuare, è importante avere in mente una questione; il server tiene traccia di tutti gli utenti online: nel momento in cui, nel selettore, viene ricevuto un comando di *login*, il server procede all'inserimento in una mappa chiave valore della coppia *idSocketChannel* associati all'utente e al client dal quale ha ricevuto il co-

mando. In questo modo saprà sempre quale utente è collegato al client dal quale ha ricevuto un determinato messaggio (feature che permette agli utenti l'utilizzo di comandi con sintassi del tipo: *mostra_punteggio* piuttosto che *mostra_punteggio mioId*). Chiarita questa piccola parentesi, torniamo a noi: il server sta per ricevere il messaggio *sfida idB*, inviato da *idA*.

Al momento della ricezione del messaggio, il server procede all'esecuzione di un nuovo thread (un *runnable*, per la precisione), **RequestManager**, che si occuperà della fase di setup della sfida tra *idA* e *idB* (verrà quindi lanciato un thread *RequestManager* per ogni sfida tra due utenti; la gestione di tutti questi thread è responsabilità di un **pool**). Prima di eseguire il setup, però, *RequestManager* procederà ad inoltrare, tramite protocollo UDP, la sfida al client destinatario. Sorge però un quesito: a chi inoltriamo il messaggio? È necessario che il server, o meglio *RequestManager* conosca l'indirizzo (inteso come coppia *IP* e *porta*) della socket sulla quale l'*UDPReceiver* interessato è in ascolto! Il problema trova soluzione nel seguente meccanismo: al momento dell'avvio, il client, prima di lanciare *UDPReceiver*, procede all'apertura della socket UDP sulla medesima porta della socket TCP con la quale comunica con il server (ciò è possibile proprio grazie alla differenza di protocollo). Quindi, se *idB* è online, per il server è molto semplice recuperare questa porta: va a vedere nella mappa di cui abbiamo parlato prima i dati della *SocketChannel* TCP associata ad *idB* (salvata al momento del *login*), e ne recupera la porta, comunicandola a *RequestManager* (verrà passata come parametro nel costruttore). Proprio a causa di questo meccanismo, è **necessario** che l'utente sfidato sia online! In caso contrario, verrà notificato ad *idA* l'impossibilità di inoltrare la sfida e quindi di giocare.

Bene, a questo punto *RequestManager* inoltra la richiesta all'utente, inviando sulla socket UDP il messaggio ***add idA***. L'*UDPReceiver*, che leggerà il messaggio, capirà che è arrivata una nuova richiesta da parte di *idA* e procederà all'aggiornamento, **sincronizzato**, della coda di sfide mantenuta nel client (nel prossimo capitolo approfondiremo, per ora basti sapere che esiste questa struttura dati e che, essendo condivisa dal main thread del client e da *UDPReceiver*, è acceduta tramite sincronizzazione). Nella struttura dati non solo viene salvato l'id dell'utente sfidante, ma anche i dati della socket UDP del *RequestManager* associato alla richiesta della corrispondente sfida, i quali saranno poi utili per l'eventuale invio dell'accettazione di essa.

Ora, possono succedere due cose:

- L'utente *idB* accetta la richiesta di sfida di *idA*, prima che, lato server, scada il timeout. Per farlo, *idB* inserisce il comando ***mostra_sfide*** (il quale stampa la coda delle richieste) ed indica il nome dell'utente del quale accettare la richiesta, ovvero *idA*. Una piccola parentesi: il timeout è impostato con un parametro predefinito, recuperato da un file di configurazione **.properties**. A fine capitolo ci soffermeremo con più attenzione su questa possiblità che *Java* offre. Tornando a noi, *idB* ha accettato la sfida. Ciò che succede è che il thread del client procede all'invio del messaggio ***accepted*** verso il *RequestManager* corrispondente (si ricordi che è attivo un *RequestManager* per ogni sfida, non serve quindi indicare quale richiesta sia stata accettata, poichè unica per quel *RequestManager*). In realtà, la comunicazione UDP non è qui conclusa: ho voluto implementare un meccanismo di ulteriore conferma della validità della sfida. Per adesso non andremo a fondo, lasciando questo aspetto al punto successivo (il caso in cui *idB* non accetta la sfida o comunque scade timeout). In ogni caso, dopo aver ricevuto il messaggio *accepted*, il *RequestManager* procede all'invio di un ulteriore **ACK** per confermare la validità della sfida appena accettata. Bene, una volta ricevuto questo *ACK* dal client, *idB*, e poi anche *idA* sapranno che la sfida si farà.

- L'utente *idB* non accetta la richiesta di sfida di *idA*. In tal caso, lato server, in *RequestManager*, scatta il timeout di cui abbiamo già parlato nel punto precedente. Il timeout è collegato alla socket UDP di *RequestManager*, la quale si mette in attesa di una risposta di accettazione subito dopo l'invio della richiesta: se questa non arriverà entro un certo lasso di tempo, la sfida sarà considerata rifiutata. In tal caso, *RequestManager* stesso si occuperà di notificare *idA* sulla socket TCP, il quale era in attesa di un esito della richiesta. È però necessario che anche *idB*, o meglio il suo *UDPReceiver*, venga notificato dello scadere della richiesta, così che questa possa essere tolta dalla coda, per evitare che *idB* la accetti in un successivo momento. Per fare ciò, *RequestManager* manderà un ulteriore messaggio UDP all'*UDPReceiver* di *idB*, della forma ***remove idA***, così che la richiesta venga rimossa. Nasce però qui un bel problema! Ho deciso di evitare la gestione di alcun timeout lato client, per ragioni di **sicurezza**, lasciando così la responsabilità al solo server (per essere precisi, *RequestManager*). Questo

implica però la possiblità che si presenti un determinato scenario, per niente piacevole, che andremo ora a esaminare. Anzitutto si ricorda che gli accessi alla coda di richieste sono coordinati tramite la keyword **synchronized**, poichè la struttura è condivisa dal main thread di *Client* (il quale ci accede nel momento in cui vuole vedere le sfide ed eventualmente accettarle) e dal runnable *UDPReceiver* (il quale aggiunge e rimuove le richieste in base alle comunicazioni che arrivano dal server). Questo mi assicura che non ci sia **incosistenza**, ma non esclude che possa succedere ciò: nello stesso momento in cui l'utente richiede di accettare una sfida, e quindi invia un messaggio UDP a *RequestManager* del tipo ***accepted*** (nel main thread di *Client*), *UDPReceiver* ha ricevuto la notifica di timeout della richiesta. Se nella race-condition il thread del *Client* arriva prima, la risposta alla richiesta viene inviata prima che essa possa essere eliminata dalla coda. Ma una volta che *idB* accetta la richiesta, esso si aspetterà di giocare (non solo se lo aspetterà l'utente a livello di modello mentale, ma il codice stesso del client comporterà una situazione di lettura da socket, operazione bloccante). Questo però non deve accadere, perchè nel momento in cui lato server scatta il timeout, *RequestManager* termina la sua esecuzione dopo l'invio della notifica del timeout stesso, e quindi non leggerà mai il messaggio di accettazione, lasciando l'utente *idB* in un'attesa infinita. Ho voluto allora implementare un ulteriore meccanismo di verifica (lo stesso nominato nel punto precedente), ispirato in un certo senso all'**hand-shake a tre vie**, visto durante il corso di teoria. Nel momento in cui il client decide di accettare una richiesta, setta un apposito flag a *false*. Questo flag è condiviso con *UDPReceiver* (e per questo è stato dichiarato di tipo **volatile**), che intanto continua a ricevere messaggi UDP. Di base, quello che succede è che *RequestManager*, dopo aver ricevuto la notifica di accettazione della sfida, nel caso in cui **non sia scattato il timeout**, invia subito un **ACK** di riconferma ad *UDPReceiver*, il quale, una volta ricevuto il messaggio, setterà il medesimo flag (modificato precedentemente dal client) a *true* (lo riconoscerà poichè della forma, in questo caso, **starting idA**) . Il client, dopo una sleep di $500ms$, pegno che bisogna pagare per esser certi che arrivi la conferma di validità, ricontrollerà il flag (che lui aveva inizialmente impostato a false): se è rimasto false anche dopo l'invio del messaggio di accettazione, allora è *incappato* proprio nella situazione prima ipotizzata, la sfida non è più valida; se è invece diventato true

allora è andato tutto liscio, si può giocare!

