Relazione Reti

Fausto Frasca

February 13, 2020

1 Premesse

Come punto di riferimento per il progetto si è scelto un computer di ultima generazione non troppo potente, ma comunque con risorse hardware minime come 8gb di ram, un processore quad-core 8 thread e un ssd.
Su tale modello si è cercato di perseguire determinate scelte che garantissero un giusto compromesso tra uso efficiente delle risorse, velocità e complessità generale del sistema.

2 Architettura generale

In questo paragrafo verrà riportata una breve introduzione di quella che è l'architettura generale del sistema. Il tutto verrà approfondito nei capitoli successivi.

2.1 Client

Il client fornisce una GUI per l'interazione con l'utetne. Si è preferita tale scelta, anzichè la cli, per via di una maggiore semplicità d'uso.
Una volta connesso al server, il client avrà un processo principale per gestire la comunicazione client-server su socket TCP e un thread a parte che sta in ascolto su una socket UDP per le richieste di sfida.

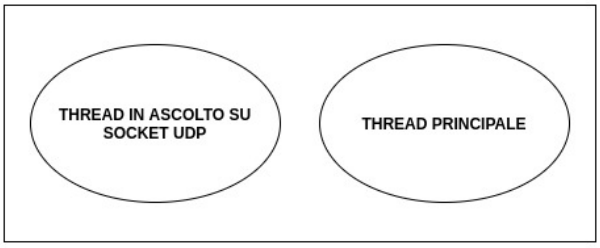


Figure 1: Schema client

2.2 Server

Il server è composto da un thread con selettore principale per la gestione della connessione TCP, un threadpool per svolgere operazioni "lunghe" delegate dal selettore e un oggetto RMI per l'operazione di registrazione.
Sotto al server è presente uno *user dispatcher*, con cui comunica richiedendo gli oggetti *User* in modo tale che sia il thread principale, sia i thread del threadpool operino sui medesimi oggetti.
Sotto allo user dispatcher è presente un *DBMS* che si occupa della gestione degli oggetti User in memoria, sia in lettura che in scrittura.
Come riportato nell'immagine sotto, il server comunica esclusivamente con lo user dispatcher e quest'ultimo comunica col DBMS.

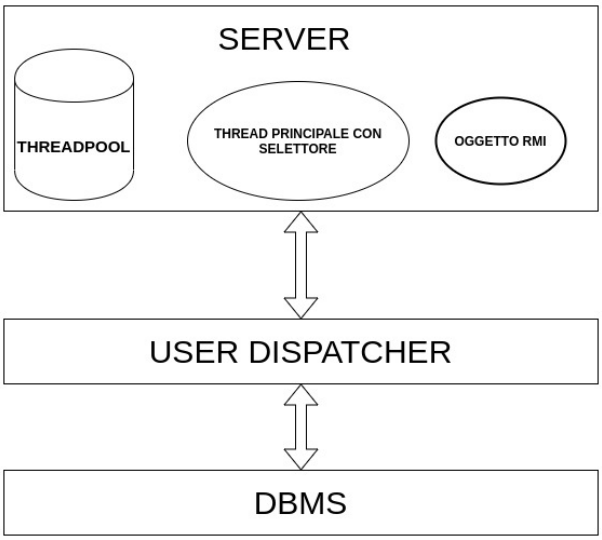


Figure 2: Schema server

3 Struttura generale delle directory del progetto

Il progetto è costituito da tre directory:

- Client
- Commons
- Server

3.1 Client

All'interno del path `Client/src/main/java` vi sono tutte le classi sfruttate dal client.
È presente anche una directory `Costanti`. Al suo interno vi è un'interfaccia con le costanti generali usate dalle varie classi del client.

3.2 Commons

All'interno del path `Commons/src/main/java` sono presenti le interfacce comuni sfruttate sia dal client che dal server.

3.3 Server

All'interno del path `Server/src/main/java` troviamo le classi sfruttate dal server. Alcune di queste sono raggruppate in sottodirectory per una maggiore comprensione del lavoro svolto da ciascuna di loro.
Anche qui è presente una directory `Costanti` contenente un'interfaccia con tutte le costanti usate dalle varie classi del server.

4 Descrizione classi e scelte implementative

In questo capitolo verranno illustrate le singole classi presenti nelle tre directory descritte in precedenza.

4.1 Commons

4.1.1 RMIRegistrationInterface.java

Questa è l'interfaccia dell'oggetto RMI che poi sarà registrato nel registry.
Al suo interno è presente il metodo che permette a un utente di registrarsi e i vari codici ritornati dal metodo stesso.
Sono inoltre definite le costanti per la porta e l'identificativo dell'oggetto remoto.

4.1.2 TCPConnection.java

All'interno di questa interfaccia sono definite le costanti per la connessione TCP tra client e server.

4.2 Client

Il metodo costruttore di tutte le classi del client contiene le istruzioni per stampare a video le varie componenti da inserire nel frame dell'interfaccia

4.2.1 MainClassClient.java

Questa classe ha il compito di settare il main frame, che poi ospiterà i vari elementi della GUI, e di richiamare un oggetto della classe *startGUI*.

