Relazione Hotelier

Mattia Piras

September 06, 2024

Contents

1. Dati

Segue un breve elenco delle entità principali che costituiscono il sistema

1.1. Hotel

La classe `Hotel` modella l'entità fondamentale dell'applicazione. Gli attributi di un istanza della classe sono gli stessi definiti nel file `.json` fornito con alcune differenze. La maggior parte degli attributi che identificano l'hotel sono privati, non appena l'hotel viene deserializzato questi non vengono modificati.

L'attributo `Score` non è presente nel file, essendo presenti invece i 5 punteggi sintetici.
Utilizzando quest'unico attributo, è più semplice gestire gli aggiornamenti del punteggio e il loro recupero. Inoltre si disaccoppia la struttura, rendendo semplice un eventuale refactoring del codice per adattarlo a nuove specifiche (e.g aggiunta o rimozione di un punteggio specifico).

Gli altri attributi: `Score`, `Rank`, `Local Rank Position` sono invece protetti; in questo modo la modifica di questi da parte di altre utility risulta più semplice (senza utilizzare setter e getters).

1.2. Utente

La classe utente modella un generico utente che utilizzerà l'applicazione. Al suo interno definisce un' `enum` statica che modella i vari badges. Questi vengono attribuiti all'utente in base al suo punteggio, inizializzato a 100 in fase di registrazione. L' `enum` possiede un costruttore che associa per ogni valore, il punteggio minimo necessario al raggiungimento del badge.

L'utente definisce inoltre una variabile statica intera che rappresenta il punteggio massimo che un'utente può avere. Questo attributo è utile per la normalizzazione dei moltiplicatori che vengono utilizzati per l'aggiornamento del rank dell'Hotel nel momento in cui l'utente lasica una recensione.

Altri attributi che definisce l'utente sono: la `fingerprint` e il `salt` stringhe che sostituiscono la memorizzazione della password, che rendono il sistema sicuro. Il meccanismo di autenticazione funziona utilizzando un Hash&Salt basato su SHA-256.

1.3. Recensione

La classe recensione mette in relazione un utente e un hotel a un determinato tempo. Una recensione memorizza lo username dell'utente, l' `ID` dell'hotel, l'esperienza attuale dell'utente, il punteggio lasciato con la recensione (utilizzando un istanza della classe `Score`) e una coppia di timestamp `LocalDateTime`. Un timestamp viene appeso nel momento in cui la recensione viene aggiunta dal client, mentre l'altro, viene posto dal `Rank Manager` nel momento in cui la recensione viene considerata per l'aggiornamento del rank dell'Hotel. Infatti, l'aggiornamento del rank è un calcolo incrementale, per cui una recensione aggiorna il rank. La recensione viene considerata, assieme a un peso. Il peso rappresenta quanto la recensione modifica il rank dell'Hotel e viene calcolato in base all'esperienza dell'utente e al lasso di tempo trascorso dall'apposizione della recensione a quando il rank viene effettivamente aggiornato.

1.4. HotelDTO

E' convenzione, per applicazioni reali, utilizzare dei `Data Transfer Objects` che incapsulano i dati non sensibili di oggetti che vengono maneggiati dal server. In questo modo, si consente ai vari client, maggiore maneggevolezza dei dati (a differenza ad esempio di codificare un oggetto come una stringa) senza esporre informazioni su come questo viene codificato dal server.

Dato che in questo contesto, le istanze hotel non contengono dati che possono essere considerati sensibili in qualsiasi circostanza, per giustificare il loro utilizzo si impone, per convenzione, il campo `Hotel_ID` come sensibile, pertanto non presente nelle istanze di `HotelDTO`.

Questo può considerarsi forzato, ma in contesti, ad esempio, in cui l'ID dell'hotel rappresenta la chiave primaria di un record su un Database, l'esposizione di quest'ultimo comporterebbe l'esposizione di informazioni che potrebbero facilitare violazioni della sicurezza.

1.5. Request & Response

Modellano generici pacchetti di richiesta e risposta che transitano tra server e client. I loro attributi seguono uno standard, per cui le richieste possiedono un metodo, indicato a partire da un' `enum` statica e un campo `Data` di tipo `Object` in modo da poter incapsulare qualsiasi tipo di oggetto derivante, restituibile tramite casting.

I pacchetti di risposta invece, contengono uno `Status` che può significare: successo, fallimento o richiesta da parte del server di ulteriori dati riguardanti la richiesta precedente. Possiedono anche un attributo `Error` (significativo solamente per `Status.FAILURE`), che codifica un errore specifico a partire da un' `enum` statica. L' `enum` possiede un costruttore, che permette di restituire un' `Error Phrase` specifica del valore. Anche i pacchetti di risposta contengono un campo `Data` generico.

2. Server

Il server è costituito da diverse componenti ognuna con una funzione differente.

2.1. Schema dei Thread

Il Server funziona secondo il seguente insieme di threads:

1. **`Main Thread`**: si occupa dell'inizializzazione delle strutture dati. Ad avvio completato, riceve pacchetti di richiesta dai client e invia loro i pacchetti di risposta.
2. **`RankManager`**: implementato come un Scheduled Single Threaded Pool, esegue periodicamente l'elaborazione delle recensioni, aggiornamento dei rank degli Hotel, aggiornamento dell'esperienza degli utenti (a recensione elaborata) e eventuale notifica su canale di multicast nel caso di cambiamento del Top Hotel nel ranking Locale (per città).
3. **`File Handlers`**: threads che eseguono il caricamento con deserializzazione dei file `.json` in memoria principale e memorizzazione persistente degli stessi. I threads vegono schedulati periodicamente da uno Scheduled ThreadPool. Il caricamento dei file avviene all'accensione del server, mentre la memorizzazione persistente è periodica, per file.
4. **`Request Handlers`**: thread che eseguono le computazioni necessarie per la soddisfazione delle richieste dei clients. Vengono attivati tramite un Cached ThreadPool.
5. **`Termination Handler`**: configurato come un `Shutdown Hook` effettua operazioni di pulizia allo spegnimento della JVM.

