UNIVERSITÀ DI PISA

Relazione Progetto Hotelier

Aprile Filippo
Corso A
580175

a.a 2023/24

Contents

1 Introduzione

Il progetto Hotelier è stato sviluppato in Java 21 ed utilizza un interfaccia a linea di comando. Le classi del progetto sono state suddivise nei seguenti tre package principali:

- **hotelier client**, contenente le classi necessarie per l'implementazione del client.
- **hotelier common**, contenente le classi condivise dal server e dal client.
- **hotelier server**, contenente le classi necessarie per l'implementazione del server.

La trasmissione dei pacchetti TCP avviene mediante **protocollo di tipo richiesta/risposta**, illustrato in seguito, utilizzando:

- **socket bloccanti**, lato client.
- **NIO**, lato server. La gestione dei singoli pacchetti viene affidata ad una threadPool al fine di migliorare il throughput.

Le notifiche dei cambiamenti delle prime posizioni dei rank locali vengono effettuate tramite protocollo UDP utilizzando delle **mutlicastSocket**, implementate nelle seguenti classi:

- **HotelierClientMulticastReciever**, lato client.
- **HotelierServerMulticastSender**, lato server.

I metodi per la registrazione e le notifiche per il cambiamento dei rank locali vengono effettuati tramite RMI. Le notifiche sono state implementante mediante il meccanismo delle callback.

Sono state utilizzate le seguenti librerie:

- **commons-lang**, per il confronto di stringhe ed il parsing dell' input dell' utente.
- **gson**, utilizzata per la serializzazione e deserializzazione in json di:
  - **files**, config server/client, dati da persistere lato server.
  - **oggetti da inviare in rete**, mediante: pacchetti TCP e callback RMI.

1.1 Setup

All' interno dello zip consegnato sono presenti le folder **HotelierServer** e **HotelierClient**, contenenti enntrambe i rispettivi file json di configurazione per il server e per il client.

Inoltre all' interno di **HotelierServer** sono presenti anche i seguenti file per la persistenza delle strutture su disco:

- **hotels.json**, contenente la lista degli hotel serializzati in json. Se mancante il server termina e ne segnala la mancanza.
- **reviews.json**, contenente la lista delle recensioni pubblicate serializzate in json, creato dal server allo startup se mancante.
- **users.json**, contenente la lista degli utenti regisitrati serializzati in json, creato dal server allo startup se mancante.

In caso si desideri modificare i config del server o del client è sufficiente modifare i rispettivi file di configurazione.

Per testare il progetto occore avviare il server e successivamente il client tramite i corrispondenti runnable jar file presenti nello zip invocando i rispettivi comandi *java -jar client.jar* per il client e *java -jar server.jar* per il sever. In caso di errore assicurasi che:

- siano settati corretamente i permessi di lettura e scrittura;
- client e server possano operare su rete privata (firewall).

2 Strutture Dati

2.1 Hotels

Gli hotel sono rappresentati dalla classe **HotelierHotel**, contenuta nel package ***unipi.aprile.filippo.hotelier.common.entities***. Ogni hotel è formato da:

- **id**, id dell' hotel.
- **name**, nome dell' hotel.
- **description**, breve descrizone dell' hotel.
- **city**, città in cui è situato l' hotel.
- **phone**, numero di telefono dell' hotel.
- **services**, lista di servizi offerti dall' hotel.
- **rate**, rate dell' hotel (compreso tra 0 e 5).
- **ratings**, ratings dell' hotel (compresi tra 0 e 5).
- **reviewCount**, numero di recensioni dell' hotel.
- **rank**, rank globale dell' hotel.
- **localRank**, rank locale dell' hotel rispetto agli altri hotel situati nella sua stessa città.

  I campi: rate, ratings, reviewCount, rank e localRank sono ***mutabili*** quindi per rendere la classe **thread-safe** i rispettivi **metodi getter e setter sono synchronized**. I restanti campi sono tutti immutabili.

2.2 Utenti

Gli utenti sono rappresentati dalla classe **HotelierUser**, contenuta nel package ***unipi.aprile.filippo.hotelier.common.entities***. Ogni utente è formato da:

- **username**, username dell' utente.
- **password**, password dell' utente.
- **badge**, distintivo dell' utente.
- **reviewCount**, numero di recensioni effettuate dall' utente.

I campi: badge e reviewCount sono ***mutabili*** quindi per rendere la classe **thread-safe** i rispettivi **metodi getter e setter sono synchronized**. I restanti campi sono tutti immutabili.

Il badge è un **enum** che assume i seguenti valori al raggiungimento del corrispondente numero di recensioni:

- **recensore**, valore di default assegnato quando utente si registra.
- **recensore esperto**, 2 recensioni.
- **contribuente**, 3 recensioni.
- **contribuente esperto**,4 recensioni.
- **super contribuente**, 5 recensioni.

Le soglie del numero di recensioni sono settate in modo da poterne testare facilmente il funzionamento da interfaccia a linea di comando.

In un caso reale si potrebbe utilizzare una funzione esponenziale come quelle usate per il calolo dei livelli in base all' esperienza.

2.3 Recensioni

Le recensioni sono rappresentate dalla classe **HotelierReviews**, contenuta nel package ***unipi.aprile.filippo.hotelier.common.entities***. Ogni recensione è formato da:

- **username**, username dell' utente che ha effettuato la recensione.
- **hotelID**, id dell' hotel riferito dalla recensione.
- **rate**, rate assegnati all' hotel riferito dalla recensione.
- **rating**, ratings assegnati all' hotel riferito dalla recensione.
- **timestamp**, timestamp della pubblicazione della recensione.

