Relazione progetto Reti 2019/20

Matteo De Francesco
562176

Gennaio 2020

Indice

1 Introduzione

Di seguito verranno descritte le funzionalità principali del servizio WordQuizzle, delle scelte implementative e del protocollo di comunicazione utilizzato.

2 Protocollo di comunicazione

Per gestire in modo efficiente la comunicazione tra Client e Server, è stato stabilito un protocollo di comunicazione ben definito, descritto nelle classi **Message** e **Connection**.

2.1 La classe Message e la classe Connection

Ogni messaggio che viene scambiato tra Client e Server utilizza le funzioni **read** e **write** della classe Connection, il protocollo definito consiste nello scambiare prima la *lunghezza del messaggio* e in seguito il contenuto. I messaggi scambiati sono definiti nella classe Message e ognuno di questi è composto da due campi: il tipo e il payload.
Il tipo del messaggio è descritto da codici interi che rappresentano la richiesta che si vuole effettuare e le risposte che si possono ricevere, questi codici sono contenuti all'interno della classe Message. Il payload sono i dati che vengono effettivamente scambiati, puo' essere di tipo intero, di tipo stringa e di tipo byte[]. La classe Message implementa *Serializable*, poiché i messaggi vengono serializzati prima dell'invio, grazie alla funzione **getByteFromObject** e deserializzati all'arrivo con la funzione **getObjectFromByte**.

3 Server

Il server è stato implementato tramite un Selector, per consentire la distribuzione tra i vari canali pronti e non. Di seguito il funzionamento generale.

3.1 Funzionamento generale

All'avviare del Server viene letto il contenuto del dizionario delle parole (che verrà usato in seguito nella sfida), vengono inizializzate le strutture dati condivise, viene creato lo stub RMI e il ServerSocketChannel in ascolto di connessioni.
Al momento della connessione ogni Client viene accettato e viene messo in

modalità lettura. Quando arriva una richiesta, questa, con il relativo attachment del Client e la SelectionKey, vengono passati a un Thread della ThreadPool che si occuperá di gestire la richiesta, settare il canale in modalitá di scrittura e notificare il selector. In modalita scrittura il selector si occuperà di scrivere il messaggio di risposta al Client.

3.2 Strutture Dati condivise

Il Server utilizza delle strutture dati e oggetti condivisi con i thread:

- **writeGsonLock**: ReentrantLock per consentire la modifica del file *infoserver.json* in mutua esclusione;
- **users**: ConcurrentHashMap con chiave nome dell'utente e valore porta UDP del Client relativo. Viene modificata al momento di login e logout, per consentire al Server un accesso veloce per capire quali sono gli utenti online e non;
- **fighting**: ConcurrentHashMap con chiave un oggetto di tipo *ValueOne* e valore un oggetto di tipo *ValueHashmap*. Viene usata al momento dell'inizio di una sfida, dopo vedremo in dettaglio le classi e il funzionamento;

3.3 L'attachment del Client

Prima di passare a descrivere il funzionamento del WorkerThread, dobbiamo parlare dell'attachment del Client. Ogni Client viene "dotato", al momento della connessione, di un oggetto ClientState che contiene i campi qui di seguito:

- requestMessage: è la richiesta effettuata dal Client;
- responseMessage: è la risposta che verrà salvata dal WorkerThread in seguito;
- user: oggetto di tipo User, contiene varie informazioni sul client, se è online o meno, se si trova in sfida o meno;
- usernameSfida: indica il nome del Client sfidato, viene inizializzato al momento di una richiesta di sfida;
- words: ArrayList di parole da tradurre durante la sfida;

- translatedWords: ArrayList di parole tradotte per verificarne la correttezza;
- iterWords: intero per scorrere l'array delle parole e per la gestione del timer;
- sfidaPoint: intero per contare i punti effeettuati da un Client;
- correct: ArrayList di due posizioni per contare le parole corrette e errate di un Client;
- timer: oggetto di tipo Timer per fermare la sfida al momento dello scadere del tempo;

Nella classe User troviamo informazioni relative al nome del Client, al suo stato di online e al suo stato di sfida. Per quanto riguarda tutti gli altri campi, vedremo nella gestione della sfida come torneranno utili.