2.2. Strutture Dati

Il server mantiene le strutture dati per la memorizzazione delle entità di riferimento come attributi della classe `Server Context` (`Singleton`).

2.2.1. Hash Maps

Per la memorizzazione di Hotels e Utenti, che dal momento del caricamento dal file, rimangono reperibili in memoria centrale si utilizzano delle `ConcurrentHashMaps`. Per la memorizzazione degli Hotels, le hash map indicizzano (come chiavi) i nomi delle città (case insensitive) su dei `ArrayList<Hotel>`. Le liste sono mantenute ordinate per rank dal `RankManager`, in questo modo l'accesso a tutti gli Hotel avviene in complessità $O(1)$ mentre l'accesso al singono Hotel avviene in complessità $O(n)$. L'utilizzo di `ArrayList<>` è puramente una scelta di comodo, dovuta ai metodi della classe che risultano molto pratici e il fatto che non abbiano capacità fissa si dimostra efficace in certe circostanze. L'applicazione infatti non supporta aggiunte/rimozioni di Hotel dopo l'accensione, quindi si sarebbero ottenuti risultati discreti utilizzando vettori statici `Hotel[]` dovuti principalmente al basso overhead di memoria. In contrapposizione gli `ArrayList<>` hanno un'overhead maggiore, pur essendo più flessibili (ad esempio l'estrazione di un elemento dalla testa del vettore implica lo *shift* di tutti gli elementi).

Gli utenti sono memorizzati utilizzando allo stesso modo delle `ConcurrentHashMaps` indicizzando lo `username` (univoco) e associando l'istanza `Utente`.

La `ConcurrentHashMap` è la scelta più sensata considerato il tempo di accesso costante al valore, e la sincronizzazione *fine-grained* che avviene sui `buckets` (porzioni della mappa) che eventualmente consentono a più threads di accedere a porzioni diverse della stessa. Grazie alla sincronizzazione intrinseca della mappa i metodi di *insertion* che quelli di *retreival* (e.g

`putIfAbsent()`, `compute()`...) sono atomici senza bisogno di meccanismi aggiuntivi. Infine tutte le strutture `Concurrent` mettono a disposizione degli iteratori *fail-safe* (non sollevano eccezioni di tipo `ConcurrentModification`) e *weakly-consistent* (riflettono lo stato della mappa al momento della creazione dell'iteratori) che si rivelano utili quando si accede alle strutture per intervalli.

Anche le recensioni vengono memorizzate (temporaneamente) in una `ConcurrentHashMap<>` indicizzata sull'`HotelID` con valori `ArrayList<Review>`. Il fatto che la mappa debba essere concorrente si deve alla necessità che questa sia scrivibile da più thread (`"Client Handlers"`) e leggibile dal `RankManager`.

Si utilizza infine una `ConcurrentHashMap<String,boolean>` per tenere traccia degli utenti loggati nel sistema. Il valore non è importante, ma non esistono nativamente soluzioni `Concurrent` della classe `Set` per cui si utilizza un valore booleano. Le operazioni eseguite sulla mappa sono solamente `.get(key), .remove(key)`, quindi il valore non è significativo.

2.2.2. LinkedBlockingQueue

Dato che il calcolo del rank dell'Hotel è incrementale (la recensione aggiorna il rank) si è deciso che le recensioni non dovrebbero risiedere in memoria centrale dopo che sono state conteggiate. Poichè anche il rank attuale dell'Hotel viene memorizzato su disco, la persistenza delle recensioni è solo una misura aggiuntiva per garantire la *fault-tolerance* dell'intero sistema. Quindi le recensioni, dopo che vengono conteggiate dal `RankManager` vengono aggiunte a una `LinkedBlockingQueue<>` (`DumpQueue`) che viene periodicamente svuotata da un thread apposito. L'utilizzo di questa struttura dati è preferibile all'utilizzo della variante `ArrayBlockingQueue<>` perchè quest'ultima implementa una coda circolare, con capienza massima indicata in fase di inizializzazione. La coda scelta invece memorizza le references agli oggetti quindi si può espandere fino a esaurimento memoria. Con un'adeguata sincronizzazione dei thread (produttore e consumatore della coda) si garantiscono buone prestazioni.

2.3. Inizializzazione

Prima che il server possa iniziare a operare devono essere configurate correttamente tutte le sue componenti. La configurazione avviene per mezzo della classe `ServerContext` (che implementa il pattern `Singleton`) che oltre a salvare i parametri, inizializza strutture dati, threadpools e in generale fornisce un "entry point" per le risorse a tutte le classi che costituiscono il server. Seguono tutti i passi che il server esegue all'accensione.

2.3.1. Lettura Parametri

Il server supporta il tuning di varie operazioni a partire da un set di parametri che possono essere indicati nel file di configurazione del server.

E' possibile reperire la lista dei parametri con brevi spiegazioni direttamente dalla sezione **Compilazione & Esecuzione**.

La lettura dal file di configurazione viene fatta per mezzo di un istanza `ConfigLoader`. La stessa classe viene utilizzata per la lettura dei parametri di configurazione del client. Il costruttore prende in input il path del file di configurazione e produce un'istanza `Properties` che estende `HashMap`. I parametri possono essere reperiti tramite dei getters appositi della classe `ConfigLoader`, alcuni che eseguono anche il casting a tipi specifi.
Abbiamo un insieme di parametri obbligatori (e.g host e porta del server) e alcuni facoltativi, per cui invocando un metodo `getter` con casting si può specificare il valore di default che il parametro deve assumere (In questo modo si può evitare di lanciare eccezioni dovute a parametri non presenti o casting erroneo).
Il costruttore di `ServerContext` avvia quindi la funzione `parseArguments()` che legge i parametri mappandoli in variabili statiche della classe (si ricordi il pattern Singleton), e successivamente valida la configurazione. La fase di validazione della configurazione permette di fare dei controlli sui parametri obbligatori, come l'esistenza e permessi del file `.json` dove sono salvati gli Hotel, e valori errati di host e porta.
Nel caso la validazione non avesse successo, `ServerContext` stampa a schermo il parametro che provoca l'errore, con eventuali delucidazioni sull'errore specifico e invoca una `System.exit(1)` terminando l'intero processo.