I campi sono tutti immutabili, quindi la classe risulta **thread-safe**.

2.4 Rank locali

I rank locali sono rappresentati dalla classe **HotelierLocalRank**, contenuta nel package ***unipi.aprile.filippo.hotelier.common.entities***. Ogni rank locale è formato da:

- **city**, città del rank locale.
- **hotels**, lista di hotel situati in quella città.

Il campo hotel è ***mutabile*** mentre il campo city è ***immutabile***. Non implementa nessun meccanismo di sincronizzazione in quanto la classe non viene utilizzata in contesto multithreading e quindi comporterebbe un overhead inutile.

2.5 Comandi

I comandi sono rappresentati dalla classe **HotelierClientCommand**, contenuta nel package ***unipi.aprile.filippo.hotelier.client.cli.command***. Ogni comando è formato da:

- **name**, nome del comando.
- **arguments**, argomenti del comando.
- **commandType**, tipo del comando.

I campi sono tutti ***immutabili***. Il badge è un **enum** utilizzato per delegare il comando all' handler di competenza, il quale può assumere i seguenti valori:

- **TCP**, comando Tcp.
- **RMI**, comando Rmi.
- **LOCAL**, comando locale (non viaggia sulla rete).

2.6 Pacchetti Tcp

I pacchetti Tcp sono rappresentati dalle rispettive classi contenute nel package ***unipi.aprile.filippo.hotelier.common.network.packets***. Ogni classe rappresentante un pacchetto estende la classe astratta **HotelierPacket** in modo da poterne effettuare il multiplexing e demultiplexing. Sono presenti le seguenti classi:

- **HotelierPacketLogin**, contiene username e password inviati nella richiesta di *login*.
- **HotelierPacketLoginResponse**, contiene risposta/esito richiesta di *login*.

- **HotelierPacketLogout**, inviato a seguito di richiesta di *logout* (vuoto).
- **HotelierPacketLogoutResponse**, contiene risposta/esito richiesta di *logout*.
- **HotelierPacketHotel**, contiene nome e città hotel richiesto tramite *searchHotel*.
- **HotelierPacketHotelResponse**, contiene hotel richiesto, null se non trovato.
- **HotelierPacketHotelList**, contiene città hotel richiesti tramite *searchAllHotels*.
- **HotelierPacketHotelListResponse**, contiene lista hotel richiesti.
- **HotelierPacketReview**, contiene username, id, rate e ratings inviati nella richiesta di *insertReview*.
- **HotelierPacketReviewResponse**, contiene risposta/esito richiesta di *insertReview*.
- **HotelierPacketBadge**, inviato a seguito di richiesta di *showMyBadges* (vuoto).
- **HotelierPacketLBadgeResponse**, contiene distintivo dell' utente.
- **HotelierPacketErrorResponse**, contiene messaggio di errore.

Per ogni richiesta possibile del client sono presenti i relativi pacchetti di richiesta e risposta. In caso di richiesta non andata a buon fine il server invia un ***HotelierPacketErrorResponse*** contenente il rispettivo messaggio di errore (mancanza di permessi, risorsa non trovata ...).

3 Protocollo comunicazione Tcp

Per la comunicazione Tcp viene utilizzato un ***protocollo di tipo richiesta/risposta***, il quale consiste nell' invio di messaggi composti nell' ordine da:

- **lunghezza paccchetto**, lunghezza pacchetto serializzato in bytes.
- **id pacchetto**, id del pacchetto utilizzato per il multplexing e demultiplexing.
- **payload**, bytes del pacchetto serializzato in json.

3.1 Multiplexing/Demultiplexing paccheti Tcp

La classe statica **HotelierPacketRegistry**, contenuta nel package ***unipi.aprile.hotelier.client*** espone i seguenti metodi per effettuare multiplexing e demultiplexing dei pacchetti Tcp:

- **getPacketFromID** (demultiplexing), prende come parametri pacchetto serializzato in bytes e relativo id restituendo il pacchetto deserializzato.
- **getIDFromPacket** (multiplexing), prende come parametro un pacchetto e ne restituisce l'id.

4 Client

Il main del client è implementato nella classe **HotelierClientMain**, contenuta nel package ***unipi.aprile.filippo.hotelier.client***, ed ha lo scopo di:

- **inizializzare il client**, andando a recuperare i config da disco se presenti oppure creando un file config con valori di default.
- **avviarlo**, instanziando le seguenti classi:
  - **HotelierClientRmi**, per l' invocazione della register remota su Server-Rmi e la ricezione delle notifche sui cambiamenti dei rank locali delle città di interesse tramite callback Rmi.
  - **HotelierClientMulticastReciever**, per la ricezione delle notifiche Udp a seguito del cambiamento dell' hotel in prima posizione di qualsiasi rank locale.
  - **HotelierClientCLI**, per la gestione dei comandi effettuati dall' utente tramite interfaccia a linea di comando.

