

Progetto RCL - Word Quizzle

Niccolò Puccinelli - 559604

a.a 2019/2020

Indice

1	Architettura generale	3
1.1	Lato server	3
1.1.1	Selector principale	3
1.1.2	Selector sfida	4
1.1.3	Gestione file JSON	4
1.1.4	Gestione utenti	5
1.1.5	Dizionario e traduzioni	5
1.2	Lato client	6
1.2.1	Schermata iniziale	6
1.2.2	Schermata home	7
1.2.3	Schermata di sfida	8
1.2.4	Schermata di gioco	9
2	Gestione dei thread e strutture dati	11
2.1	Gestione della concorrenza	11
2.2	La classe Utente()	11
2.3	La classe Attachment()	11
3	Descrizione delle classi (Server)	12
3.1	Gestione operazioni	12
3.1.1	Server	12
3.1.2	OperazioniServer	12
3.2	Gestione sfida	12
3.2.1	ThreadSfida	12
3.2.2	Sfida	12
3.2.3	TimerSfida	13

4	Descrizione delle classi (Client)	13
4.1	Client e WQGUI: Registrazione e Login	13
4.1.1	Registrazione	13
4.1.2	Login	13
4.2	LoggedGUI: Operazioni varie	13
4.3	StartGUI e GameGUI: Sfida	14
4.3.1	StartGUI	14
4.3.2	GameGUI	14
5	Istruzioni	15

1 Architettura generale

Il progetto si compone di due entità principali che comunicano tra loro, Server e Client. Tale comunicazione avviene, a seconda dei casi, tramite RMI, UDP e TCP.

1.1 Lato server

Il server è costituito da due selector: uno per la gestione delle operazioni principali e uno per la gestione della sfida.

1.1.1 Selector principale

OP_ACCEPT: Dopo aver accettato la connessione, viene registrata la chiave in lettura, cui viene attaccato un buffer da cui leggere/scrivere i dati.

OP_READ: La lettura è effettuata nel seguente modo: ogni stringa da leggere è passata dal client al server con `\n` concatenato. La lettura prosegue finché il messaggio letto non contiene `\n`. In tal caso, significa che ha finito di leggere la stringa.

Una volta terminata la lettura, il messaggio viene passato alla classe *OperazioniServer*. Un `CachedThreadPool` si occupa della gestione delle richieste, al termine di ognuna delle quali viene restituito un esito. Tale esito è una stringa terminata dal carattere `\n`, in modo da poter essere letto dal metodo `readLine()` nel client. L'esito è scritto nell'attachment della chiave, il cui `interestops` viene settato ad `OP_WRITE`.

Nel caso sia stata mandata una richiesta di sfida tramite UDP, oltre alla scrittura dell'esito si provvede alla creazione di un Thread `t` che si occuperà della gestione della sfida, attraverso la classe *Sfida*. Se entro 15 secondi non si riceve un pacchetto di risposta UDP dall'utente sfidato, la sfida è considerata rifiutata.

Viene infine risvegliato il selector principale con una `wakeup()`.

OP_WRITE: L'esito dell'operazione viene letto dal buffer e scritto sul Socket-Channel. Per controllare la terminazione della scrittura viene usato il metodo `hasRemaining()`.

Se l'esito contiene la stringa "Accettata", significa che è stata accettata una richiesta di sfida mandata in precedenza. Viene quindi settato a 0 l'`interestOps` della chiave, così da essere ignorata dal selector. In caso contrario la chiave viene rimessa in `OP_READ`, pronta per una nuova richiesta.

1.1.2 Selector sfida

Viene inizialmente eseguito il controllo sull'interestOps dei 2 sfidanti, in modo da iniziare la sfida solo quando questo equivale a 0.

Il ciclo termina quando la variabile booleana stop diventa true, ovvero quando il numero di giocatori è 0 (int inGame, inizialmente settato a 2).

Le chiavi sono poi registrate in OP_READ e viene loro associato un attachment di tipo Attachment, che oltre ad un buffer per leggere e scrivere le parole contiene due campi utili per la gestione del timer (Timer time e AtomicBoolean t). Per ogni chiave viene infatti fatto partire un timer (contenuto in Attachment, così da essere diverso per ogni chiave) in modo che, passati 60 secondi, cancelli la key (non prima di aver mandato il messaggio di tempo scaduto al client).