2.3.2. Inizializzazione Risorse

Utilizzando i parametri letti dal file di configurazione il costruttore `ServerContext` inizializza le strutture dati principali del server (vuote) e i vari threadpool, poi termina. Il popolamento delle strutture dati viene eseguito concorrentemente invocando il metodo `ServerContext.ResourcesInit()`. Vengono quindi sottoposte due nuove istanze delle tasks `UsersLoad` e `HotelsLoad`. Dato che il server non può considerarsi avviato senza che le task vengano competate, si raccoglie il valore di ritorno della `pool.submit(Runnable)` di tipo `Future<?>` e si e si attende passivamente la terminazione delle tasks con un metodo privato della classe `waitForTaskCompletion(Future<?>...)` che è variadico e interamente invoca `task.get()`. Infine può essere invocato `ServerContext.scheduleTasks()` che crea un `ArrayList<ScheduledFuture<?>>` e utilizza `pool.scheduleWithFixedDelay(Runnable)` aggiungendo i valori di ritorno alla lista. Questi vengono memorizzati e verranno utilizzati in fase di terminazione.

L'utilizzo di `pool.scheduleWithFixedDelay()` è preferito a `pool.scheduleWithFixedRate()` dato che il parametro `delay` viene interpretato come intervallo di tempo dalla fine di esecuzione della Task quindi offre maggior controllo rispetto a un intervallo di tempo fisso.

2.3.3. Parsing & Marshalling

La classe `Parser` viene utilizzata come "contenitore" per le classi che implementano le task `Runnable` di deserializzazione e serializzazione `.json`. La classe non contiene un costruttore esplicito, si affida infatti al costruttore vuoto predefinito dal compilatore, limitandosi a dichiarare un set di variabili statiche inizializzate nella dichiarazione, che indicano dei `TypeToken` che assicurano un corretto parsing. I metodi che eseguono il parsing a `.json` sono analoghi tra loro così come quelli che eseguono il marshalling, perciò segue una spiegazione generica (con eventuali richiami a caratteristiche specifiche).

- **Parsing**: i runnable che eseguono il parsing caricano `Hotel` e `Utenti` nelle rispettive Hash Maps. Si ricorda che il caricamento delle recensioni non è necessario come gia spiegato precedentemente nella specifica entry nella sezione **Dati**. Il parsing viene eseguito, creando un `JsonReader` a partire da un `FileReader` a partire dalla stringa rappresentante il path del file da cui si esegue il caricamento. Nel caso degli Utenti questi vengono aggiunti direttamente alla mappa con il metodo `.put()` specificando come chiave lo `username` specifico. Per gli Hotel invece, si utilizza il metodo `.compute()` che crea l'`ArrayList<>` nel caso non esistesse la mappatura alla chiave `city` (resa case insensitive tramite il metodo `.toLowerCase()`) e aggiunge l'`Hotel` in coda. Il `JsonReader` è dichiarato in un blocco `try-with-resources` in questo modo la chiusura è automatica all'uscita dal blocco. Nel caso degli Hotel, se si verificasse un'eccezione durante il caricamento, il comportamento di default è la stampa della `StackTrace` associata e l'uscita tramite una `System.exit(1)`. Nel caso degli utenti invece il comportamento di default alla ricezione di un eccezione è quello di settare la HashMap a vuota (nel caso di eccezione durante il caricamento esegue `.clear()` altrimenti per `FileNotFound` non si effettuano mappature).
- **Marshalling**: per la serializzazione, invece, si necessitano meccanismi di sincronizzazione aggiuntivi, dato che questa viene eseguita periodicamente. Per l'esecuzione ci si affida agli iteratori delle `ConcurrentHashMaps` per la caratteristica *fail-fast/weak-consistence*. Dato che la serializzazione non è un processo atomico, si vorrebbe cercare di minimizzare i casi in cui il file viene sovrascritto, ma non completato (a causa di un interruzione). Dato che le cause per cui si ha un salvataggio parziale sono eventi asincroni (e.g crash del sistema per causa imprecisata), il problema non si può risolvere ma limitare. Il salvataggio non viene quindi effettuato direttamente sul file specificato, ma su un file temporaneo, appositamente creato. Non appena il salvataggio è avvenuto, se non ci sono state eccezioni, il path del temporaneo viene rimpiazzato con quello del file indicato utilizzando il metodo `File.move()` indicando le flags `REPLACE_EXISTING` e `ATOMIC_MOVE`.

```java
1    public static final StandardCopyOption ATOMIC_MOVE
```

Move the file as an atomic file system operation

In questo modo si evitano sovrascritture parziali al file (Eventualmente in caso di crash si può ricostruire parte del contenuto del file temporaneo). Il file temporaneo viene automaticamente eliminato se non si verificano eccezioni. Il caso del salvataggio delle recensioni risulta un'po più complicato. Dato che le recensioni si accumulano nel tempo, risulta poco efficiente per ogni salvataggio periodico, leggere il file, e allo stesso tempo scrivere (su nuovo file o sovrascrivere se il file di destinazione è lo stesso del file sorgente) aggiungendo alla fine le nuove recensioni. Per questo, i salvataggi periodici creano nuovi file temporanei (salvati di default in una directory temporanea), e eseguendo dei `.poll()` dalla `LinkedBlockingQueue` contenente le recensioni da scrivere su file. Nel caso in una specifica run la lista fosse vuota, il file temporaneo non viene creato e la task termina immediatamente. Per convenzione si è scelto di fare al più 500 estrazioni dalla lista per run, ma questo parametro può essere facilmente modificato dal file di configurazione. Allo spegnimento del server, si esegue quindi la task `MergeReviews()`

che scrive il nuovo file di destinazione, o eventualmente sovrascrive il file sorgente, con lo stesso meccanismo di file temporanei descritto per la serializzazione di hotel e utenti. Se la serializzazione ha successo allora la directory temporanea viene elminiata, cosiccome i file al suo interno.