4.1 Config

I config del client sono rappresentanti dalla classe **HotelierClientConfig**, contenuta nel package ***unipi.aprile.filippo.hotelier.client.config***, la quale presenta i seguenti campi:

- **tcpPort**, porta server Tcp;
- **rmiPort**, porta del registro Rmi;
- **mcastPort**‘, porta della socket mutlicast;
- **serverAddress**, indirizzo sever socket Tcp e registro Rmi:
- **rmiRemoteReference**, nome che identifica stub server nel registro Rmi:
- **mcastAddress**, indirizzo della sockert multicast:

Nel package è presente inoltre la classe **HotelierClientConfigManager**, la quale espone i metodi *createDefaultConfig* e *loadClientConfig*, aventi rispettivamente i compiti di:

- *createDefaultConfig*, crea file config con valori di default (sono gli stessi presenti nel file config fornito) e lo salva su disco;
- *loadClientConfig*, deserializza config da disco;

4.2 Interfaccia a linea di comando

La classe **HotelierClientCLI**, contenuta nel package ***unipi.aprile.filippo .hotelier.client.cli*** avvia un thread per la gestione dei comandi inseriti dall' utente tramite interfaccia a linea di comando. Il thread esegue un ciclo continuo in cui attende l'input dell'utente. Quando l'input viene ricevuto, il thread esegue le seguenti operazioni:

1. controlla che input non sia vuoto, nel caso skippa iterazione;
2. ottiene il comando tramite parsing dell' input implementato nella classe *HotelierClientCommandParser*;
3. controlla che il comando sia supportato, nel caso skippa iterazione;
4. controlla se è stato inserito un comando di exit, nel caso;
    - chiude tutte le risorse associate al client;
    - termina il client.
5. gestisce il comando e stampa la relativa risposta;
6. tramite la risposta controlla se è stato inserito un comando di login gestito con successo, nel caso:
    - richiede all' utente di inserire le città di interesse per le quali vuole ricevere notifiche dei cambiamenti dei rank locali;
    - controlla se non è stata inserita nessuna città di interesse, nel caso segnala un warning tramite stampa;
    - invoca il metodo remoto tramite Rmi per registrare callback su serverRmi per le città di interesse;
    - aggiunge il client al gruppo multicast per la ricezione delle notifiche Udp sul cambiamento dell' hotel in prima posizione di qualsiasi rank locale.
7. tramite la risposta controlla se è stato inserito un comando di logout gestito con successo, nel caso:
    - controlla se l' utente aveva registrato delle città di interesse, nel caso invoca il metodo remoto tramite Rmi per deregistrare la callback su serverRmi e resetta la mappa dei rank locali;

- rimuove il client al gruppo multicast per la ricezione delle notifiche Udp sul cambiamento dell' hotel in prima posizione di qualsiasi rank locale.

4.2.1 Parsing dei comandi

La classe *statica* **HotelierClientCommandParser**, contenuta nel package ***unipi.aprile.filippo.hotelier.client.cli.command.parser*** espone il metodo *parseCommand* per effuare il parsing dell' input inserito dall' utente e restituirne il relativo comando. Il metodo è implementato come segue:

1. setta il **nome del comando** all substring di input precedente al primo spazio;
2. setta **argomenti del comando** ad array delle stringhe contenute tra "" all' interno di input;
3. crea il relativo comando controllando che gli argomenti validi (interi compresi tra 0 e 5 per rate e ratings) e corretti in numero ;
4. restituisce il comando creato.

4.2.2 Gestione dei comandi

La classe **HotelierClientCommandHandler**, contenuta nel package ***unipi.aprile.filippo .hotelier.client.cli.command.handler*** espone il metodo *handleCommand* per smistare i comandi all' handler di competenza, implementato come segue:

- filtra i comandi in base al loro **tipo**:
  - **TCP**, viene delegato ad handler comandi Tcp;
  - **RMI**, viene delegato ad handler comandi Rmi;
  - **LOCALE**, viene delegato ad handler comandi Locali.

Funge inoltre da handler per i comandi locali, per i quali viene invocata la funzione di gestione specifica in base al loro nome.

All' interno di hotelier sono disponibili i seguenti comandi locali:

- **help**, restituisce la lista dei comandi disponibili;
- **showLocalRanks**: restituisce la lista dei rank locali aggiornati e ordinati per cui l' utente si è registrato. L' aggiornamento dei rank locale avviene mediante callback Rmi.

4.2.3 Gestione dei comandi Tcp

La classe **HotelierClientTcpHandler**, contenuta nel package ***unipi.aprile.filippo .hotelier.client.cli.command.handler*** espone il metodo *handleTcpCommand* per la gestione dei comandi Tcp, implementato come segue:

1. **crea pacchetto di richiesta Tcp** in base al nome del comando, avente come parametri gli argomenti del comando (null in caso di pacchetto non supportato);
2. **invia il pacchetto di richiesta Tcp**, crea il messaggio di richiesta contenente: lunghezza pacchetto serializzato + id + pacchetto serializzato e lo invia sulla socket bloccante al serverTcp;
3. **attende il pacchetto di risposta Tcp** attende il messagio di risposta e una volta ricevuto demultiplexa il pacchetto tramite id ricevuto e lo deserializza;
4. **restituisce risposta/esito della richiesta**, ottiene la risposta dal pacchetto di risposta Tcp e la restituisce;

All' interno di hotelier sono disponibili i seguenti comandi Tcp:

- **login**, richiesta di login avente username e password passati;
- **logout**, richiesta di logout;
- **searchHotel**, richiesta hotel avente nome e città passati;
- **searchAllHotels**, richiesta lista di hotel aventi città passati;
- **insertReview**, richiesta di inserzione di una recensione aventer parametri passati;
- **showMyBadges**, richiesta distintivo dell' utente;

In caso di ***IOException*** segnala all' utente impossibilità di contattare server Tcp tramite stampa.

