RELAZIONE PROGETTO DI RETI DI CALCOLATORI E LABORATORIO

DI ALESSIO BREGLIA, MATRICOLA 536164

WORD QUIZZLE 2019/2020

1. Introduzione e principali scelte di implementazione

Word Quizzle è un sistema di sfide di traduzioni italiano-inglese tra gli utenti registrati al servizio. Il server è un server *multithreaded* in cui ogni richiesta da parte di un client viene gestita attraverso un thread separato. Ho deciso di svilupparlo come *multithread* invece di utilizzare il *multiplexing NIO* poiché, secondo me, ciò rende molto più semplice la gestione dell’interfaccia grafica (che analizzeremo nel seguito). Per il salvataggio dei dati degli utenti (nome, password, lista amici, punteggio) nei file ho usato la libreria *simple.json* la quale, pur non essendo thread-safe, non crea dei conflitti poiché i thread sono separati gli uni dagli altri (non esiste più di un thread che salva i dati su uno stesso file), eliminando così eventuali problemi di sincronizzazione. Inoltre, i file JSON vengono modificati ogni volta ce ne sia bisogno (ad esempio per aggiornare un punteggio o per aggiungere un amico). Ho utilizzato il metodo *Thread.sleep()* per mettere un thread in attesa di un comando da parte dell’utente oppure in attesa che l’utente invii qualcosa (come ad esempio un amico da aggiungere, oppure di una parola tradotta); l’ho utilizzata poiché non sono variabili condivise su cui far aspettare i vari thread, ma eventi. I thread delle sfide sono stati implementati attraverso la classe *ThreadPoolExecutor* con un *NewCachedThreadPool*, poiché non ho voluto mettere un vero e proprio limite sul numero di sfide che si possono eseguire in contemporanea; inoltre così facendo riutilizzo i thread che hanno terminato la propria esecuzione. Nel task passato all ThreadPoolExecutor sono passati come parametri il nome dello sfidato, gli ArrayList *TranslatedWords* contenente le parole tradotte della sfida e *arrWords* contenente le parole da tradurre (selezionate dalla funzione *PrepareArrays*), e le porte TCP dello sfidante e dello sfidato (per inviare le parole da tradurre a ciascuno di essi).Il timer per la sfida è stato impostato a 60 secondi; per ogni parola corretta assegno +3 punti, per ogni parola sbagliata assegno -2 punti. Il punteggio salvato nel file dell’utente non può mai essere sotto lo 0 (sia per non interferire con i vari codici di errore presenti, sia perché avere punteggi minori di 0 non mi sembrava stilisticamente corretto). Nel caso in cui un client chiuda la finestra per l’invio della parola, gli viene assegnato un punteggio di 0. Le parole vengono inviate dal thread gestore della sfida ai client attraverso una connessione TCP, affinché sia il protocollo stesso a gestire ritardi e perdita di eventuali pacchetti. All’interno del thread gestore della sfida ho implementato un *selector* utilizzando la classe *ServerSocketChannel* di Java NIO, in modo tale da avere delle *readline()* non bloccanti, ed è usato solo in lettura. L’invio delle parole ai due partecipanti, come abbiamo già detto, avviene mediante due connessioni TCP distinte, una per ciascuno dei due.

Per ogni client, dopo aver eseguito la login, viene attivato un *thread listener* per le sfide. Esso si blocca sul metodo *receive()* in attesa che gli arrivi un DatagramPacket contenente o il nome dello sfidante oppure una stringa contenente “si” nel caso di accettazione della sfida, “no” altrimenti (ecco perché lo username deve avere lunghezza maggiore di 3, altrimenti il thread, se un utente si chiama “si”, non è in grado di distinguere tra sfidante e sfidato). In base alla stringa ricevuta si attiva il codice corrispondente allo sfidante o allo sfidato. La parole inviate al thread sfida vengono concatenate con la stringa “/uno/” se ad inviarla è stato lo sfidante, “/due/” se è stato lo sfidato (così il thread sfida può aggiornare correttamente i punteggi). Ad ogni iterazione il *ByteBuffer* contenente di volta in volta la parola ricevuta viene ripulito attraverso il metodo *Arrays.fill* con byte 0, in modo tale da non avere dei rimasugli delle precedenti stringhe ricevute. Il timer della sfida è stato implementato attraverso la classe *java.util.timer*, il quale, se scade, invia una stringa composta da “TIMEOUT” più il punteggio ottenuto fino a quel momento. Ovviamente nel momento in cui vengono tradotte tutte le parole il timer viene cancellato.

