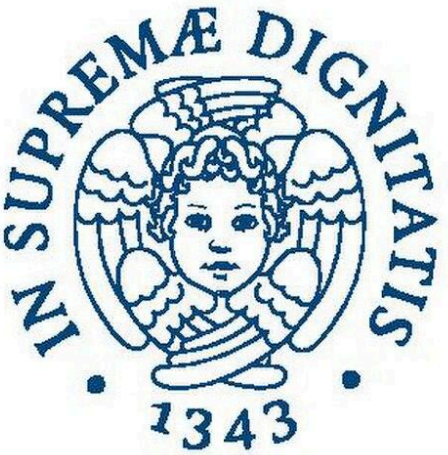
Progetto finale Laboratorio III

Alessio Pardini

637814



UNIVERSITÀ DI PISA

1. Introduzione

Il progetto denominato Hotelier637814 è suddiviso in due package, denominati "Client" e "Server". Ognuno di questi package rappresenta un modulo distinto del progetto e contiene il codice sorgente associato e i file necessari per l'esecuzione corretta delle funzionalità del sistema.

Il package "Client" contiene:

- lib: Una cartella che contiene:
  - gson-2.10.jar: Libreria esterna Gson, utilizzata per interpretare alcune risposte in formato JSON fornite dal server.(ad esempio nel caso login, approfondiremo di seguito).
- src: Una cartella che contiene file:
  - ClientHotelierMain.java: File contente il codice sorgente del client, responsabile principalmente delle operazioni di interazione e comunicazione con l'utente.
  - client.properties.txt: File di configurazione contente i parametri necessari per la corretta esecuzione del client:
    - Hostname su cui risiede il server;
    - Porta per contattare il server con connessione UDP;
    - Indirizzo Multiast per contattare il server con connessione UDP;
    - Porta per contattare il server con connessione TCP;
    - Dimensione del buffer utilizzato lato client per invio/ricezione messaggi su connessione TCP;
    - Dimensione del buffer utilizzato lato client per ricezione datagrammi UDP.
- Client.jar: Il .jar relativo al Client.
- 2 cartelle del folder java project create automaticamente da vs code, che mantengo nella consegna nel caso volessimo fare dei test con il mio pc tramite vs code: bin, .vscode.

Il package "Server" contiene:

- lib: Una cartella che contiene:
  - gson-2.10.jar: Libreria esterna Gson, utile per la gestione semplificata della comunicazione tramite formato JSON e per la persistenza dei dati in file JSON.
- src: Una cartella che contiene file:
  - ServerHotelierMain.java: File contente il codice sorgente relativo al server.
  - server.properties.txt: File di configurazione contente i parametri necessari per la corretta esecuzione del client
    - Porta su cui il server si pone in ascolto di connessioni TCP;
    - Porta per il multicast UDP;
    - Dimensione del buffer di lettura per le connessioni TCP;
    - Messaggio di default che ci viene inviato dal client per terminare la connessione
    - Intervallo di tempo tra gli aggiornamenti dei file JSON.

        - Frequenza di aggiornamento del ranking locale degli hotel.
        - Soglia di tempo dopo la quale si può effettuare una nuove recensione.
        - nome del file JSON per gli hotel.
        - Nome del file JSON per gli utenti.
        - Nome del file JSON per le recensioni.
        - Tempo massimo di attesa per la chiusura del ThreadPool
- Server.jar: Il .jar relativo al Server.
- Hotels.json: File JSON che contiene l'elenco di tutti gli hotel registrati nel sistema.
- Reviews.json: File JSON che contiene l'elenco di recensioni fino a quel momento persistite.
- Users.json: File JSON che contiene l'elenco degli utenti registrati fino a quel momento persistiti nel sistema
- 2 cartelle del folder java project create automaticamente da vs code, che mantengo nella consegna nel caso volessimo fare dei test con il mio pc tramite vs code: bin, .vscode.

2. Scelte Effettuate

In questa sezione mi soffermerò sulle scelte di maggiore importanza effettuate durante la progettazione di questo sitema.

Partiamo da un punto fondamentale richiesto per la redazione di questa relazione:

Spiegazione algoritmo per il calcolo del punteggio degli hotel:

Idea dell'algoritmo:

L'algoritmo di ranking che ho deciso di sviluppare per il calcolo del punteggio degli hotel si basa come richiesto su tre fattori:

- **Qualità**: Il testo a noi fornito definisce il fattore qualità un qualcosa riguardante i punteggi sintetici attribuiti da ogni utente a quella singola struttura.
  Nell'algoritmo fornito la Qualità viene modellata come la media dei punteggi sintetici associati a quell'hotel.
  Dove i punteggi sintetici associati all'hotel rappresentano la media dei punteggi sintetici associati da ogni utente per quell'hotel, tramite le recensioni.
- **Quantità**: Il punteggio di quantità è semplicemente il numero totale di recensioni ricevute dall'hotel.
- **Attualità**: il punteggio di attualità è rappresentato dalla media del punteggio di attualità di ogni recensione per quell'hotel, dove quest'ultimo è inversamente proporzionale ai giorni trascorsi dalla sua pubblicazione fino al momento attuale.
  Dunque in altre parole più la recensione è recente maggiore è il suo punteggio di attualità.
  Osservazione punteggio attualità:

Tale parametro è aggiornato all'ultima recensione effettuata per gli hotel di quella città, dunque l'ultima recensione per gli hotel di quella città avrà il massimo valore di attualità (= 1) e non un valore aggiornato a quel momento in cui si richiede l'aggiornamento del ranking(<=1), questo perchè non avendo altre recensioni si dovrebbero traslare solamente i valori che comunque rimarrebbero ordinati allo stesso modo e questo causerebbe soltanto un costo computazionale maggiore senza ricevere alcun beneficio tangibile.