OP_READ: Prima di tutto si effettua il controllo dell'AtomicBoolean contenuto nell'attachment della chiave. Se equivale a true, significa che sono passati 60 secondi e la chiave viene subito messa in **OP_WRITE**, altrimenti si esegue la lettura del buffer con lo stesso metodo del selector principale (attraverso la verifica della presenza del carattere \n).

Viene analizzato il messaggio: se contiene la stringa "Inizio", significa che deve ancora essere inviata la prima parola, altrimenti il messaggio letto viene confrontato con la giusta traduzione corrispondente. Se il confronto dà esito positivo, si prosegue con l'aumento del punteggio (+3).

Successivamente viene scritta la prossima parola da tradurre nel buffer associato alla chiave (o la parola "Fine", nel caso le parole siano finite). La chiave viene quindi messa in **OP_WRITE**.

OP_WRITE: Si controlla il valore dell'AtomicBoolean contenuto nell'attachment della chiave. Se è true, allora viene scritta nel channel la stringa "Tempo scaduto" (verifica terminazione mediante metodo `hasRemaining()`), viene decrementata la variabile inGame e cancellata la chiave. Se inGame == 0, stop assume valore true e il ciclo termina.

Se il timer non è ancora scaduto, si prosegue con la scrittura del buffer nel channel di comunicazione col client. Se il messaggio contiene la parola "Fine", viene decrementata la variabile inGame e cancellata la chiave. Se inGame == 0, stop assume valore true e il ciclo termina. Diversamente, se "Fine" non è presente all'interno del messaggio, allora la sfida deve continuare e la chiave viene rimessa in **OP_READ**, pronta per leggere la prossima traduzione inviata dal client.

Una volta terminata la sfida, viene risvegliato il selector principale e le chiavi dei due sfidanti vengono reimpostate ad **OP_READ**.

1.1.3 Gestione file JSON

Le informazioni relative agli utenti (nome, password, punteggio e lista degli amici) vengono mantenute in un file json e in una `ConcurrentHashMap<String, Utente>`

UserDB, in cui sono presenti ulteriori informazioni da non memorizzare nel json, ovvero il numero di porta per l'attesa di messaggi UDP e un booleano (logged) che equivale a true se l'utente è loggato.

L'aggiornamento del json avviene attraverso la cancellazione del file precedente, sostituito dalla versione aggiornata. Se non è presente il file JsonDB.json, la creazione è effettuata al lancio del server.

Per gestire le conversioni in json è stata utilizzata la libreria gson, in particolare i metodi `fromJson()` e `toJson()`.

1.1.4 Gestione utenti

Gli utenti vengono memorizzati al momento della registrazione in una `ConcurrentHashMap<String,Utente>`, chiamata UserDB, in cui la chiave è rappresentata dal nome dell'utente e il valore da un oggetto della classe *Utente*. Ad un oggetto di tipo *Utente* sono associati il nome, la password, il punteggio, la lista degli amici, il numero di porta UDP e un booleano che indica se l'utente è loggato o meno. Il database degli utenti viene creato all'avvio del server con l'interpretazione del json. Se quest'ultimo non esiste, allora viene creato e UserDB verrà inizializzato ad un valore diverso da null una volta registratosi il primo utente.

Il server mantiene inoltre una `LinkedBlockingQueue<String>` per controllare se un utente sia già impegnato in una sfida e una `ConcurrentHashMap<String,SocketChannel>` in cui sono memorizzati i channel di comunicazione di ogni utente, in modo da poter settare l'interestOps della chiave a 0 all'inizio della sfida e rimetterlo ad `OP_READ` alla fine.