4.2.2 StartGUI.java

Tale classe ha il compito di mostrare la schermata iniziale.
Le operazioni offerte all'utente sono: *login* e *registrazione*

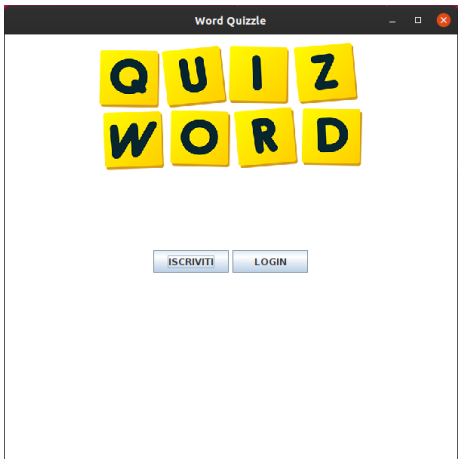


Figure 3: Schermata avvio client

4.2.3 Register.java

Questa classe sfrutta l'oggetto RMI remoto inserito nel registry, richiamando il metodo `registra_utente` che permette di effettuare la registrazione di un utente. Successivamente, tramite uno switch, si parsa il codice di ritorno del metodo richiamato.
Se il codice indica un errore, questo viene stampato nella parte inferiore del frame. Se invece la registrazione è andata a buon fine, viene creata in automantico un'istanza della classe *Login.java*.

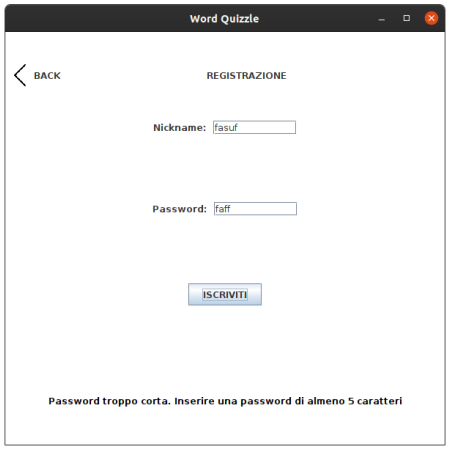
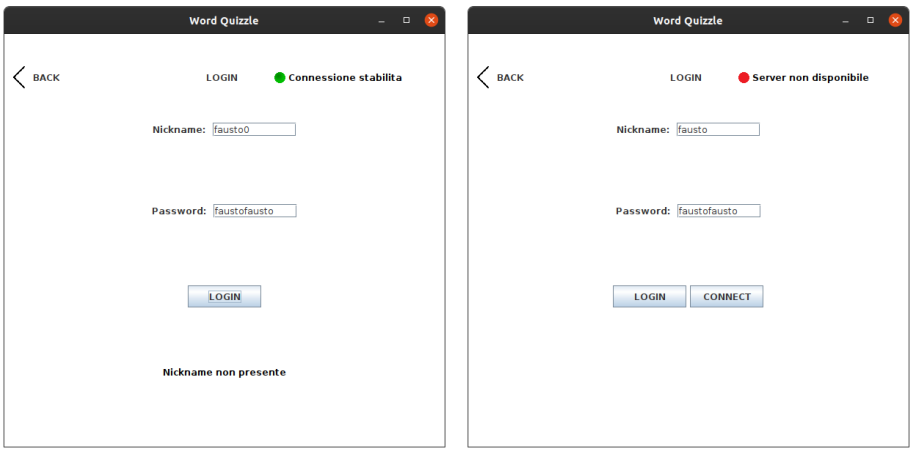


Figure 4: Schermata di registrazione con un eventuale errore riportato in basso

4.2.4 Login.java

Una volta entrati nella schermata di login, la classe instaura una connessione TCP col server, in base ai parametri definiti nell'interfaccia *TCPConnection* (4.1.2). L'avvenuto successo o meno della connessione viene riportato in alto, nel lato destro dell'interfaccia.
Qualora si fosse verificato un errore durante la connessione TCP, verrà mostrato un secondo pulsante denominato `CONNECT` che permette di provare nuovamente a instaurarla.
Anche in questo caso, se ci fossero degli errori nell'autenticazione dell'utente, questi verrebbero riportati nella parte inferiore.
Se invece l'autenticazione del client è andata a buon fine si richiamano le classi *UDPListener.java* (e la si passa a un thread) e *HomePage.java*



(a) (b)

Figure 5: Schermata di login con un eventuale errore riportato in basso (a) Schermata di login con errore sulla connessione TCP (b)

4.2.5 HomePage.java

Superato il login, viene fatta comparire la home page. Qui l'utente può richiedere diverse operazioni:

- Aggiungere un amico
- Visualizzare la lista degli amici
- Visualizzare il proprio punteggio
- Sfidare un amico
- Richiedere il logout

Alcune risposte (come per esempio la conferma dell'avvenuta aggiunta di un amico) ed eventuali errori vengono riportati nella parte inferiore della schermata. Da qui in avanti, da qualsiasi punto della schermata, se la connessione TCP risultasse interrotta verrebbe mostrato un pop up di avviso e l'utente sarebbe reindirizzato alla schermata iniziale (4.2.2).