L'idea dunque dell'algoritmo è stata quella di combinare questi 3 parametri in modo tale che quest'ultimi apportassero al punteggio finale dell'hotel uno "stesso impatto".

Questi 3 fattori vengono considerati per ciascun hotel di ogni città.
Ad ogni città a livello logico associamo 3 intervalli numerici, *relativi ai valori dei 3 fattori di ciascun hotel per quella città*, di cui ci interessa il massimo attuale, si ha dunque 1 intervallo per ciascun fattore.

Ogni intervallo ha un minimo pari a 0 ed un MAX non sempre prevedibile che dipende dalle recensioni inserite dagli utenti.
ad esempio l'intervallo quantità ha un massimo che dipende dal numero di recensioni effettuate e cresce man mano che il sistema è in uso, in casistiche reali si può arrivare anche a più di 10.000 recensioni per una certa struttura, l'intervallo qualità rimane comunque limitato tra 0 e 5, mentre l'intervallo attualità è limitato tra 0 a 1.

Quello che viene fatto è quindi una normalizzazione MIN/MAX dei valori presenti in questi intervalli in modo tale che ricadino tutti in un intervallo [0,10]

I punteggi di quantità, qualità, attualità vengono normalizzati tenendo conto del massimo attuale per quella città. Questo permette di confrontare gli hotel basandosi anche sulle specifiche condizioni locali per quella città; inoltre la normalizzazione basata sul massimo attuale consente una valutazione che non dipende da quelli che sono *i massimi attuali possibili*, che sono conosciuti almeno per quanto riguarda i punteggi di qualità(5) e di attualità(1), dunque l'algoritmo è robusto a questa eventuale modifica.

Una volta normalizzati i valori per ciascun hotel questi vengono sommati, il valore che si ottiene detto ***rankingScore*** $\in$ **[0,30]** è il fattore determinante per il posizionamento all'interno del ranking.
Gli hotel con un ***rankingScore*** più elevato saranno posizionati a livelli più alti. Nel caso in cui due hotel abbiano lo stesso ***rankingScore*** , l'ordinamento avviene mediante un confronto lessicografico basato sul nome dell'hotel per determinarne l'ordine.

Andiamo ora nel dettaglio dell'algoritmo:

Ogni volta che viene inserita una recensione per un certo hotel, vengono aggiornati i valori relativi ai punteggi di qualità e di quantità per quel determinato hotel, inoltre si vanno a ricalcolare i punteggi di attualità di ogni singolo hotel di quella città, questo viene fatto perchè se così non fosse, il risultato che avremo è che per diversi hotel si considera un punteggio di attualità non aggiornato al momento attuale dell'inserimeto di tale recensione, ma relativo ad un **currentTime** differente, ciò farebbe sì che l'ultima recensione di ogni hotel avrebbe il

massimo punteggio di attualità, il che sarebbe corretto se il confronto sarebbe tra recensioni dello stesso hotel, ma in tal caso il confronto deve avvenire tra gli hotel, cioè tra le medie dei punteggi di attualità dei vari hotel.
Durante il calcolo del punteggio di attualità per ogni hotel calcolo il massimo valore ottenuto, questo mi serve per normalizzare inseguito i punteggi di attaulità sul massimo attuale.

Una volta calcolati questi punteggi, quest'ultimi vengono memorizzati negli hotel tramite opportuni metodi:

>.setOriginalQualityScore(qualityScore);
>.setOriginalQuantityScore(quantityScore);
>.setOriginalRelevanceScore(relevanceScore);

Si utilizza la notazione *Original* in quanto valori non ancora normalizzati.
si tiene conto di tali valori poichè in caso di nuovo massimo di uno qualsiasi dei punteggi questi devono essere rinormalizzati.

Terminata questa fase si ottiene l'oggetto CityStats dalla mappa cityStastMap, relativa alla città dell'hotel per il quale sta avvenendo l'inserimento della recensione.
L'oggetto CityStats mantiene memorizzati le informazioni dei massimi attuali per i punteggi di quella città:

>maxQuantityScore
>maxQualityScore
>maxRelevanceScore

Una volta ottenuto tale oggetto, viene settato il nuovo maxRelevanceScore e viene chiamato il metodo:

> *cityStats.**normalizeAndUpdateMaxScores(hotelsInCity, hotel)**;*

Al quale viene passato la lista degli hotel per quella città, questo mi serve per normalizzare sicuramente i punteggi di attualità mentre gli altri 2 punteggi li rinormalizziamo solo in caso di nuovo massimo, inoltre viene passato al metodo anche l'hotel per il quale chiaramente si normalizzano tutti i nuovi punteggi.

Per normalizzare i valori utilizzo il metodo:

>***normalizeValue(value, maxValue)***:

il quale prende in input il valore da normalizzare e il massimo attuale. In caso di value == 0 restituisce 0, sennò restituisce 10*(valure/maxValure).

Una volta normalizzati modifico il ***rankingScore*** tramite il metodo

>setRankingScore(nuovoRankingScore).