2.3.3.1. Approccio Alternativo

In alternativa alla deserializzazione dell'intero file, e serializzazione periodica di tutte le strutture in memoria temporanea si poteva optare per l'utilizzo della `API.stream` offerta da `gson` che consente di leggere e scrivere parti di oggetto `.json` senza serializzare o deserializzare l'intero file. Questa risulta molto versatile per files di grandi dimensioni e inizialmente l'idea di implementazione era proprio questa. Infatti si sarebbe gestita la richiesta di oggetti tramite delle hash maps che emulavano delle memorie cache con rimpiazzamento `LRU`. La questione in questo caso sarebbe stata la granularità di caricamento e scaricamento per gli Hotel. Infatti la realizzazione di una cache a granularità di Hotel avrebbe svantaggiato le richieste che richiedevano un hotel singolo avvantaggiando i le richieste che richiedevano gruppi di Hotel e viceversa. La principale ragione per cui quest'idea è stata scartata è il fatto che la cache si sarebbe dovuta adattare al contesto multithread del sistema. Le `Linked Hash Maps` offrono una base solida e semplice per l'implementazione di cache con rimpiazzamento `LRU` infatti consentono di rimuovere con complessità $O(1)$ l'elemento acceduto più lontano del tempo (si può eseguire l'override del metodo `removeEldestElement()`) dato che, se configurate con l'apposita flag ordinano gli elementi in base all'ultimo accesso. Il drawback è che non esiste a oggi un implementazione nativa `Concurrent` (nonostante esistano diverse repository OpenSource che offrono alternative) il che avrebbe comportato sincronizzare l'intera mappa per ogni thread che richiedeva un elemento. Dato che quindi, sarebbe stato complicato o poco efficiente realizzare l'intero sistema si è optato per la serializzazione e deserializzazione intera dei file, considerato anche il caso specifico di file di piccole dimensioni, il che garantisce prestazioni efficienti.

2.4. Esecuzione

Illustrato il contesto all'accensione del server, espone il funzionamento del server a tempo di esecuzione, successivamente quindi, all'ininizializzazione delle risorse da parte dei metodi di `ServerContext`.

2.4.1. Socket Channel

Per la gestione delle connessioni è stato utilizzato il paradigma di `channel-multiplexing` offerto dai `SocketChannel` della libreria di `java.nio`. Il server apre quindi una `ServerSocketChannel` che servirà da welcome socket, e un selector in un blocco `try-with-resources`. All'interno dello scope del blocco viene quindi definito un `ShutdownHook` che esegue quindi alla ricezione di un'interruzione. Questo è un tipico meccanismo che consente di fare pulizia e garantire una terminazione corretta dei processi allo spegnimento dalla JVM. Il funzionamento del `ShutdownHook` viene illustrato nella sezione **Terminazione**. Procedendo, il canale viene configurato, associando la porta specificata sul file di configurazione e configurando il canale in modalità non bloccante. Infine il canale viene registrati al selettore settando l'option set a `OP-ACCEPT`;

2.4.2. Selector

Si entra quindi in un `while-loop` controllato da una variabile `AtomicBoolean`. Si esegue quindi la `selector.select()`. La selezione implica un'attesa passiva finchè che uno dei canali registrati, non è "pronto" (accordatamente al suo option-set). Nella prima `.select()` il selettore viene risvegliato solo nel caso ci sia una richiesta di connessione, in questo caso, viene aperto un nuovo `SocketChannel` e viene registrato nel selettore associando come `attachment` un'istanza della classe `Session`.

2.4.2.1. Gestione Eccezioni

Il comportamento di default alla ricezione di un'eccezione, è quello di rimuovere la chiave dall'insieme di monitoraggio del selettore e chiudere la socket di comunicazione con il client. Questo viene effettuato sia dal thread che opera il selettore sia dal thread del `MainPool` che gestisce la richiesta.

2.4.3. Sessione

La classe sessione rappresenta istanze che vengono utilizzate per tracciare lo stato di comunicazione di un generico canale, e mappare lo stato di elaborazione delle richieste (qualora queste richiedessero multipli scambi di dati tra il server e il client).

```java
Session {
            String      Username;
            Object      Data;
            Method      LastMethod;
            String      Message;
            ByteBuffer  Buffer;
            boolean     PendingBufferInit;
            boolean     PendingMessageCollection;
            ByteBuffer  SizeBuffer;
        }
```

La sessione viene utilizzata sia dal thread che opera il selettore che maneggia i `ByteBuffer`, sia dal thread che gestirà la richiesta specifica del threadpool. Eventuali dettagli vengono discussi nella sottosezione **Gestione Richieste**.

2.4.4. Channel: Lettura e Scrittura

Le letture e le scritture su canale sono gestite dal thread che opera il selettore. Quando un canale è configurato in modalità non bloccante, le operazioni di lettura e scrittura restituiscono immediatamente il controllo, anche se non ci sono dati disponibili. Questo consente un vero multiplexing dei canali, permettendo di progettare routine che possono scrivere e leggere porzioni di messaggi. In questo modo, il server può gestire più connessioni contemporaneamente, sfruttando i tempi di scrittura e lettura, che sono generalmente molto più rapidi rispetto all'uso di semplici `Socket`.