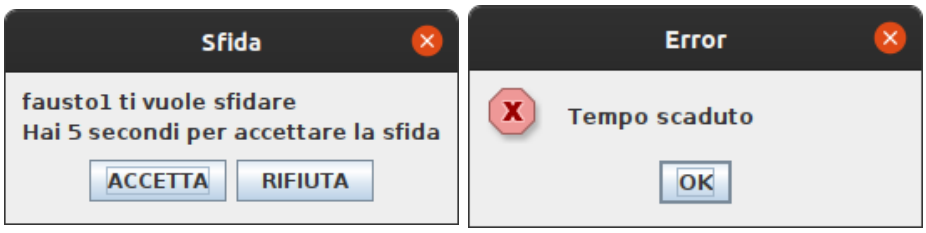
(a)

(b)

Figure 6: Home page (a)
Pop up che indica l'interruzione della connessione TCP (b)

4.2.6 UDPListener.java

Questa è una classe di tipo singleton.
Qui viene gestita tutta la parte riguardante la socket UDP per le richieste di sfida tra utenti. Una volta avviata, la classe si occupa di creare la socket e di aprirla sulla stessa porta su cui è stata aperta la socket TCP.
Non appena riceve un datagramma, che rispetta il protocollo definito (5.8), viene settato il flag di sfida della classe `ChallengeFlag.java`(4.2.7) e si mostra a video un pop up che permette all'utente di accettarla o meno. All'interno del pop up viene mostato chi è lo sfidante e il tempo massimo che ha per accettare la sfida.
Se la sfida viene accettata viene creato un oggetto della classe *Challenge.java* e la sfida parte, altrimenti il flag di sfida viene resettato.
Se invece l'utente accetta oltre il tempo previsto, verrà mostrato un altro pop up che indica l'avvenuto timeout.



(a) (b)

Figure 7: Richiesta di sfida (a)
Pop up che indica l'avvenuto timeout (b)

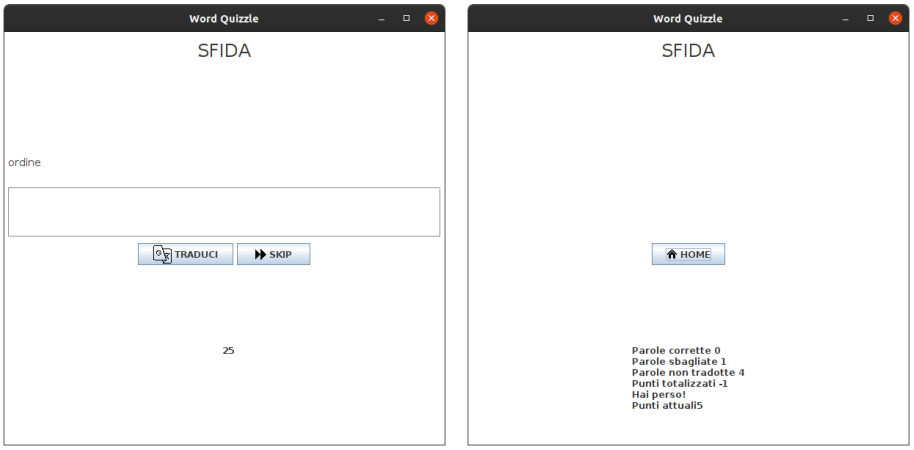
Il flag di sfida serve per evitare che l'utente riceva richieste mentre a sua volta è già impegnato in una sfida o ha già una richiesta di sfida in sospeso. Quando il flag è settato, tutte le nuove richieste sono rifiutate in automatico.
Una volta che l'utente effettua il logout, o viene disconnesso per un errore sulla connsessione TCP, il thread che gestisce questa classe e la socket UDP vengono chiusi.

4.2.7 ChallengeFlag.java

Classe singleton, per far si che il thread principale e quello UDP operino sulla medesima istanza.
Quando si invia o si riceve una richiesta di sfida, tale flag viene settato a 1 e una volta terminata la sfida o rifiutata la richiesta viene posto a 0.
Il flag è un `AtomicInteger` che garantisce il cambio di valore in maniera atomica, facendo si che il valore sia sempre consistente per entrambi i thread.

4.2.8 Challenge.java

Questa classe si occupa della gestione vera e propria della sfida.
Legge le parole in italiano da tradurre e invia al server le parole tradotte dall'utente. Permette anche di poter saltare la traduzione di una parola tramite l'apposito pulsante `SKIP`.
Nella parte inferiore è anche riportato il tempo massimo a disposizione per l'utente per terminare la sfida.
Finita la sfida, sempre nella zona inferiore, viene riportato un riassunto di quelle che sono le parole tradotte in modo corretto, errato o non tradotte. Viene riportato anche un riassunto di quello che è il punteggio guadagnato e quello attuale dell'utente.

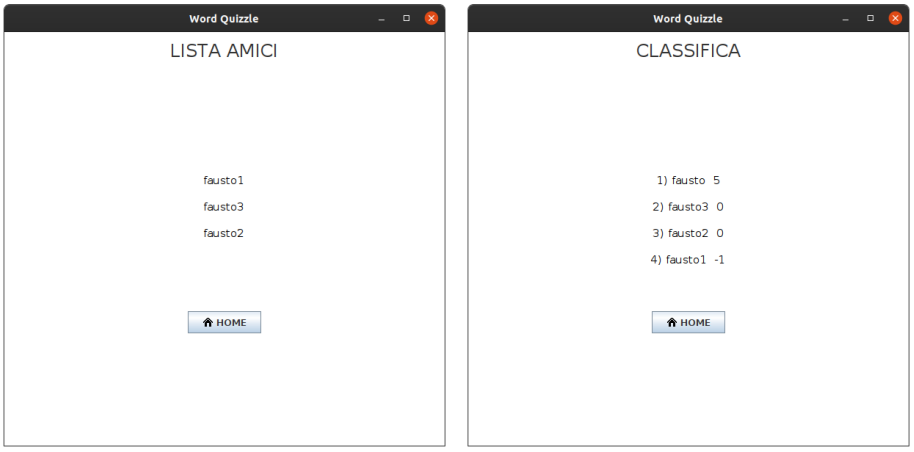


(a) (b)

Figure 8: Schermata durante la sfida (a)
Riassunto post sfida (b)

4.2.9 ShowFriends.java e ShowRank.java

Queste due classi hanno caratteristiche simili.
ShowFriends mosta la lista degli amici dell'utente, mentre la seconda mostra la classifica in base ai punti accumulati dall'utente e dai suoi amici con le varie sfide.