Infine come richiesto da specifica:
"Il server aggiorna periodicamente il ranking degli alberghi di ogni città. Il periodo di tempo che intercorre tra un aggiornamento e il successivo è un parametro di input della applicazione "
Periodicamente viene eseguito il thread "HotelRankingUpdater" il quale ottiene per ogni città la lista di hotel associata ed esegue il sort di tale lista che si baserà sui valori rankingScore, in caso di nuovo primo invierà una notifica agli utenti loggati.(piccolo approfondimento sarà trattato nella sezione ServerHotelierMain punto 5).

Ragioni dietro l'Implementazione del Server con Java NIO e Multiplexing dei Canali:

Il testo a noi fornito cita:“il server può essere realizzato con JAVA I/O e threadpool oppure può effettuare il multiplexing dei canali mediante NIO (eventualmente con threadpool per la gestione delle richieste).”

Il server proposto è realizzato mediante il multiplexing dei canali tramite NIO, questa scelta è stata dettata in primo luogo da una curiosità puramente didattica, in secondo luogo si hanno dei chiari vantaggi nell’utilizzo di questa soluzione che abbiamo discussi attentamente a lezione.

Con Java NIO,è possibile avere operazioni di I/O non bloccanti che permettono ai thread di gestire altre attività mentre attendono il completamento delle operazioni di I/O, evitando il blocco del thread e migliorando l'efficienza generale del server rispetto ad una implementazione con java IO bloccante.

Tramite JAVA NIO abbiamo il multiplexing dei canali, ciò consente di gestire più operazioni di I/O su un singolo thread, potenzialmente aumentando il numero di connessioni simultanee che possono essere gestite dal singolo thread. Questo riduce l'uso eccessivo di thread separati per ogni richiesta I/O, ottimizzando l'uso delle risorse del sistema.

Il multiplexing consente a un singolo thread di monitorare diversi canali di I/O per eventi di lettura/scrittura pronti, rendendo più efficiente la gestione delle richieste senza dover allocare un thread separato per ciascuna operazione.

In generale in situazioni di carico elevato, Java NIO con il multiplexing dei canali tende a offrire prestazioni superiori rispetto all'approccio basato su threadpool. Questo grazie alla minore latenza e alla migliore gestione delle risorse di sistema.

3. ServerHotelierMain

il file ServerHotelierMain.java contiene tutto il codice sorgento relativo al server, incluse diverse classi annidate di cui parleremo di seguito.

Il codice sorgente è strutturato nel seguente modo:

1. **Attributi della classe ServerHotelierMain:**
   tra cui troviamo
   - >Il percorso del file di configurazione
   - >I parametri di configurazione del server
   - >I parametri relativi alla configurazione di rete
   - >Il ServerSocketChannel e il Selettore
   - >I Nomi relativi ai file json
   - >Le strutture dati usate lato server:

Come richiesto per questo punto particolare, effettuerò una disamina più curata.
Tra le strutture dati manetenute dal server troviamo:
->ConcurrentHashMap<String, User> registeredUsers:

Questa struttura dati mantiene le coppie chiave,valore username, utente. Qui si memorizzano *i dati relativi a tutti gli utenti registrati nel sistema.*
La decisione di utlizzare questa struttura dati è stata presa considerando i vantaggi offerti per le operazioni di ricerca.
Queste operazioni di ricerca vengono eseguite più volte all'interno del codice, ad esempio per il login o per la registrazione, allo scopo di verificare se l'utente è già registrato (cioè presente nella struttura dati) o già loggato (attributo LoggedIn = true dell'oggetto user trovato tramite la ricerca per username).
Inoltre non si è scelto di utilizzare una struttura dati Map qualsiasi che per definizione da questo ventaggio relativo alle operazioni di ricerca, bensì si è scelto di usare una ConcurrentHashMap in modo da gestire efficientemente la sincronizzazione, difatti questa si basa su una fine-grain locking; si ha un array di segmenti in cui ogni segmento punta ad una HashMap e si ha un lock per ogni segmento, dunque modficihe simultanee sono possibili se modificano diversi segmenti.
le altre due soluzione possibili potevano essere l'HashTable e Synchronized Collections che usano entrambe un unica lock per gestire l'intera collezione.

->ConcurrentHashMap<String, ConcurrentLinkedQueue <Hotel>> hotelsByCity:

Questa struttura dati mantiene le coppie chiave,valore città, coda di hotel per quella città. Qui si memorizzano *i dati relativi a tutti gli hotel registrati nel sistema.*
La decisione di utlizzare questa struttura dati è stata presa considerando i vantaggi offerti per le operazioni di ricerca relativi all'ottenimento di tutti gli hotel per quella città utile sicuramente per quanto riguarda l'operazione *SearchAllHotels(city)* ma chiaramente non solo.
Avere tale struttura dati mi permette di mantenere ordinati gli hotel in base al rankingScore questione che non potevo effettuare allo stesso prezzo computazionale nel caso di ConcurrentHashMap<String,ConcurrentHashMap<String,Hotel>> , in tal caso dovevo comunque avere una qualsiasi altra gestione, si poteva mantenere ad esempio il primo oggetto per città memorizzato a parte o comunque gli hotel ordinati a parte con un altra struttura.
Le motivazione relative al perché di una ConcurrentHashMap e non di una qualsiasi altra Map threadSafe vengono già

delineate nel paragrafo superiore.
Per quanto riguarda la scelta di una ConcurrentLinkedQueue al posto di una qualsiasi struttura dati List thread safe o Queue thread safe, sta nel fatto che quest'ultima è ottimizzata per una sincronizzazione efficiente ed è stata preferita alla struttura dati CopyOnWriteArrayList, poichè è vero che nel nostro sistema la registrazione degli hotel da parte degli utenti non è permessa dunque questa tipologia di scritture sulla struttura dati non vengono effettuate, ma c'è il bisogno di riordinare questa struttura dati ogni qual volta viene chiamato eseguito il thread per l'aggiornamento del ranking locale.

