HOTELIER → an HOTEL advIsor sERvice

Francesco Fiaschi, 636697, Informatica - UNIPI

Indice

1 Introduzione

Il progetto consiste nella realizzazione di una versione semplificata del famoso servizio TripAdvisor. Qui ci limiteremo a gestire hotel, come tipo di struttura, login, registrazione, ricerca e scrittura di recensioni solo tramite punteggio, a livello di funzionalità.

2 Definizione delle Scelte Implementative

Il programma all'avvio deve caricare una lista di hotel e le relative informazioni da un file json; questo viene effettuato tramite l'uso della libreria esterna GSON di Google, oltre all'uso in lettura la libreria viene usata in scrittura per salvare periodicamente il file degli utenti registrati e le modifiche del ranking e delle recensioni associate agli hotel.

Le proprietà del server sono passate tramite lo specifico file di configurazione: *config.inputServer*.

Questo avviene tramite la funzione *caricaProprieta* che usa la classe *Proprieties* per fare il parsing delle proprietà nel file impostando così le seguenti costanti:

1. `PORT` → che indica la porta della connessione con i client;
2. `indirizzoMulticast` → è l'indirizzo dove si trovano tutti gli utenti loggati;
3. `portaMulticast` → porta corrispondente della connessione multicast;
4. `fileUtentiPath` → path della posizione del file delle informazioni degli utenti registrati;
5. `fileHotelPath` → path della posizione del file dell'informazione sugli hotel;
6. `intervalloAggiornamento` → durata del timer che permette il ricalcolo del ranking.

Per lo scambio di messaggi tra client e server ho utilizzato JAVA I/O e per gestire le varie connessioni in modo indipendente e concorrente ho utilizzato un *CachedThreadPool* per gestire più utenti contemporaneamente allocando dinamicamente nuovi thread al bisogno.

2.1 Algoritmo di Ranking

L'algoritmo che calcola il punteggio del ranking si basa su l'attualità delle recensioni; ossia più una recensione è stata creata recentemente, maggiore sarà la sua influenza nel punteggio; infatti, ogni recensione ha l'attributo che rappresenta il tempo di creazione ottenuto con la funzione `System.currentTimeMillis()`. Quest'ultimo viene poi diviso per il prodotto tra il numero delle recensioni moltiplicato per 10, per fare un calcolo equilibrato tra hotel con diverso numero di recensioni. In seguito, al calcolo aggiungiamo il numero delle recensioni, quindi la quantità, e il rate di ogni recensione, ossia la qualità delle recensioni stesse.

$$diffTempo = \frac{1}{(TempoAttuale - TempoCreazione) * 10}$$

$$somma = \sum_{i=0}^{nRecensioni} rate * \left(\frac{diffTempo}{nRecensioni * 10}\right)$$

$$Punteggio = (nRecensioni * 0.2) + (somma * 0.6) + (generalRate * 0.2)$$

Dove:

- *TempoAttuale* → tempo attuale del sistema;
- *TempoCreazione* → corrisponde al tempo in cui è stata creata la recensione.

2. 2 Persistenza dei File

Le informazioni come utenti registrati, recensioni, punteggi e ranking, devono essere salvate periodicamente, quindi, oltre ad aggiornali ogni qual volta viene effettuata un'azione è stato implementato un timer quando questo scatta vengono ricalcolati i ranking locali di ogni città e vengono salvati utenti ed hotel sui rispettivi file json per avere sempre i file consistenti.

2.3 Scelte Lato Server

- Per conservare le informazioni sugli utenti registrati utilizziamo una lista: `List<Utente>` che contiene tutti gli utenti registrati. Questa lista è inizializzata vuota nel caso non ci sia un file utente al percorso indicato e in questo caso ne crea uno vuoto; altrimenti il file viene caricato e scritto in questa lista. Quest'ultima non è implementata con Concurrent poiché solo il server, allo scadere del timer, chiama e modifica la lista degli utenti.
- Per le informazioni degli hotel utilizziamo anche qui una lista: `ConcurrentLinkedQueue<Hotel>`. Questa viene inizializzata tramite GSON dal file specificato nel *fileHotelPath* e diamo per scontato che il file passato contenga informazioni adeguate su un insieme di hotel
- Creiamo un set delle città ammesse estraendole dal file degli hotel passato in input per verificare se cerchiamo città che non sono gestite nel nostro servizio
- Inizializziamo anche una variabile globale `ConcurrentHashMap<String, String> firstRanking` che indica i primi classificati per ogni città. Infatti il primo attributo della mappa è la città dell'hotel ed il secondo è il nome dell'hotel; questo viene utilizzato per verificare se il primo classificato cambia quando scriviamo una nuova recensione, in questo caso viene mandata la notifica.
- La gestione delle connessioni avviene tramite i thread con codice: *RichiesteClient.java* dove tramite l'invio di richieste di una specifica funzione viene restituito il risultato richiesto chiamando i metodi del ServerMain.
- Dopo abbiamo le funzioni chiamate dai thread che gestiscono i client:
- `registrazioneUtente(username, password)` → gestisce la registrazione di un nuovo utente restituendo un intero corrispondente ad un messaggio di verifica:
    1. Password mancante
    2. Utente già registrato
    3. Utente registrato con successo
- `loginUtente(username, password)` → gestisce il login di un utente già registrato, verificando se è già loggato e restituendo true o false in base al risultato dell'operazione;
- `showMyBedges(username)` → che restituisce l'ultimo bedge acquisito dall'utente tramite l'username (username passato dal thread del server e non dall'utente);
- `logOut(username)` → effettua il logout dell'utente con quell'username;
- `recensione(username, nameHotel, citta, rate, cleaning, position, services, quality)` → inserimento della recensione nell'hotel corrispondente e aumento del numero delle recensioni effettuate dall'utente con quell'username. Questa operazione ricalcola anche i punteggi totali aggiornando le medie di tutte le recensioni effettuate su quell'hotel, e il