(a) (b)

Figure 9: Lista di amici (a)
Classifica (b)

4.3 Server

Prima di proseguire, c'è da fare una precisazione.
Le classi presenti nella directory `Task` sono classi le cui istanze sono eseguite da un thread del threadpool.
Sarà inoltre riportato nel titolo la directory in cui è contenuta la classe (`Directory/Classe`)

4.3.1 Database/DBMS.java

Questa classe si occupa di gestire gli utenti salvati in memoria.
È l'unica classe che legge e scrive sulla memoria.
Ogni utente è salvato all'interno cartella `Dati` e tutti i suoi dati sono scritti su un file json denominato secondo lo schema `nicknameUtente.json` (ad esempio fausto.json).
Si è optato per avere un file per ogni utente, anzichè uno contenente tutti gli utenti, per velocizzare tutte quelle operazioni semplici (come per esempio controllare se un utente è già registrato). Infatti, sfruttando la struttura ad albero del filesystem, la ricerca di un file relativa a un utente è effettuata in tempi brevi.
Per serializzare e deserializzare l'oggetto User, si fa uso della libreria *Gson*.
La classe presenta due metodi thread-safe: `existUser` e `registerUser`.
Questo perchè, in realtà, l'oggetto RMI per la registrazione bypassa lo user dispatcher, comunicando direttamente col DBMS.
Di conseguenza, per gestire la conccorrenza del server RMI, e tra RMI e user dispatcher, tali metodi sono stati resi `syncronized`.
Perciò una vera rappresentazione dell'architettura server è riportata nell'immagine in basso

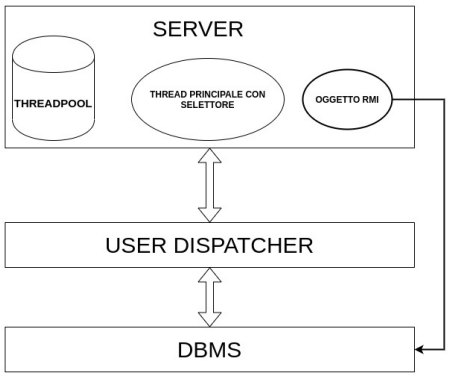


Figure 10: Architettura server reale

4.3.2 User/User.java

Questa è la classe che rappresenta l'utente. Al suo interno sono presenti dati come nickname, password, lista di amici e punteggio.

Poichè gli oggetti di tale classe saranno risorse condivise da più thread, il punteggio è un `AtomicInteger`.
Inoltre le istanze della classe sono "rientranti", ovvero che contengono un contatore denominato `use` (`AtomicInteger` anche lui) che viene incrementato quando l'istanza viene usata da un thread e decrementato quando non ne ha più bisogno.
Il motivo di tale scelta è riportato a seguire.

4.3.3 User/UserDispatcher.java

Questa classe, singleton, gestisce tutte quelle che sono le istanze degli utenti richiesti dal server a livello superiore, richiedendole al DBMS a sua volta qualora non risultassero presenti nella propria struttura dati.
La struttura dati usata è una `ConcurrentHashMap`.
La scelta è ricaduta su tale struttura per la velocità e l'efficienza offerta.
All'interno dello user dispatcher è presente anche un thread daemon cleaner, che viene risvegliato a intervalli di tempo regolari per ripulire la map da tutti quegli utenti il cui contatore `use` risulta azzerato.
Se si guarda bene il codice, si può notare che, nonostante la struttura dati apparentemente thread-safe, ci siano metodi `syncronized`.
Questo perchè, com'è riportato nella documentazione ufficiale, tale struttura dati ammette che alcune operazioni, come per esempio la `get` e la `delete`, possano avvenire in contemporanea. Di conseguenza per garantire la consistenza della map, si è dovuto optare per tale scelta.

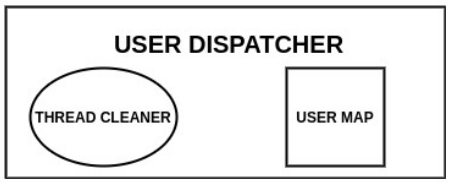


Figure 11: Architettura user dispatcher

4.3.4 Server/ConKey.java

Gli oggetti di questa classe vengono allegati alla key che viene generata quando un client si connette al server.
Contiene diverse informazioni e flag utili per la gestione delle request client-server e response server-client.
È presente anche un flag che indica se l'utente è già impegnato in una sfida, in modo da porre un ulteriore filtro alle richieste quando un utente ne sta già svolgendo una.

4.3.5 Server/ConChallenge.java

Gli oggetti di questa classe invece vengono allegati alle key generate quando i due sfidanti vengono registrati sul selettore di sfida.
Anche in questo caso sono presenti diversi flag utili per la gestione della sfida.

