RELAZIONE HOTELIER

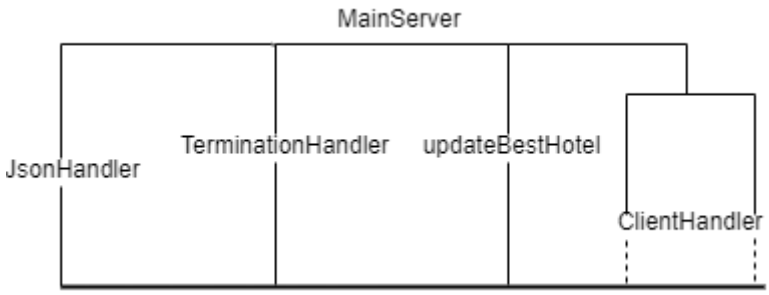
Carolina Ferrari-Laboratorio di reti 2023/2024

Sommario

1. Introduzione

Il progetto è stato realizzato con **Java 21** e la libreria **Gson,** in particolare la versione **2.11.0,** per la serializzazione e deserializzazione delle informazioni relative a utenti, hotel e recensioni.
Consiste in una interazione tra client e server tramite una **socket TCP** in cui le operazioni di lettura e scrittura vengono gestire tramite **BufferReader** e **PrintWriter**.
Presenta inoltre una parte dedicata a un canale multicast gestito tramite **socket UDP** a cui si uniscono tutti gli utenti una volta che si sono loggati in attesa di notifiche sul cambio del migliore hotel nelle città supportate.

2. Main server



Il server fa uso delle funzionalità di **Java I/O** e di **ThreadPool.** All'avvio il server legge dal file *server.properties* i parametri di configurazione, ossia:

- la porta su cui sarà in ascolto il server
- il tempo massimo per la terminazione dei task non terminati alla chiusura del pool
- il numero minimo di giorni dopo i quali un utente può inserire una recensione per lo stesso albergo
- ogni quanti minuti si controllano i ranking di tutti gli hotel autonomamente
- la porta di multicast.

Viene creata un'istanza della classe ***JsonHandler*** per la gestione della serializzazione e della deserializzazione passando come parametri i nomi dei file json di hotel, utenti e recensioni e il nome del file di testo in cui si specificano le città supportate per l'inserimento degli hotel.
Si crea anche un'istanza della classe ***updateBestHotel*** per la gestione dell'invio dei messaggi sul canale multicast. Si avvia quest'ultima attraverso l'uso dell'interfaccia **SchedulerExecutorService** con l'uso di **scheduleAtFixedRate** che dà la possibilità di eseguire un task periodicamente dopo un certo periodo di tempo.
Alla chiusura del Server si avvierà la classe ***TerminationHandler*** per gestire correttamente la terminazione.
A questo punto il server si mette in ascolto sulla socket e ogni volta che arriva una nuova richiesta da un client avvia un **ClientHandler** per soddisfare le richieste di un determinato utente. I thread dedicati sono gestiti usando un **CachedThreadPool**.

2.1. JsonHandler

La classe JsonHandler all'avvio **deserializza** gli utenti presenti nel file Json tramite *userReader*. Successivamente inizializza l'HashMap degli hotel che si occupa di tenere in

memoria gli hotel in base alla città. Inizialmente si inseriscono solo le città consentite dal file city.txt come chiave e si inizializza la lista degli hotel a nulla tramite la funzione *initializeHotels*. Non sarà possibile inserire hotel in città diverse da quelle specificate all'avvio. Successivamente si deserializzano gli hotel dal file Json tramite *hotelReader* (se la città dell'hotel deserializzato non è presente l'hotel non viene inserito). Quando si hanno tutti gli hotel in memoria si inizializza l'HashMap relativa alle recensioni con le chiavi degli hotel, poi si deserializzano le recensioni e si associa a ogni recensione l'hotel e l'utente corrispondente rispettivamente tramite nome Hotel/città e username.
Una volta che tutte le recensioni sono presenti nel programma si inizializza il punteggio degli hotel in base a esse. Si aggiorna il numero di recensioni per i singoli utenti e si ordinano gli hotel per città in base al punteggio calcolato precedentemente.
Quando si avvia la terminazione del server si **serializzano** tutte le informazioni su hotel, utenti e recensioni nei rispettivi file. Come scelta implementativa non si serializzano i punteggi degli alberghi in quando si inizializzano tramite le recensioni e nemmeno l'istanze di Hotel e User relativi alle recensioni in quanto saranno poi inizializzati al successivo avvio.