1.1.5 Dizionario e traduzioni

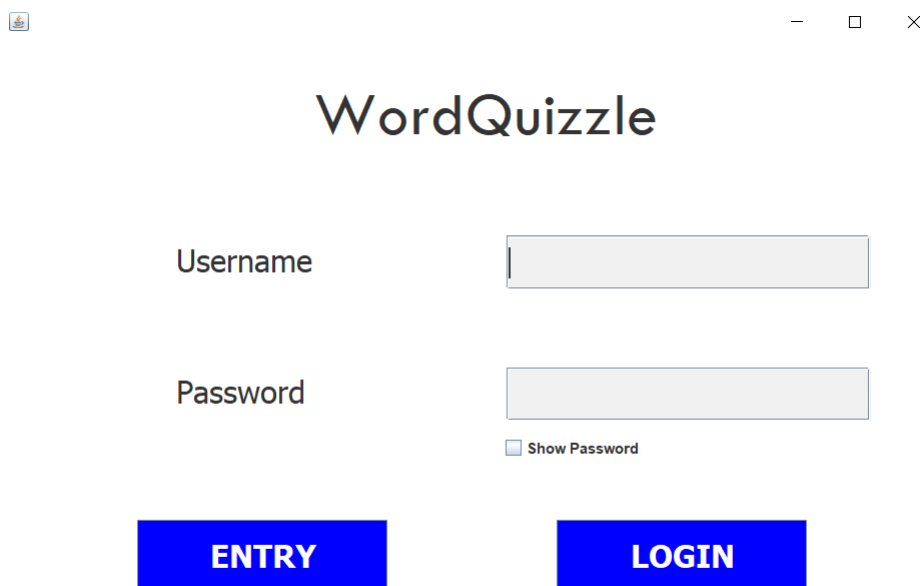
All'avvio della sfida, il server legge 13 parole in italiano da un dizionario in modo casuale e le inserisce in un `ArrayList<String>`. Viene successivamente invocato il metodo `itaToEng(String itaWord)`, che si occupa della traduzione delle parole dall'italiano all'inglese attraverso una richiesta GET effettuata ad un sito esterno (<https://mymemory.translated.net/doc/spec.php>). Se il sistema non è connesso ad Internet viene sollevata una `IOException`, altrimenti la traduzione viene restituita al chiamante (dopo aver effettuato il parsing). Il server inserisce quindi ogni parola tradotta dall'italiano all'inglese in un `ArrayList<String>` parallelo a quello usato per memorizzare le parole in italiano.

1.2 Lato client

Il client è stato implementato mediante una semplice interfaccia grafica (libreria *Swing*), divisa principalmente in 4 parti: la schermata iniziale di registrazione/login, la home con tutte le operazioni eseguibili, la schermata di sfida per inviare e ricevere richieste e la schermata di gioco vera e propria.

Per uscire dal gioco è necessario essere nella schermata iniziale. Questo per evitare problemi dovuti alla chiusura delle finestre mentre sono in corso delle operazioni.

1.2.1 Schermata iniziale



WordQuizzle

Username

Password

☐ Show Password

ENTRY **LOGIN**

Prima dell'avvio vero e proprio dell'interfaccia grafica viene eseguito il lookup dell'oggetto remoto e l'apertura di un channel TCP con cui il client comunicherà le proprie richieste al server.

La GUI iniziale propone all'utente di registrarsi o loggarsi con username e password. La registrazione avviene tramite RMI, il login tramite comunicazione TCP con il server sul SocketChannel aperto precedentemente.

Durante la fase di login viene inoltre creato un DatagramSocket UDP, con conseguente bind sulla prima porta trovata libera dal sistema, a partire dalla numero 6780. Questo per permettere l'ascolto di eventuali richieste di sfida (classe *ThreadSfida*) attraverso messaggi UDP.

Se il login va a buon fine, viene inizializzata la schermata home.

1.2.2 Schermata home



Al momento dell'inizializzazione viene settato a true il valore `logged` e fatto partire un thread che riceve come task un'istanza della classe *ThreadSfida*, incaricata di gestire le sfide ricevute da parte di altri utenti. In particolare, fino a quando il valore di `logged` precedentemente settato equivale a true, il thread si mette in attesa di pacchetti UDP contenenti la richiesta di sfida tramite una *receive*. Se un pacchetto viene ricevuto, tale thread invoca il metodo *sfidaRicevuta()* della schermata di sfida, che si occupa della gestione della richiesta.