La **socket non bloccante** viene creata nel costruttore il quale viene invocato nel metodo ***startClient*** del main. **OutputStream e InputStream** della socket vengono salvate come campi della classe in modo da poter essere utilizzati nei metodi che implementano invio e ricezione dei pacchetti Tcp.

HotelierClientTcpHandler implementa inoltre un metodo privato per assicurare la lettura di tutto il messaggio di risposta in quanto il server potrebbe effettuare scritture parziali utilizzando NIO.

La classe espone infine il metodo ***close*** per la chiusura della socket e delle stream associate. Viene invocato dalla classe **HotelierClientCLI** quando utente immette comando di *exit*.

4.2.4 Gestione dei comandi Rmi

La classe **HotelierClientRmiHandler**, contenuta nel package ***unipi.aprile.filippo.hotelier.client.cli.command.handler*** espone il metodo *handleRmiCommand* per la gestione dei comandi Rmi, il quale filtra i comandi in base al loro nome e invoca la relativa funzione di gestione.

All' interno di hotelier sono implementati i seguenti comandi Rmi:

- **register**, invoca metodo remoto per registrazione nuovo utente su server Rmi;

In caso di ***RemoteException*** segnala all' utente impossibilità di contattare server Rmi tramite stampa

4.3 Ricezione notifiche Udp

La classe **HotelierClientMulticastReciever**, contenuta nel package ***unipi.aprile.filippo.hotelier.client.multicast*** avvia un thread per la ricezione delle notifiche Udp riguardanti i cambiamenti degli hotel in prima posizione di un qualsiasi rank locale. Il thread esegue un ciclo continuo in cui attende le notifiche. Alla ricezione del *datagramPacket*, deserializza la risposta contentuta in esso e la stampa.

La **mutlicastSocket** viene creata nel costruttore il quale viene invocato nel metodo ***startClient*** del main.

HotelierClientMulticastReciever espone inoltre i metodi *joinGroup* e *leaveGroup* per permettere al client di unirsi/rimuoversi dal gruppo multicast.
.
La classe espone infine il metodo ***close*** per la chiusura della socket se ancora aperta. Viene invocato dalla classe **HotelierClientCLI** quando utente immette comando di *exit*.

4.4 Comunicazione Rmi

4.4.1 Interfaccia client Rmi

L' interfaccia **HotelierClientInterface**, contenuta nel package ***unipi.aprile.filippo.hotelier.common.network.rmi*** contiene i metodi esposti dal client che possono essere invocati remotamente da server.

Espone il metodo **notifyInterest** per la ricezione delle notifiche riguardanti cambiamenti rank locali di interesse.

4.4.2 Implementazione interfaccia Rmi

La classe **HotelierClientRmiImpl**, contenuta nel package ***unipi.aprile.filippo .hotelier.client.rmi*** implementa i metodi esposti dall' interfaccia *HotelierClientInterface* per invocazione remota.

Utilizza una mappa per i rank locali, avente:

- **chiave**, città di interesse;
- **valore**, relativa lista di hotel;

Il metodo **notifyInterest** è implementato come segue:

- deserializza il rank locale passato;
- aggiorna la entry della mappa avente chiave: città rank locale con la lista di hotel in esso contenuta.

HotelierServerRmiImpl espone inoltre il metodo *getLocalRankMap*, il quale restituisce la mappa dei rank locali.

L' accessso alla mappa dei rank locali viene eseguito in un *blocco sychronized* al fine di evitare *race-conditions*, in quanto Rmi può gestire invocazioni su thread diversi.

4.4.3 Invocazione metodi remoti

La classe **HotelierClientRmi**, contenuta nel package ***unipi.aprile.filippo .hotelier.client.rmi*** si occupa dell' implementazione dei metodi Rmi, fruitori dello stub del server, e dell' esportazione dello stub del client per permettere invocazione metodi remota al server (callback).

Il costruttore, invocato nel metodo ***startClient*** del main, esegue le seguenti operazioni:

- recupera stub del server da registro Rmi tramite *rmiRemoteReference* passata;
- crea ed esporta lo stub del client, utilizzato dal server per invocazione dei metodi remoti (callback);

Sono implementati i seguenti metodi remoti:

- **requestRegister**, effettua invocazione remota del metodo *registerUser* tramite stub del server. Vengono passati username e password utente da registrare;
- **registerInterests**, effettua invocazione remota del metodo *registerCallback* tramite stub del server. Vengono stub del client e lista città di interesse;

- **registerCallback**, effettua invocazione remota del metodo *unregisterCallback* tramite stub del server. Viene passato stub del client.

La classe espone inoltre i metodi *localRankMapToString* e *close*, dove:

- il primo restituisce la mappa dei rank locali formattata come stringa per la stampa;
- il secondo implementa la chiusura risorse associato alla comunicazione Tcp, ovvero:
  - controlla se utente aveva inserito delle città di interesse, nel caso deregistra la callback;
  - de-esporta stub del client.

Viene invocato dalla classe **HotelierClientCLI** quando utente immette comando di *exit*.