4.3.6 Server/DictionaryDispatcher.java

Lo scopo di questa classe, anch'essa singleton, è quello di scegliere un determinato numero di parole italiane casuali dal dizionario, richiedere la traduzione in inglese e ritornare la lista contenente le coppie {`parola italiana, traduzione inglese`}.
Per quanto riguarda la fase di traduzione, si è notato che alcune traduzioni restituite dal servizio *MyMemory* non erano esatte. Perciò è stato aggiunto un ulteriore servizio a supporto del primo. Il servizio è Yandex, che garantisce una traduzione più precisa.
Anche con tale servizio la richiesta di traduzione è effettuata tramite il metodo `Get` e la risposta è restituita in json.
Lo schema scelto è il seguente:

1. Richiedo la traduzione della parola a Yandex
2. Richiedo la traduzione della parola a MyMemory
3. Confronto la lista delle possibili traduzioni restituita da MyMemory con la traduzione fornita da Yandex.
   Se trovo una traduzione di *MyMemory* che combacia con quella di *Yandex* allora ritorno la prima taduzione, altrimenti la seconda.

Vengono gestiti diversi errori, tra cui il caso in cui uno dei servizi risultasse irraggiungibile.

4.3.7 Task/AddFriend.java

Il compito di questa classe è quello di aggiungere l'username di due utenti nelle rispettive liste di amici.
Prima di tutto vengono effettuati diversi controlli, come il caso in cui un utente provi ad aggiungersi tra gli amici.
Superati tali controlli, si acquista la lock dell'oggetto che rappresenta la lista di amici e si effettua l'operazione.

4.3.8 Task/ShowFriends.java

In un primo momento viene acquistata la lock sull'oggetto che rappresenta la lista di amici di un utente.
Successivamente viene creata una stringa json contente la relativa lista, in modo da essere inviata al client che ne ha fatto richiesta.

4.3.9 Task/ShowRank.java

Anche qui viene acquisita la lock sull'oggetto che rappresenta la lista di amici di un utente. In questo caso però vengono fatte delle richieste allo user dispatcher per ottenere gli oggetti associati ai nickname degli amici, in modo tale da ottenere il punteggio di ogni amico e creare una lista contenenti le coppie {`nickname,score`}.
Tale lista poi viene ordinata in maniera decrescente di punteggio e convertita in stringa json, in modo da essere spedita in risposta al client che ne ha fatto richiesta.

4.3.10 Task/ChallengeRequest.java

Tale classe è quella che si fa carico di tutta la parte pre-sfida.
Crea una socket UDP in modo da inviare la richiesta di sfida allo sfidato, secondo il protocollo stabilito (5.8).
Successivamente registra la key dello sfidato in un selettore temporaneo ed attende per un periodo di tempo predefinito.
Se non si riceve risposta entro l'arco di tale periodo allora la richiesta di sfida si considera rifiutata.
In caso contrario, si legge e si interpreta la risposta dello sfidato. Se la risposta è negativa, viene riportata allo sfidante e il tutto termina.
Se invece la risposta è positiva viene informato lo sfidante e si crea un'istanza della classe *Challenge.java*.
Vengono tenuti in considerazioni diversi casi, come per esempio il caso in cui lo sfidato si disconnette senza dare una risposta alla richiesta di sfida, oppure il caso in cui lo sfidante, una volta inviata la richiesta, si disconnette senza attendere la risposta dello sfidato.

4.3.11 Task/Challenge.java

Prima di avviare effettivamente la sfida, vengono eseguite determiante operazioni.
Innanzitutto si richiede la lista di parole da usare per la sfida alla classe *DictionaryDispatcher* (4.3.6). Se la lista viene restituita senza che si verifichino errori si prosegue registrando le key dei due sfidanti su un selettore creato appositamente per la sfida.
Alle nuove key generate viene allegata un'istanza della classe *ConChallenge* (4.3.5).
Terminata tutta questa fase di setup la sfida comincia.
Agli sfidanti viene comunicato il timer che indica il massimo tempo a disposizione per completare la sfida e poi, in successione, avviene uno scambio richieste-risposte tra client e server in cui il client invia la parola tradotta e il server invia la nuova parola da tradurre.
Se uno dei due sfidanti si disconnette, la partita continua comunque per l'altro utente.
Una volta che entrambi gli sfidanti hanno concluso le parole da tradurre, o è scattato il timeout, viene eseguita la parte finale della sfida.

Consiste nel contare le parole tradotte correttamente, quelle sbagliate e quelle non date e assegnare il relativo punteggio (2 punti per ogni parola corretta, -1 per quelle errate e 0 per quelle non tradotte). Il tutto viene inserito in una stringa e spedito al client come risposta finale.
Finita la sfida le due key vengono nuovamente registrate sul selettore principale del server.
La scelta è ricaduta nel selettore anche per la sfida poichè quello che conta è il numero di parole correttamente tradotte e non chi riesce a tradurle nel minor tempo possibile.
Si è perciò evitato dedicare un singolo thread per utente, risparmiando di conseguenza risorse.

4.3.12 RMIRegistrationImpl.java

Questa classe è l'implementazione dell'interfaccia definita al punto 4.1.1. L'istanza di tale classe è usata per l'operazione di registrazione.
Prima di richiamare i metodi opportuni offerti dal *DMBS* (come già detto nel paragrafo 4.3.1) si effettuano diversi controlli, come il caso in cui la password sia minore di 5 caratteri.
Ad ogni controllo è associato un codice di ritorno, definito nell'interfaccia stessa (5.1).

