Relazione progetto Reti 2019/20
Word Quizzle

Sacco Giuseppe
559370

Gennaio 2020

Contents

1 Introduzione **2**
1.1 Descrizione Generale 2

2 Protocollo di comunicazione **2**
2.1 Message 3
2.2 MessageManager 3

3 Classi condivise **3**
3.1 Classe User 3

4 Server **3**
4.1 Funzionamento del Server 4
4.2 Classe Server 5
4.3 Classe UserAttach 6
4.4 Classe StubRegistration 6
4.5 Classe Challenge 6
4.6 Classe ServerManager 6
4.7 Classe JModel 8
4.8 Classe MyTimer 8
4.9 Classe TSfida 9

5 Client **9**
5.1 Classe Client 10
5.2 Classe ClientManager 10
5.3 Classe TUdp 10
5.4 GUI 10
5.4.1 LoginWQ 10
5.4.2 HomeWQ 11
5.4.3 FriendsWQ 11
5.4.4 ListFriendsWQ 11
5.4.5 ClassificaWQ 11
5.4.6 TSfidaClient 11

6 Compilazione **12**

1 Introduzione

Il servizio Word Quizzle mette a disposizione un sistema di sfide di traduzione di parole italiano-inglese tra gli utenti registrati con la possibilità di sfidare i propri amici.

1.1 Descrizione Generale

Per la gestione del Server è stato implementato un Selector per la gestione delle connessioni.
Dal lato del Client è stata implementata una semplice interfaccia grafica che mostra all'utente le possibili funzioni che l'applicazione può svolgere tramite un menù, solo dopo aver effettuato l'accesso con il proprio account precedentemente creato.
In seguito verrà spiegato approfonditamente il funzionamento del Client, del Server e il protocollo di comunicazione utilizzato.

2 Protocollo di comunicazione

Lo scambio di messaggi tra Client e Server è stato definito mediante un protocollo descritto tramite le classi **Message** e **MessageManager**

2.1 Message

La classe Message è costituita da due campi: un intero *type* e un'array di byte *payload*.
Il *type* è indentificato da codici della classe che indicano il tipo di richiesta/risposta che il mittente invia.
Per questa classe sono stati definiti più tipi di costruttori, dando così la possibilità di creare un messaggio con un *payload* diverso: un int, string e array di byte. Nel caso di un int o una stringa il costruttore trasformerà in array di byte le informazioni date.
Per farsi restituire il *payload* con il tipo corretto basta chiamare una tra le funzioni get messe a disposizione dalla classe che restituiscono il tipo desiderato.
La classe Message implementa Serializable, poiché i messaggi vengono serializzati prima dell'invio e deserializzati all'arrivo.

2.2 MessageManager

Ogni messaggio scambiato tra Client e Server utilizza le funzioni *readMsg* e *sendMsg* della classe MessageManager.
Il protocollo al momento dello scambio di un messaggio consiste nel mandare prima la *dimensione* di quest'ultimo e successivamente il suo *contenuto*.
Nella classe sono implementate le funzioni di serializzazione del messaggio con la funzione *getByteFromObject*, e deserializzazione dei messaggi grazie alla funzione *getObjectFromByte*.

3 Classi condivise

3.1 Classe User

Viene usata per la gestione degli utenti nell'hash map. Contiene l'username e password dell'utente, la sua lista di amici e il punteggio ottenuto.

4 Server

Come detto in precedenza il Server gestisce le connessioni con un Selector.
Per coordinare il funzionamento del Server sono state create le seguenti

classi: *Server, ServerManager, StubRegistration, UserAttach, TSfida, Challenge, MyTimer, JModel*. Come librerie esterne è stato utilizzato il pacchetto **Gson** di Google, che consente di leggere un file JSON e creare delle classi apposite per convertire da stringa o oggetto. È stato utilizzato per la lettura del file *database.json* e per la lettura delle risposte HTTP di traduzione delle parole.