- **Lettura**: L'operazione di lettura da un canale viene eseguita dal thread che opera il selettore, al risveglio da una `.select()` selezionando una chiava in stato *ready* con option-set `OP_READ`. Viene quindi invocato il metodo `handleRead` che esegue il *retreival* dell'*attachment* della chiave, che è la `Session` associata al canale. La lettura avviene

direttamente dalla sessione, invocando il metodo `Session.readFrom(SocketChannel)`, che restituisce un valore booleano che indica se la lettura del messaggio è terminata. Senza considerare i casi di letture parziali di messaggio, la lettura del messaggio implica almeno due `read` considerando che per convenzione il client prima di inviare un pacchetto invia la sua dimensione in formato intero. Dato che tecnicamente anche la lettura di un intero, potrebbe essere parziale la lettura restituisce `false` finchè il `SizeBuffer` non ha più elementi (tra la posizione corrente e il limite). Non appena il `SizeBuffer` è stato scritto tutto, si recupera il valore intero associato al buffer. Se la lunghezza del messaggio supera un valore predefinito configurabile dal file di configurazione il metodo lancia un eccezione. Altrimenti si ottiene un buffer (come viene effettivamente ottenuto il buffer viene illustrato nella prossima sessione). Si setta quindi il limite del buffer alla lunghezza del messaggio. La lettura del messaggio è analoga a quella della sua lunghezza, per cui si restituisce `false` fino a che il buffer non è stato totalmente riempito, in questo momento si pulisce il `SizeBuffer` per la prossima scrittura e si chiama il metodo `getMessageFromBuffer()` che esegue la decodifica del contenuto del buffer in formato stringa e la si salva nel campo `Message` della sessione settando una flage `PendingMessageCollection`. Dopo il setting della flag tutte le successive letture del buffer restituiscono `true` immediatamente fino a che, viene eseguito il metodo `getMessage()`. Questa è una misura di sicurezza che assicura che il contenuto del buffer non sia sovrascritto finchè un thread non ha recuperato il messaggio associato al buffer. Non appena la lettura è terminata si effettua la `.submit(...)` al threadpool per la gestione della richiesta.

- **Scrittura**: la scrittura è analoga alla lettura. La prima volta che si invoca una scrittura, questa prende il contenuto del messaggio e lo scrive su un buffer, scrivendo anche la sua lunghezza nel `SizeBuffer`, questa operazione è controllata da una flag `PendingBufferInit` che viene risettata a `false` dalla prossima read, a fine lettura. Anche in questo caso la flag è un meccanismo di sicurezza che assicura una corretta sincronizzazione tra letture e scritture

2.4.5. Buffer Pool

In server reali, vengono spesso utilizzati dei `Buffer Pool` che permettono di avere una gestione efficiente della memoria. Sarebbe impensabile in un server che gestisce un numero alto di client contemporaneamente, avere un buffer personale al client, considerando scenari in cui la lunghezza dei messaggi scambiati è decisamente più alta di questo contesto.

Consideriamo uno scenario in cui un server che opera con le modalità descritte debba gestire, un migliaio di client (numero molto basso per scenari reali), scegliamo di allocare un buffer di $2KB$ per ogni client, la memoria utilizzata solamente per l'allocazione dei buffer sarebbe di $2GB$ senza considerare l'overhead dell'istanza

---

Allo stesso modo, non è contemplato riallocare il buffer a ogni comunicazione, dato che questo comporterebbe un overhead sul Garbage Collector, che dovendo gestier continue deallocazioni e allocazioni di segmenti di memoria piccoli dovrebbe continuamente gestire di frammentazione esterna della memoria.

La soluzione implementata è stata quella di realizzare una struttura che implementi una sorta di memoria cache, memorizzando un certo numero di buffer, fino a capienza massima e fornendo sempre un buffer di dimensione "sub-ottimale" in tempo $O(1)$. La struttura utilizzata è una `Linked Hash Map` configurata con flag di accesso. La mappa mantiene uno stato interno in cui "ordina" le entry per tempo di accesso, e permette di rimuove automaticamente l'elemento più "vecchio" con il metodo `RemoveEldestEntry()` di cui si può eseguire l'override. In questo caso specifico si è definita la classe `BufferPool` che estende `LinkedHashMap`. Al costruttore vengono passati due parametri, `MaxSize` che serve per configurare la capienza massima del buffer, e `threshold` che serve a stabire un intervallo per cui consideriamo un buffer gia presente nella mappa ottimale. La mappa memorizza dei buffer indicizzati a partire dalla loro dimensione.

Nel momento in cui si invoca la `BufferPool.get(key)`, come attributo chiave si specifica un limite inferiore alla dimensione del buffer che si vuole ricevere. La dimensione viene quindi arrotondata per eccesso al `threshold` e viene fatta una ricerca nella mappa. Se una mappatura per la chiave esiste allora viene restituita, altrimenti viene allocato un nuovo buffer, inserito nella mappa e restituito al chiamante. L'inserimento nella mappa agisce da trigger per `RemoveEldestEntry()` che rimuove effettivamente il buffer utilizzato più lontano del tempo se la capienza fissata della cache supera la capienza attuale.

E' importante notare che il `bufferPool` può essere utilizzato da un unico thread, dato che l'implementazione non include alcun sistema di sincronizzazione. Nonostante ciò, la soluzione si adatta al meglio al contesto

---

2.4.6. Pacchetti: Ricezione e Inivio

Nonostante le richieste vengano ricevute dal thread gestore come stringhe c'è un passo ulteriore da eseguire prima della gestione della richiesta. Le stringhe in entrata infatti codificano istanza di `Request` in formato `.json`. Queso passaggio ulteriore semplifica la leggibilità del codice, infatti sono definiti dei `TypeAdapter` condivisi tra server e client, molto utili soprattutto in fase di deserializzazione considerando che per costruzione il pacchetto può contenere un campo `Data` di tipo `Object`. In fase di debug e testing si è riscontrato il problema per il quale in mancanza di regole di deserializzazione esplicite, la deserializzazione della stringa di `gson` produceva pacchetti per cui il campo `Data` non veniva deserializzato in formato `Object` ma convertito a istanze di classe errate.

Ad esempio, pacchetti in cui il campo `Data` era di tipo `Object` `istanceof String[]` veniva convertito in `String[]` oppure campi data di classi più complesse come `Score` venivano convertiti in `LinkedHashMap`.

La serializzazione a `.json` oltre a trasferire oggetti complessi ha importanti implicazioni riguardo alla portabilità (e viene utilizzata anche in contesti reali), infatti una stringa è il costrutto base di qualsiasi linguaggio di programmazione; ciò determina il fatto che sarebbe possibile implementare il client utilizzando qualsiasi linguaggio, senza modificare l'implementazione del server.

Il thread alla gestione delle richieste produce dei pacchetti `Response` che vengono serializzati a `.json` prima di essere inviati.