Chiaramente in quest'ultimo caso c'è poco da fare, la sfida non avrà luogo e i due client torneranno alla loro comunicazione TCP con il server. Vediamo invece che succede successivamente al primo punto. Dopo aver inviato l'ulteriore *ACK* ad *idB*, *RequestManager* può ora procedere con l'esecuzione della fase di setup della sfida. All'inizio, quando abbiamo parlato del *Server* e del suo avvio, ho omesso un particolare, che credo sia più sensato indicare adesso: sul file system è memorizzato un file *.txt* contenente circa 1000 parole in lingua italiana. Questo file viene letto dal server all'avvio, il quale crea da esso (con le parole in esso) una struttura dati (un semplicissimo *ArrayList* di stringhe).

Nel momento in cui *RequestManager* arriva al punto di dover implementare il setup della sfida, la prima cosa che fa è scegliere da questa struttura *numberOfWords* parole (*numberOfWords* è una costante che specifica il numero di parole di cui una sfida è composta, anch'essa viene recuperata da un file di configurazione, ne parleremo dopo). Una volta scelte le parole, è necessario recuperare dal servizio *API* indicato dai docenti le traduzioni di queste. Ho deciso di parallelizzare quest'operazione: *RequestManager* procederà al lancio di *numberOfWords* **APIFetcher**, runnable creato appositamente per eseguire questo task. Ciò che l'*APIFetcher* fa è una semplice richiesta al server **http** dell'API, recuperando la traduzione (eventualmente le traduzioni, nel caso ce ne dovessero essere più d'una) della parola italiana ad esso affidata. Ad ogni fetch, ogni componente andrà a popolare un *ArrayList* di stringhe con le proprie traduzioni. Tutti gli *ArrayList* sui quali lavorano gli *APIFetcher* sono in realtà elementi di un ulteriore *ArrayList*: proprio per questo motivo non c'è bisogno di sincronizzare alcuna struttura; ogni runnable inserisce le parole su un singolo *ArrayList*, disgiunto dagli altri, ognuno dei quali è già stato creato precedentemente e quindi già ben posizionato all'interno della struttura più esterna. Quando tutti gli *APIFetcher* avranno terminato la loro operazione, proseguirà l'esecuzione di *RequestManager* (che intanto si era messo in attesa della terminazione dei vari *fetcher*), il quale si ritroverà tra le mani la struttura dati appena citata con tutte le traduzioni necessarie. La cosa non è però così semplice: potrebbe benissimo accadere che la richiesta *http* fallisca! Dobbiamo coprire questa situazione, comunicando ai due utenti che la sfida non può avere luogo! Chiaramente, chi ha la possibilità di incontrare problemi durante le richieste, sono proprio

gli *APIFetcher*: in questo caso, colui che ha incontrato il problema, setta a *true* un flag condiviso tra *RequestManager* e tutti gli *APIFetcher* (dichiarato **volatile** per evitare inconsistenze). Così facendo, dopo aver aspettato la terminazione degli *APIFetcher*, *RequestManager* potrà fare un check sul flag di cui sopra, così da capire se si sono verificati errori durante il fetching. In tal caso, si procederà a notificare (tramite comunicazione TCP) i due client dell'accaduto e alla cancellazione della sfida. Piccola nota: gli *APIFetcher* si accorgono dell'errore controllando lo status code della risposta *http*. Se questo è maggiore di 299 allora capiranno che c'è stato un problema.

Supponiamo però che non ci siano stati problemi durante il recupero delle traduzioni, che succede ora? *RequestManager* procede a lanciare un ulteriore (ed ultimo!) thread, **MatchManager**. Questo, sarà il responsabile della gestione della sfida tra i due utenti, e provvederà a comunicare con essi tramite lo stesso channel con il quale sono collegati al server (i channel dei due utenti vengono infatti passati da thread in thread fin dall'esecuzione di *RequestManager*). Ora, così come la classe *Server*, anche *MatchManager* utilizza un **selettore** per comunicare con i client, il che implica un piccolo problema da risolvere: nel momento in cui il client invia una traduzione, entrambi i selettori notificheranno un channel pronto (difatti il channel non è mai stato deregistrato dal primo selettore) e potrebbe accadere che il selettore di *Server* "rubi" il messaggio a *MatchManager*. Per risolvere questo intoppo, ci aiutiamo con una struttura dati che tiene traccia degli utenti in gioco, **inGameUsers**. Accedendo ad essa in maniera sincronizzata, *MatchManager* inserisce al suo interno *idA* e *idB*. Quando il *Server* proverà a interagire con il channel di uno dei due utenti (e lo farà, poichè il channel sarà effettivamente pronto non appena l'utente invierà una traduzione) controllerà prima l'id dell'utente associato al channel rilevato pronto: se questo id è contenuto nella struttura *inGameUsers*, allora capirà che non è il caso di interferire, saltando quel ciclo di selettore.

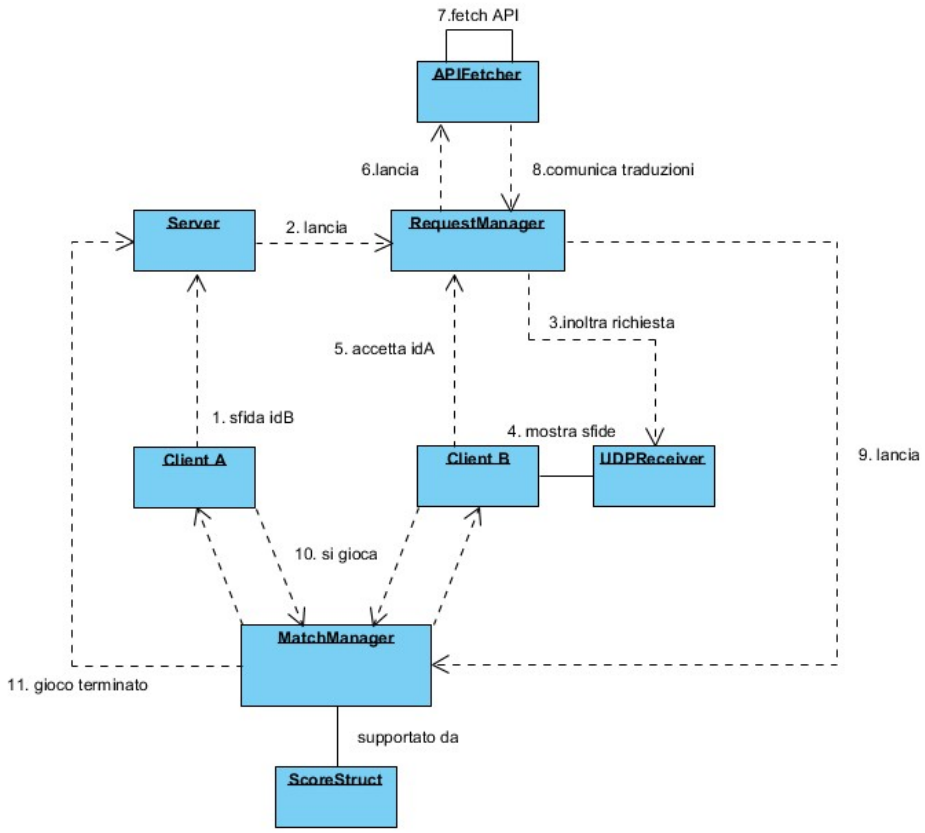
Ottimo! I due client possono ora comunicare le proprie traduzioni a *MatchManager*, il quale procederà ad implementare tutto il meccanismo della sfida. Per adesso non soffermiamoci su come questo sia stato realizzato, l'idea di base è comunque quella di avere due indici, uno per ogni utente, che tengono traccia della domanda corrente per essi (ne parleremo più a fondo nel capitolo 3). È però doveroso dedicare un secondo alla gestione del **timeout** della sfida: quando *RequestManager* fa partire *MatchManager*, lo fa

attraverso un **Executor**. Questo mi ha permesso, tramite l'utilizzo della keyword **Future** di poter settare un timeout alla terminazione del nuovo runnable (*MatchManager*). Ora, solitamente la keyword *Future* viene utilizzata insieme ad un *Callable*, per dare un limite di tempo all'attesa di un qualche dato di ritorno. In questo caso ho però deciso di prendere una strada leggermente diversa, continuando ad utilizzare un *Runnable*, poichè non avevo nessun dato da restituire. Ciò è stato possibile grazie all'utilizzo di **Future<?>**. Effettivamente, non è forse l'approccio più corretto a livello teorico; lo ho però preferito per questioni di leggibilità e progettuali. In ogni caso, utilizzare un *Callable* non cambierebbe di molto la situazione, potremmo per esempio far tornare una semplice stringa di debug che indichi l'esito della sfida. Tornando a noi, nel momento in cui scade il timeout, *RequestManager* chiamerà un metodo offerto da *MatchManager*, il quale comporta il *flip* di un booleano appositamente mantenuto per l'eventuale notifica di timeout: il *MatchManager*, prima di inviare la successiva traduzione (oppure di aggiornare i punti, ed in altri "strategici" punti) controllerà il valore di questo flag, così da eventualmente far terminare il match. I due client verranno a questo punto notificati e la sfida terminerà. Purtroppo, essendo il client a *linea di comando*, non è possibile far sì che l'interfaccia di ogni utente reagisca a ciò in tempo reale, poichè bloccata in attesa di input della prossima traduzione (difatti, i client visualizzeranno la notifica del timeout solo dopo l'invio della traduzione corrente, eventualmente notificati della validità o meno della traduzione appena inviata). Purtroppo non si può far molto, si noti però che la cosa importante è che sia il server a reagire in tempo reale al timeout, per poter dedurre quando terminare la partita e quindi aggiornare eventuali punteggi all'interno del database (la situazione viene gestita molto meglio nella versione del client con interfaccia grafica, che però lasceremo come ultimo argomento di questa relazione).