5 Server

Il main del server è implementato nella classe **HotelierServerMain**, contenuta nel package ***unipi.aprile.filippo.hotelier.server***, ed ha lo scopo di:

- **inizializzare il server**, andando a:
  - recuperare i config da disco se presenti oppure creando un file config con valori di default.
  - recuperare liste di hotel, utenti registrati e recensioni dal disco deserializzanodo i corrispettivi file json. Se il file degli hotel non è presente viene terminato il server notificandone l' assenza tramite stampa, mentre se non sono presenti i file relativi a utenti registrati e recensioni vengono creati e inizializzati con un lista vuota.
- **avviarlo**, instanziando le seguenti classi:
  - **HotelierServerNIO**, avvia thread per la gestione della comunicazione Tcp gestita tramite multiplexing di canali non bloccanti (NIO).
  - **HotelierServerRmi**, per l' invocazione remota delle callback riguardanti cambiamenti dei rank locali delle città interesse e ricezione delle richieste di registrazione remota effettuate dai client.
  - **HotelierServerMulticastSender**, per l' invio delle notifiche Udp riguardanti cambiamento dell' hotel in prima posizione dei rank locali a tutti gli utenti loggati.
  - **HotelierServerRanking**, avvia thread per il calcolo e update dei ranking degli hotel. Il tempo che deve intercorrere tra un calcolo e il successivo, in secondi, equivale al campo ***rankingInterval*** presente nel file config del server.

In caso di eccezzione nella fase di avvio viene terminato il server segnalando di controllare che non sia già stato avviato tramite stampa.

5.1 Config

I config del server sono rappresentanti dalla classe **HotelierServerConfig**, contenuta nel package ***unipi.aprile.filippo.hotelier.server.config***, la quale presenta i seguenti campi:

- **tcpPort**, porta della socket Tcp;
- **rmiPort**, porta del registro Rmi;
- **mcastPort**, porta della socket mutlicast dei client;
- **rankingInterval**‘, secondi da intercorrere tra calcoli successivi dei rank degli hotel;
- **serverAddress**, indirizzo socket Tcp e registro Rmi:
- **rmiRemoteReference**, nome per identificare stub server nel registro Rmi:
- **mcastAddress**, indirizzo socket multicast dei client:

Nel package è presente inoltre la classe **HotelierServerConfigManager**, la quale espone i metodi *createDefaultConfig* e *loadClientConfig*, aventi rispettivamente i compiti di:

- *createDefaultConfig*, crea file config con valori di default (sono gli stessi presenti nel file config fornito) e lo salva su disco;
- *loadClientConfig*, deserializza config da disco;

5.2 Registri

5.2.1 Registro hotels

La classe *singletone* **HotelierServerRegisterHotels**, contenuta nel package ***unipi.aprile .filippo.hotelier.server.register*** espone i seguenti metodi per la gestione della lista di hotel:

- **getHotelByID**, restituisce hotel avente id passato, null se non trovato;
- **getHotelsByCity**, restituisce lista di hotel aventi città passata;
- **getHotelByNameAndCity**, restituisce hotel avente nome e città passati, null se non trovato;
- **getCities**, restituisce la lista di tutte le città degli hotel senza duplicati;
- **getHotels**, restituisce la lista degli hotel;
- **serialize**, periste la lista degli hotel sul disco, nel file **hotels.json** presente nella folder ***HotelierServer***;

- **deserialize**, deserializza gli hotel da disco e li aggiunge alla lista di hotel;

Gli accessi e le modifiche alla lista di hotel vengono effettuati in *blocchi synchronized* sulla lista al fine di rendere la classe *thread-safe*.

5.2.2 Registro utenti

La classe *singletone* **HotelierServerRegisterUsers**, contenuta nel package ***unipi.aprile .filippo.hotelier.server.register*** espone i seguenti metodi per la gestione della lista degli utenti:

- **auth**, restituisce hotel avente username e password passati, null se non trovato;
- **register**, restituisce risposta/esito registrazione nuovo utente avente username e password passati, implementando i vari controlli (username/password vuoti, username/password contenenti, username già registrato). In caso di registrazione con successo aggiunge utente alla lista e la serializza su disco;
- **getUserByName**, restituisce utente avente username passata, null se non trovato;
- **serialize**, periste la lista degli utenti sul disco, nel file **users.json** presente nella folder ***HotelierServer***;
- **deserialize**, deserializza gli utenti da disco e li aggiunge alla lista di utenti;

Gli accessi e le modifiche alla lista di utenti vengono effettuati in *blocchi synchronized* sulla lista al fine di rendere la classe *thread-safe*.

5.2.3 Registro recensioni

La classe *singletone* **HotelierServerRegisterReviews**, contenuta nel package ***unipi.aprile .filippo.hotelier.server.register*** espone i seguenti metodi per la gestione della lista delle recensioni:

- **addReview**, aggiunge recensione passata alla lista delle recensioni;
- **getUserReviews**, restituisce la lista di recensione effettuata dall' utente passato;
- **getHotelReviews**, restituisce la lista di recensione riguardanti hotel passato;
- **serialize**, periste la lista delle recensioni sul disco, nel file **reviews.json** presente nella folder ***HotelierServer***;
- **deserialize**, deserializza le recensioni da disco e li aggiunge alla lista di utenti;

Gli accessi e le modifiche alla lista delle recensioni vengono effettuati in *blocchi synchronized* sulla lista al fine di rendere la classe *thread-safe*.

5.3 Comunicazione Tcp

La classe **HotelierServerNIO**, contenuta nel package ***unipi.aprile.filippo .hotelier.server.network*** avvia un thread per la gestione della comunicazioni TCP tramite multiplexing di canali non bloccanti (NIO). Il thread esegue un ciclo continuo in cui attende delle chiavi pronte registrate al selettore e le gestisce.