2.4.7. Gestione Richieste

La gestione di una singola richiesta viene delegata a un thread del `MainPool`. La classe `Request Handler` implementa i metodi che vengono utilizzati l'elaborazione delle richieste. La classe implementa `Runnable` e il costruttore prende in input la chiave del selettore, ricavando implcitamente la sessione accedendo all' `attachment`. La classe definisce definisce variabili statiche che vengono inizializzate alla prima istanziazione della classe tramite dei blocchi `static{}`. Il metodo `run` include la deserializzazione della stringa a pacchetto di `Request`. Viene creata nel blocco `static` una `EnumMap<Method,BiFunction<Request,Session,Response>>` utilizzando le `Functional Interfaces` per cui il thread, utilizza il metodo del pacchetto come chiave della mappa, andando a ricavare l'handler specifico per la richiesta, che viene invocato producendo a `return` un pacchetto di risposta che viene serializzato. Dopo ciò il thread setta l' `Option-Set` della chiave a `OP-WRITE`, invoca `selector.wakeup()` e termina.

```
La chiamata selector.wakeup() è presente perchè evita situazioni di
race-condition del thread che opera il selector. Infatti La chiamata
key.interestOps(SelectionKey.OP_WRITE) modifica il set di interesse della chiave. Se
il selector è bloccato su una .select() potrebbe non identificare immediatamente il
cambiamento nell'interest set. Questo può portare a casi in cui il selettore perde
quest'informazione. Chiamando selector.wakeup() il selector viene svegliato e forzato
a rivalutare l'Interest Set di tutte le chiavi registrate, includendo quello appena
modificato.
```

Gli handler delle richieste sono pressochè analoghi tra loro per cui segue la spiegazione dettagliata solo di due di loro.

2.4.7.1. Richiesta Registrazione

Una richiesta di registrazione viene gestita dal server con una doppia comunicazione con il client:

Arrivata la richiesta di Registrazione, si controlla il campo `LastMethod` della sessione, se diverso da `Method.REGISTER` allora è la prima invocazione. In tal caso si effettua il casting del parametro username dal pacchetto di richiesta, se l'username è non valido (non soddisfa l'espressione regolare del file di configurazione) oppure solleva `ClassCastException` si restituisce un errore. Successivamente si controlla se nella hash map esiste gia un utente, se la `.get()` restituisce `null` allora si genera un `SALT`, stringa che verrà concatenata dal client alla password e verrà passata come input alla funzione di hash producendo la fingerprint che verrà mandata nel prossimo messaggio di richiesta. Si crea un nuovo utente e lo si salva temporaneamente nella sessione. Si setta il campo `LastMethod` della sessione al metodo corrente per tracciare lo stato di elaborazione della richiesta. Il `return` in caso di successo è un messaggio di risposta con campo `Status` `AWAIT_INPUT`. Il client invierà il prossimo messaggio contenente la fingerprint generata tramite hash. Arriva quindi un nuovo messaggio di richiesta da parte del client. In questo caso il thread capisce che è il secondo messaggio

controllando `session.LastMethod`, esegue un controllo preliminare sulla lunghezza della password e ricontrolla se l'utente (recuperato dalla sessione) è gia registrato altrimenti setta il campo `fingerprint` e lo inserisce nella mappa.

```java
private static Response handleRegister(Request request, Session session) {
        //prima comunicazione
        if(session.getMethod() != Method.REGISTER){
            String username = (String) request.getData();
            //controlla validità username
            if(!user_regex.matcher(username).matches())
                return new Response(Error.INVALID_PARAMETER);
            try{
                User u = UsersTable.get(username);
                if(u != null) return new Response(Error.USER_EXISTS);
                else{ //genera il salt e lo invia al client
                    u = new User(username, null, HashUtils.generateSalt(SALT_LENGTH));
                    //salva l'utente nella sessione
                    session.Data = u;
                    session.LastMethod = Method.REGISTER;
                    return new Response(Status.AWAIT_INPUT,u.salt);
                }
            } catch (NullPointerException e){
                session.flush(); return new Response(Error.INVALID_REQUEST);
            }
        }
        else{//seconda comunicazione
            String password = null;
            try{   //controlla validità dell'Hash generato dal client
                  password = (String) request.getData();
                  if(password.length() != HashUtils.HASH_LEN)
                    return new Response(Error.INVALID_PARAMETER);
            } catch(ClassCastException e){
                return new Response(Error.INVALID_REQUEST);
            }
            //recupera l'utente dalla sessione e lo salva nella mappa
            User u = (User) session.Data;
            u.setFingerprint(password);
            try {
                if(UsersTable.putIfAbsent(u.username,u) == null)
                    return new Response(Status.SUCCESS);
                else return new Response(Error.USER_EXISTS);
            } catch (NullPointerException e){
                return new Response(Error.BAD_SESSION);
            } finally{
                session.flush();
            }
        }
    }
```

Dato che la registrazione non immagazzina dati dell'utente nella sessione, si esegue `Session.flush()` che setta a `null` tutti i campi della sessione utilizzati negli handler;

La richiesta di Login è analoga, alla prima invocazione, si controlla che l'utente esista e non sia già loggato, in tal caso si restituisce al client il `salt` memorizzato per la generazione dell'hash. Al secondo messaggio si controlla la validità della password con quella memorizzata e in tal caso si aggiunge l'utente alla Hash Map degli utenti loggati e si inizializza il campo `username` della sessione per richieste successive.