->ConcurrentHashMap<String, ConcurrentLinkedQueue<Review>> reviews:
Questa struttura dati mantiene le coppie chiave,valore nomeHotel, coda di recensioni. Qui si memorizzano *i dati relativi a tutti le recensioni registrate nel sistema.*
L'unica discussione che rimane interessante da affrontare è relativa alle motivazioni riguardanti del perchè di una ConcurrentLinkedQueue, questa è stata scelta in quanto ottimizzata per la sincronizzazione, ed in tal caso si effettuano sia letture che scritture con una frequenza considerevole.
Una ulteriore mappa non avrebbe senso perchè ci serve sapere tutte le recensioni associate ad un certo hotel per il calcolo del rankingScore.

->ConcurrentHashMap<String, CityStats> cityStatsMap:
Questa struttura dati maniene le coppie chiave, valore città,CityStats.
Abbiamo già discusso dell'importanza di mantenere questa struttura dati nel paragrafo "2.Scelte effettuate".

>Il pool di thread
>L'oggeto di blocco per sincronizzare l'output su console.

2. Main:

Il metodo inizia con la lettura della configurazione del server da un file serverproperties.txt grazie al metodo *readConfig()*, abbiamo già discusso nella sezione introduzione i parametri importanti per il corretto funzionamento del server presenti in questo file.
Una volta fatto c'ho si ha l'aggiunta di uno ShutdownHook al Runtime dell'applicazione. Un ShutdownHook è un thread che viene eseguito quando la JVM sta terminando, tipicamente in risposta a un arresto del sistema, una chiamata del metodo System.exit o in risposta ad un segnale di interruzione come SIGINT.
il ServerTerminationHandler classe che estende Thread di cui un istanza viene passata allo shutdownhook, prende come parametro il riferimento al thred main, questo per far si di chiamare su quest'ultimo interrupt().
Ciò è necessario perchè se così non fosse a seconda dell'interleaving delle istruzioni se arrivasse un segnale SIGINT prima che il thread main sia arrivato a utilizzare la select() sull'oggetto selector avremmo lanciata un eccezione in quanto si effettua una select() su un oggetto che è già stato chiuso da serverTerinationHandler.

Inseguito vengono chiamati 3 metodi, che ci permettono di recuperare le informazioni che si trovano sui file JSON, in modo da popolare le strutture dati del server

>*loadHotelsFromJSON()*

>*loadUsersFromJSON()*

>*loadReviesFromJSON()*

Una volta letti i file JSON, si crea un pool di thread per l'esecuzione periodica di 2 attività, la persistenza dei dati su file JSON e l'aggiornamento del ranking locale degli hotel. Questa operazione consente al server di eseguire queste 2 attività in background senza bloccare la gestione delle richieste dei client.

Infine, il metodo apre un ServerSocketChannel per gestire le connessioni dei client e un selettore per gestire l'I/O multiplexato dei canali. Il ServerSocketChannel consente al server di accettare richieste di connessione da parte dei client, mentre il selettore consente al server di monitorare più canali contemporaneamente e di gestire le richieste dei client in modo efficiente.

Il server entra quindi in un ciclo infinito nel quale:

- Attende (e accetta) richieste di connessione da parte dei client.
- Controlla (tramite il selettore) se ci sono canali pronti per essere letti o scritti.

Se il selettore rileva una richiesta di connessione, il server accetta la connessione e registra il canale del client sul selettore, grazie al metodo handleAccept()

Se il selettore rileva che un canale è pronto per essere letto, il server legge i dati dal canale e li elabora, grazie al metodo handleRead().

Se il selettore rileva che un canale è pronto per essere scritto, il server scrive i dati sul canale. grazie al metodo handleWrite().

Il ciclo infinito termina quando il server viene interrotto.

3. Handle:

Qui troviamo sostanzialmente tutti i metodi di “tipo” handle

>*handleAccept()*: gestisce la connessione in arrivo da un client accettando la connessione tramite il metodo *accept()* chiamato sul serverSocketChannel, configurando il SocketChannel restituito dalla accept() per operazioni non bloccanti e registrandolo nel selettore per leggere dati dal client. Associando come oggetto att un oggetto State per gestire una corretta lettura e la memorizzazione dell’ID del client associato alla connessione.

>*handleWrite(Selection key)*: Il metodo handleWrite gestisce l'operazione di scrittura su un canale di tipo SocketChannel associato a una chiave di selezione (SelectionKey).

Sostanzialmente dopo aver recuperato dalla chiave di selezione il SocketChannel e l'oggetto State associati, si utilizza il metodo channel.write(state.buffer) per scrivere i dati contenuti nel buffer dell’oggetto State sull'oggetto SocketChannel.