Allo start del thread, prima del ciclo, esegue le seguenti operazioni:

- crea un *socketAddress* avente indirizzo e porta passati;
- apre un *serverSocketChannel* e lo binda a *socketAddress* creato;
- apre un selettore per la gestione dei canali non bloccanti;
- registra la *serverSocketChannel* al selettore per operazione di accept;
- registra la serverSocketChannel al selettore per ope;

Il thread esegue un ciclo continuo in cui attende delle chiavi pronte registrate a selettore, a quel punto:

- itera le chiavi pronte. Per ogni chiave controlla per quale operazione è pronta ed esegue le rispettive operazioni:
  - ***accept***, accetta la connessione del client, setta il *socketChannel* dedicato a non bloccante e lo registra sul selettore per operazione di lettura e scrittura. Infine allega alla chiave del canale dedicato un nuovo **HotelierServerClientHandler**;
  - ***lettura***, esegue la lettura del messaggio di richiesta tramite il metodo *handleRead* del clientHandler ottenendo il pacchetto Tcp. Controlla che il pacchetto sia supportato e delega la sua gestione ad una cached Threadpool. Infine controlla se il client si è disconesso, nel caso: aggiorna lista utenti loggati, chiude la *socketChannel* dedicata e richiede la cancellazione della relativa chiave dal selettore.
  - ***scrittura***, esegue la scrittura del messaggio di risposta tramite il metodo *handleWrite* del clientHandler. Infine controlla se il client si è disconesso, nel caso chiude: aggiorna la lista di utenti loggati, chiude la *socketChannel* dedicata e richiede la cancellazione della relativa chiave dal selettore.
- una volta gestita, rimuove la chiave dal set delle chiavi pronte;

5.3.1 Gestione del client

La classe **HotelierServerClientHandler**, contenuta nel package ***unipi.aprile.filippo .hotelier.server.network*** gestisce la comunicazione Tcp tra il server ed il singolo client, esponendo i seguenti metodi:

- **handleRead**, effettua le lettura del messaggio di richiesta e restituisce il pacchetto di richiesta deserializzato;
- **handlePacket**, invoca il metodo *handlePacket* della classe **HotelierServer-PacketHandler** per la gestione del pacchetto di richiesta e inserisce il pacchetto di risposta ottenuto in una coda;
- **handleWrite**, recupera il pacchetto di risposta dalla coda, costruisce il relativo messaggio di risposta e ne effettua la scrittura;

Utilizza un booleano *isConnected* per gestire la stato del client, il quale viene settato a false se le lettura/scrittura sul canale generano delle eccezzioni.

I field di tipo *byteBuffer* vengono utilizzati per gestire correttamente letture/scritture parziali che possono essere effettuate su canali non bloccanti.

La coda dei pacchetti dei risposta è stata implementanta per possibili sviluppi futuri, in modo che il server sia compatibile anche in caso di client che effettua molteplici richieste e solo successivamente attenda relative risposte. Al fine di evitare *race-conditions* le operazioni di modifica della coda sono realizzate in *blocchi synchronized* su di essa.

La classe espone infine il metodo *close*, il quale: aggiorna la lista di utenti loggati e chiude il canale dedicato al client.

5.3.2 Gestione del login

La classe *singletone* **HotelierServerLoginHandler**, contenuta nel package ***unipi.aprile.filippo.hotelier.server.network*** espone i seguenti metodi per le gestione degli utenti loggati.

- **addUser**, aggiunge utente passato alla lista degli utenti loggati;
- **removeUser**, rimuove utente passato dalla lista degli utenti loggat;
- **isLoggedIn**, controlla se utente passato è loggato o meno;

I metodi sono tutti *synchronized* al fine di rendere la classe thread-safe.

5.3.3 Gestione dei pacchetti

La classe **HotelierServerPacketHandler**, contenuta nel package ***unipi.aprile.filippo .hotelier.server.network*** espone il metodo *handlePacket* per la gestione dei pacchetti di richiesta, implementato come segue:

- filtra il pacchetto in base alla sua istanza;
- per ogni pacchetto invoca relativo metodo di gestione;
- restituisce pacchetto di risposta ottentuto;

La gestione dei pacchett dei vari pacchetti di richiesta viene effettuata tramite una *cachedThreadpoll* al fine di aumentare il througput del server.

La classe implementa i seguenti metodi di gestione:

- **handleLoginPacket**, controlla che username e passoword passati siano validi, nel caso aggiunge utente alla lista degli utenti loggati restituendo pacchetto login di risposta;
- **handleLogoutPacket**, controlla che utente sia loggato, nel caso rimuove utente dalla lista degli utenti loggati restituendo pacchetto logout di risposta;
- **handleHotelPacket**, controlla che hotel richiesto esista, nel caso restituisce pacchetto hotel di risposta contenente hotel richiesto;
- **handleHotelListPacket**, controlla che siano presenti hotel per la città richiesta, nel caso restituisce pacchetto hotelList di risposta contenente lista hotel richiesti;
- **handleReviewPacket**, controlla che utente sia loggato e che hotel esista, nel caso:
  1. inserisce la recensione nella lista di recensioni del registro corrispondente;
  2. persiste la lista di recensioni del registro su disco;
  3. incrementa numero di recensioni effuate dall' utente di 1;
  4. controlla se utente ha raggiunto soglia per nuovo badge e nel caso lo aggiorna;
  5. persiste la lista di utenti del registro su disco;
  6. incrementa numero di recensioni effuate dall' utente di 1;
  7. aggiorna rate e ratings medi dell' hotel a seguito della nuova recensione;
  8. incrementa numero di recensioni riguardanti l' hotel di 1;
  9. persiste la lista di hotel del registro sul disco;
- **handleBadgePacket**, controlla che utente sia loggato, nel caso restituesce pacchetto badge di risposta contenente distintivo utente.