3.4 Gestione delle richieste: WorkerThread

Ongi volta che viene letto un messaggio, il Server passa il "lavoro" da svolgere a un Thread della ThreadPool. Il **WorkerThread** implementa l'interfaccia WordQuizzleInterface, che contiene tutte le funzioni che puo' svolgere il Server. All'avvio del Thread,viene letto il file *infoserver.json* e salvato in un ArrayList *info*, viene diviso il messaggio di richiesta e viene analizzato il codice del Message. Fondamentale è l'inizio del metodo run, qui viene controllato se il Client si trova in una ConcurrentHashMap condivisa e se non si trova in sfida, lo vedremo dopo.
Passiamo a parlare dei metodi:

- requestnotvalid(): viene chiamato se il messaggio inviato dal Client consiste in una richiesta non supportata dal Server, viene restituito un messaggio di richiesta non valida;
- savePortNumber(): primo messaggio scambiato tra Client e Server, serve per salvare all'interno dell'attachment del Client il numero della sua porta UDP dove eventualmente sará mandata la richiesta di sfida;
- login(nickUtente,password): consente a un utente di effettuare il login, controlla se è contenuto in *info*, se non è gia online tramite la classe User dell'attachment e se la password è corretta. Se il tutto va a buon fine, restituisce un messaggio di tipo LOGINOK, altrimenti restituisce un messaggio di errore a seconda del tipo, password errata, utente non registrato oppure utente già online;

- logout(nickUtente): consente a un utente di effettuare il logout, controlla se l'utente è contenuto in *info*, se non era già offline. Se va tutto a buon fine, restituisce un messaggio di tipo LOGOUTOK, altrimenti un messaggio di errore, utente non registrato oppure utente non online;

- aggiungiamico(nickUtente,nickAmico): consente di aggiungere nickAmico alla lista degli amici di nickUtente e viceversa. Controlla che entrambi siano contenuti in *info* con la funzione di supporto *check*, se va tutto a buon fine l'amicizia viene aggiunta ambo i lati e viene riscritto il file JSON, altrimenti ritorna un messaggio di errore;

- listaamici(nickUtente): restituisce la lista degli amici di nickUtente in formato JSON dopo aver controllato che nickUtente sia registrato e online;

- mostrapunteggio(nickUtente): restituisce il punteggio totale di nickUtente, recupera il valore da *info* e lo restituisce dopo i vari controlli;

- mostraclassifica(nickUtente): restituisce la classifica con i punteggi in ordine decrescente di nickUtente e tutta la sua lista di amici;

- translateWords(words): traduce le parole contenute nell'array words che verranno poi usate nella sfida;

- updateDB(user,points): metodo usato al termine di una sfida per aggiornare i punteggi dell'utente user all'interno del file *infoserver.json*;

3.5 Caso sfida

Per quanto riguarda il caso della sfida, il WorkerThread utilizza tre metodi per la gestione: il metodo *sfida(nickUtente,nickAmico)*, chiamato da nickUtente quando vuole iniziare una sfida contro nickAmico, il metodo *sfidaduring* per gestire il trasferimento delle parole durante la sfida, e infine *checkFinished*, per sincronizzare il termine della sfida di entrambi. Quando nickUtente inizia una sfida, dopo i vari controlli di registrazione, di login, e se nickAmico è online, viene lanciato un Thread che si occuperà di mandare la richiesta UDP al Client. Se non riceve risposta entro 10 secondi, la sfida si considera rifiutata. Se invece la risposta è di tipo 0, allora il Client ha rifiutato la sfida. Se 2, allora nickAmico è gia impegnato in una sfida e bisogna attendere. Se 1 invece, la sfida è stata accettata e puo' partire l'inizializzazione dei vari parametri.
È qui che entra in gioco l'hashmap **fighting**. Il Client che sfida aggiunge