Si Verifica dunque se dopo la scrittura nel canale ci sono ancora dati non completamente consumati nel buffer,

questo potrebbe capitare perchè la write non da nessuna garanzia.
Se ci sono dati rimanenti nel buffer dopo la scrittura, si compatta il buffer (sposta i dati non scritti all'inizio del buffer) e si termina la funzione per ritentare la scrittura più tardi.
Se la scrittura è stata completata e non ci sono più dati da inviare, si stampa un messaggio di conferma dell'invio al client e si resetta il buffer per prepararsi alla prossima operazione di lettura con conseguente switch di interesse per la chiave a operazioni di lettura, in attesa dunque di ulteriori richieste da soddisfare.

>*handleRead(Selection Key)*: Il metodo handleRead gestisce l'operazione di lettura su un canale di tipo SocketChannel associato a una chiave di selezione (SelectionKey), in seguito elabora le informazioni lette e scrive la risposta elaborata nel buffer dell'oggetto State.
Sostanzialmente dopo aver recuperato dalla chiave di selezione il SocketChannel e l'oggetto State associati, si utilizza il metodo channel.read(state.buffer) per leggere i dati contenuti sul canale e scriverli nel buffer dell'oggetto State. L'oggetto State in tale metodo è di fondamentale importanza in quanto ci permette di tenere conto dei byte letti grazie a state.count e della lunghezza del messaggio ricevuto grazie a state.length, dunque è possibile gestire eventuali chiusure forzate del client, verificare se si è letto tutto il messaggio o meno, in quel caso il metodo termina e si effettuerà la lettura successivamente, quest'ultima questione è importante in quanto la lettura da canale non è bloccante, il che significa che potrebbe non essere disponibile l'intero messaggio quando il server tenta di leggere.
In caso di lettura avvenuta con successo si estrae il messaggio dal buffer e si gestisce l'eventuale richiesta chiamando il metodo *buildReplyBuffer(String request, SocketChannel socketChennel, int idClient)* che si occuperà di chiamare il metodo handleRequest di cui di seguito parleremo, il quale restituirà la stringa di risposta, e il metodo buildReplyBuffer si occuperà di porla all'interno del buffer.
Di seguito si rinizializzano i valori di State relativi a count e length e si swithca l'interesse per la chiave alla scrittura.
Nel caso di richiesta di disconnessione questa viene gestita contestualmente, sostanzialmente si disconnette l'utente associato a quel client, chiaramente solo se se client era loggato, in tal caso si resetta anche il suo

idClient a -1; dopo aver fatto ciò si chiude semplicemente il socketChannel.

>*handleRequest(String request, SocketChannel clientSocket, int idClient):*

Tale metodo riceve la stringa di richiesta effettuata dal client, estrae i parametri della richiesta usando il delimitatore "," e successivamente chiama opportuni metodi in base al tipo di richiesta.
Se le richiesta non corrisponde a nessuna delle opzioni presenti viene generato un oggetto 'JsonObject' con un messaggio di richiesta non valida e la risposta viene costruita grazie ad una funzione toJson di cui parleremo.
Questa casistica in realtà per come è gestito il codice lato client non si verifica mai, ho deciso comunque di averla nel codice lato server per renderlo robusto ad eventuali modifiche lato client.

4. Gestione richieste:

In questa sezione abbiamo tutti i metodi per la gestione delle richieste del client.

>*register(String username, String password)*

booelan isValidPassword(String password)

String hashPassword(String password)

>*login(String username, String password, int idClient)*

>*logout(String username)*

>*searchHotel(String nomeHotel, String citta)*

>*inserReview(String username, String nomeHotel, String citta, String globalScore, List<String> singleScores)*

>*void updateHotelReviewsScores(Review review)*

>*void updateHotelScoresAndRanking(Hotel hotel)*

>*showMyBadge(String username)*

Tutti questi metodi restituiscono una Stringa, che viene costruita grazie al metodo toJson, in modo tale da fornire un formato unico di risposta

>*toJson(ServerResponse serverResponse, JsonObject data)*:

Il metodo toJson ha lo scopo di convertire le informazioni provenienti da un oggetto ServerResponse e un oggetto JsonObject in una stringa JSON formattata. Questo processo viene esguito utilizzando la libreria Gson, con la creazione di un'istanza configurata per una formattazione più leggibile.
Chiariremo di seguito l'oggetto ServerResponse.
Osserviamo che il metodo prende in input un JsonObject e non una stringa formattata JSON questo perchè nel caso di associamento di questa stringa ad una proprietà dell'oggetto JSON che viene costruito nel metodo si avrebbe una formattazione errata, la libreria di serializzazione potrebbe interpretare i caratteri speciali nella stringa come parte della sintassi JSON, invece di trattarli come dati effettivi, in tal caso la stringa risultante

dalla conversione JsonObject->String non sarebbe facilmente leggibile.

Un altra porzione di codice importante di questa sezione è sicuramente:

>*enum ServerResponse:*

L'enumerazione 'ServerResponse' rappresenta un insieme di risposte possibili che il server può restituire in risposta alle diverse operazioni richieste dai client.
Ogni costante dell'enumerazione rappresenta una possibile risposta ed è definita con un codice numerico e una frase associata.

5. Classi:

In questa sezione abbiamo tutti le sottoclassi annidate della classe ServerHotelierMain.

>*CityStats*:

Di questa classe abbiamo già parlato ampiamente, non mi dilungherò ulteriormente.