4.3.13 MainClassServer.java

Questa è la classe in cui è definito il main.
Si occupa di tutto il setup necessario per avviare il server.
In maniera sequenziale effettua le seguenti operazioni:

- Creazione del registry su una porta fissata ed esportazione dell'oggetto RMI su di esso.
- Creazione della socket TCP e associazione del selettore che fungerà da main selector
- Creazione di un threadpool

Se tutte le operazioni sono andate a buon fine, verrà creata un'istanza della classe *Server.java* a cui verrà passato sia il threadpool sia il selector.

```
[START] Server avviato
[START] Server RMI configurato
[START] Socket TCP configurata, in ascolto sulla porta 5000
[START] Threadpool avviato
[START] Server ready
```

Figure 12: Report che viene mostrato dopo aver terminato la fase di setup del server

Il threadpool è di tipo *newCachedThreadPool*. Il motivo di tale scelta sta nel fatto che il core principale del progetto è la sfida, perciò, rispetto a un

threadpool con un numero massimo di thread, qui si riesce a non imporre un limite superiore al numero di sfide che possono essere svolte in contemporanea. Un'altra scelta è il selettore anzichè il multithread.
Si è perseguita tale strada sempre considerando il fatto di sfruttare al meglio le risorse. Più nel dettaglio si è scelta tale architettura per il semplice fatto che, in un'architettura multithread dove ad ogni utente viene assegnato un thread, se l'utente non interagisce con l'interfaccia grafica si avrà un thread passivo che occupa semplicemente risorse.

4.3.14 Server.java

Questa classe costituisce il core del server.
Accetta le richieste di connessione dei client, legge le loro request e restituisce loro delle response secondo il protocollo stabilito (5).
In base alla richiesta ricevuta, decide se eseguire direttamente l'operazione (come per esempio il logout) oppure delegarla a un thread del threadpool (come per esempio ricavare la lista di amici di un determinato utente).
Le strutture dati usate in questa classe sono due, entrambe `ConcurrentHashMap`.
Una racchiude tutte le istanze degli utenti online ed è costituita dalle coppie {`username,User`}.
L'altra invece racchiude le chiavi associate agli utenti online. È composta dalle coppie {`username, keyUser`}.

5 Operazioni e protocollo di comunicazione

Le comunicazioni tra client e server prevedono una request e una response.
I messaggi di request sono strutturati secondo la regola: `OP\nMessage\n`.
Quelli di response sono strutturati nel seguente modo: `LEN_MSG\nESITO\nMessage\n`.
All'interno di `Message` viene inserita una stringa che da maggiori informazioni riguardo l'esito dell'operazione, mentre `LEN_MSG` indica la lunghezza che va da `ESITO` fino all'ultimo `\n`.

5.1 Registrazione

Questa è l'unica operazione che differisce nel protocollo di comunicazione client-server.
Il metodo offerto per la registrazione ritorna un codice che indica l'esito dell'operazione:

- `INVALID_NICK = -101`: Il nickname passato non è valido
- `EXISTS_NICK = -102`: Il nickname passato risulta già registrato
- `INVALID_PWD = -103`: La password passata non è valida
- `TO_SHORT_PWD = -104`: La password passata è troppo corta. La lunghezza minima è di 5 caratteri.
- `TOO_LONG_PWD = -105`: La password passata è troppo lunga. La lunghezza massima è di 20 caratteri.
- `SPACE_IN_NICK = -106`: Il nickname contiene spazi
- `GENERAL_ERROR = -1`: Errore generico
- `OK = 1`: Operazione andata a buon fine.

5.2 Login

- Request: `LOGIN\nNickname\nPassword\n`
- Response:
  - `LEN_MSG\nOK\nMessage\n` se l'operazione ha avuto successo.
  - `LEN_MSG\nKO\nMessage\n` altrimenti.

5.3 Logout

- Request: `LOGOUT\nNickname\`
- Response: `LEN_MSG\nOK\nMessage\n`

5.4 Aggiungi amico

- Request: `ADDFRIEND\nNickname\nNicknameFriend\n`
- Response:
  - `LEN_MSG\nOK\nMessage\n` se l'operazione ha avuto successo.
  - `LEN_MSG\nKO\nMessage\n` altrimenti.

5.5 Lista amici

- Request: `SHOWFRIENDS\nNickname\n`
- Response: `LEN_MSG\nOK\nJsonMessage\n`

5.6 Mostra punteggio

- Request: `SHOWSCORE\nNickname\n`
- Response: `LEN_MSG\nOK\nMessage\n`

5.7 Mostra classifica

- Request: `SHOWRANK\nNickname\n`
- Response: `LEN_MSG\nOK\nJsonMessage\n`

5.8 Challenge

Protocollo in cui un utente richiede di cominciare una sfida:

- Request: `CHALLENGE\nNickname\nNicknameFriend\n`
- Response:
  - `LEN_MSG\nOK\nMessage\nItalianWord\nChallengeTimer\n` se la sfida è stata accettata.
  - `LEN_MSG\nKO\nMessage\n` altrimenti.