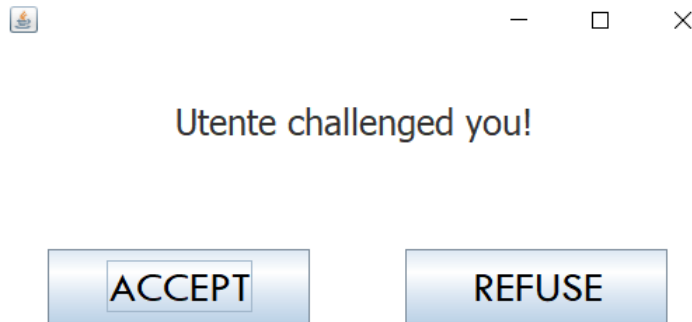
La GUI home propone all'utente di scegliere fra varie operazioni: "ADD FRIEND", "FRIENDS LIST", "SCORE", "RANKING", "LOGOUT".

Per ognuna di queste, viene inviato un messaggio al server con una *write* (controllo terminazione con *hasRemaining()*), che restituirà un esito corrispondente a un codice di successo/fallimento o ad una particolare stringa richiesta dal client (es: classifica, lista degli amici).

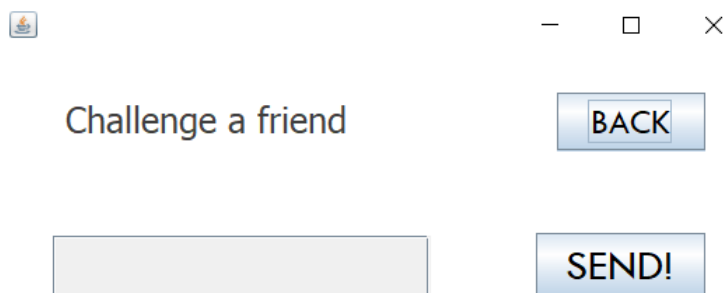
Se l'operazione scelta è invece "PLAY!", si procede con l'invocazione del metodo *inviaSfida()* della schermata di sfida, che si occuperà dell'invio di una richiesta ad un particolare utente. Viene nascosta la schermata home.

1.2.3 Schermata di sfida

Questa schermata differisce a seconda che la richiesta di sfida sia stata inviata oppure ricevuta.

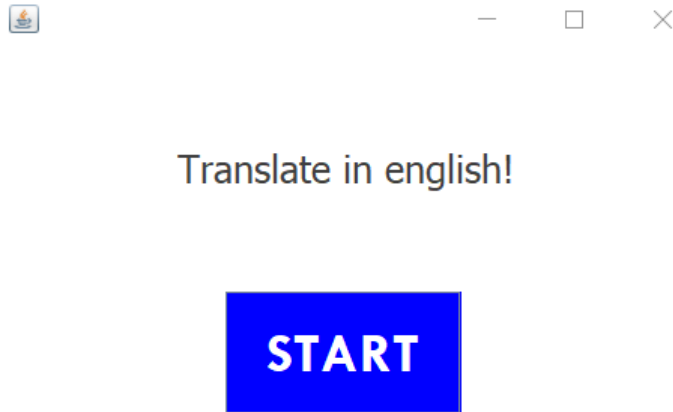


- Sfida ricevuta: Viene inizialmente settato a true un booleano ("ricevuta"), che indica l'avvenuta ricezione della sfida. Questo per evitare che un utente possa inviare una sfida prima di aver deciso se accettare o rifiutare una richiesta pendente. L'utente può decidere se accettare o rifiutare di giocare. In entrambi i casi, la decisione viene mandata al server tramite messaggio UDP. Se l'utente ha accettato, il server fa partire il thread che gestisce la sfida, mentre il client inizializza la schermata di gioco. In caso di rifiuto, si viene riportati alla schermata home.



- Sfida inviata: Per inviare una sfida è sufficiente inserire il nome dell'amico da sfidare e cliccare sul tasto "SEND!". Viene controllato il valore del booleano "ricevuta" per evitare il problema sopracitato e, se risulta uguale a false, si procede con la scrittura della richiesta sul channel di comunicazione col server. Se la stringa di risposta inviata dal server e letta dal client contiene la scritta "Accettata", significa che l'altro utente ha accettato la sfida e il client inizializza la schermata di gioco.