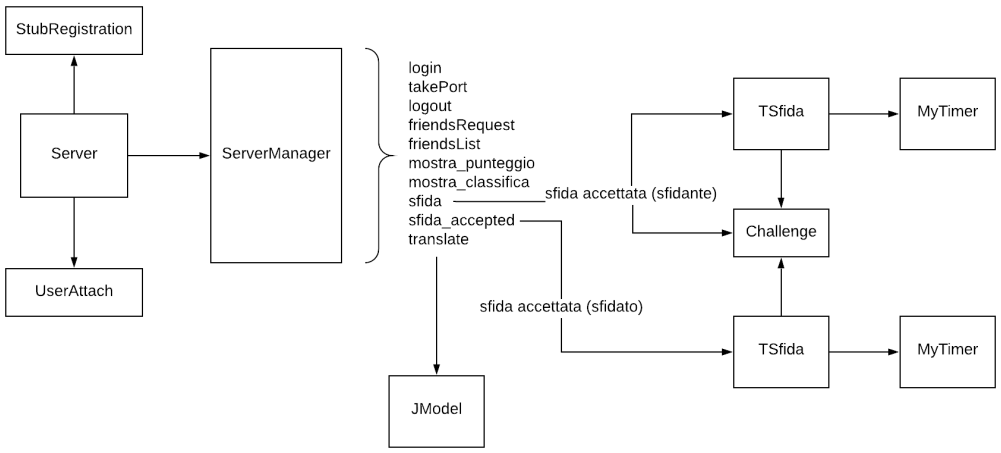

Figure 1: Funzionamento del Server ed interazione con tutte le classi nel momento di un avvio di sfida.

4.1 Funzionamento del Server

Il Server nel Selector accetta le connessioni da parte dei Client. Quando riceve un qualsiasi messaggio da parte del Client, delega ad un thread di occuparsi della richiesta. Questo thread esegue il metodo *run()* della classe **ServerManager** e viene creato ogni qual volta vi è una richiesta da un client. Tutti questi thread sono raccolti in una *threadpool*. Inoltre il thread dopo aver generato l'adeguata risposta alla richiesta del Client aggiorna la chiave del Selector, delegando la scrittura della risposta sul socket di comunicazione con il client.

4.2 Classe Server

È la main class del Server. Inizializza il dizionario di parole per le sfide e le strutture dati condivise. Avvia lo stub RMI e un threadpool per i thread *ServerManager* con un array blocking queue. Crea il server socket channel e inizializza il selector utilizzando come attachment per la chiave l'oggetto definito dalla classe *UserAttach*.
Oltre alla gestione del Selector mette a disposizione varie funzioni di accesso alle variabili condivise tra i thread.
Tra le variabili condivise ci sono:

- *ConcurrentHashMap<String, Integer> ports:* per le porte UDP dei client
- *ArrayList<Challenge> sfide: raccoglie le sfide attive* (con un'opportuna lock per gestire la concorrenza)
- *ReentrantLock filelock:* lock per l'accesso al file
- *ConcurrentHashMap<String, User> usersDB*: dove sono salvati tutti i dati relativi agli utenti
- *ArrayList<String> online*: contiene gli username di tutti gli utenti online

Le funzioni messe a disposizione dal Server sono:

- *updateDB* aggiorna l'HashMap contenente le informazioni degli utenti
- *checkDB* controlla se nel db esiste un determinato utente per il login
- *checkOnline* controlla se uno specifico utente è online
- *goOnline* inserisce un utente tra quelli online nel momento in cui verrà eseguito l'accesso
- *goOffline* rimuove dagli utenti online un utente che ha effettuato il logout o si è disconnesso
- *checkUser* verifica che all'interno del database ci sia un determinato utente
- *updateFile* aggiorna il file .json utilizzando l'HashMap
- *updatePoint* aggiorna il punteggio dell'utente nell'HashMap.

4.3 Classe UserAttach

Viene usata nel Selector per tenere traccia del messaggio ricevuto, quello di risposta e l'username dell'utente.

4.4 Classe StubRegistration

Implementa l'interfaccia *Registration*.
Nel metodo costruttore se non esiste viene creato il *database.json*. Se già esistente viene letto e salvato nella ConcurrentHashMap, condivisa dal Server, che raccoglie le informazioni relative agli utenti.
La funzione *registra_utente* permette di registrare un utente: viene letto il file .json sotto forma di array, se esiste già nell'array un utente con lo stesso nome ritorna 0, altrimenti viene aggiunto e vengono aggiornati l'hashmap e il file.