il suo nome come chiave e il nome dello sfidato come valore, insieme ad altre informazioni sulla sfida, overro la lista delle parole da tradurre e quelle tradotte. Questo servirà al Client sfidato al momento della richiesta per far partire la sfida. Infatti, come detto prima, all'inizio del metodo run viene controllato se il nome del Client attualmente attivo si trova nell'hashmap, se cosi accade infatti viene inizializzata anche per lui la partita prendendo le parole da tradurre e quelle tradotte dal valore dell'hashmap.
Una volta iniziata la sfida, il Server controlla la validitá della traduzione con il metodo *sfidaduring*, in caso di successo aumenta di 2 il valore *sfidaPoint* dell'attachment, altrimenti lo decrementa di 1. Viene modificato allo stesso modo anche il valore dei parametri *pointsSfida* nella chiave o valore di fighting a seconda dell'utente, viene incrementato anche il valore all'indirizzo 0 o 1 dell'ArrayList *correct* a seconda che la risposta sia corretta o sbagliata. Parametro importante qui è **iterWords** contenuto nell'attachment. Se questo valore è minore di 7, allora vengono contati i punti come detto sopra, altrimenti se vale 7 o 9, vuol dire rispettivamente che le parole sono finite o che il timer è scaduto. In questi casi il Client controlla che l'altro abbia finito, tramite i valori contenuti nel valore di fighting **hasFinished** e **hasFinishedFirst**. Se entrambi questi valori sono true, allora entrambi hanno terminato la sfida e viene restituito il resoconto della sfida, con i punti effettuati, le risposte corrette e sbagliate e l'esito della partita. Se cosi non fosse, allora viene resituito al Client un messaggio di tipo SFIDAWAITING. Quando un Client vede arrivarsi questo messaggio, allora manda richieste ogni x secondi al Server per sapere se l'altro avversario ha terminato. Questo viene controllato tramite il metodo *checkFinished*, che controlla la terminazione dell'avversario e in caso di fine restituisce il resoconto, altrimenti continua con messaggi di tipo SFIDAWAITING. Al termine della sfida, vengono resettati tutti gli oggetti utilizzati nella sfida.

4 Client

Passiamo ora a parlare del funzionamento del Client.

4.1 Funzionamento generale

Le classi principali utilizzate lato Client sono il **ClientManager** e le varie Form della GUI. Come prima cosa, viene istanziata la classe ClientManager, che si connette al Server e scambia come primo messaggio la porta UDP del DatagramSocket creato in precedenza e il recupero dello stub RMI.

La prima form che viene mostrata al Client è la StartForm, dove puo' effettuare il login o la registrazione . Una volta effettuato il login, viene mandato nella HomeForm dove puo' effettuare le varie operazioni descritte in precedenza nel Server. Se decide di sfidare un amico, viene mandato nella SfidaFrame dove inserirà la traduzione delle parole e l'invio di quest'ultima.

4.2 La classe ClientManager

La classe ClientManager si occupa della logica del Client, contiene vari metodi che vengono chiamati a seconda dell'operazione che si vuole effettuare, e all'avvio crea il DatagramSocket per l'ascolto delle richieste di sfida, si connette al Server e recupera lo stub RMI. I vari metodi e il loro funzionamento sono i seguenti:

- login(username,password): manda un messaggio di tipo LOGINCODE con username e password al server e attende la risposta, se è di tipo LOGINOK viene settato l'username del Client. Restituisce il codice di risposta;
- registra(username,password): chiama la funzione *registranuovoutente* dello stub RMI, ritorna 1 in caso di successo 0 altrimenti;
- addFriend(name): manda un messaggio di tipo ADDFRIENDCODE al Server e ritorna il codice del messaggio di risposta;
- showFriends(): manda un messaggio con codice FRIENDSLISTCODE al Server e ritorna il messaggio di risposta;
- showScore(): manda un messaggio con codice SHOWPOINTSCODE al Server e ritorna il messaggio di risposta;
- showScoreBoard(): manda un messaggio con codice SHOWSCORECODE al Server e ritorna il messaggio di risposta;
- logout(): manda un messaggio con codice LOGOUTCODE al Server e ritorna il messaggio di risposta;
- sfida(name): manda un messaggio con codice SFIDACODE al Server e ritorna il messaggio di risposta;
- sendReceive(word): manda la parola tradotta al Server con codice SFIDADURING e riceve la prossima parola o la terminazione;

- receiveWord(): utilizzata in fase iniziale di sfida per leggere la prima parola mandata del Server;
- waitFriend(): manda un messaggio con codice WAITFRIEND se ha ricevuto una risposta uguale. Serve ad aspettare la terminazione della sfida dell'avversario;
- reqNotValid(): manda un messaggio di REQUESTNOTVALID. Serve in fase di accettazione della sfida da parte dello sfidato;

4.3 Il thread UDP

Il thread UDP viene lanciato al momento del login di un Client, si occupa di restare in ascolto di eventuali richieste di sfida. Quando arriva una richiesta di sfida, la receive si "sblocca" e viene chiamato un metodo dello stub RMI *notificaClient* per avvertire il Client della richiesta di sfida. Se la sfida viene accettata, risponde al Server con 1 e aggiunge un valore a una coda condivisa *queue*. Questa coda serve alla HomeForm per sapere se ci sono richieste pendenti di sfida e farla partire. Se la sfida viene rifiutata la risposta al Server sarà 0, se il Client è gia impegnato in una sfida sarà 2.