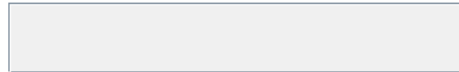
1.2.4 Schermata di gioco



Una volta premuto sul bottone "START!", il client invia al server la parola "Inizio", legge la prima parola in italiano inviata dal server e chiede all'utente di inserire la traduzione corrispondente. La traduzione viene inviata al server e quest'ultimo invia un'altra parola da tradurre. Le letture/scritture di traduzioni proseguono fino allo scadere del timer o alla fine delle parole, dopo di che vengono mostrati i risultati e la schermata home è resa di nuovo visibile.



articolo



3/13

49 seconds left



E' stato implementato anche un timer lato client, così da cliccare automaticamente sul pulsante "SEND!" un secondo dopo lo scadere del timer lato server (metodo `doClick()`). Il server riceve un messaggio vuoto a tempo scaduto, quindi notifica il client dell'evento, il quale mostra così i risultati finali.

2 Gestione dei thread e strutture dati

2.1 Gestione della concorrenza

La gestione della concorrenza è effettuata mediante l'utilizzo della classe `ConcurrentHashMap` per la memorizzazione dei dati relativi agli utenti e dei `SocketChannel` su cui avviene la comunicazione TCP. Inoltre i dati potenzialmente rischiosi sono acceduti con metodi `synchronized`.

Per quanto riguarda la concorrenza nello svolgimento delle operazioni, vengono utilizzati due selector (vedi 1.1.1 e 1.1.2) e un `cachedThreadPool` che gestisce richieste multiple da parte del client.

2.2 La classe `Utente()`

`Utente` è una struttura dati che contiene i dati relativi agli utenti registrati (vedi 1.1.4) e vari metodi `set()`/`get()` per modificare/ottenere tali dati. I campi `port` e `logged` sono stati dichiarati `transient` per non essere memorizzati all'interno del file `json`.

2.3 La classe `Attachment()`

E' stata definita una nuova struttura dati `Attachment` da associare alle chiavi del selector della sfida, contenente un `ByteBuffer` su cui scrivere/leggere dati e altri 2 campi per la gestione del timer lato server (vedi 1.1.2).

3 Descrizione delle classi (Server)

3.1 Gestione operazioni

3.1.1 Server

All'inizio viene effettuata la lettura del file json, i cui dati vengono inseriti nel database degli utenti (vedi 1.1.3). Successivamente si procede con il bind dell'oggetto remoto e la gestione della richiesta attraverso il selector principale (vedi 1.1.1). La richiesta viene passata ad un `cachedThreadPool()`, a cui passo un'istanza della classe *OperazioniServer* come task. La classe *Server* contiene inoltre i vari metodi per eseguire le operazioni richieste dal client, invocati dalla classe *OperazioniServer*.

3.1.2 OperazioniServer

Classe in cui viene elaborato il messaggio passato dalla classe *Server*. La prima parola del messaggio determina in quale `case` dello `switch` continuare l'esecuzione (vedi 1.1.1). Una volta fuori dallo `switch`, viene risvegliato il selector principale e il controllo passa di nuovo alla classe *Server*, eventualmente dopo aver concluso una sfida (in questo caso il controllo passa prima alla classe *Sfida*).

3.2 Gestione sfida

3.2.1 ThreadSfida

Thread avviato subito dopo il login dell'utente. Si mette in attesa di un messaggio di richiesta di sfida UDP da parte di un altro utente e, una volta ricevuto, chiama il metodo `sfidaRicevuta()` (vedi 1.2.2 e 1.2.3).

3.2.2 Sfida

Dopo aver tradotto le parole (vedi 1.1.5) e aver gestito la sfida con un selector (vedi 1.1.2), il programma esegue il calcolo del punteggio (+5 al vincitore finale) e invia l'esito della sfida ai due sfidanti con una write sulle loro sockets. I due utenti vengono quindi rimossi dalla `LinkedBlockingQueue` `busyUsers`, in cui erano stati aggiunti all'inizio della sfida per evitare sfide multiple in corso (vedi 1.1.4).

3.2.3 TimerSfida

Classe il cui compito è, una volta invocata, quello di settare a true una variabile booleana, utile per il termine della sfida in caso di tempo scaduto (vedi 1.1.2).