4.5 Classe Challenge

Definisce un tipo di sfida. Quando si avvia una sfida viene creato un tipo Challenge che andrà a far parte di una lista condivisa tra i thread che la gestiscono. I thread interessati accederanno solo all'oggetto Challenge che li riguarda. Nella classe si raccolgono gli username degli avversari, la lista di parole da mandare e le parole tradotte, il punteggio di entrambi gli sfidanti e due variabili *AtomicBoolean* che indicano rispettivamente quando le sfide sono finite.

4.6 Classe ServerManager

Viene attivato ogni qual volta vi è una richiesta da parte del Client.
Nel metodo costruttore prende l'attachment della key *(UserAttach)*, la key relativa e il selector.
Nel metodo run viene fatto il controllo su quale richiesta viene mandata dal client e ad ognuna di queste corrisponde la relativa funzione che la implementa.

- *login*: nel caso di login il payload del messaggio conterrà username e password dell'utente. Dopo averli ricavati tramite la funzione *checkDB* del Server viene fatto un controllo sull'esito. In base all'esito viene

mandato il messaggio opportuno. In caso di avvenuto login viene salvato nell'attachment l'username dell'utente e viene messo nella lista degli utenti online.

- *takePort*: nel caso di ricezione della porta UDP viene salvato nella HashMap delle porte la coppia <username, porta>.

- *logout*: nel caso di logout il payload avrà il nome dell'utente che esegue il logout. L'utente viene rimosso sia dalla lista degli utenti online che dall'HashMap contenente le porte UDP.

- *friendsRequest*: nel caso di una rihiesta di amicizia il payload conterrà il nickUtente e nickAmico con il quale si vuole creare l'amicizia. Viene controllato se l'utente "nickAmico" esiste nel database e se non fa già parte della lista degli amici. Se le condizioni precedenti sono verificate allora viene creato il legame di amicizia tra i due utenti e aggiornato il database.

- *friendsList*: nel caso di richiesta della lista degli amici il payload conterrà il nome dell'utente. La lista degli amici viene trasformata in json e poi mandata come stringa.

- *mostra_punteggio*: nel caso di richiesta del punteggio il payload conterrà l'username dell'utente. Viene ricavato il punteggio e mandato in un messaggio come intero.

- *mostra_classifica*: nel caso di richiesta del punteggio il payload conterrà l'username dell'utente. Si ricava la lista degli amici con il relativo punteggio ottenuto, viene trasformato in json e successivamente inviato come stringa.

- *sfida*: Nel caso di sfida il payload conterrà gli username dello sfidante e dello sfidato. Dato la richiesta di traduzione di parole da effettuare viene fatto prima un test di rete.
  In seguito viene testato che nickAmico esista nel database, si controlla se nickAmico appartiene alla lista degli amici di nickUtente e viene testato se nickAmico è online.
  A questo punto viene creato il datagram socket con il relativo timer e mandata in UDP la richiesta di sfida contenente come payload nickUtente. Se non viene ricevuta risposta la richiesta di sfida viene rifiutata.

Oppure se viene ricevuta una risposta viene controllato il tipo di messaggio ricevuto ovvero se l'utente è occupato in un'altra sfida, se è stata rifiutata oppure se è stata accettata. Nel caso che la sfida sia stata accettata vengono scelte le parole per la sfida e vengono tradotte con la funzione *translate*, in seguito viene creato un oggetto di tipo *Challenge* che conterrà tutte le informazioni della sfida tra cui gli username degli sfidanti e la lista di parole. Infine viene creata una nuova connessione TCP e un nuovo thread che gestirà la sfida per lo sfidante che comunicherà sul socket appena creato. Come messaggio di risposta verrà mandato in payload la porta appena creata per la connesione TCP.

- *sfida_accepted*: nel caso in cui uno sfidato accetta la sfida ricevuta il payload conterrà il nome dello sfidante. Viene controllato se nella lista di oggetti *Challenge* esiste un oggetto che possiede il nome dello sfidante e dello sfidato. Se non viene trovata alcuna corrispondenza viene mandato un messaggio di annullamento sfida, altrimenti viene creato un socket TCP e un thread per la gestione della sfida che utilizzerà il socket appena creato per comunicare con il client. Nella risposta al client viene mandato nel payload la porta per la connessione TCP.