Il dizionario da cui vengono prese le parole è un semplice file di testo (*Dizionario.txt*), il quale comprende 30 parole diverse da cui ne vengono scelte 4 (se la funzione seleziona una già presa, ne sceglie un’altra, affinché abbia sempre 4 parole diverse tra loro).

I file dei vari utenti sono salvati nella cartella *Users* la quale è dedicata al salvataggio dei dati. All’interno di essa troviamo inoltre il file *PortMap.txt* che al suo interno contiene un *HashMap <NomeUtente, Porta>* che mi salva la porta UDP di ciascun utente registrato. Questa HashMap viene ovviamente salvata in una HashMap creata all’interno della classe *ServerQuizzle*, così da non dover, ogni volta che serve, accedere al file. Ogni porta viene creata convertendo il nome in un valore numerico utilizzando la codifica ASCII (converto ogni carattere nel numero corrispondente e li sommo), aggiungendo 1000 al valore ottenuto (così da ottenere un numero di porta compreso tra 1024 e 65535, cosicché non vengano utilizzate porte note). Ho inoltre aggiunto un controllo nel caso ci fossero 2 porte uguali : sommo 1 alla seconda porta fin quando non trovo una porta non utilizzata.

Abbiamo 5 strutture dati condivise tra i vari client : la *PortMap* (la HashMap contenente le porte UDP), l’ArrayList *utentiLoggati* (contenente gli utenti attualmente loggati e connessi), l’ArrayList *sfidaInCorso* (contenente gli utenti che sono attualmente impegnati in una sfida), l’ArrayList *TranslatedWords* (contenente le parole tradotte) e l’ArrayList *arrWords* (contenente le parole da tradurre). Per rendere sicuro l’accesso in modo concorrente ho utilizzato delle *ReentrantLock*. Ho preferito queste ultime rispetto al *synchronized* per due motivi :

1. La lock viene data al thread che sta aspettando da più tempo;
2. Ho una migliore gestione quando devo testare se la lock è stata presa oppure no (usando il metodo *trylock()*).

Ho quindi usato 4 ReentrantLock :

1. lockPortMap : lock relativa alla *PortMap*.
2. lockUtentiLoggati : lock relativa a *utentiLoggati*.
3. lockSfidaInCorso : lock relativa a *sfidaInCorso*.
4. lockPrepareArrays : lock relativa al metodo *PrepareArrays*, che prepara gli array con le parole tradotte (e quelle da tradurre) da passare al thread sfida.

Il nome dell’utente deve avere una lunghezza maggiore di 3; questo perché determinate parole di lunghezza minore potrebbero interferire con il corretto funzionamento del programma (inoltre deve essere minore di 128 caratteri, per evitare problemi con i vari buffer). Il perché è spiegato nella sezione successiva.

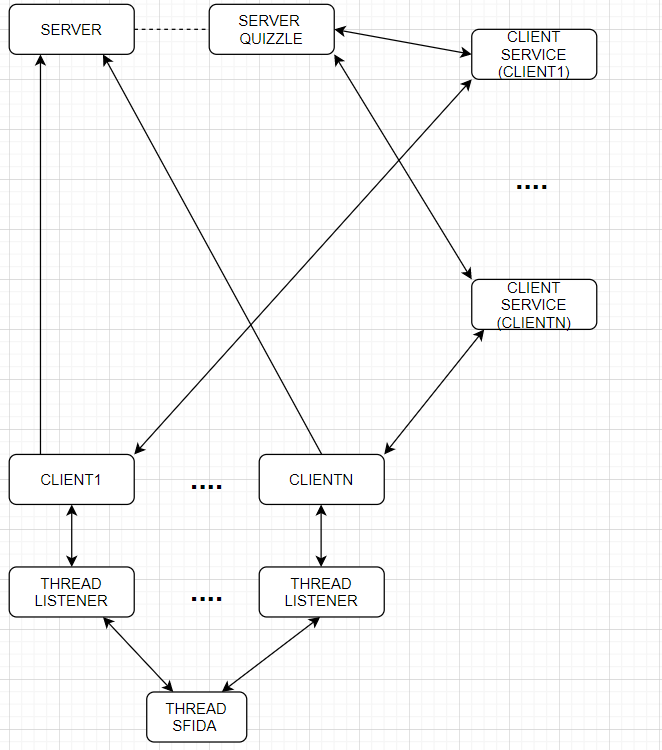
L’unico metodo che prende più parametri rispetto a quelli definiti da progetto è il *registra\_utente* : oltre a passargli il nome utente e la password, gli viene passata anche la porta UDP del client. Questo serve affinché essa venga salvata nel file PortMap.txt (è la porta relativa al *thread listener* delle sfide).