ranking di quella città invocando il metodo `listHotelRanking(citta)` e salvando i dati aggiornati sul file degli hotel;

- `loggato(username)` → restituisce true o false se un utente è loggato o no;
- `listHotelRanking(citta)` → questo metodo calcola il punteggio per il ranking degli hotel e ordina la lista in base al ranking. All'interno viene chiamato il metodo `localListRanking(citta)` che restituisce la lista degli hotel di quella città. Viene anche chiamata la funzione `invioNotifiche(notifica)` → che invia a tutti gli utenti loggati l'avvenuto cambiamento del primo classificato indicando la città e il nuovo primo classificato.

2.4 Scelte Lato Client

- Parametri passati tramite il file di proprietà:
  - `SERVER_HOST` → l'indirizzo a cui collegarsi con il socket TCP;
  - `SERVER_PORT` → Porta dove il server è in ascolto;
  - `indirizzoMulticast` → indirizzo del socket multicast;
  - `portaMulticast` → porta associata al socket multicast;
- Una volta stabilita la connessione con il server viene stampato subito il menu (inviato dal server) con le possibili azioni che l'utente può effettuare:
  0 LogIn
  1 Registrati
  2 Mostra il mio bedge
  3 Cerca Hotel
  4 Cerca tutti gli Hotel
  5 Inserisci Recensione
  6 LogOut
  9 Exit
- Successivamente alla scelta dell'azione desiderata si invia al server la richiesta corrispondente all'azione e i dati inseriti da CLI, mettendoci in ascolto per la risposta
- Ricevuta la risposta verrà analizzata e stampato a video il messaggio corrispondente
- Nella funzione di scrittura della recensione viene controllato che i punteggi delle recensioni siano compresi tra 0 e 5 e nel caso di un numero errato chiediamo la re-immissione dei punteggi.
  - Sono previsti errori di digitazione, come l'inserimento di un numero non compreso nell'intervallo e l'immissione di una lettera, evitando così errori inaspettati, questo tramite la funzione: `leggiInputRecensione(userInput, richiesta)`
- Nella fase di logIn se la procedura va a buon fine viene fatta l'iscrizione al multicast tramite il metodo: `iscrizioneMulticast()` per permettere di ricevere le notifiche.
- Per ricevere le notifiche dal server utilizziamo un thread, sempre attivo, che, stampa ogni stringa significativa ricevuta sulla connessione UDP.

3 Strutture Dati

3.1Mantenimento Recensioni

Per il mantenimento dei dati del server come utenti, recensioni e hotel ho utilizzato i file json; le recensioni sono inserite negli hotel nell'apposita proprietà in una lista di recensioni che contengono il nome dell'hotel, la città, il rate, dato dal recensore, e una mappa con le proprietà valutate.

3.2 Mantenimento Utenti Registrati

Per gli utenti invece ho creato un file json composto da una lista con username, password, numero di recensioni e un flag che indica se l'utente è loggato.

Al primo avvio del server viene chiamata la rutine `inizializzaFileUtenti()` che verifica la presenza del file nel path specificato e, nel caso in cui non fosse presente lo crea. Nel caso in cui il file esista provvede a caricarlo nella variabile *utentiRegistrati* che, come prima cosa, forza lo stato di tutti gli utenti a "non loggato" per non creare problemi in caso di interruzione, per errore, del server con utenti ancora loggati.

3.3 Classe Hotel

La classe Hotel contiene informazioni di base e informazioni aggiuntive:

- `punteggio` → utilizzato per calcolare la posizione nel ranking;
- `nRecensioni` → numero di recensioni effettuate per quell'hotel;
- `recensioni` → con tipo: `List<String>`;
- `posizioneRanking` → che corrisponde alla posizione nel ranking.

Viene implementato il confronto in base al punteggio tra hotel tramite l'interfaccia Comparable.