2.4.7.2. Richiesta Gruppo Hotel

```java
@SuppressWarnings("unchecked")
    public static Response handleSearchAll(Request request, Session session) {
        Queue<HotelDTO> toReturn = null;
        if(session.LastMethod != Method.SEARCH_ALL){
            toReturn = new LinkedList<HotelDTO>();
            String city = ((String) request.getData()).toLowerCase().trim();
            ArrayList<Hotel> hmap = HotelsTable.get(city);
            if(hmap == null) return new Response(Error.NO_SUCH_CITY);
            //creo una copia in locale della lista
            synchronized(hmap){for(Hotel h : hmap){toReturn.add(h.toDTO());}}
            session.setMethod(Method.SEARCH_ALL);
            session.setData(toReturn);
        }
        //recupera la lista dall'iteratore
        else{
            //l'unico caso in cui la sessione contiene un arrayList
            if((session.getData() instanceof LinkedList<?>)){
            toReturn = (LinkedList<HotelDTO>) session.getData();
            } else return new Response(Error.BAD_SESSION);
        }
        HotelDTO[] batch = new HotelDTO[DEF_BATCH_SIZE];
        //restituisco un batch di hotels
        for(int i = 0; i< DEF_BATCH_SIZE; i++){
            batch[i] = toReturn.poll();
            if(batch[i] == null) break;
        }
        //se gli hotel da inviare sono finiti pulisce la sessione
        if(toReturn.isEmpty()){
            session.clearData();
            session.clearMethod();
            session.LastMethod = null;
            return new Response(Status.SUCCESS,batch);
        } else{
            return new Response(Status.AWAIT_INPUT,batch);
        }

    }
```

La richiesta di *Gruppo Hotel* è l'unica che supporta un numero variabile di scambi di comunicazione tra il server e il client. Infatti il numero di comunicazioni dipende dal numero di hotel che vengono inviati a ogni `run` e dalla dimensione totale del gruppo. Il parametro `BATCH_SIZE` si può modificare tramite il file di configurazione. La richiesta non richiede il login e genera errore (causa client) solamente nel caso in cui la città posta nel campo data del pacchetto non sia indice nella mappa degli Hotel.

Il metodo si propone di offrire un esempio esplicativo di invio di informazioni in `batches` (emulando figurativamente le query con limit e offset a database) utilizzato anche in casi reali per far si che il destinatario possa iniziare l'elaborazione dei dati prima che questi siano stati totalmente trasmessi. Infatti il client, inizia la visualizzazione degli Hotel nel momento in cui riceve il primo gruppo. Solo nel caso l'utente ne richiede altri, innesca richieste successive.

Alla prima chiamata, viene effettuato uno snapshot protetto da `synchronized` della lista degli hotel della mappa.

```
La creazione di uno snapshot locale della lista è necessaria per soddisfare il
requisito di invio ordinato in base al rank degli hotel. Dato che nanche l'iterfaccia
Collection.synchronized(ArrayList<>) supporta iteratori weak-consistent è necessario
salvare una copia locale della lista, dato che l'ordine degli elementi della lista
condivisa potrebbe cambiare dato l'eseguire del Rank Manager
```

La struttura dati che viene utilizzata per effettuare lo snapshot è una `LinkedList<Hotel>`. L'utilizzo di questa, permette di semplificare il codice, dato che la struttura si presta molto bene a essere utilizzata come una coda. Infatti gli inserimenti vengono tutti effettuati in testa alla lista mentre le estrazioni in coda, senza necessità di mantenere un indice.

Se ad esempio si fosse utilizzato un `ArrayList<>` in primis le estrazioni dovevano essere in coda, per non provocare lo shift degli elementi, inoltre per non aggiungere complessità, il rank manager avrebbe dovuto ordinare le liste per rank crescente, il che risulta poco intuitivo, dato che in questo caso il `topHotel` si troverebbe in ultima posizione alla lista.

Eseguendo lo snapshot, si effettua il casting da `Hotel` a `HotelDTO`, utilizzando un metodo della classe che crea la nuova istanza a partire dai campi dell'Hotel, a eccezione del campo `ratings` istanza della classe `Score` che deve essere clonato. Per questo motivo la classe `Score` implementa `Clonable`, ridefinendo il metodo `clone()`;

Man mano che il client effettua le richieste, il metodo rimuove dalla lista e invia, in questo modo il garbage collector può eventualmente effettuare la deallocazione di porzioni della memoria associate allo snapshot. Questo rende il metodo meno *memory-consumptive* e lo rende altamente scalabile, a differenza di invio dell'intero gruppo in un unico pacchetto, che per grossi gruppi di Hotel (assunzione di un caso reale) potrebbe generare problemi di latenza.

2.4.8. Rank Manager

Il rank manager è l'utility che si occupa di aggiornare lo score degli Hotel, mantenendo il ranking locale ordinato. Se il ranking locale cambia, viene inviata una notifica su un canale di multicast, che indica il nuovo `TopHotel` nel ranking locale.

`RankManager` implementa il pattern singleton (implementando `Runnable`), per cui definisce un costruttore privato, utilizzato per inizializzare i parametri utili all'esecuzione oltre che ai parametri necessari per inizializzare un'istanza della classe `UDPNotifier`, che implmenta metodi per l'invio della notifica sul canale di multicast.

Il metodo `run()` viene eseguito su un `SingleThreadedScheduledExecutor` a intervalli regolari dato che inizialmente viene inserito tramite la `.scheduleWithFixedDelay()`. All'avvio, il rank manager cerca di aggiornare il rank locale di ogni città, aggiornando una variabile locale `LocalDateTime` che viene utilizzata per calcolare il peso di ogni recensione. Avviene un update del timestamp per ogni esecuzione rank manager, in questo modo si evitano disparità (seppur piccole) nell'attribuzione dei rank agli Hotel.

scandendo la hash map degli Hotel e chiamando per ogni lista la funzione `UpdateTopHotel(ArrayList<Hotel>)`. I metodi del rank manager richiedono sincronizzazione aggiuntivas dato che si scrivono valori che potrebbero essere letti da altri thread. Si cerca però di di limitare la serializzazione dell'accesso alle strutture dati, prediligendo, sincronizzare per acquisire una copia locale di ciò che verrà scritto, aggiornare il valore in locale, e sincronizzare ancora una volta per fare lo swap della copia locale aggiornata con il valore condiviso obsoleto.