L’intero codice è stato sviluppato su IntelliJ IDEA 2019.2.3 (Ultimate Edition), Build #IU-192.6817.14, Runtime version: 11.0.4+10-b304.69 amd64, VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o.

Nella sezione successiva verrà analizzata la struttura del progetto e le classi presenti.

2. STRUTTURA DEL PROGETTO

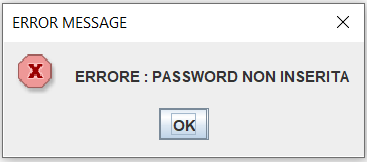
La struttura del progetto viene mostrata dal seguente schema :



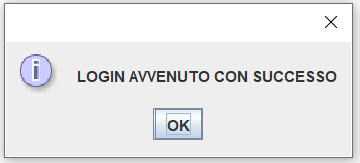
Abbiamo la classe *MainServer* che rappresenta il server che accetta le connessioni da parte dei vari client, oltre ad avere il *registry RMI*. A ciascuno di essi assegna un *ClientService* che gestirà i comandi del client. In base al comando ricevuto, ClientService chiama il metodo corrispondente presente nella classe *ServerQuizzle*, che implementa l’interfaccia *ServerWQ* contenente i metodi richiesti da progetto, e restituisce al client un *JSONObject* contenente il risultato della computazione. All’interno della classe *MainClient* sono chiamate, all’occorrenza, le classi che rappresentano i frame dell’interfaccia grafica (analizzate nella sezione successiva). Ho deciso di implementare i feedback attraverso dei semplici JOptionPane, piuttosto che fare una classe, poiché a parer mio andavano bene per far vedere all’utente se un comando è andato a buon fine oppure no. Ho usato 3 tipi di JOptionPane : INFORMATION\_MESSAGE, ERROR\_MESSAGE, QUESTION\_MESSAGE (l’uso di quest’ultimo viene analizzato nell’interfaccia grafica).

Se il JSONObject risultato contiene un messaggio di errore, esso verrà stampato a schermo attraverso un *JOptionPane* di tipo ERROR\_MESSAGE (assieme al codice dell’errore), altrimenti verrà stampato a schermo un *JOptionPane* di tipo INFORMATION\_MESSAGE.

Qui sotto troviamo un esempio di JOptionPane di tipo ERROR\_MESSAGE e un altro di tipo INFORMATION\_MESSAGE.



Questo JOptionePane di tipo ERROR\_MESSAGE ci appare, per esempio, se ci dimentichiamo di scrivere la password quando registriamo un utente.



Questo JOptionePane di tipo INFORMATION\_MESSAGE invece ci appare quando ci siamo loggati con successo.

Questi servono per fornire feedback all’utente e per fargli capire quando un comando è andato a buon fine. L’unico momento in cui non viene mostrato alcun JOptionPane di conferma è nel comando *sfida,* quando il client sfidato accetta la sfida; questo perché una volta accettata, subito viene inviata la prima parola, e mi sembrava ridondante far apparire un JOptionPane che indicava al client sfidante l’avvenuta accettazione della sfida, dato che se ne accorge dal fatto che appare il frame contenente la prima parola da tradurre.

Le strutture dati condivise si trovano all’interno della classe *ServerQuizzle*, che è condivisa tra tutti i *ClientService*. Per questo motivo è stato necessario l’inserimento di alcune *ReentrantLock*, per evitare problemi di sincronizzazione tra thread quando accedono a queste strutture.

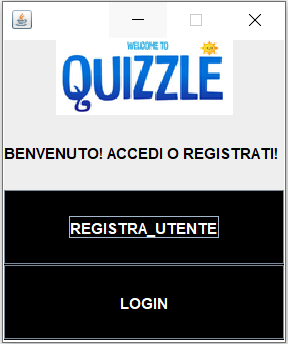
Il thread delle sfide è implementato dalla classe *Sfida*. Dentro essa è implementato il *selector* e le due connessioni TCP analizzate nella sezione precedente.