4.4 La GUI

Il Client sceglie le operazioni da fare tramite tre GUI che si scambiano a seconda del suo "stato".

4.4.1 StartForm

La StartForm è la GUI che viene mostrata all'avvio, vi sono due campi di testo, uno per l'username e uno per la password, e due tasti, uno per il login e uno per la registrazione. In caso di login, viene chiamato il metodo *login()* contenuto nel ClientManager, in caso di registrazione, viene chiamata la funzione dello stub *registranuovoutente*, di cui parleremo in seguito. Se il login ha avuto successo, il Client viene mandato nella HomeForm.

4.4.2 HomeForm

La HomeForm è la GUI che appare a seguito di un login avvenuto con successo o al termine di una sfida. È la "home principale", dove il Client puo' scegliere quale operazione eseguire. A seguito di una operazione, viene mandato un determinato messaggio con un certo codice e la risposta ricevuta

viene mostrata tramite dialog. In caso di sfida, se questa viene accettata viene lanciato SfidaFrame per la gestione della sfida.

4.4.3 SfidaFrame

SfidaFrame è la GUI che viene mostrata all'inizio di una sfida, è composta da una linea di testo contenente la parola da tradurre, un campo di testo per inserire la traduzione e un bottone per inviarla. Al termine della sfida, se il Client deve attendere l'altro avversario allora il suo frame si "blocca" in attesa dell'altro. Una volta terminata la sfida, viene mostrato un dialog con il resoconto e viene rimandato alla Home.

5 Stub RMI

Lo Stub RMI è formato dalle classi Registration e RegistrationImplementation. Le due funzioni caricate su RMI sono *registranuovoutente* e *notificaClient*: la prima legge il contenuto del file *infoserver.json* in mutua esclusione utilizzando il lucchetto *writeJsonLock* e controlla che l'utente non sia giá registrato, in caso positivo lo aggiunge; la seconda, invece, mostra all'utente sfidato un dialog dove puo accettare o rifiutare la sfida, ritorna 1 in caso di sfida accettata, 0 altrimenti.

6 Altre classi e librerie

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. In questo particolare caso le classi usate per leggere i JSON sono **InfoServer**, per il file *infoserver.json*, e la classe **GetJson**, per il risultato restituito dalla GET al sito api.mymemory.translated.net.
Per **InfoServer** i campi sono il nome dell'utente, il suo punteggio, la sua password e la sua lista di amici.
Per **GetJson** i campi sono responseData, oggetto di tipo Info che contiene la parola tradotta.
Per quanto riguarda le classi *ValueOne* e *ValueHashmap*, queste sono usate rispettivamente come chiave e valore dell'hashmap **fighting**, di cui abbiamo parlato in precedenza.
Per **ValueOne** abbiamo:

- user: nome dell'utente che sfida;

- sfidaPoint: per contare i punti dell'utente nella sfida corrente;
- isInterrupted: per gestire il caso in cui la traduzione delle parole non è andata a buon fine;

Per **ValueHashmap** abbiamo invece:

- user: il nome dell'utente sfidato;
- sfidaWords: le parole da tradurre;
- translated: le parole tradotte;
- pointsSfida: per tenere conto dei punti effettuati dall'utente sfidato;
- hasFinished: per controllare se lo sfidato ha terminato;
- hasFinishedFirst: per controllare se lo sfidante ha terminato;
- hasInterrupted: per tenere conto di eventuali disconnessioni dello sfidato durante una sfida;
- hasInterruptedFirst: per tenere conto di eventuali disconnessioni dello sfidante durante una sfida;

7 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/WordQuizzle.java
javac -d bin -cp lib/gson-2.8.6.jar:src src/Client/NewClient.java

Per eseguire il programma invece lanciamo:

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