>*User:*

La classe 'User' rappresenta un utente nel sistema, con attributi come username, password, stato di accesso, numero di recensioni scritte, livello di esperienza e un identificatore client.
Il livello di esperienza è gestito attraverso un'enumerazione interna 'Level', che include distintivi e requisiti di recensioni per ciascun livello. La classe fornisce metodi per accedere e modificare le informazioni dell'utente, nonchè per aggiornare automaticamente il livello in base al numero di recensioni scritte.

>*Hotel:*

La classe 'Hotel' rappresenta un hotel nel sistema e implementa l'interfaccia 'Comparable' per definire un ordinamento tra gli oggetti Hotel basato come già discusso sull'attributo rankingScore in ordine decrescente.
Gli attributi includono idenitificativo, nome, descrizione, città, numero di telefono, lista di servizi, rate, ratings, punteggio di ranking, numero di recensioni, e un oggetto di normalizzazione.
La classe '*Normalization*' è una classe interna privata a 'Hotel' utilizzata per gestire i valori originali e normalizzati relativi alla qualità, quantità e rilevanza dell'hotel.

>*Review:*

La classe 'Review' rappresenta una recensione di un utente su un hotel all'interno del sistema. Gli attributi della classe includono l'ID dell'utente, il timestamp della recensione, il nome dell'hotel, la città, il punteggio globale e le valutazioni specifiche tramite un oggetto 'Ratings'.

>*State:*

La classe ‘State’ rappresenta uno stato associato a una connessione in questo contesto di comunicazione asincrona. Gli attributi della classe includono un contatore di byte letti, la dimensione del messaggio da ricevere, un buffer, la sua dimensione, e un identificativo(‘id’).

>*ServerTerminationHandler:*

La classe ‘ServerTerminationHandler’ estende la classe ‘Thread’ e rappresenta un gestore di terminazione per il server.
Prende in input il riferimento al thread main, abbiamo già discusso delle motivazioni.
All’interno del metodo run(), viene chiuso il pool di thread scheduler, viene invocato il metodo ‘saveDataToJson’ per persistere i dati in formato JSON, viene chiuso il ‘ServerSocketChannel’ e il ‘Selector’.
All’interno del metodo saveDataToJson vengono sloggati tutti gli utenti connessi e dissociati dall’idClient, grazie al metodo setAllLoggedOutUsers() e persistiti i dati su JSON.

>*HotelRankingUpdater:*

La classe HotelRankingUpdater implementa l'interfaccia Runnable e si occupa dell'aggiornamento del ranking degli hotel.
All'interno del metodo run(), avviene l'iterazione attraverso tutte le città presenti nella mappa hotelsByCity. Per ogni città, si ottiene la lista di hotel associati e si procede a ordinare tale lista in base al rankingScore degli hotel.
Successivamente, viene confrontato il primo hotel prima e dopo l'ordinamento. Se il nuovo hotel primo classificato è diverso da quello precedente, viene inviato un messaggio UDP per segnalare l'aggiornamento del ranking locale per quella città tramite il metodo *sendUDPMessage*.
Il metodo *sendUDPMessage* si occupa di inviare un messaggio UDP a un indirizzo IP multicast specifico e a una porta specifica.Questa operazione viene eseguita grazie ad un ‘MulticastSocket’.

>*DataUpdater:*

La classe DataUpdater implementa l'interfaccia Runnable e si occupa di persistere i dati relativi a hotel,utenti recensioni su appositi file JSON.
All'interno del metodo run(), vengono chiamati i metodi writeHotelsToJson, writeUsersToJson e

writeReviewsToJson per scrivere le informazioni degli hotel, degli utenti e delle recensioni nei rispettivi file JSON.
I metodi writeHotelsToJson, writeUsersToJson e writeReviewsToJson convertono le mappe in liste di oggetti corrispondenti, al fine di ottenere un formato di output appropriato per la serializzazione JSON mediante la libreria Gson, in seguito si scrivono tali liste nei file JSON specificati.

6. Utility Generali:

In questa sezione abbiamo sostanzialmente i metodi da me delineati come utility generali

>*readConfig()*:

Il metodo 'readConfig()' di cui all'interno di questa relazione abbiamo avuto modo di parlare, sostanzialmente legge le configurazioni da un file utilizzando un oggetto 'Properties' e assegna i valori alle variabili corrispondeni nel programma, sfruttando 'FileInputStream' per accedere al file di configurazione.

Di seguito si hanno 3 metodi usati nel main per caricare i dati da JSON a struttura dati usate lato server

>*loadHotelsFromJSON():*

>*loadUsersFromJSON():*

>*loadReviewsFromJSON():*

Per terminare questa sezione abbiamo il metodo

>*sort()*:

sostanzialmente prende in input una 'ConcurrentLinkedQueue' di oggetti 'Hotel', la converte in una lista standard 'ArrayList' la ordina usando il metodo 'Colletions.sort', svuota la lista originale e successivamente aggiunge gli elementi ordinati dalla lista temporanea alla 'ConcurrentLinkedQueue'

7. Eccezioni:

In questa ultima sezione abbiamo una sola eccezione

>*class EmptyFileException extends Exceptio*n:

utile per gestire i casi di file vuoti.

4. ClientHotelierMain

il file CerverHotelierMain.java contiene tutto il codice sorgento relativo al server, incluse diverse classi annidate di cui parleremo di seguito.