- *translate*: funzione di traduzione delle parole scelte, con la richiesta in HTTP al sito.

4.7 Classe JModel

Rappresenta la risposta in Json da parte del sito di traduzione. Viene utilizzata per semplificare la lettura dell'oggetto json, tramite GSon, con all'interno la parola tradotta precedentemente richiesta al sito.

4.8 Classe MyTimer

Viene creato il timer nel thread di sfida. Quando viene avviato dopo T2 (variabile che indica la durata della partita) "setta" la variabile di iterazione del ciclo del thread di sfida ad un valore TIMER_CODE che notifica la fine della sfida.

4.9 Classe TSfida

Classe che implementa la sfida tra due avversari.
Vengono avviati due thread (ogni qual volta si avvia una sfida) di questa classe: uno per lo sfidante e uno per lo sfidato, e si controlla se il thread sia dell'uno o dell'altro. In seguito si attiva il timer e inizia l'invio delle parole da tradurre al client. Alla fine della sfida se l'avversario non ha finito viene fatto attendere per sapere il punteggio. Una volta che entrambi hanno concluso la sfida viene inviato un messaggio con l'esito della partita e di conseguenza viene rimosso l'oggetto *Challenge* dall'arraylist delle sfide.
In seguito si ha l'aggiornamento del punteggio dell'utente al database e quello del file *database.json*.

5 Client

La gestione del Client è stata implementa tramite un'interfaccia grafica ed una classe *ClientManager* che gestisce tramite varie funzioni l'invio e la ricezione di messaggi dal Server. In più tramite la classe *TUdp* che implementa un metodo *run* viene gestita la ricezione di richieste di sfida da parte del Server.

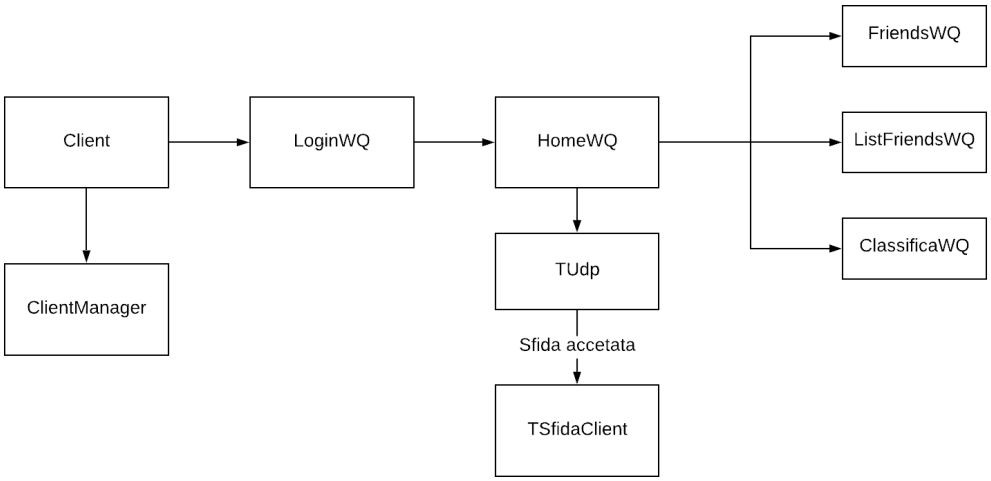

Figure 2: Funzionamento del Client ed interazione con tutte le classi nel momento di un avvio di sfida.

5.1 Classe Client

Richiede l'inizializzazione della connessione al client, tramite la funzione *initClient* della classe *ClientManager*, e iniziallizza la GUI per l'utente.

5.2 Classe ClientManager

Gestisce tutte le operazioni principali offerte da WQ tranne le richieste di sfida.
Nel metodo costruttore viene creata la connessione per la registrazione con la porta RMI.
Il metodo *initClient* inizializza le connessioni TCP e UDP, quella TCP su una porta definita dal Server, quella UDP su una porta libera che poi verrà inviata in fase di login al Server. Il resto delle funzioni hanno il compito di inviare una richiesta con il codice appropriato e restituire il messaggio di risposta restituito dal Server.