I metodi sono quasi tutti get e set standard tranne:

- `setRate()` → che imposta il punteggio dell'hotel approssimando alla seconda cifra decimale;
- `toString1()` → restituisce le informazioni dell'hotel sottoforma di stringa;
    - Le informazioni restituite sono volontariamente limitate (senza possibilità di visualizzare punteggio, posizione nel ranking e recensioni)

Ogni recensione ha un attributo che indica la sua data di creazione. Questo ci servirà nel calcolo del ranking dato che più la recensione sarà recente e più influirà maggiormente sul calcolo del punteggio.

3.4 Classe Utente

La classe utente contiene le seguenti informazioni:

- `username` → username dell'utente creato;
- `password` → password dell'utente;
- `nRecensioni` → numero di recensioni effettuate dall'utente;
- `badge` → ultimo riconoscimento;
- `loggato` → flag per sapere se l'utente è loggato;
- `tempCreazione` → corrisponde al tempo di creazione della recensione, viene inizializzato al momento della creazione di una nuova istanza tramite `System.currentTimeMillis()`.

In particolare, osserviamo il metodo:

- `getBadge()` → quando viene invocato, aggiorna il badge dell'utente dividendo il numero delle recensioni per 3 e restituendo il badge corrispondente.

3.5 Classe Recensioni

La classe recensioni contiene le informazioni relative ad una recensione di uno specifico hotel, ed ha le seguenti proprietà:

- `nameHotel` → contiene il nome dell'hotel recensito;
- `citta` → citta dell'hotel recensito;
- `rate` → voto generale assegnato all'hotel;
- `ratings` → voto per ogni categoria, di tipo: `Map<String, Integer> ratings` dove il primo attributo di tipo stringa corrisponde alla categoria e l'intero al voto corrispondente.

In particolare osserviamo il metodo:

- `setRatings(cleaning, position, services, quality)` → questo metodo imposta i voti relativi alle categorie specifiche inserendoli nella mappa.

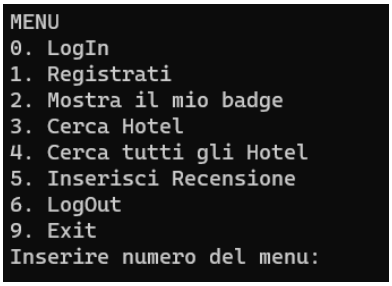
4 Uso del Servizio HOTELIER

All'avvio l'utente si troverà di fronte un menu di scelta con cui può indicare l'azione desiderata con la scrittura di un numero tra 0 e 6 oppure uscire dal servizio con il numero 9.

Il menù viene riportato ogni volta alla fine di un azione e del suo relativo messaggio di feedback.

Non tutte le azioni sono possibili da utenti non loggati. Infatti, le funzioni per scrivere una recensione, di mostrare il badge e il logout (corrispondenti ad i numeri: 2, 5, 6) richiedono di essere loggati.

Il resto delle funzioni sono sempre disponibili in ogni momento, di modo che ogni utente possa ricercare informazioni sull'hotel e nel caso registrarsi o fare il login (0,1,3,4,9).



5 Panoramica Thread e Sincronizzazione

Lato **server**:

- Thread
  - Nel server utilizziamo un *CachedThreadPool* per la gestione dinamica di un pool di thread per ogni richiesta di connessione client.
  - Viene schedulato anche un thread per il ricalcolo del ranking e per il salvataggio dei dati, mediante il *timer*, dato che java esegue il task in un thread separato.
- Sincronizzazione
  - Per sincronizzare le parti critiche ho inizializzato le variabili utilizzate simultaneamente, che potrebbero creare problemi in lettura e scrittura, in modo concorrente: `ConcurrentLinkedQueue<Hotel>` e `ConcurrentHashMap<String, String>`

Lato **client**:

- Thread
  - Nel client utilizziamo un thread per gestire la ricezione di messaggi, indipendentemente dal thread principale: `Thread riceveTask`

6 Comandi Compilazione ed Esecuzione

Per compilare uso un unico comando visto che il server e il client sono nella stessa cartella:

```
javac -cp "gson-2.10.1.jar" *.java
```

Successivamente per eseguirli dobbiamo usare:

- **Server**: `java -cp ".;gson-2.10.1.jar" HotelierServerMain`
- **Client**: `java HotelierClientMain`

Per creare il file jar del **server** ci sono due possibilità:

1. Senza l'uso del *manifest*:
   1. `jar -cvfe HotelierServerMain.jar HotelierServerMain *.class`
   2. Per eseguirlo dobbiamo quindi dichiarare la libreria esterna:
      `java -cp "HotelierServerMain.jar;gson-2.10.1.jar" HotelierServerMain`
2. Con l'uso del *manifest* (Preferibile):
   1. `jar cvfm HotelierServerMain.jar MANIFEST.MF *.class`
   2. Per eseguirlo:
      `java -jar HotelierServerMain.jar`

Per creare il file jar del **client**:

- `jar cvfe HotelierClientMain.jar HotelierClientMain HotelierClientMain.class`
- Per eseguirlo:
  `java -jar HotelierClientMain.jar`