Il codice sorgente è strutturato nel seguente modo:

1. **Attributi della classe CerverHotelierMain:**
   tra cui troviamo
   >Il percorso del file di configurazione
   >I parametri di configurazione del server(TCP;UDP, exitMessage, …)
   >Variabile per memorizzare l'username dell'utente loggato
   >Oggetto di blocco per gestire l'accesso al gruppo multicast
   >Oggetto di blocco per la stampa sincronizzata
   >SocketChannel per comunicare col server su connessione TCP
   >Riferimento al thread UDP
   >ByteBuffer per la lettura e scrittura dei dati su connessione TCP
   >BufferReader per la lettura dell'input fornito dall'utente per effettuare le richieste
2. **Main:**
   Il main inizia eseguendo la lettura della configurazione del programma dal file, gestendo eventuali eccezioni durante questa operazione. Una volta acquisite le configurazioni, viene presentato un messaggio di benvenuto.
   Successivamente, vengono inizializzati diversi componenti cruciali per il funzionamento del client. Un gestore di terminazione viene aggiunto al Runtime, consentendo la pulizia delle risorse in caso di chiusura improvvisa del programma, viene inoltre inizializzato un socket multicast per ricevere messaggi UDP e allocato un buffer per l'invio e la ricezione di dati. Vengono anche istanziati oggetti per la lettura dell'input da tastiera e la connessione al server tramite socket TCP.
   Di seguito un thread separato viene avviato per gestire la ricezione dei dati tramite UDP. Successivamente, si entra in un loop che presenta un menu delle funzionalità disponibili all'utente.
   Di questo tipo:
   "Cosa desideri fare?
   1. Registrati
   2. Effettua il login
   3. Effettua il logout
   4. Cerca Hotel per nome e città
   5. Cerca tutti gli hotel per città.
   6. Inserisci una review
   7. Mostra i tuoi badge
   8. Esci

   Scegli:"
   Le scelte dell'utente vengono gestite tramite uno switch-case, eseguendo metodi opportuni che corrispondono alle opzioni selezionate.
   In particolare nel caso in cui l'utente dichiara di voler terminare il client e dunque preme '8' viene chiamato il metodo closeConnection(), tramite il quale si notifica il server la volontà di terminazione e si imposta il flag stato = false in modo da uscire dal ciclo while.

Una volta usciti dal ciclo while viene chiamata una System.exit così da mandare in esecuzione lo shutDownHook, per una corretta terminazione

Il loop continua finché la variabile "stato" è true.
Nel caso di interruzione del programma sarà chiuso il bufferReader e se eravamo in attesa di input dell'utente, la lettura restituirà null, per questo c'è un controllo sulla variabile 'choice'.

3. Handle:

Qui troviamo sostanzialmente tutti i metodi di "tipo" handle, tra cui un metodo fondamentale che discuteremo subito:

>*sendRequest(String request)*:

Il metodo sendRequest accetta una richiesta come stringa, si occupa di inviarala su un canale SocketChannel e di leggere e visualizzare la risposta del server. Una volta aver inviato la richiesta al server, il metodo attende e legge la risposta dal SocketChannel. Inizialmente, legge la lunghezza della risposta e quindi recupera il resto della risposta in base a quest'ultima.
La procedura potrebbe richiedere più letture del SocketChannel per garantire che l'intera risposta sia stata ricevuta poichè potrebbe avere una lunghezza maggiore della dimensione del buffer.
Il buffer rimane dunque con dimensione fissa, al massimo si itera per leggere tutto.

Di seguito abbiamo tutti i metodi "handle" che permettono di preparare la stringa di richiesta da mandare al server, spedita grazie al metodo appena visto, sendRequest(). Gli unici metodi di cui accennerò una piccola discussione saranno handleLogin() e handleLogout(), i quali risultano più interessanti.

>*handleRegistration()*:

>*handleSearchHotel()*:

>*handleSearchHotelAllHotels():*

>*handleInsertReview():*

>*handleLogin():*

In questo metodo oltre alla preparazione della stringa, si aggiorna il valore di username, valore che nel caso di login fallito viene rimesso a null.
Per verificare il caso di login fallito si utiliza la libreria Gson per deserializzare la stringa JSON restituita da sendRequest(), ovvero la risposta del server, in un oggetto Java di tipo JsonResponse.
Inoltre in caso di login avvenuto con successo si notifica il thread che gestisce UDP del successo del login, che solo in tal solo caso potrà accedere al gruppo multicast, come specificato dal testo.

>handleLogout():

In questo metodo oltre alla preparazione della stringa, si verifica il successo o meno del logout con la stessa strategia usata per verificare il login.
In caso di logout avvenuto con successo allora si setta username a null e si chiude il multicastSocket, questo perchè in tal modo il

thread UDP che era in attesa sulla receive di tale multicastSocket avrà generata un eccezione che sarà gestita in modo tale da non appartenere più al gruppo multicastSocket come richiesto dal testo (solo gli utenti loggati devono ricevere le notifiche).