Protocollo in cui il server avverte un utente di una richiesta di sfida

- Request: `CHALLENGE\nNicknameSfidante\nTimerT1\n`
  dove `TimerT1` indica il tempo massimo che lo sfidato ha per accettare la sfida.
- Response:
  - `ACCEPTED\nMessage\n` se la sfida viene accettata.
  - `REFUSED\nMessage\n` altrimenti.

Nel caso in cui la sfida venisse accettata, verrebbe inviato un messaggio di conferma `LEN_MSG\nOK\nMessage\nItalianWord\nChallengeTimer\n` anche allo sfidato.

6 Comandi per compilare ed eseguire il progetto

Il progetto è stato costruito usando Maven. Per poter compilare ed eseguire il progetto vi sono due modi:

- Usare l'IDE Intellij
- Usare il terminale

6.1 Da terminale su Ubuntu

È necessario installare Maven tramite il comando da terminale:
`sudo apt-get install maven`.

Una volta installato, recarsi nella cartella `Costanti` del server (**??**), commentare la riga 27 e decommentare la riga 28.
Fare la stessa cosa con la cartella `Costanti` del client, commentando la riga 5 e decommentando la riga 6.
Fatto ciò aprire un terminale e recarsi nella cartella del progetto.
Eseguire prima lo script: `./compile&server.sh` per compilare ed eseguire il server.
Successivamente aprire un altro terminale, recarsi nella cartella del progetto ed eseguire lo script `./client.sh` per far partire un client.
Ripetere l'ultima operazione tante volte quanti client si vogliono avviare.

6.2 Intellij

È possibile scaricare la versione gratuita del programma ed è presente per tutti i sistemi operativi.

Per Windows basta andare su link e scaricare la versione free open-source.

Per quanto riguarda Linux ci sono diverse modalità:

- Digitare `Intellij Community` su *Ubuntu software center*.
- Scaricare la versione free open-source al seguente link.
- Digitare sul terminale il comando:
  `sudo snap install intellij-idea-community --classic`

Una volta installato il programma basterà aprirlo, selezionare dalla schermata principale `Import project` e selezionare il path dove si trova il progetto `word-quizzle`.

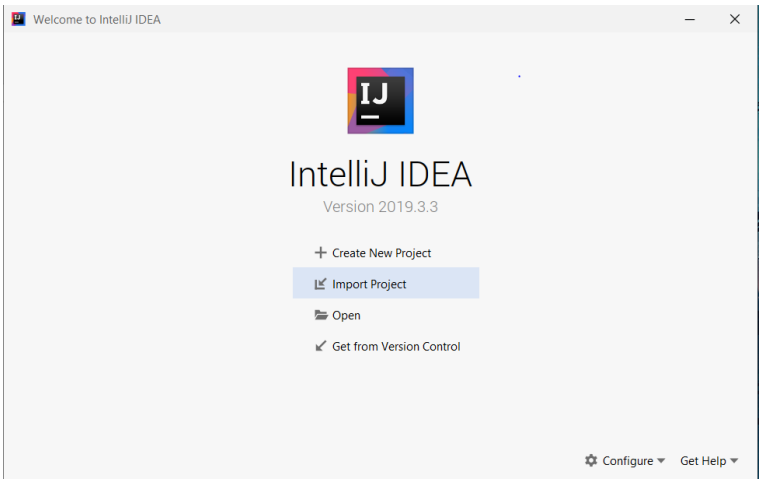


Figure 13: Schermata iniziale di intellij

Nella schermata che compare, selezionare `Import project from external model` e selezionare `Maven`, come riportato nella foto in basso. Successivamente cliccare su `Finish`.

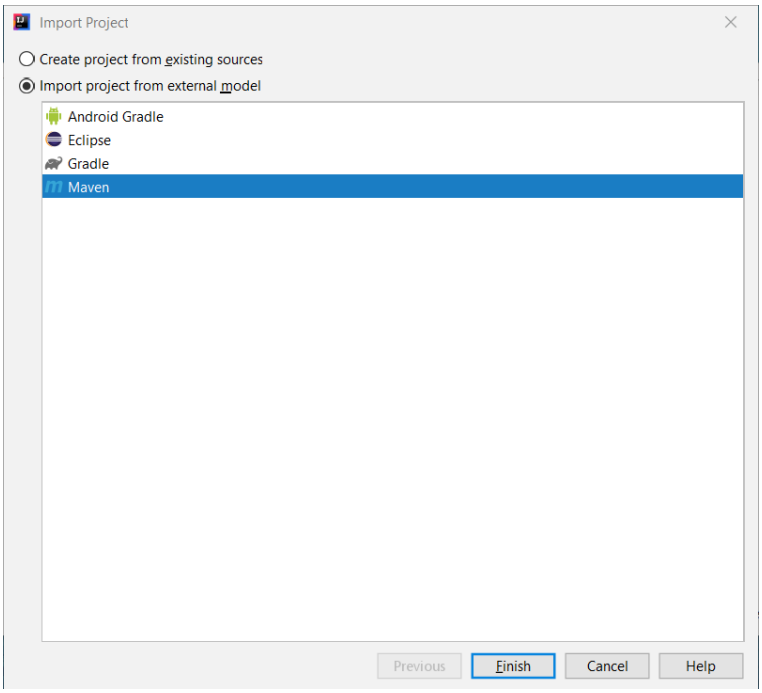


Figure 14: Importare il progetto con maven

Terminati i passaggi iniziali, verrà mostrato l'IDE.
Da qui, prima cliccare sul bottone in alto a destra con su scritto *Maven* e successivamente premere sulle freccie che formano un cerchio. Questo consentirà a maven di costruire e scaricare tutto ciò che è necessario per il progetto.
I passaggi sono illustrati nella foto sotto.