5.3 Classe TUdp

Viene attivato il thread di questa classe al momento del login che si occupa di gestire tutte le richieste di sfida che l'utente riceverà.
Riceve il pacchetto UDP di richiesta di sfida dal Server. Se l'utente è occupato in una sfida manda in automatico un messaggio che notifica l'utente occupato. Se l'utente non è occupato allora viene presentato un message box dove l'utente può decidere se accettare o meno la richiesta. Nel caso in cui la richiesta viene accettata viene fatta una richiesta al server se può cominciare effettivamente la partita oppure se è stata annullata nel caso in cui la risposta non è arrivata in tempo oppure se la risposta in UDP non è arrivata. Se dal Server si riceve la risposta che la partita può avere inizio, allora viene creato un thread della classe *TSfidaClient* che gestirà l'intera partita e l'interfaccia grafica stessa.

5.4 GUI

5.4.1 LoginWQ

Presenta l'interfaccia per effettuare il login o la registrazione. L'interfaccia presenta i campi username e password da inserire e due bottoni che performano rispettivamente il login e la registrazione. La classe utilizza le funzioni

messe a disposizione dal Client Manager per mandare le richieste al Server. A registrazione o login avvenuti con successo si passa alla home.

5.4.2 HomeWQ

Presenta un'interfaccia per richiedere tutte le operazioni prestabilite: *aggiugni amico, lista amici, mostra punteggio, mostra classifica, sfida, logout.* Premere un bottone tra "Aggiungi amico", "Lista amici", "Mostra classifica" e "Sfida" porterà ad un nuovo form per la visualizzazione delle richieste. Tramite il bottone "Logout" è possibile tornare al form *LoginWQ*.

5.4.3 FriendsWQ

Presenta un'interfaccia con il campo dell'username amico da inserire affinchè venga creata l'amicizia. Dopo aver inserito il nome dell'utente basta cliccare sul bottone "Aggiungi" per creare l'amicizia. In fondo al form vi è il bottone "Home" per tornare al form *HomeWQ*.

5.4.4 ListFriendsWQ

Mostra la lista degli amici dell'utente collegato. Una volta eseguita la lettura della risposta del Server, viene prima deserializzato il messaggio, letto come json di tipo ArrayList di stringhe e successivamente viene mostrato all'utente scorrendo l'array. In fondo al form vi è il bottone "Home" per tornare al form *HomeWQ*.

5.4.5 ClassificaWQ

Viene mostrata la classifica in base al punteggio ottenuto da tutti gli amici dell'utente che ha effettuato l'accesso. Una volta eseguita la lettura della risposta del Server, viene prima deserializzato il messaggio, letto come json di tipo ArrayList di *User* e successivamente verrà mostrato all'utente scorrendo l'array. In fondo al form vi è il bottone "Home" per tornare al form *HomeWQ*.

5.4.6 TSfidaClient

Il thread viene attivato nel momento in cui la sfida ha inizio. Nella pagina è mostrato il numero di parole inviate fino a quel momento. Vi è un text field che permette l'inserimento della risposta. Tramite i bottoni "Submit"

e "Skip" è possibile inviare la risposta oppure saltare la parola proposta che verrà segnata come non data. Il thread riceve ad ogni iterazione un messaggio dal server tra i possibili:

- nuova parola da tradurre, con successivo aggiornamento dell'interfaccia
- timer scaduto, con esito della partita come prossimo messaggio
- attesa dell'esito della partita se l'avversario non ha finito
- esito della partita e di conseguenza la fine della sfida e ritorno alla home.

6 Compilazione

Per quanto riguarda la compilazione e l'esecuzione, partendo dalla directory principale del progetto, compiliamo Server e Client utilizzando:

javac -d bin -cp lib/gson-2.8.6.jar:src src/Server/Server.java
javac -d bin -cp lib/gson-2.8.6.jar:src src/Client/Client.java

Per eseguire il programma invece lanciamo:

java -cp bin:lib/gson-2.8.6.jar Server.Server
java -cp bin:lib/gson-2.8.6.jar Client.Client