Il thread listener delle sfide è implementato dalla classe *listenForSfida*. Infine abbiamo la classe *ValueComparator* che contiene un comparatore che serve per ordinare correttamente la classifica da mostrare all’utente quando preme il comando MOSTRA\_CLASSIFICA.

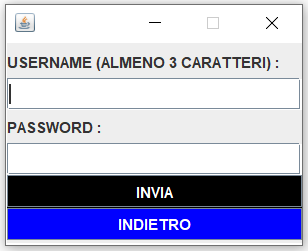
La sezione successiva analizza l’interfaccia grafica, mostrando tutti i frame che sono stati usati per realizzarla.

3. INTERFACCIA GRAFICA

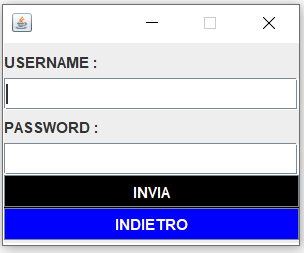
Analizziamo ora l’interfaccia grafica. Tutti i frame non sono espandibili per scelta stilistica. Ho deciso di non mettere subito a disposizione tutti i comandi per guidare l’interazione dell’utente : all’inizio esso può registrarsi oppure effettuare il login, attraverso il seguente frame; questo frame è implementato dalla classe *GraphicInit*.



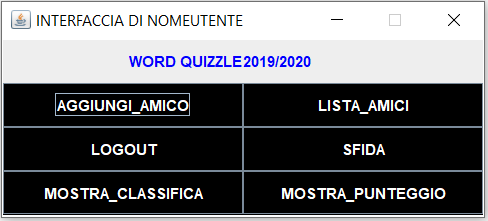
Ho voluto aggiungere un immagine (Quizzle\_real.png), scaricata da Internet e successivamente modificata per adattarla al contesto, per rendere più accattivante il frame iniziale. In base al pulsante premuto compariranno due frame diversi; se viene premuto REGISTRA\_UTENTE, apparirà il seguente frame :



L’utente dovrà semplicemente digitare uno username (implementato con un JTextField) di almeno 3 caratteri (per i motivi spiegati in precedenza), ed una password (implementata con un JPasswordField, in modo tale che non si vedano i caratteri effettivi quando viene scritta). Se invece l’utente vuole effettuare il login, verrà mostrato il seguente frame (questi ultimi due sono implementati dalla classe *GraphicRegLogin*).

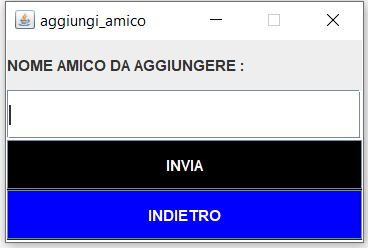


Questo frame è molto simile al precedente tranne per il fatto che non viene specificata la lunghezza dell’username; questo perché si presuppone che al momento della registrazione esso sia già di almeno 3 caratteri (un utente deve essere necessariamente già registrato per potersi loggare con successo). Questi ultimi due frame hanno anche il tasto indietro per tornare al frame precedente. Una volta effettuato il login, viene mostrato il seguente frame (implementato dalla classe *GraphicInterfaceWQ*), contenente i comandi richiesti da progetto :



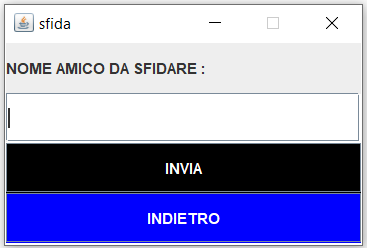
I comandi MOSTRA\_CLASSIFICA, LISTA\_AMICI e MOSTRA\_PUNTEGGIO fanno apparire un JOptionPane contenente, rispettivamente, la classifica dell’utente ed i suoi amici, la lista amici dell’utente e il punteggio dell’utente. Il comando LOGOUT equivale al classico tasto “esci dal gioco” presente in ogni gioco moderno, infatti ha la stessa semantica (viene chiuso il frame ed il client si disconnette dal server).