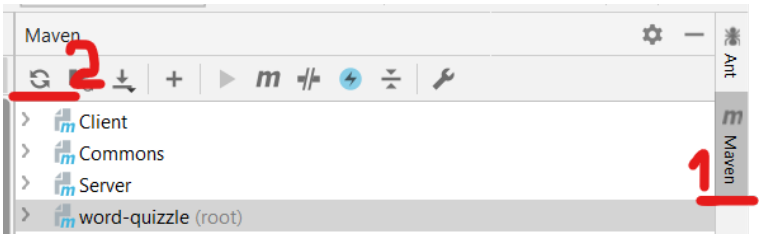


Figure 15: Build del progetto

Una volta terminato il download di tutto ciò che è necessario a maven, cliccare sul pulsante in alto a sinistra `Project`, premere sulla tendina `Project` come riportato nell'immagine e selezionare `Packages`.

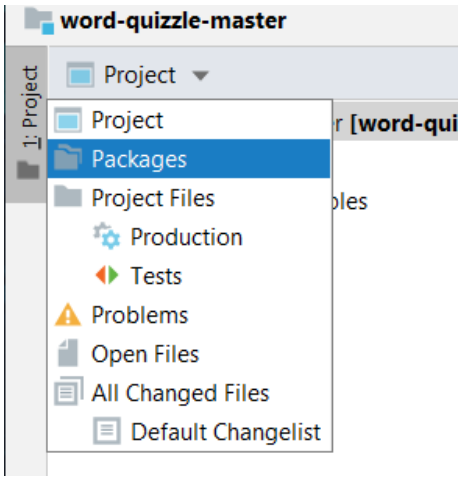
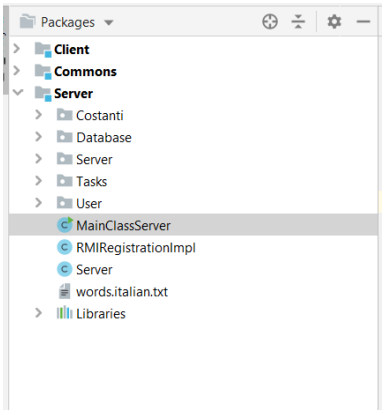
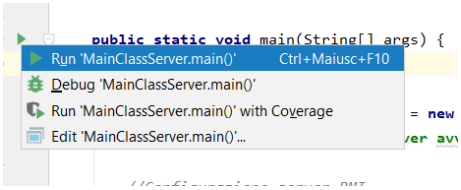


Figure 16: Selezione package

Fatto ciò, avviare prima il server.
Aprire la directory *Server*, fare doppio click sulla classe *MainClassServer* e cliccare sul pulsante verde che compare alla sinistra del metodo main. Da qui scegliere 'Run'.



(a)



(b)

Figure 17: Directory server (a)
Avviare il main del server (b)

Successivamente ripetere la stessa cosa per il client.
Aprire la directory *Client* e fare doppio click su `MainClassClient`, cliccare il

pulsante verde a fianco del main e scegliere 'Run'.
Si aprirà solo un client e per poterne aprire un altro è necessario duplicare la configurazione.
Per farlo, recarsi nella parte superiore della barra, sulla destra, e cliccare `Edit Configurations...`, come mostrato nella figura in basso.

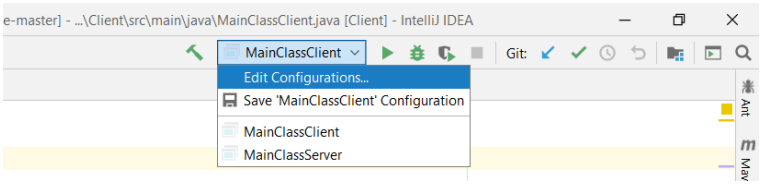


Figure 18: Duplicazione client

Si aprirà una finestra.
Da qui basta selezionare la configurazione MainClassClient e premere la sequenza di tasti CTRL+D o il pulsante segnato in rosso. Ripetere l'operazione tante volte quante copie dei client che si vogliono creare.
Finito il tutto clicchiamo su `Apply`.

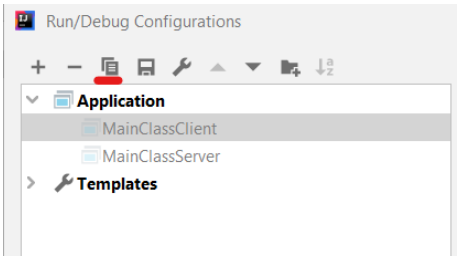


Figure 19: Build del progetto

Per avviare le varie istanze basta scegliere dal menù a tendina le varie configurazioni create e premere sul pulsante verde di play posto sulla destra.

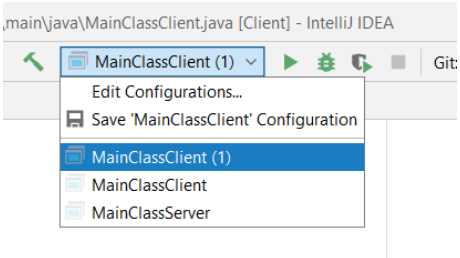


Figure 20: Avvio client