3. Classi:

Qui troviamo sostanzialmente tutti le classi annidate:

>*UDPReceiver()*:

La classe 'UDPReceiver' rappresenta un componente fondamentale del sistema client, la quale si occupa della ricezione di messaggi tramite il protocollo UDP da un gruppo multicast.
La classe implementa l'interfaccia Runnabel difatti viene eseguito il metodo run() da un thread separato come già abbiamo discusso nella sezione main.
All'interno del metodo run, si configurano inizialmente i parametri necessari per la ricezione dei dati tramite UDP.
Successivamente, all'interno di un ciclo while, si ha un' ulteriore while che fa si che si attenda su un oggetto loginLockForMulticast fino a quando non si è loggati.
Questo permette di sospendere il thread e di svegliarlo e farlo attendere per la ricezione di un messaggio UDP dal server solo nel caso in cui il client è loggato. Questa difatti è una condizione richiesta dal testo.
Si utilizza questa gestione per evitare di effettuare attesa attiva del thread.
Una volta che la condizione di login è true allora ci si connette al gruppo multicast e ci si pone in attesa di ricevere pacchetti UDP.
Qualsiasi messaggio ricevuto viene poi stampato su console.
Una questione fondamentale di tale metodo è la gestione delle eccezioni, difatti il multicastSocket viene chiuso dal thread main in 2 casi.
Il primo di cui abbiamo già discusso, in caso di logout eseguito con successo, ciò ci permette di non ricevere le notifiche UDP come richiesto.
Il secodo caso invece riguarda la terminazione del client.
Difatti per far si di terminare il thread UDP, non basta chiamare l'interrupt(), il thread se bloccato sulla receive() a meno che non riceva un messaggio dal server non si sbloccherebbe e dunque il programma non terminerebbe.
Quello che si fa è chiudere il multicastSocket in modo tale che il thread UDP sia costretto a gestire l'eccezione, controllerà dunque se il thread è stato interrotto e in tal caso terminerà correttamente.

>*ClientTerminationHandler extends Thread*:

La classe 'ClientTerminationHandler' è una sottoclasse utilizzata per la terminazione corretta del client.

Quest'ultima nel metodo run() setta lo stato del client a false, chiude tutte le risorse aperte, tra cui il flusso di input da tastiera, il SocketChannel e interrompe il thread UDPReceiver responsabile della ricezione dati tramite UDP.

>JsonResponse:

Questa è una classe interna di cui abbiamo già parziamente discusso che è stata progettata per rappresentare la struttura della stringa JSON ricevuta dal server.
I campo di questa classe corrispondono alle chiavi della stringa formattata JSON.
L'annotazione '*@SerializedName'* è utilizzata per specificare il nome della chiave corrispondente nel JSON.

4. readConfig():

In questa ultima parte troviamo il metodo

>readConfig()

5. Schema Generale dei Thread

Questa sezione in realtà dalla lettura delle sezioni precedenti si dovrebbe essere capita, allo scopo però di adempire in modo ottimale alla richiesta di delineare lo schema generale dei thread, dedicherò una sezione apposita.

Lato Server

Lato server, abbiamo il thread Main che chiaramente viene iniziata la sua esecuzione quando il programma Java viene avviato.
Sostanzialmente è il thread che gestisce il flusso principale del server, abbiamo già descritto nelle sezioni precedenti di che cosa si occupa.
All'interno del thread Main come sappiamo si crea un pool di thread (scheduler) per l'esecuzione periodica delle attività e si pianifica e avvia le attività periodiche.
In questo momento si ha quella che chiamiamo la Thread Activation dei due thread DataUpdater e HotelRankingUpdater, cioè vengono assegnate le risorse dal sistema operativo ai 2 thread.
I 2 thread non vengono però eseguiti immediatamente, si ha un delay iniziale di 30 secondi per l'esecuzione di entrambi ed una volta terminata quest'ultima, i due thread vengono eseguiti ogni 60 secondi.
All'interno del thread Main si ha anche la creazione di un'istanza del thread UDPReceiver il quale viene avviato grazia al metodo start().
Inoltre per terminare correttamente nel thread Main si Aggiunge un hook di terminazione (ServerTerminationHandler).
Questo thread viene esguito nel momento nel momento in cui il server viene interrotto da un segnale SIGINT.

Lato Client

Lato Client, il programma inizia con l'esecuzione del thread Main, il quale gestisce il flusso principale del client.

Il Thread Main come ben sappiamo ormai aggiunge Aggiunge un hook di terminazione (ClientTerminationHandler) per gestire la terminazione del client in modo ordinato.
Dunque in caso di SIGINT o comunque di terminazione del client viene attivato tale thread.
Il thread Main inoltre avvia il thread UDPReceiver per ricevere dati tramite UDP.
Quest'ultimo si sospende nel caso di condizione di login == false, cioè il thread passa in stato di attesa.
Viene quindi deschedulato e posto nella coda di attesa fino a quando non verrà chiamata la notify() grazie alla quale verrà spostato in coda pronti.

6. Istruzioni per la Compilazione ed Esecuzione del Progetto

CLIENT
Si naviga le directory fino a ritrovarci su "Hotelier637814/Client/src"
Compilazione:
javac -cp ../lib/gson-2.10.jar ClientHotelierMain.java
Esecuzione:
java -cp ../lib/gson-2.10.jar ClientHotelierMain.java
Esecuzione JAR:
Si naviga le directory fino a ritrovarci su "Hotelier637814/Client"
java -jar Client.jar

SERVER
Si naviga le directory fino a ritrovarci su "Hotelier637814/Server/src"
Compilazione:
javac -cp ../lib/gson-2.10.jar ServerHotelierMain.java
Esecuzione:
java -cp ../lib/gson-2.10.jar ServerHotelierMain.java
Esecuzione JAR:
Si naviga le directory fino a ritrovarci su "Hotelier637814/Server"
java -jar Server.jar