2.2. updateBestHotel

Questa classe ha il compito di **mandare messaggi** per l'aggiornamento dell'hotel migliore in una determinata città. È avviata dal server e ogni 1440 minuti (un giorno). Il metodo run della classe itera su tutte le città e chiama il metodo updateBest che tramite l'uso di una HashMap conosce l'hotel migliore precedente e restituisce l'istanza di hotel se quest'ultimo è cambiato altrimenti null. Il metodo updateBest è anche chiamato ogni volta che si effettua una nuova recensione. Gli altri metodi della classe servono invece per mandare il messaggio del cambiamento sul gruppo multicast.

2.3. TerminationHandler

Si occupa di far **terminare** tutti i **thread istanziati** di ClientHandler e updateBestHotel.
Prima esegue lo *shutdown*, se dopo *maxDelay* se ancora non sono terminati invoca *shutdownnow*.
Successivamente si occupa di chiamare la funzione per la serializzazione delle informazioni e, se ci sono utenti loggati, forza il logout.

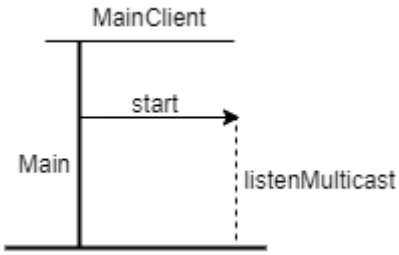
2.4. ClientHandler

Questa classe ha il compito di **gestire le richieste** di un determinato utente finché non esce o effettua il logout. Inizialmente manda i comandi che l'utente ha a disposizione poi si mette in attesa di riceverli.
Le richieste dell'utente non sono case sensitive tranne il logout e register per permettere una maggiore flessibilità nella scelta dell'username e della password.
I possibili comandi sono:

- Register <username> <password>: effettua la registrazione di un utente e il successivo login. Controlla se l'utente è già loggato o se l'username è già in uso.

- Login <username> <password>: effettua il login di un utente già registrato in precedenza che non sia già loggato. Quando si effettua il login si setta il Badge dell'utente grazie al numero di recensioni. Finché l'utente non è loggato il badge sarà RECENSORE anche se ha effettuato delle recensioni in precedenza.
- Logout: se l'utente è loggato esegue il logout e chiude la connessione con il Client.
- SearchHotel "<Hotel da cercare>" "<Città>": cerca un hotel tramite il nome e la città in cui si trova e restituisce le relative informazioni. Il nome dell'Hotel e la città devono essere inseriti tra virgolette ("...") sia in questo che nei successivi comandi. Se la città è presente ma l'hotel non è stato trovato si restituisce una lista di tutti gli hotel presenti nella città.
- SearchAllHotels "<Città>": cerca tutti gli hotel presenti in una città e li restituisce insieme alle relative informazioni.
- InsertReview"<Hotel da cercare>" "<Città>" x x x x x: le x rappresentano i punteggi da 0 a 5 relativi rispettivamente pulizia, posizione, servizi, qualità e punteggio globale. Se l'utente è loggato inserisce una recensione a un determinato hotel e si aggiorna il Badge dell'utente. Se si inseriscono punteggi sbagliati (numeri non compresi tra 0 e 5 o caratteri non numerici) la recensione non viene inserita.
  Se lo stesso utente ha effettuato una recensione per lo stesso hotel entro 30 giorni dalla data attuale la recensione non viene inserita. Quando si inserisce la recensione vengono aggiornati i punteggi globali e parziali dell'hotel, la data media di tutte le date delle recensioni dell'hotel, si ricalcola il ranking della relativa città e si controlla se è cambiato il migliore hotel.
- ShowMyBadges: mostra a un utente loggato il relativo badge.
- Exit: se l'utente è loggato effettua anche il logout, altrimenti chiude solo la connessione con il Client.

3. Main client



Il client fa uso delle funzionalità di **Java I/O.** All'avvio il client legge dal file *client.properties* i parametri di configurazione, ossia:

- la porta su cui sarà in ascolto il server
- la porta di multicast.

Viene creata una socket e finché la connessione non viene chiusa il client invia i comandi al server e attende risposta. Quando il server manda la conferma del login o della registrazione effettuati con successo il client avvia un **ListenMulticast** il cui compito è di rimanere in ascolto delle notifiche sul

canale multicast. Quando il server manda in risposta “exit” il client esce dal gruppo multicast, se il cliente era loggato, e interrompe il thread, successivamente stampa il messaggio di uscita e chiude la socket.

3.1. ListenMulticast

E’ una classe che implementa Runnable con il solo scopo di rimanere in **ascolto delle notifiche** sul canale multicast. Inizialmente il thread fa la join al gruppo tramite l’indirizzo IP poi attende i messaggi, li elabora e li stampa finché l’utente non lascia il gruppo e chiude la socket.

4. Scelte Implementative

L’implementazioni delle classi *User, Hotel* e *Review* hanno degli aspetti in comune. Ogni classe utilizza dei getter e dei setter per accedere ai relativi campi.
Utilizzano una **ConcurrentHashMap** in una variabile statica condivisa a tutte le istanze per tenere in memoria e elaborare i dati delle rispettive classi.