Durante tutta la sfida, *MatchManager* deve tenere traccia, per entrambi gli utenti, di alcune statistiche per ogni giocatore, tra cui: punti attuali, risposte corrette, risposte errate o solamente non date (nel caso di timeout, le risposte rimaste saranno conteggiate come non date). Inoltre queste statistiche vanno aggiornate in base all'evolversi del match. Viene allora in aiuto un ulteriore componente, **ScoreStruct** che incapsula in un unico oggetto tutte queste statistiche e i vari metodi per gestirli (l'utilizzo di questo modulo ha migliorato di molto la leggibilità di *MatchManager*).

Abbiamo concluso! Al termine del match, verranno aggiornati i punti totali dei due utenti in base all'esito di esso: chi vincerà guadagnerà un tot di punti (parametro arbitrariamente predefinito), che si andranno ad aggiungere ai punti già racimolati nel tempo (con altre sfide) dall'utente.

Come per la prima parte, allego qui sotto un semplice schema che riassume il funzionamento appena discusso.



Prima di proseguire con il prossimo capitolo, discutiamo la piccola parentesi precedentemente rimandata: in questo progetto sono presenti alcuni valori, o per meglio dire **parametri**, che ne regolano e caratterizzano il funzionamento. Tra questi ci sono sicuramente, in primo piano, i vari *settaggi* del match (si pensi al numero di parole che i client devono tradurre, il tempo disponibile per il match e altri...). I file **.properties** sono, almeno per *Java*, perfetti per questo tipo di utilizzo: storage di parametri, valori di config-

urazione e perchè no anche piccoli dati (rigorosamente di tipo primitivo). Inoltre, sono offerte dal linguaggio stesso una serie di API che ne permettono l'interazione in maniera molto semplice ed efficace, attraverso la classe **Properties** (per adesso ne vedremo il funzionamento concettuale, per vederne i codici consultare i successivi capitoli o i file sorgenti). L'idea è quella di scrivere all'interno del file delle coppie chiave valore, del tipo:

$$number\_of\_words = 10$$

$$udp\_timeout = 30$$

Tramite codice *Java*, attraverso un *input stream* (recuperato nel mio caso da un *FileChannel*, scelto per la sua efficienza), un oggetto di tipo *Properties* riesce a ricostruire una serie di coppie chiave-valore coerenti con i parametri salvati all'interno del file, rendendo così semplice la lettura di essi.

Durante tutto questo capitolo ho volutamente evitato di inserire alcun codice. Ho infatti preferito, per questa prima parte, provare a comunicare un modello concettuale del funzionamento di tutta l'applicazione, così che sia più semplice nel prossimo capitolo passare da un punto all'altro del progetto (questa volta ispezionando alcuni pezzi di codice) senza dover contestualizzare granchè. Ci concentreremo adesso sugli aspetti riguardanti il multithreading e la concorrenza.

Chapter 2

Threads, strutture dati e concorrenza

Arrivati a questo punto, dovrebbe essere più o meno chiaro il meccanismo di tutto il progetto, nonchè l'insieme di componenti che in esso agiscono. Cercheremo adesso di esaminare, da un punto di vista più tecnico, alcuni punti fondamentali riguardanti il multithreading e la programmazione concorrente.

Il primo oggetto nominato nel capitolo precedente è stato **UsersRegister**. Abbiamo detto che questo incapsula al suo interno la struttura dati contenente gli utenti registrati al servizio (con le relative informazioni) e tutti i metodi per gestire tale struttura. Abbiamo però anche accennato all'implementazione tramite pattern **Singleton** del costruttore: partiremo proprio da questo.

```
/**
 * L'unica istanza della classe {@link UsersRegister}.
 * Ho deciso di implementare un pattern Singleton poich
 * l'oggetto rappresenta un punto di riferimento per tutto il
    programma.
 * E' sensato avere un'unica istanza per tutti, che
   rappresenta il nostro
 * database di utenti.
 */
private static UsersRegister instance;

//Costruttore privato, cos  che sia inutilizzabile da altre
   classi.
```

```
private UsersRegister() {}

/**
 * Il metodo statico che sar  utilizzato per recuperare l'
   unica istanza
 * di {@link UsersRegister}.
 *
 * @return L'unica istanza di {@link UsersRegister}
 * @throws RemoteException Lanciata in caso di problemi con l
   'RMI.
 */
@SuppressWarnings("RedundantThrows")
public static synchronized UsersRegister getInstance() throws
    RemoteException {
    if (instance == null) {
        instance = new UsersRegister();
        try {
            instance.usersRegister = ServerUtilities.readJson
   ();
        } catch (IOException e) {
            //C'   stato qualche problema durante la lettura
   del file!
            e.printStackTrace();
        }
        if (instance.usersRegister == null) {
            instance.usersRegister = new ConcurrentHashMap
   <>();
        }
    }
    return instance;
}
```

Si noti l'utilizzo, molto importante, della keyword **synchronized** per il metodo **getInstance**. Potrebbe infatti capitare che due thread differenti tentino di recuperare l'istanza di *UsersRegister* allo stesso momento; supponiamo che questo non sia ancora stato creato la prima volta: senza la keyword *synchronized* potrebbe succedere che entrambi i thread tentino (ed effettivamente lo facciano) di istanziare entrambi un nuovo oggetto di tipo *UsersRegister*, situazione *proibita* dal pattern *Singleton*, poichè a questo punto non sarebbe unica l'istanza nel processo!

Come già detto, *UsersRegister* incapsula al suo interno la struttura che contiene tutti i dati degli utenti. Questa è una **ConcurrentHashMap**: una

versione della classica *HashMap* specializzata (da *Java* stesso) per gestire al meglio situazioni concorrenti. Riesce infatti a garantire un buon trade-off tra gestione della concorrenza e scalabilità del programma. Inoltre, è molto più efficiente delle strutture *Synchronized*, poichè evita di bloccarsi su tutta la mappa, concentrandosi piuttosto sui *buckets* (le entry, per capirci) sul quale i thread lavorano. Per esempio, supponendo che due thread stiano modificando il *value* di due chiavi diverse, nel caso si dovesse utilizzare la *SynchronizedHashMap* la lock verrebbe richiesta, da ogni thread, su tutta la struttura. Con la *ConcurrentHashMap*, invece, vengono acquisite due lock differenti, relative ai precisi due buckets in questione (e quindi i due thread possono lavorare in parallelo, aumentando l'efficienza del codice). Le due tipologie di strutture condividono però un piccolo punto, da tenere bene a mente: sono entrambe **conditionally thread-safe**. Questo vuol dire che possiamo assumere come *safe* le sole operazioni singole. Non abbiamo nessuna garanzia in caso di operazioni composte! A breve ne vederemo un esempio, esaminando il metodo **registerNewUser**; si sappia comunque che, anche in questo caso, la *ConcurrentHashMap* si rivela di maggiore aiuto, definendo alcune operazioni normalmente composte, come singoli comandi.

Vediamo allora proprio il metodo appena citato: **registerNewUser**. Una possibile race-condition può verificarsi nell'eventualità in cui due client tentino di registrarsi al servizio nello stesso momento. La tecnologia *RMI* gestisce le varie richieste tramite un pool di thread; ciò implica che, molto probabilmente, quelle due richieste verranno gestite da due thread differenti. È allora necessario coordinare due aspetti: l'aggiunta dei due utenti nella mappa, e la scrittura del file *.json* per persistere i dati (ogni qual volta c'è un cambiamento all'interno della struttura, essa verrà riscritta per salvare le modifiche).