Analizziamo adesso il comando AGGIUNGI\_AMICO. Una volta premuto, apparirà il frame seguente :

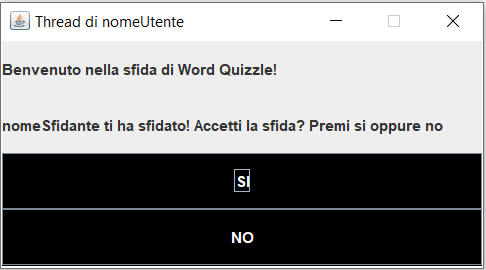


Basta scrivere nell’apposito spazio il nome dell’amico da aggiungere e, se quest’ultimo è registrato, verrà aggiunto in lista; ovviamente comparirà un JOptionPane di conferma oppure di errore (ad esempio se il nome digitato non esiste nella cartella Users).

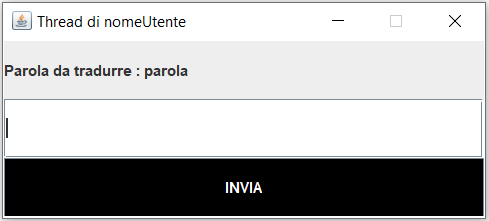
In ultimo vediamo il comando sfida. Una volta premuto verrà mostrato il seguente frame :



Questi ultimi due sono implementati dalla classe *GraphicInsertName*.In quest’ultimo ci basterà scrivere il nome dell’utente per inviargli una sfida. Ovviamente anche lo sfidato dovrà essere loggato affinché abbia successo il comando; inoltre, se l’utente è già impegnato in un’altra sfida, ci apparirà un JOptionPane che ci comunica, per l’appunto, che l’utente è già impegnato e quindi di riprovare in seguito. Ho deciso di non mettere nessuna coda delle sfide per due motivi : uno, può darsi che un’utente debba aspettare troppo per una sfida e quindi decida o di sfidare qualcun’ altro (e magari all’utente sfidato all’inizio gli arriva la notifica della sfida nel mentre che l’utente sfidante è impegnato in un’altra), oppure di effettuare il logout, rendendo quindi molto più tediosa la gestione di questa coda; secondo, a parer mio è meglio comunicare subito all’utente che la persona che si vuole sfidare è occupata e di riprovare in seguito. Anche questi ultimi due possiedono il tasto indietro per tornare al frame precedente. Una volta premuto il tasto invia, all’utente sfidato apparirà il seguente frame (implementato dalla classe *GraphicInterfaceSfida*).



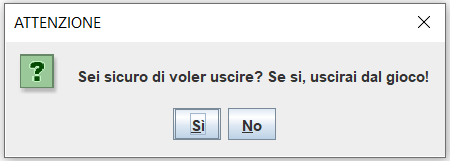
In questo frame l’utente sfidato deve decidere se accettare o meno la sfida; se preme no, all’utente sfidante apparirà un JOptionPane che gli comunica che la sfida è stata rifiutata, oppure, se la sfida verrà accettata, ad entrambi apparirà il seguente frame (implementato dalla classe *GraphicInterfaceTraductions*).



All’interno del riquadro l’utente dovrà scrivere la traduzione che, secondo lui, è quella della parola inviata. Una volta terminata la sfida apparirà un JOptionPane contenente il punteggio ottenuto.

All’interno di ciascuna classe che implementa i vari frame ho messo (oltre agli *ActionListener* relativi ai JButton) anche dei *WindowListener* in modo tale da capire quando l’utente preme sul tasto X. In ogni frame (tranne quest’ultimo) premere sul tasto X farà apparire un JOptionPane di tipo QUESTION\_MESSAGE che ci chiederà la conferma dell’azione : se premiamo sì usciremo dal gioco, altrimenti non succederà niente. In quest’ultimo frame invece premere sul tasto X farà terminare anzitempo la sfida con un punteggio di 0 (ovviamente anche qui apparirà un JOptionPane di conferma). Sono presenti dei JOptionPane anche per tutte le eccezioni eventualmente lanciate durante l’esecuzione del Quizzle : essi sono composti dalla stringa “ERRORE : eccezione.*toString()*” in modo tale da mostrare che tipo di errore è accaduto.

Ad esempio il seguente JOptionPane di tipo QUESTION\_MESSAGE appare quando si preme sulla X nel frame iniziale :



Infine ho implementato, per scelta puramente stilistica, anche un *MouseMotionListener* : esso fa apparire, se lasciamo il cursore per qualche secondo su un pulsante, una frase che ci spiega esattamente che cosa fa quel particolare tasto. Nel frame seguente viene mostrato il messaggio che appare quando il puntatore è sopra il pulsante LISTA\_AMICI.