4.1. User

Per la classe User l’HashMap “registerUser” ha come chiave l’username che identifica l’utente e come valore l’utente stesso in modo tale da poter risalire ai suoi dati attraverso l’username.

Il metodo *setUser* inserisce un utente nell’HashMap se non già presente.

È implementato il metodo *getAllUsers* che restituisce una lista di tutti gli utenti presenti nel sistema.

Per quando riguarda il Badge di un utente ogni 2 recensioni aumenta di livello, per esempio fino a 2 recensioni è recensore, da 2 a 4 è recensore_esperto... da 8 recensioni in poi è considerato contributore_super.

Per facilitare la comparazione di due utenti si è effettuato l’override dei metodi equals e hashCode per far sì che gli utenti siano confrontati attraverso l’uername.

4.2. Hotel

Per la classe Hotel l’HashMap “hotels” ha come chiave la città e come valore una **CopyOnWriteArrayList** di tutti gli hotel presenti in essa. Si è scelto la città come chiave per semplificare la ricerca e l’ordinamento degli hotel.

La classe presenta un’altra HashMap che ha come chiave la città e valore l’hotel migliore. È necessaria per il confronto quando si aggiornano i punteggi degli hotels e notificare se c’è stato un cambiamento.

La variabile privata *timeReview* indica la data media di quando sono state effettuate tutte le recensioni e serve per calcolare il punteggio dell’hotel.

Il metodo *getAllHotels* restituisce una lista di tutti gli hotel presenti nel sistema, mentre il metodo *getAllCity* restituisce tutte le città in cui sono presenti gli hotel.

Il metodo setHotel inizializza i ratings di un hotel a 0 e lo inserisce nella lista corrispondente dell'HashMap.

Il metodo *initializeHotel* inizializza l'HashMap con tutte e sole le città che il sistema supporta.

I metodi sortHotel, initializeBest e updateBest sono di supporto per l'aggiornamento del migliore hotel nella città. sortHotel ordina gli hotel presenti nella lista in base alla città in relazione allo score totale. initializeBest inizializza l'HashMap dei migliori hotel per la prima volta inserendo il primo della lista. updateBest controlla se il primo hotel nella lista di una determinata città è sempre uguale al precedente e in caso contrario restituisce il nuovo hotel.

Per facilitare la comparazione di due hotels si è effettuato l'override dei metodi equals e hashCode per far sì che due hotels siano confrontati attraverso il nome e la città.
Per avere una rappresentazione in Stringa dell'hotel si è effettuato l'override anche del metodo toString.

4.3. Review

Per la classe Review l'HashMap "hotelReview" ha come chiave l'hotel e come valore una **CopyOnWriteArrayList** di tutte le recensioni che sono state fatte per quell'hotel. Si è scelto l'hotel come chiave per semplificare l'aggiornamento del punteggio relativo all'hotel.

La variabile statica MIN_DAYS_BETWEEN_REVIEWS indica quanti giorni devono passare tra una recensione e l'altra dello stesso utente per lo stesso hotel.

Il metodo *setReview* inserisce una recensione nella lista appropriata di "hotelReview"

Il metodo *initializeHotelScore* inizializza il punteggio degli hotel in base alle recensioni lette dal file Json, setta l'utente per ogni recensione e aggiorna il suo numero delle recensioni effettuate.

Il metodo *calculateHotelScore* calcola il punteggio di un albergo facendo una somma pesata di vari parametri. Si calcola la media dei punteggi singoli e, come per il punteggio globale, si normalizza a 100. Le due medie nella somma pesata avranno peso di 0,5. Gli altri parametri sono i voti totali delle recensioni con peso 0,2 e i giorni medi di differenza tra le date delle recensioni e la data odierna con punteggio negativo di -0,2 per penalizzare gli hotel che hanno recensioni più vecchie. Non ci possono essere punteggi negativi, quindi se alla fine del calcolo il punteggio è minore di 0 si setta pari a 0.

Il metodo *addReview* inserisce una nuova recensione nella lista appropriata di "hotelReview" se l'utente non ne ha effettuata un'altra per lo stesso hotel entro 30 giorni. Si aggiornano i parametri relativi all'hotel, si ricalcola il punteggio, si aggiorna la lista degli hotel nella città e si controlla se è cambiato il primo classificato per mandare una notifica.

Il metodo *getAllReviews* restituisce una lista con tutte le recensioni presenti nel sistema.

Il metodo *initializeReview* inizializza l'HashMap con tutti e solo gli hotel che il sistema supporta.

I metodi *stringTime* e *dataTime* servono per la gestione delle date e la trasformazione in stringa o viceversa per facilitarne l'utilizzo e la serializzazione.