Per quanto riguarda il primo punto, dobbiamo stare molto attenti a ciò che abbiamo già detto precedentemente. Pensiamo per un attimo a quali operazioni dovrà fare un thread per aggiungere un utente alla mappa: controlla anzitutto l'eventuale presenza dell'*id* richiesto (per notificare ad esempio che non è disponibile, poichè già registrato. Si ricordi che la mappa è composta di coppie *id-User* e gli *id* sono univoci) ed eventualmente aggiunge il nuovo utente! Queste sono però **due** operazioni, e non possiamo quindi utilizzarle senza preoccuparci di gestirne la sincronizzazione, poichè la mappa ne garantisce la *safety* nei soli casi di operazioni individuali. Fortunatamente,

ConcurrentHashMap offre alcuni comandi specifici, che servono ad astrarre operazioni normalmente composte (feature che *SynchronizedHashMap* non offre). Ho quindi utilizzato il metodo **putIfAbsent** per inserire in maniera corretta il nuovo utente nella struttura.

```
/**
 * Registra un nuovo utente al servizio WordQuizzle,
   inserendolo all'interno
 * di {@link #usersRegister} e riscrivendo sul file la
   struttura, cos  da
 * persistere le nuove modifiche. Per ulteriori informazioni
   riguardo l'implementazione
 * della scrittura su file, consultare la classe {@link
   ServerUtilities}, metodo
 * {@link ServerUtilities#writeJson(ConcurrentHashMap)}
 *
 * @param nickUtente L'username da registrare.
 * @param password La password relativa all'utente.
 * @throws RemoteException In caso di problemi con la
   tecnologia RMI.
 * @throws AlreadyRegisteredUserException In caso di id gi
   registrato.
 * @throws NullPointerException Nel caso in cui qualche
   parametro sia null.
 */
@SuppressWarnings({"RedundantThrows"})
@Override
public void registerNewUser(String nickUtente, String
   password) throws RemoteException,
        AlreadyRegisteredUserException, NullPointerException
   {
    if (nickUtente == null || password == null) throw new
   NullPointerException();
    //Le operazioni composte non sarebbero di norma thread-
   safe. Ci viene per  in aiuto la ConcurrentHashMap.
    if (this.usersRegister.putIfAbsent(nickUtente, new User(
   nickUtente, password)) != null)
        throw new AlreadyRegisteredUserException();
    try {
        //Thread-safe grazie all'utilizzo dei FileChannel.
        ServerUtilities.writeJson(this.usersRegister);
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Exception: trying writing on GSON
    after new sign up. [IO EXCEPTION]");
```

```
    }

    if(Server.DEBUG_MODE) System.out.println("Nuovo utente
   registrato: " + nickUtente);
}
```

Dobbiamo ora risolvere la questione legata alla scrittura su file della struttura. È chiaro che se due thread cercassero di scrivere il medesimo file in maniera non sincronizzata, si creerebbe un potenziale pericolo di inconsistenza. La situazione a questo punto è tragica: se dovessero infatti esserci incosistenze nel file, tutti i dati andrebbero persi. Al successivo avvio, il server non riuscirebbe a deserializzare il file, lanciando una **MalformedJsonException**. Gestire questo aspetto, in realtà, è stato semplice: l'interazione con il file è stata implementata tramite **FileChannel**, i quali sono per loro natura **thread-safe**, oltre ad essere molto efficienti (altra motivazione del loro utilizzo).

Bisogna ammettere che l'utilizzo del selettore e della *ConcurrentHashMap* alleggeriscono non di poco le attenzioni da riservare al tema della concorrenza. Ci sono però, chiaramente, alcune situazioni particolari che è comunque necessario gestire, come abbiamo appena visto riguardo la registrazione. Altre situazioni di possibile race-condition, sono causa della presenza del thread *MatchManager*. Cerchiamo di capire quale potrebbe essere il problema: a fine partita, i vari thread che gestiscono il match, procedono all'eventuale incremento del punteggio totale dell'utente vincitore. Questo potrebbe collidere con la richiesta da parte di un client, per esempio, della classifica dei propri amici, tra i quali potrebbe risiedere proprio l'utente di cui *MatchManager* sta aggiornando i punti. Questa situazione potrebbe tranquillamente portare ad un'incosistenza, e va quindi gestita: è stato necessario sincronizzare sia il metodo per la costruzione della classifica (usato dal server quando il client la richiede tramite apposito comando), sia i metodi di gestione del punteggio di un utente (utilizzati dai vari *MatchManager*). Qui sotto inserisco il codice di **buildRank**, che reputo leggermente più interessante dei vari metodi di gestione dei punti, i quali non verranno allegati poichè banali (si recupera l'utente tramite la *get* "già sincronizzata" fornita dalla *ConcurrentHashMap*, si acquisisce la lock sull'oggetto *User* e si aggiorna il punteggio; dopodichè, si rilascia la lock).

```
/**
```

```
 * Costruisce e restituisce una classifica degli amici dell'
   utente, sottoforma di {@link LinkedHashMap}
 * ordinata in maniera decrescente sui punteggi.
 *
 * @param id Utente che richiede la classifica.
 * @return La classifica,
 */
public LinkedHashMap<String, Integer> buildRank(String id) {
    LinkedHashMap<String, Integer> rank = new LinkedHashMap
   <>();

    User user = this.usersRegister.get(id);
    ArrayList<String> friends = user.getFriends();

    rank.put(id, user.getPoints());

    for (String friend : friends) {
        /*
               necessario sincronizzarci su ogni utente,
   poich  tra i miei amici potrebbe essere che
            qualcuno abbia appena finito di giocare e quindi
   il thread MatchManager ne stia aggiornando
            il punteggio totale.
         */
        User u;
        int points;
        synchronized (u = this.usersRegister.get(friend)) {
            points = u.getPoints();
        }
        rank.put(friend, points);
    }

    /*
        Ordinamento della classifica, cos  da restituirla in
    ordine decrescente di punteggio.
     */
    List<Map.Entry<String, Integer>> list = new ArrayList<>(
   rank.entrySet());
    list.sort((userA, userB) -> -(userA.getValue().compareTo(
   userB.getValue())));
    rank.clear();
    for (Map.Entry<String, Integer> entry : list) {
        rank.put(entry.getKey(), entry.getValue());
    }
```

```
    return rank;
}
```

Nel precedente capitolo, abbiamo parlato della struttura dati **inGameUsers**, e di come questa giochi un ruolo fondamentale nel far sì che il primo selettore (della classe main, e quindi thread, *Server*) e il secondo (del thread *MatchManager*) siano coordinati nel modo corretto. Anche in questo caso, è necessario far sì che gli accessi alla struttura, in lettura per *Server* e in scrittura per *MatchManager*, siano sincronizzati. Non inserisco alcun codice, poichè in entrambi casi composto da un semplice blocco **synchronized** all'interno del quale si lavora sulla struttura.

Bene! Per quanto riguarda il server, abbiamo finito. Ultimo appunto, ricordo la definizione tramite keyword **volatile** di due flag:

- Il flag utilizzato dai vari *APIFetcher* per notificare a *RequestManager* eventuali errori durante la richiesta *http*;
- Il flag controllato da *MatchManager*, settato eventualmente da *RequestManager* al momento opportuno, per gestire la chiusura della sfida a causa dello scadere del timeout;

Esaminiamo adesso il client (faremo molto prima, non c'è molto da dire). Come abbiamo visto nel primo capitolo, all'avvio il *Client* lancerà il runnable *UDPReceiver*, che sarà sempre in attesa di ricevere messaggi UDP. Abbiamo inoltre visto il meccanismo con il quale le sfide vengono ricevute e "notificate" anche al client: tramite una struttura dati **condivisa** tra i due thread. Ma questo è forse uno dei più classici scenari di programmazione concorrente: una forma particolare di **produttore** (*UDPReceiver*) - **consumatore** (*Client*). Ogni accesso alla struttura, sia dal thread di *Client*, sia da *UDPReceiver* è contenuto in un blocco **synchronized**, così che non ci sia alcun rischio di inconsistenza. Come per i flag del server, anche in questo caso è stato necessario dichiarare il flag per il controllo (ulteriore controllo, per meglio dire) della validità della sfida tramite la keyword **volatile**, poichè condiviso anch'esso dai due thread (non mi soffermo sul meccanismo, poichè già discusso nel capitolo precedente). I codici relativi non sono di grande interesse; inserisco solo alcune righe di *UDPReceiver*, per completezza (riguardano la fase in cui il thread esamina il messaggio ricevuto, cercando di capire che tipo di notifica sia arrivata da *RequestManager*; una *add* comporta l'inserzione di una nuova sfida, una *remove* la rimozione).

```
case "add":
    synchronized (this.requests) {
        String[] addressInfo = new String[2];
        addressInfo[0] = serverAddress.toString().substring
   (1);
        addressInfo[1] = String.valueOf(serverPort);
        requests.put(id, addressInfo);
    }
    break;
case "remove":
        synchronized (this.requests) {
        requests.remove(id);
    }
    break;
```

Chapter 3

La sfida

Soffermiamoci adesso sull'ultimo aspetto che esamineremo in maniera più dettagliata: l'implementazione del meccanismo di sfida. Seguiranno poi una lista dei vari moduli implementati ed alcune linee guida per l'esecuzione dell'applicazione.