In caso di errore (login mancante, risorsa non trovata ....) viene restituito un pacchetto di errore contenente il problema riscontrato.

La classe espone infine il metodo *handleClientDisconnect*, il quale controlla se utente si era loggato e in caso lo rimuove dalla lista.

5.4 Calcolo ranking hotel

La classe **HotelierServerRanking**, contenuta nel package ***unipi.aprile.filippo.hotelier.server.ranking*** avvia un thread per il calcolo e aggiornamento dei rank degli hotel.

Allo start del thread, prima del ciclo, inizializza lista rank locali con copie delle liste hotel. Con copie si intende che ogni lista di hotel di ogni rank locale contiene le copie dei relativi hotel (un rank locale per ogni città degli hotel). Le copie sono necesssarie in quanto gli aggiornamenti dei rank delgi hotel del registro non si manifestano in esse e quindi possano essere usate per il confronto.

Il thread itera ogni *rankingInteval* secondi, ed esegue le seguenti operazioni:

1. itera tutti gli hotel presenti nel registro
2. controlla che hotel abbia delle recensioni
3. calcolo il nuovo *rank* di ogni hotel e li aggiorna;
4. aggiorna il *localRank* di tutti gli hotel. Il campo viene settato alla loro posizione rispetto agli altri hotel della stessa città ordinati decrescentemente per rank;
5. persiste la lista di hotel aggiornata tramite il metodo *serialize* esposto dal registro;
6. itera la lista di tutti i rank locali, e per ognuno:
    - controlla se è cambiato hotel in prima posizione, nel caso invia una notifica Udp a tutti i client loggati tramite il metodo *notifyFirstPosition* esposto dal sender multicast;
    - controlla se è avvenuto cambiamento del rank locale, nel caso notifica tutti i client che avevano registrato interesse per esso tramite callbackRmi. Per notificare il cambiamento usa il metodo *notifyLocalRank* esposto dal server Rmi. Inoltre rinnova la lista di hotel del rank locale con la rispettiva copia della lista di hotel aggiornati (anche in questo caso con copia della lista si intende nuova lista contenente le copie degli hotel);

Le copie degli hotel vengono create utilizzando il *copyConstructor* esposto dalla classe **HotelierHotel**.

5.4.1 Algoritmo di ranking

Il calcolo del nuovo rank viene effuato dal metodo *calculateRank*, che esegue le seguenti operazioni:

1. ottiene la lista di recensioni dell' hotel passato;

2. calcola la media dei minuti trascorsi dalla pubblicazione delle recensioni;

3. restiusce il nuovo rank calcolato applicando la seguente formula:

$$\log_{10}(1 + reviewCount) \cdot rate + e^{-\frac{avgMinutes}{60}}$$

L' algoritmo tiene traccia, in ordine di importanza, di: **qualità**, **numero recensioni** e **attualità**.
I parametri sono calcolati come segue:

- *qualità*, rate. Il rate è la media dei rate delle recensioni riguardanti l' hotel;
- *numero recensioni*, log10(1 +numero di recensioni). Viene calcolato tramite log10 in modo che non risulti predominante sul rank. Viene sommato 1 al numero di recensioni al fine che il fattore moltiplicativo risultante sia sempre positivo.
- *attualità*, decremento esponenziale della media dei trascorsi dalla pubblicazione/60. Il fattore risultante è valore compreso tra 1 e 0 che diventa praticamente nullo se media minuti trascori maggiori di 60 minuti. Ha effetto ha parità di rate e numero di recensioni.

I parametri sono settati in modo da poterne testare facilmente il funzionamento da interfaccia a linea di comando.

In un caso reale si potrebbe calcolare la media dei giorni intercorsi dalla publicazione delle recensioni e portare il *tau* del decremento esponsenziale da 60 a 365.

Al fine di migliorare l' algoritmo è possibile introdurrre dei pesi per i vari fattori in modo da equilibrare l' apporto di ognuno di essi sul rank.

5.5 Invio notifiche Udp

La classe **HotelierServerMulticastSender**, contenuta nel package ***unipi.aprile.filippo.hotelier.server.network.multicast*** espone il metod *notifyFirstPosition* per notificare nuovo hotel in prima posizione di un rank locale. Il metodo è implementanto come segue:

- costruisce la stringa contenente città e nome hotel passato;
- serializza la stringa in bytes;
- invia il *datagramPacket* contenente stringa serializzata alla socket multicast su cui si sono aggounti tutti gli utenti loggati;

La **mutlicastSocket** viene creata nel costruttore il quale viene invocato nel metodo ***startClient*** del main.

La classe espone infine il metodo ***close*** per la chiusura della socket se ancora aperta.

5.6 Comunicazione Rmi

5.6.1 Interfaccia server Rmi

L' interfaccia **HotelierServerInterface**, contenuta nel package ***unipi.aprile.filippo .hotelier.common.network.rmi*** contiene i metodi esposti dal server che possono essere invocati remotamente da client.