5. Concorrenza e strutture dati

Come già detto in precedenza si è scelto di utilizzare una ConcurrentHashMap per memorizzare Hotel, utenti e revisioni in modo tale da poter gestire accessi concorrenti da più client grazie alla concurrent collection di java. Per hotel e utenti i valori sono memorizzati in una CopyOnWriteArrayList sempre per gestire gli accessi concorrenti senza avere effetti collaterali.

Per quanto riguarda la classe Hotel sono presenti altre due HashMap.

L'HashMap per i punteggi singoli è implementata come una ConcurrentHashMap per evitare effetti collaterali nel caso due utenti effettuino una recensione per un hotel nello stesso momento.

L'HashMap per gli hotel migliori invece non ha problemi di concorrenza quando viene inizializzata in quanto l'inizializzazione la effettua il server stesso. Visto che non viene utilizzata spesso si è pensato che non fosse necessaria una ConcurrentHashMap ma si è solo sincronizzato il pezzo di codice per il relativo aggiornamento.

Per quanto riguarda il metodo addReview si è ritenuto necessario sincronizzare una parte di esso per l'aggiornamento della data media di recensioni per hotel nel caso due utenti inseriscano nello stesso momento una recensione per lo stesso hotel.

I metodi della classe Hotel updateVote, addRatings e addRate sono stati sincronizzati per evitare aggiornamenti concorrenti.

Per la gestione dei client da parte del server si è deciso di avviare un newCachedThreadPool in quanto non ha limiti nella dimensione del pool e quindi non rifiuta nessuna connessione.

6. Compilazione

Per quanto riguarda la corretta esecuzione del programma c'è da tenere presente che va eseguito prima il server e dopo il client, in caso contrario il client non si riesce a collegare.

```
PS C:\Users\ferra\OneDrive\Desktop\lab3\hotelier> java MainClient
Errore: Connection refused: connect
```

6.1. Esecuzione da jar

Per quanto riguarda la compilazione da file jar per eseguire il server il comando è:
java -jar MainServer.jar

```
PS C:\Users\ferra\OneDrive\Desktop\lab3\hotelier2> java -jar MainServer.jar
[SERVER] In ascolto sulla porta: 12000
```

Mentre per quanto riguarda l'esecuzione del client il relativo comando è:
java -jar MainClient.jar

```
PS C:\Users\ferra\OneDrive\Desktop\lab3\hotelier2> java -jar MainClient.jar
Benvenuto, questi sono i possibili comandi:
Register <username> <password>
Login <username> <password>
Logout
SearchHotel "Nome Hotel" "Nome Città"
SearchAllHotels "Nome Città"
InsertReview "Nome Hotel" "Nome Città" x x x x x
ShowMyBadges
Exit
Le x rappresentano numeri interi tra 1 e 5 per pulizia, posizione, servizi, qualità e punteggio globale
```

6.2. Compilazione ed esecuzione da file sorgente

Per la compilazione da file sorgente i comandi per creare il file .class del Server ed eseguirlo sono:

javac -cp .;gson* MainServer.java

java -cp .;gson* MainServer

```
C:\Users\ferra\OneDrive\Desktop\lab3\hotelier>javac -cp .;gson\* MainServer.java

C:\Users\ferra\OneDrive\Desktop\lab3\hotelier>java -cp .;gson\* MainServer
[SERVER] In ascolto sulla porta: 12000
```

Per la compilazione e l'esecuzione del Client invece i comandi sono:

javac MainClient.java

java MainClient

```
C:\Users\ferra\OneDrive\Desktop\lab3\hotelier>javac MainClient.java

C:\Users\ferra\OneDrive\Desktop\lab3\hotelier>java MainClient
Benvenuto, questi sono i possibili comandi:
Register <username> <password>
Login <username> <password>
Logout
SearchHotel "Nome Hotel" "Nome Città"
SearchAllHotels "Nome Città"
InsertReview "Nome Hotel" "Nome Città" x x x x x
ShowMyBadges
Exit
Le x rappresentano numeri interi tra 1 e 5 per pulizia, posizione, servizi, qualità e punteggio globale
```

7. Esempio

Di seguito è riportato un esempio minimo di come si può utilizzare il servizio hotelier.

```
showmybadges
Utente non loggato
login carolina gr
Login effettuato
insertreview "hotel roma 4" "RoMa" 4 5 3 4 5
Recensione inserita con successo
Nuovo hotel migliore a Roma: Hotel Roma 4
showmybadges
CONTRIBUTORE
searchhotel "hotel roma" "roma"
Hotel non trovato.
Altri hotels presenti a roma:
Hotel Roma 4
Hotel Roma 3
Hotel Roma 1
Hotel Roma 2
Hotel Roma 5
Hotel Roma 6
Hotel Roma 7
Hotel Roma 8
Hotel Roma 9
Hotel Roma 10
```