4 Descrizione delle classi (Client)

4.1 Client e WQGUI: Registrazione e Login

4.1.1 Registrazione

La registrazione è implementata mediante RMI. Il metodo `entryUser(String nickUtente, String password)` presente nella classe `Server` permette di registrare un utente, grazie all'oggetto remoto `"USER_SERVER"`. Tale oggetto è definito nella classe `ServiceInterface`.

In caso di esito positivo, l'utente viene aggiunto al database `UserDB` e il file `json` viene aggiornato nelle modalità sopra descritte. Se l'utente è già registrato viene visualizzato un messaggio di errore.

4.1.2 Login

Il server controlla la correttezza di `user` e `password`, che l'utente sia registrato e che non sia già loggato. In caso di esito positivo, scrive su `UserDB` la porta UDP dell'utente (vedi 1.2.1), aggiorna il campo `logged` dell'utente a `true` e aggiunge il corrispondente `channel` ad un array di `SocketChannel`, che servirà poi per la sfida.

Viene quindi visualizzato un messaggio di avvenuto login e aperta la schermata home del gioco (*LoggedGUI*), dalla quale è possibile aggiungere amici, sfidarli, visualizzare la classifica, il proprio punteggio e la lista di amici e fare logout.

Una volta loggati sarà necessario fare logout per chiudere le finestre della GUI, in modo da evitare comportamenti indesiderati durante la gestione delle richieste del client da parte del server.

4.2 LoggedGUI: Operazioni varie

Una volta inizializzata la schermata e lanciato il thread per gestire le richieste di sfida (vedi 1.2.2), il menù propone di scegliere fra varie opzioni, per ognuna delle quali il client scrive al server una stringa costituita dal nome dell'operazione e da uno o più parametri utili per il soddisfacimento della richiesta.

L'esito della richiesta viene infine visualizzato a schermo.

- "LOGOUT": Viene richiesto il logout e si ritorna alla schermata iniziale. La connessione UDP (in attesa per ricevere sfide) è chiusa e l'attributo `logged` è settato a `false`.
- "ADD FRIEND": E' possibile aggiungere un utente alla propria lista di amici, a meno che non sia già presente o non sia registrato.
- "FRIENDS LIST": Il server invia la `GList` contenente la lista degli amici dell'utente, che viene deserializzata (metodo `fromJson()`) e stampata a schermo.
- "SCORE": Si richiede al server il punteggio totale dell'utente, che viene poi stampato a schermo.
- "RANKING": Il server invia la `GList` contenente la classifica dei punteggi, che viene deserializzata (metodo `fromJson()`) e stampata a schermo.
- "PLAY!": Viene inizializzata la GUI per l'invio della sfida (vedi 1.2.2 e 1.2.3) e nascosto il `JFrame` corrispondente alla schermata home.

4.3 StartGUI e GameGUI: Sfida

4.3.1 StartGUI

A seconda che la sfida sia stata ricevuta o inviata, viene visualizzata una differente schermata (vedi 1.2.3).

In caso di sfida ricevuta appare una finestra che chiede se accettare o meno la richiesta. A risposta negativa corrisponde un ritorno al `JFrame` *LoggedGUI*, altrimenti si prosegue con l'avvio della *GameGUI*.

In caso di sfida inviata invece, il client analizza l'esito ricevuto dal server e, se negativo, restituisce un messaggio di errore (ad esempio: timer scaduto, richiesta rifiutata o utente già occupato). Se l'esito è positivo viene avviata la *GameGUI*.

4.3.2 GameGUI

Interfaccia grafica che si occupa di leggere le parole immesse dall'utente per poi inviarle al server attraverso `SocketChannel`. Una volta finite le parole da tradurre o, in alternativa, allo scadere del timer, l'esito finale della sfida viene notificato ai due utenti attraverso un messaggio a schermo e si viene riportati alla schermata home (*LoggedGUI*).

5 Istruzioni

Il progetto è stato sviluppato interamente su Eclipse. Entrare nella cartella Word Quizzle. In "src" si trovano i file .java. Eseguire prima la classe Server e poi una o più classi Client. Si aprirà una finestra con una semplice interfaccia grafica, con la quale è possibile interagire per richiedere l'esecuzione delle operazioni descritte sopra.