Abbiamo già detto che il thread **MatchManager** utilizza un selettore per comunicare con i due client in game. L'intero meccanismo di sfida si basa su due indici, uno per ogni client, che servono a tenere traccia della situazione (a livello del match) di un certo utente. Questi indici saranno poi usati come *attachment* dei vari channel, così che il selettore possa capire la situazione in cui si trova l'utente collegato al client con il quale sta interagendo. Vediamo per prima cosa quali sono questi indici e cosa significano:

- FIRST_QUESTION_INDEX → 0 ... Server.numberOfWords - 1 → se l'indice è compreso tra 0 (incluso) e il numero di parole da tradurre (escluso), allora l'utente sta ancora inviando le sue traduzioni. Il server procede in questo caso a comunicare la parola di indice corrispondente;
- ERROR_INDEX → -1 → se l'indice è uguale a -1 qualcosa è andato storto! In realtà, ciò non accade mai (o almeno non dovrebbe);
- START_INDEX → -2 → nel momento in cui i due channel vengono registrari nel selettore, viene inserito come *attachment* l'indice -2, così che il selettore possa capire quando l'utente deve ancora ricevere il ***primissimo*** messaggio: non sarà una parola, bensì il messaggio di

sistema *"via alla sfida di traduzione! Avete 'x' secondi per tradurre 'y' parole..."*;

- FINISHED_QUESTIONS_INDEX → Server.numberOfWords → se l'indice è uguale al numero di parole da tradurre, l'utente ha concluso tutte le sue traduzioni. Gli verrà allora inviato il messaggio contenente le statistiche della sua prestazione, invitandolo ad attendere che anche lo sfidante termini le sue traduzioni o che scada il timeout;

- END_INDEX → Server.numberOfWords + 1 → se l'indice è uguale al numero di parole da tradurre **+ 1**, allora *MatchManager* invia il messaggio di esito del match (il messaggio finale). Questa situazione ha luogo quando entrambi i client hanno finito oppure quando scade il timeout.

Vediamo adesso ciò che fa, in maniera molto schematica (quasi come fosse uno *pseudocodice*), *MatchManager*. Si ricordi che viene messo a disposizione di *RequestManager* un metodo, **timeout()**, che verrà chiamato nel momento in cui scadrà il tempo massimo consentito per la sfida: ciò comporterà la modifica dell'ononimo flag, il quale verrà impostato a **true**.

- Per prima cosa si procede all'inserzione di *idA* e *idB* all'interno della struttura **inGameUsers** (in maniera **sincronizzata**). Si ricordi che questo passaggio è fondamentale per far sì che il selettore di *Server* non "rubi" i messaggi a *MatchManager*;

- Una volta fatto ciò, si registrano i due channel nel nuovo selettore, impostando l'*attachment*, ovvero l'indice, a **START_INDEX** e la modalità a **WRITE** (sarà infatti *MatchManager* il primo a implementare la comunicazione, inviando il messaggio di inizio match ai due client);

- Ora, **fino a quando** entrambi gli utenti non hanno terminato di giocare (due flag appositi, uno per ogni utente, contengono questa informazione) fai ciò:

  - Se il channel pronto è in modalità **WRITE**:

    * Recupera l'indice *attached* al channel;

    - È scaduto il **timeout**? Se sì, l'indice è **minore** di Server.numberOfWords (ovvero, l'utente deve ancora finire di tradurre le parole)? Se sì, allora **non** devo inviare la prossima parola, bensì il messaggio di notifica di timeout. Inoltre, registro il channel **nuovamente** in modalità **WRITE**, con indice **FINISHED_QUESTIONS_INDEX**;
    - L'indice è uguale a **START_INDEX**? Bene, allora siamo proprio all'inizio del match, il selettore invia all'utente il messaggio di inizio sfida (*"via alla sfida di traduzione..."*) e registra **nuovamente** il channel in modalità **WRITE**, con indice **FIRST_QUESTION_INDEX**;
    - L'indice è uguale a **FINISHED_QUESTIONS_INDEX**? Se sì, l'utente ha concluso la traduzione di tutte le parole oppure è scaduto il timeout. *MatchManager* invia ad esso le statistiche riguardo le traduzioni date, invitandolo ad attendere la conclusione del match. Il channel viene registrato **nuovamente** in modalità **WRITE**, con indice **END_INDEX**;
    - L'indice è uguale ad **END_INDEX**? Vediamo, se è scaduto il timeout o se entrambi i giocatori hanno terminato le domande allora procediamo a concludere realmente il match per questo client, altrimenti aspettiamo che termini la sfida (quindi che scada il timeout o che anche l'altro utente termini le traduzioni). Se possiamo procede a concludere il match, invieremo l'esito finale, deregistreremo il channel dal selettore e, importantissimo, **rimuoviamo** l'utente dalla struttura **inGameUsers**, così che l'utente appena notificato della terminazione possa subito iniziare a comunicare con il *Server*;
    - L'indice è minore di **FINISHED_QUESTIONS_INDEX** ed è diverso da **ERROR_INDEX**? Bene, allora l'utente sta giocando, inviamo la prossima parola da tradurre. Una volta fatto ciò, registriamo il channel in modalità **READ**, inserendo come *attachment* l'array di *ByteBuffer* che useremo per implementare la fase di lettura del messaggio (un buffer per la size del messaggio e uno per il messaggio stesso);
    - L'indice è uguale a **ERROR_INDEX**? Qualcosa è andato storto! Tuttavia, non dovremmo mai capitare in questa situazione.
- Se il channel pronto è in modalità **READ**:

    - Leggi il messaggio;
    - Se il **timeout** è scaduto, **salta questo passaggio**. Altrimenti, controlla la correttezza della traduzione ricevuta, ed eventualmente aggiorna le statistiche partita relative all'utente collegato al channel con il quale stiamo interagendo;
    - Incrementa l'indice relativo all'utente corrispondente al channel pronto e registra il channel in modalità **WRITE** inserendo come attachment l'indice incrementato;

- Recupera l'unica istanza di *UsersRegister* e procedi all'aggiornamento del punteggio totale dell'eventuale utente vincitore del match;

Chapter 4

Le classi implementate

Andremo adesso a fare un rapido escursus di tutte le classi implementate nel progetto, esaminandole nello stesso ordine in cui sono state divise nei vari package. Partiremo dal package *common_src*, poichè contiene ciò che gli altri due package (quello del client e del server) utilizzano nelle loro classi.

4.1 Common Package

4.1.1 CommonUtilities.java

È una classe di utilità, creata per incapsulare i metodi **writeIntoSocket** e **readFromSocket** (così da favorire *riusabilità* e *leggibilità*), utilizzati da *Client* e *RequestManager* per scrivere e leggere dalla socket TCP (le altre classi che interagiscono su una socket, ovvero *MatchManager* e *Server* utilizzano un selettore, pertanto hanno un'implementazione propria per leggere e scrivere). Come già accennato nei precedenti capitoli, ho implementato un protocollo di comunicazione che prevede l'invio della size del messaggio prima del messaggio stesso. Ciò permette, in fase di lettura, l'allocazione del *ByteBuffer* relativo della giusta dimensione, evitando così alcun spreco di spazio. Inserisco il codice del metodo *readFromSocket*, poichè da esso si può notare l'utilità del meccanismo.

```
public static String readFromSocket(SocketChannel
    socketChannel) throws IOException {
     /*
          Le successive 4 righe si occupano di recuperare la
    lunghezza
```

```
        del messaggio da leggere. Si noti che allocare un
    ByteBuffer
        per ricevere un intero non porta a sprechi, grazie
    alla costante Java.
     */
    ByteBuffer sizeBuffer = ByteBuffer.allocate(Integer.BYTES
    );
    socketChannel.read(sizeBuffer);
    sizeBuffer.flip();
    int size = sizeBuffer.getInt();
    //Stampa di debug, nel caso in cui DEBUG_MODE sia
    impostato a true
    if (DEBUG_MODE) System.out.println("Size da leggere: " +
    size);
    /*
        Si procede alla ricezione del messaggio vero e
    proprio. Si
        noti che la size recuperata prima ci permette di
    allocare
        il ByteBuffer della corretta dimensione, evitando
    alcun spreco.
     */
    ByteBuffer readBuffer = ByteBuffer.allocate(size);
    //Stampa di debug, nel caso in cui DEBUG_MODE sia
    impostato a true
    if (DEBUG_MODE) System.out.println(readBuffer);
    socketChannel.read(readBuffer);
    readBuffer.flip();
    //Stampa di debug, nel caso in cui DEBUG_MODE sia
    impostato a true
    if (DEBUG_MODE) System.out.println(readBuffer);
    /*
        Passaggio da ByteBuffer a String. Importante l'
    utilizzo di trim() per evitare
        problemi "strutturali" della stringa.
     */
    String message = new String(readBuffer.array()).trim();
    //Stampa di debug, nel caso in cui DEBUG_MODE sia
    impostato a true
    if (DEBUG_MODE) System.out.println("Messaggio ricevuto: "
    + message);
    readBuffer.clear();
    return message;
}
```

4.1.2 UsersRegisterInterface

Rappresenta l'**interfaccia remota** dell'oggetto *UsersRegister*, fondamentale per far sì che il *Client* sappia il nome del/i metodo/i invocabili sull'oggetto remoto (ed è proprio per questo che è contenuta in questo package, la classe *Client* dovrà importarla).