Vengono esposti i seguenti metodi:

- **registerUser**, richiesta di registrazione di utente avente username e password passati. Restituisce la stringa contenente risposta/esito della richiesta;
- **registerCallback**, richiesta di registrazione delle callback per le città di interesse passate. Viene inoltre passato lo stub del client per permettere al server di invocare remotamente le callback su di esso;
- **unregisterCallback**, richiesta di deregistrazione delle callback per lo stub client passato;

5.6.2 Implementazione interfaccia Rmi

La classe **HotelierServerRmiImpl**, contenuta nel package ***unipi.aprile.filippo .hotelier.server.network.rmi*** implementa i metodi esposti dall' interfaccia *HotelierServerInterface* per invocazione remota.

Utilizza una mappa per le callabacks, avente:

- **chiave**, stub del client;
- **valore**, relativa lista delle città di interesse;

I metodi sono implementati come segue:

- **registerUser**, invoca il metodo *register*, esposto dal registro degli utenti, il quale gestitsce la registrazione dell' utente avente username e password passati. Restituisce risposta/esito della richiesta;
- **registerCallback**, aggiunge entry nella mappa delle callback avente chiave: stub client e valore: lista città di interessa passati;
- **unregisterCallback**, rimuove entry avente chiave: stub client passato;

HotelierServerRmiImpl espone inoltre il metodo *getClientsCallback*, il quale restituisce la mappa delle callbacks.

I metodi che accedono/modificano la mappa delle callbacks sono tutti *sychronized* al fine di evitare *race-conditions*, in quanto Rmi può gestire invocazioni su thread diversi.

5.6.3 Invocazione metodi remoti

La classe **HotelierServerRmi**, contenuta nel package ***unipi.aprile.filippo.hotelier.server.network.rmi*** si occupa dell' implementazione di metodi Rmi, fruitori dello stub del client, e dell' esportazione dello stub del server per permettere invocazione metodi remota al client.

Il costruttore, invocato nel metodo ***startSever*** del main, esegue le seguenti operazioni:

- crea ed esporta lo stub, utilizzato dal client per invocazione dei metodi remoti;
- crea il registro Rmi sulla porta passata;
- effettua il binding dello stub a rmiRemoteReference passata;

La classe espone il metodo *notifyLocalRank* per invocazione remota dei metodi sul client al fine di notificare cambiamenti dei rank locali. Il metodo è implementato come segue:

- ottiene la mappa delle callbacks, presente nella classe *HotelierServerRmiImpl*;
- crea una lista per le callbacks da rimuovere;
- itera la mappa delle callbacks. Per ogni entry:
  - controlla se il client aveva registrato interesse per il rank locale, nel caso esegue callback relativa per notificarne il cambiamento.
    In caso di ***RemoteException*** aggiune il client alla lista delle callback da rimuovere;
- itera la lista delle callbacks e rimuove della mappa le entry corrispondenti.

L' accesso alla mappa della callbacks viene fatto utilizzando un *blocco synchronized* su di essa al fine di evitare race condition con i metodi implementati nella classe ***HotelierServerRmiImpl***.

6 Guida all' uso

L' utente interagisce con il client mediante interfaccia a linea di comando. Sono disponibili i seguenti comandi:

- **register**, effettua richiesta di registrazione di un nuovo utente. Richiede i seguenti parametri:
  - *usermane*, nome utente;
  - *password*, password utente.

- **login**, effettua richiesta di login. Richiede i seguenti parametri:
  - *usermane*, nome utente;
  - *password*, password utente.
- **logout**, effettua richiesta di logout;
- **searchHotel**, effettua richiesta di un hotel. Richiede i seguenti parametri:
  - *nome*, nome hotel;
  - *città*, città hotel.
- **searchAllHotels**, effettua richiesta di una lista di hotel ordinati per rank locale. Richiede i seguenti parametri:
  - *città*, città degli hotel;
- **insertReview**, effettua richiesta di inserzione di una recensione. Richiede i seguenti parametri:
  - *nome*, nome hotel da recensire;
  - *città*, città hotel da recensire;
  - *rate*, rate hotel da recensire (int compreso tra 0 e 5);
  - *cleaning*, punteggio pulizia hotel da recensire (int compreso tra 0 e 5);
  - *position*, punteggio posizione hotel da recensire (int compreso tra 0 e 5);
  - *services*, punteggio servizi hotel da recensire (int compreso tra 0 e 5);
  - *quality*, punteggio qualità hotel da recensire (int compreso tra 0 e 5);

  ;
- **showMyBadges**, effettua richiesta del badge dell' utente;
- **showLocalRanks**, stampa gli hotel delle città di interesse ordinati per rank locale;
- **help**, stampa lista dei comandi disponibili;
- **exit**, termina il client;

Per utilizzare un comando deve esserne inserito il nome, seguito da uno spazio ed infine la lista di argomenti. Gli argomenti devono essere contenuti tra virgolette ("argomento1" "argomento2" ...). Esempio:

- *login "username" "password"*, **corretto**;
- *login username password*, **errato**.

Lo stesso vale per le città di interesse, anch' esse devono esere passate tra virgolette.

Consentito l' inserimento di nessuna città di interesse, nel caso non viene registrata la callback Rmi per l' utente e viene notificato un warning a stampa.

I nomi e gli argomenti dei comandi sono ***case insensitive***