4.1.3 Exceptions

AlreadyRegisteredUserException

Eccezione custom, lanciata dal metodo (invocabile da remoto) *registerNewUser* in caso si dovesse richiedere la registrazione di un id già registrato. A livello di codice non c'è molto da dire, è semplicemente una sottoclasse di *Exception* che richiama il costruttore della superclasse tramite il comando *super()*.

4.1.4 config.properties

In tutto il progetto, ho utilizzato due file di questo tipo. Abbiamo già visto il modo in cui vengono utilizzati e le motivazioni della loro presenza. All'interno di questo file è possibile configurare la **porta TCP** del *Server* (che deve però essere nota anche al *Client*), la **porta RMI** relativa al registro sul quale verrà esportato l'oggetto (o meglio il riferimento ad esso) e un **limite** di lunghezza per i comandi del *Client*. Quest'ultimo parametro serve da riferimento per l'allocazione dei *ByteBuffer* di lettura lato server, nei selettori (in tal caso, non è possibile allocare di una precisa dimensione i buffer di lettura, poichè le scritture da parte dei client potrebbero tranquillamente **non** essere atomiche, bensì essere distribuite su più cicli di selettore).

4.2 Frontend Package

All'interno del package *front_end_src* sono contenute tutte le classi appartenenti al mondo del client. Nella presentazione di queste, ometteremo due sub package: *gui* (ne parleremo a fine relazione) e *tests* (ne parleremo nel prossimo capitolo).

4.2.1 Client

Il cuore del package. Implementa il client *a linea di comando* che gli utenti potranno utilizzare per usufruire del servizio. Durante l'analisi dell'intero funzionamento ci siamo fatti una chiara idea di ciò che permette di fare, quindi non ci soffermeremo oltre.

4.2.2 ClientUtilities

Classe di **utilità** per *Client*. Contiene due metodi:

- **printMenu**: stampa il menù del client, così che l'utente possa interagire nel miglior modo possibile. Niente di speciale, semplicemente una serie di stampe che cercano di spiegare all'utente quali operazioni può compiere e con quali comandi;

```
----------------------------------------
BENVENUTO IN WORD QUIZZLE!
----------------------------------------

--- REGISTRARSI/LOGGARSI PRIMA DI ESEGUIRE ALTRE OPERAZIONI! ---
--- USAGE: command [args ...] ---
registra_utente <nickUtente> <password> --- Registra un nuovo utente, con le credenziali indicate.
login <nickUtente> <password> --- Effettua il login.
logout  --- Effettua il logout.
aggiungi_amico <nickAmico> --- Aggiungi <nickAmico> alla tua lista amici.
lista_amici --- Visualizza la tua lista amici.
sfida <nickAmico> --- Richiedi di sfidare <nickAmico>.
mostra_punteggio --- Visualizza il tuo punteggio globale attuale.
mostra_classifica --- Mostra una classifica dei tuoi amici (incluso te stesso).
mostra_sfide --- Mostra le richieste di sfida attive.
esci --- Chiudi l'applicazione.
```

- **checkCommandArguments**: controlla l'input inserito dall'utente, lanciando una **WrongNumberOfArgumentsException** in caso di un cattivo utilizzo di qualche comando. Dal momento che i controlli da fare sono diversi, incapsularli in un metodo esterno ha reso molto migliore la **leggibilità** del codice del *Client*;

4.2.3 UDPReceiver

Abbiamo già discusso il componente esaminando il funzionamento del progetto, pertanto non ci soffermeremo a parlarne. Si ricordi comunque che il task di questo runnable è ricevere messaggi UDP dal server.

4.2.4 Exceptions

WrongNumberOfArgumentsException

Eccezione custom, lanciata all'interno del metodo *checkCommandArguments* (dell classe *ClientUtilities*) in caso di un errato utilizzo dei comandi da parte dell'utente.

4.3 Backend Package

Senza dubbio il package con più classi; non ci soffermeremo troppo su di esse, poichè già incontrate durante la presentazione dell'architettura generale del progetto.

4.3.1 Server

Il **main** del *mondo* del server: crea il selettore e comunica con i vari client, eseguendo le operazioni da essi richieste. Oltre al metodo principale, è stato impementato **manageQuit**, un metodo ausiliario per gestire il caso in cui un utente (o meglio un client) si sconnetta dal server in maniera inaspettata (e non): implementa la deregistrazione del channel dal selettore e soprattutto l'aggiornamento della struttura dati degli utenti online.

4.3.2 ServerUtilities

Classe di **utilità** per il *Server*. Vediamone, in breve, i vari metodi:

- **writeJson**: preso in input un riferimento alla struttura dati degli utenti (la *ConcurrentHashMap*) la **serializza**, persistendo su file le informazioni. Il metodo è chiamato ogni qual volta la struttura dati subisce una qualche tipo di modifica;
- **readJson**: implementa il meccanismo di **deserializzazione** per il punto precedente. Ciò vuol dire che restituirà la *ConcurrentHashMap*. Il metodo è chiamato dal *Server* all'avvio, così da recuperare la versione più aggiornata dei dati degli utenti;
- **tokenizeString**: un semplice *tokenizer* per interpretare e suddividere il comando ricevuto dal client;

- **getKey**: avente come parametro la mappa degli utenti online, permette di recuperare la chiave (ovvero l'*id* dell'utente collegato ad un certo channel) dal value (ovvero il channel). Il metodo è utilizzato dal server per capire da quale utente arriva il comando appena ricevuto;

- **tokenizeAddress**: è utilizzato dalla classe *Server* per recuperare la **porta** del channel TCP di un certo utente, così da poterla comunicare al *RequestManager* (abbiamo già discusso nel primo capitolo questo meccanismo);

- **readingDictionary**: viene utilizzato dal *Server*, all'avvio, per recuperare la lista delle parole italiane dal file di testo *dictionary.txt*;

- **generateRandomNumbers**: viene utilizzato da *RequestManager* nel momento in cui deve scegliere una certa quantità di parole casuali dal dizionario. Questo metodo restituisce una serie di numeri (tanti quanti le parole da dover scegliere) casuali tutti diversi tra loro, così che si possano poi utilizzare come indici per recuperare le diverse parole dal dizionario.

4.3.3 User

La classe *User* astrae un generico utente che usufruirà del servizio, incapsulando le sue informazioni e i vari metodi per leggerle e gestirle.

4.3.4 UsersRegister

La classe *UsersRegister* rappresenta il *database* degli utenti, incapsulando la struttura dati vera e propria (la *ConcurrentHashMap*) e una serie di metodi per gestirla. Rappresenta senza dubbio il punto chiave del mondo del server. I metodi in essa definiti (non viene indicato il metodo per l'implementazione del pattern *Singleton*, poichè già discusso in precedenza):

- **registerNewUser**: registra un nuovo utente all'interno della collezione; invocabile da remoto dai vari client.

- **searchUser**: cerca un utente all'interno della *ConcurrentHashMap*, restituendo l'esito della ricerca. Viene utilizzato per controllare la validità di alcuni comandi, per esempio l'aggiunta di una nuova amicizia

(chi ci dice che l'utente da cui la richiesta è arrivata abbia indicato un destinatario realmente esistente?);

- **checkPsw**: controlla la validità della *password* relativa ad un certo *id*;
- **buildRank**: costruisce (con ordinamento decrescente sui punteggi) la classifica amici di un certo utente;
- **isFriendOf**: controlla l'esistenza di un'amicizia tra due utenti. È utilizzato nel momento in cui utente ne sfida un altro: è necessario che i due siano amici affinchè si possano scontrare!
- **getPointOf**: preso come parametro l'*id* di un utente, ne restituisce il punteggio totale;
- **getFriendsOf**: preso come parametro l'*id* di un utente, ne restituisce la lista amici;
- **addFriends**: aggiunge una nuova amicizia tra due utenti. Può lanciare una *AlreadyFriendException* nel caso in cui questa dovesse già esistere;
- **incrementPointsOf**: preso come parametro l'*id* di un utente, ne incrementa il punteggio di una certa quantità (anch'essa passata come parametro al metodo);
- **printRegister**: stampa la struttura dati in una forma leggibile. Viene invocato (la prima ed unica volta) dal *Server* all'avvio;

Prima di passare al prossimo modulo, vorrei fare un'osservazione: come si può facilmente notare, molti dei metodi appena citati sembrano praticamente uguali (sia di nome che di fatto) ad alcuni già citati per la classe *User*. Effetivamente è così! Se prendiamo per esempio il metodo *incrementPointsOf* della classe *UsersRegister*, questo utilizzerà il metodo *incrementPointsOf* della classe *User* dopo aver recuperato il riferimento all'utente in questione dalla struttura dati. Ho deciso però di creare quest'ulteriore livello di astrazione perchè in realtà quella non è l'unica cosa fatta dal metodo di *UsersRegister*: deve anzitutto sincronizzarsi sull'utente per recuperarne il riferimento (facendolo qui, non dovremmo farlo nel *Server*, il quale potrà usufruire del metodo senza preoccuparsi della concorrenza); successivamente deve procedere alla chiamata di *writeJson*, per serializzare la struttura appena modificata (è stato modificato il punteggio totale dell'utente in questione) per persistere le nuove informazioni.

4.3.5 RequestManager, APIFetcher, MatchManager, StructManager

Le quattro classi responsabili dell'implementazione del match. Le abbiamo già discusse parlando del funzionamento generale del progetto, in particolare delle sfide tra gli utenti.

4.3.6 Exceptions

AlreadyFriendException

Eccezione custom, lanciata dal metodo *addFriend* nel caso in cui si tenti di aggiungere un'amiciza già esistente.

4.3.7 config.properties

Abbiamo già incontrato un file di questo tipo, nel *Common Package*. In questa occasione, *config.properties* offre la possibilità di customizzare i vari parametri di configurazione dei match:

- Il numero di parole da tradurre;
- Il timeout di una richiesta;
- Il timeout di un match;
- I punti guadagnati ad ogni traduzione corretta;
- I punti persi ad ogni traduzione errata;
- I punti guadagnati in caso di vittoria del match;

Chapter 5

Compilazione, esecuzione e test

Per utilizzare l'applicazione, è necessario anzi tutto decidere in quale modalità verrà aperto il client: ci interfacceremo tramite linea di comando o tramite interfaccia grafica? Nel primo caso non bisogna fare altro che avviare *Server*, e successivamente *Client*. Nel secondo caso è necessario fare una piccola modifica: si apra il codice sorgente di *Server* e si cerchi la dichiarazione del flag **GUI_MODE** (riga 38 circa). Si cambi il valore del flag, settandolo a *true*. Ciò è necessario perchè il server agirà (in alcuni, pochi, punti) in maniera diversa nel caso in cui il client sarà utilizzato nella sua versione ad interfaccia grafica. Poche righe sopra, nel codice, si trova anche il flag **DEBUG_MODE**: se *true*, il server stamperà a linea di comando alcuni messaggi inerenti alla sessione corrente (per esempio, quando un utente logga, quando un utente abbandona, e così via...). Si setti a piacere il booleano e si lanci poi *Server* (questo vale per entrambi i tipi di client). Ultima differenza, nel caso in cui si voglia utilizzare l'interfaccia grafica, **non** bisognerà lanciare *Client*, ma **ClientGUI**.

In realtà, prima di avviare i due programmi, si può agire sui file *config.properties* per personalizzare eventuali *settings* dell'applicazione. Le impostazioni relative alle connessioni di rete (porta TCP ed RMI) si trovano nel file di configurazione contenuto nel package *common_src*. Le impostazioni relative al match (tempo del match, timeout delle richieste, numero di parole da tradurre, ...) si trovano invece nel file di configurazione contenuto nel package *back_end_src*.

È chiaro che questo tipo di utilizzo non ci può dire molto: l'applicazione non è realmente in rete! Lanciando un client, finiremo per essere i soli utenti ad usufruire del software in quel momento, non potremmo fare granchè! Durante l'implementazione del progetto, ho deciso di creare alcuni **test** per simulare alcuni scenari in cui il software potrebbe trovarsi se questo fosse attivo ed utilizzato da un certo numero di utenti. I vari programmi hanno l'obiettivo di testare la correttezza delle principali funzionalità del server *WordQuizzle*. Ne vedremo adesso una breve descrizione, si tenga a mente però che il concetto base è uguale per tutti: tramite l'utilizzo di un pool si lanceranno una certa quantità di thread che fungeranno da client (il numero di thread da lanciare è personalizzabile). In questo modo si potrà verificare il corretto funzionamento di tutta l'applicazione, piuttosto che procedere manualmente tramite l'utilizzo di un solo client (i programmi di test si trovano nel package *front_end_src*, nel sub-package *tests*).

- **PoolTestMisc**: vengono fatte **NUM_CLIENT_TEST** iterazioni, all'interno delle quali vengono potenzialmente lanciati due thread: **RegisterTest**, il quale simulerà un client richiedente una registrazione, e **RandomTests**, il quale simulerà un client che, previa login, richiederà alcune operazioni di carattere sociale al server. Dico potenzialmente perchè i due thread verranno lanciati solo se i rispettivi flag sono settati a true: **TEST_REGISTRATION** e **TEST_MISC** rispettivamente. Questo permette di poter testare i due aspetti sia simultaneamente sia in maniera mutuamente esclusiva;

- **TestingUDPReceiver**: **non utilizzabile**. È stato di aiuto per testare la correttezza di UDPReceiver nell'atto della ricezione dei vari tipi di messaggi. La domanda alla quale cercavo risposta è: ma siamo sicuri che non si perda nessun messaggio per strada e che riesca correttamente a leggerne il tipo di ognuno? (Per esempio, *add*, *remove*, ...)

- **TestMatches**: con *PoolTestMisc* non abbiamo testato un'unica funzionalità: la sfida! Il compito è stato affidato a *TestMatches* che fa una cosa molto semplice: una volta scelto il numero di match da testare (**NUMBER_OF_MATCHES**) esegue le rispettive iterazioni. All'interno di ogni iterazione lancia un **Requester** (colui che richiederà la sfida) e un **Accepter** (colui che la accetterà). Per prima cosa i due eseguono il login, poi scambiano l'amicizia e infine il primo richiede di sfidare il secondo. Una volta fatto ciò, *Accepter* accetterà la sfida,

la quale inizierà: i due client invieranno una parola casuale (per semplicità; alla fine a noi interessa sapere che tutto funziona correttamente, indipendentemente dalle prestazioni dei giocatori) fino a quando questa non termina, portando così alla terminazione anche del test.

- **PoolTestMatchesStress**: **non utilizzabile**. Questa è una versione *raw* di *TestMatches*. Consiglio di non utilizzarla poichè richiede alcune accortezze a livello di codice: si basa molto sul tempismo in cui i thread si richiedono le sfide o le amicizie, quindi è importante impostare in maniera corretta alcune *Thread.sleep* affinchè tutto funzioni correttamente. Per *TestMatches* la situazione è invece totalmente differente, poichè implementa dei meccanismi di sincronizzazione (automatici) tra un *Requester* e il corrispondente *Accepter*.

Il modo in cui questi test vanno utilizzati è molto semplice: si faccia partire la classe *Server* e successivamente la classe relativa al test (ricordo che *TestingUDPReceiver* e *PoolTestMatchesStress* **non** sono utilizzabili). I test vanno a buon fine quando i programmi terminano correttamente: in tal caso, non ci sono stati errori durante la comunicazione.

Il progetto è stato sviluppato utilizzando **Intellij IDEA**. È stata inoltre utilizzata la libreria esterna **Gson** di Google per implementare il meccanismo di serializzazione. Nella stessa cartella in cui è contenuto il progetto è presente la liberia di cui sopra, in formato *.jar*. Nel caso in cui l'esecuzione venisse fatta tramite *Intellij* non dovrebbe essere necessario fare nulla, se non lanciare i programmi desiderati. Nel caso in cui l'esecuzione venisse fatta tramite **Eclipse** sarà necessario, prima di poter utilizzare il software, importare la libreria usata:

- Si prema con il tasto destro sull'intero progetto, nel *project explorer* di *Eclipse* (si prema quindi su *'WordQuizzle'*);
- Si vada con il cursore sopra **Build Path >Add External Archives...**
- Si importi la libreria contenuta nella directory consegnata (*gson-2.8.6.jar*);
- Fine! Adesso sarà possibile eseguire i programmi;

Chapter 6

L'interfaccia grafica

Concludiamo la relazione parlando, in maniera molto sintetica, dell'aspetto grafico del client. Per la realizzazione di tutta l'interfaccia grafica è stata utilizzata la tecnologia **Swing** di *Java*. Soffermarsi su tutti gli aspetti realizzati porterebbe via molto tempo e spazio di questa relazione, pertanto indicheremo solo le reali differenze (a livello funzionale) rispetto all'approccio con client a linea di comando. In ogni caso, la feature più utilizzata durante tutta l'implementazione è stata l'API **SwingWorker**, la quale permette di eseguire alcuni task in background (nei quali si tenta la comunicazione con il server) per poi aggiornare la UI tramite il metodo **done** o il metodo **SwingUtilities.invokeAndWait**. Ciò è molto importante, perchè è concettualmente sbagliato eseguire operazioni potenzialmente bloccanti nel thread UI o eseguire cambiamenti all'interfaccia grafica in thread che non siano quello UI.

Per prima cosa, nel caso in cui si utilizza un client a interfaccia grafica, verrà invocata una *nuova* versione dell'*UDPReceiver*: **UDPReceiverGUI**. Questo perchè, in questo scenario, il server invierà ulteriori messaggi UDP al client rispetto alla sola richiesta di sfida:

- Nel caso in cui *idA* richieda l'amicizia di *idB*, e nel caso in cui questa venga concessa con successo, il server invierà un messaggio di notifica all'*UDPReceiverGUI* di *idB*, il quale aggiornerà in tempo reale la lista amici visualizzata a schermo (tutto ciò se *idB* è online). Questa feature non era necessaria con il client a linea di comando, poichè la lista amici non era costantemente visualizzabile, bensì veniva richiesta tramite apposito comando in un preciso momento.

- Nel caso in cui dovesse scadere il timeout di un match, *RequestManager* cambia leggermente il proprio comportamento, inviando un messaggio UDP all'*UDPReceiverGUI* dei due utenti (continua a fare ciò che faceva anche prima, ovvero invocare il metodo *timeout* di *MatchManager*). Ciò permette a *ClientGUI* di poter reagire in tempo reale al timeout della sfida, letteralmente *facendo sparire* il box di testo e il bottone di invio traduzione. Questa feature non è possibile averla con il client a linea di comando, poichè il programma stesso si trova in una situazione bloccante sia quando deve inviare traduzioni, sia quando deve ricevere messaggi dal server. Difatti, il client a linea di comando riceve la notifica di tempo scaduto al posto dell'eventuale parola successiva, solo dopo che questo ha mandato la traduzione corrente al server (eventualmente, starà al server precisare che la traduzione appena inviata non è stata conteggiata come valida).

Questi *nuovi* messaggi UDP sono inviati dal server tramite un nuovo componente: **UDPSender**. Adesso, dovrebbe essere più chiaro l'utilità del flag nominato nel precedente capitolo, ovvero **GUI_MODE**. Nel caso in cui quest'ultimo sia true, il server procederà all'invocazione dell'oggetto *UDPSender* e all'invio dei messaggi UDP nelle due situazioni appena analizzate.

Infine, sono stati realizzati alcuni componenti custom. Niente di particolare, semplicemente degli oggetti che estendono i componenti già presenti nel framework *Swing*, personalizzati automaticamente al momento della loro creazione. Un approccio di questo tipo incrementa la **riusabilità** del codice.

Concludiamo con alcuni screen delle varie schermate realizzate.

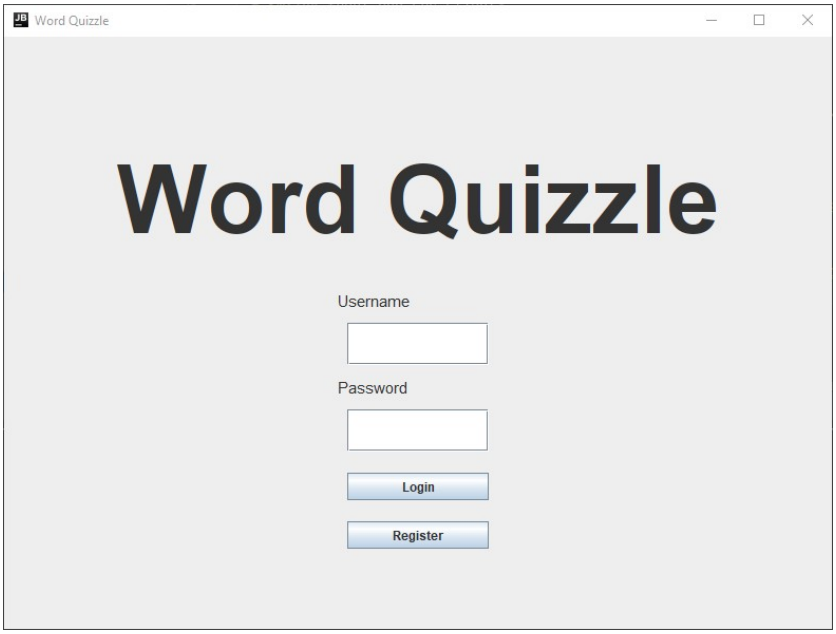


Figure 6.1: Login page

Figure 6.2: Home page

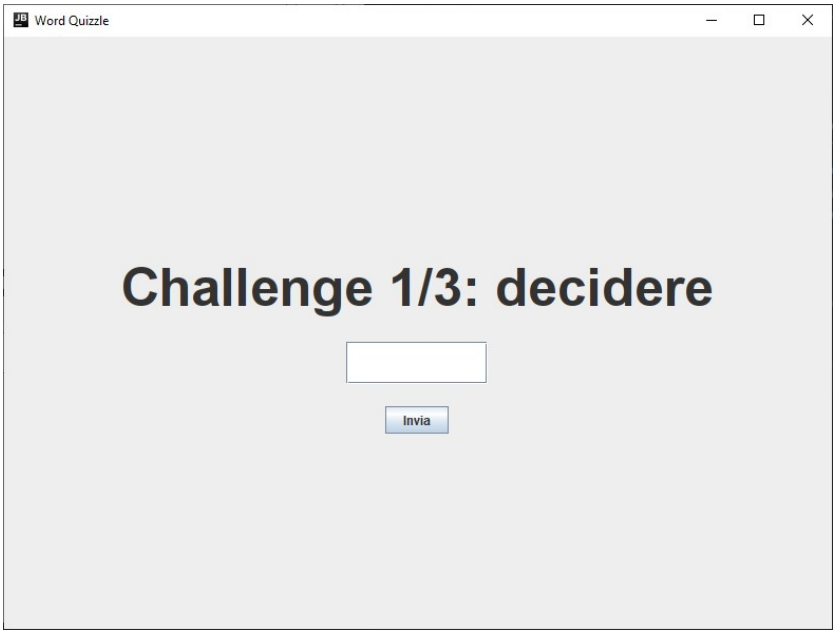


Figure 6.3: Match page

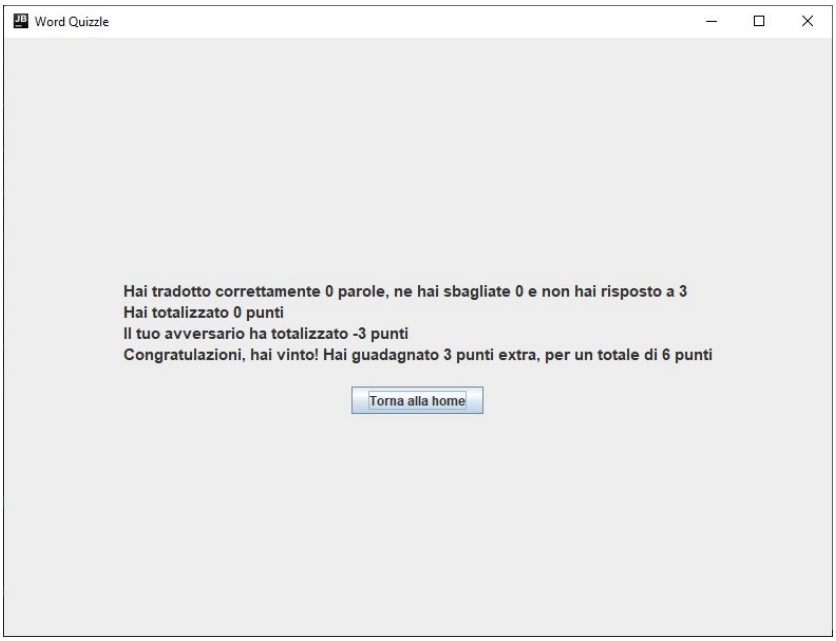


Figure 6.4: Match-End page

Figure 6.5: Adding new friend