2.4.8.1. Top Hotel Update

La funzione prende in input l' `ArrayList<Hotel>` rappresentante il ranking locale che potrebbe necessitare aggiornamento. Fino a che il ranking locale non è stato aggiornato l'Array List è protetto da un blocco `synchronized`.

```
La sincronizzazione della lista è una misura che previene a un thread di acquisire
una lista di hotel, il cui rank degli hotel è stato aggiornato parzialmente. Potrebbe
essere in questi casi, che il thread legga la lista e in un momento in cui questa deve
essere ancora ordinata. Se ciò succedesse, il meccanismo di invio degli Hotel ordinati
in base al rank verrebbe meno, dato che l'ordinamento della lista viene eseguito dal
rank Manager.
```

Viene chiamata la funzione `updateRankDumpReviews(Hotel, LocalDateTime)` per ogni Hotel del rank Locale. Questa restituisce un valore booleano, nel caso l'Hotel avesse delle recensioni pendenti, che viene messo in *OR* con una flag booleana locale. Se alla fine del ciclo di chiamate, la flag vale `true`, allora almeno un Hotel ha modificato il suo rank, per cui è necessario rieseguire l'ordinamento della lista. All'esecuzione della funzione, questa rimuove la lista delle recensioni associate all'Hotel dalla mappa. Se la lista restituita dalla `.remove()` è `null` allora il metodo termina immediatamente restituendo `false` dato che il rank dell'Hotel non è stato modificato.

Altrimenti dichiara una nuova variabile di tipo `Score` e in un blocco `synchronized` la inizializza al rating attuale dell'Hotel clonato.

```java
Score score = null;
synchronized(h){
  Score local = h.getRating();
  score = local == null ? Score.Placeholder() : local.clone();
}
```

L'operatore ternario controlla se il rating attuale dell'Hotel è null, in questo caso inizializza la variabile locale a un `placeHolder`. Questo caso non dovrebbe mai verificarsi, ma la sua aggiunta è risultata comoda in fase di testing, per cui, tra esecuzioni differenti del server, il campo rating dell'Hotel sul file `.json` veniva eliminato (per effettuare controlli) e di default, durande la deserializzazione se il valore della proprietà non esiste, questa viene inizializzata a `null`.

Dopo la clonazione in locale del punteggio dell'hotel si elabora ogni recensione aggiornando il rank locale, e successivamente eseguendo il dump della recensione sulla `LinkedBlockingQueue` che viene scaricata in fase di serializzazione. Si effettua anche l'aggiornamento del punteggio dell'utente, che di default viene eseguito incrementando l'esperienza di un valore fissato che può essere scelto randomicamente (simulando un giudizio in base alla qualità della recensione) rispetto a un intervallo di cui si specificano gli estremi nel file di configurazione oppure di un valore specifico.

Per evitare di controllare per ogni aggiornamento del punteggio una delle due opzioni specificata sopra, il costruttore del `RankManager` inizializza una variabile di tipo `Consumer<String>` (interfaccia funzionale senza return value a un parametro) a uno dei due metodi predefiniti (incremento punteggio randomico, incremento fisso) in questo modo, dopo l'inizializzazione il consumer può essere eseguito senza verificare l'opzione.

Dopo l'elaborazione di tutte le recensioni, si può tramite `synchronized` sostituire il nuovo rank con quello precedente.

2.4.8.2. Aggioramento Rank

L'aggiornamento dei campi del rating viene eseguito nel seguente modo:

$$p_{i,n} = \frac{1}{\omega_r+1} \cdot \left(\omega_r r_i + p_{i,n-1}\right) \mid \forall i \in \{\text{global, price, position, service, cleaning}\}$$

Il campo ω indica un peso che viene associato alla recensione ed è calcolato come segue:

$$\omega_r = \exp\Big(\left(-\tau \cdot \Delta t_{\text{days}}\right) + \left(-\varepsilon \cdot \frac{e}{\text{max-exp}}\right)\Big) | \tau, \varepsilon \in [0, 1]$$

Il peso viene quindi calcolato utilizzando una funzione di decadimento esponenziale che prende come parametri τ moltiplicato la distanza in giorni tra l'inserimento della recensione e l'elaborazione della stessa e ε moltiplicato per il rapporto tra l'esperienza dell'utente e l'esperienza massima come valore costante.

In questo modo si vanno considerare più indicatrici le recensioni più recenti e apposte da utenti con una certa esperienza.

L'aggionamento viene eseguito su tutti i campi del rating e il rank viene calcolato come la media aritmetica dei punteggi dei ratings.

2.5. Terminazione

La terminazione del serve può essere eseguita dall'utente invocando un interruzione `SIG-INT` da tastiera o `SIG-TERM` specificando il pid del processo che viene visualizzato a schermo al momento dell'accensione del server.

Infatti queste interruzioni in general provocano l'invocazione dei `Shutdown Hooks` definiti dal processo. Il thread di terminazione definito da `ServerMain` è il seguente:

```java
new Thread(( )->{
                try{ System.out.println("Shutdown Hook triggered");
                      running.set(false);selector.wakeup();
                      t.join();
                      System.out.println("Waiting for  server to close...");
                      ServerContext.Terminate();
                      System.out.println("Server closed");
                   } catch(Exception e){e.printStackTrace();}
                })
```

Il thread si occupa di fermare il ciclo della select utilizzando un contatore atmico `"running"` condiviso con il Thread. Esegue quindi la `selector.wakeup` e inizia procedura di terminazione `ServerContext.terminate()` dopo aver eseguito la `.join()` sul thread principale. Questa routine contiene internamente tutti i passaggi per garantire una corretta terminazione del server. Vengono illustrati in maniera generale.

1. **Terminazione delle tasks `scheduled`**: utilizzando i riferimenti `ScheduledFuture<?>` memorizzati in fase di scheduling delle task, per ognuna di queste si esegue `.cancel(false)`, che specifica non verrà più schedulata. Il parametro false, indica che non si vuole forzare l'interruzione di task gia avviate (importante per garantire la consistenza dei dati)
2. **Terminazione del `MainPool`**: si esegue la `.shutdown() & awaitTermination(...)` specificando un tempo di attesa caricato dal file di configurazione. In questo momento il threadpool non accetta nuove richieste, generando un eccezione `RejectedExecutionException` che non viene gestita avendo la certezza che non verrà mai sollevata dato che il thread `ServerMain` è l'unico sottomette le tasks al pool e questo è terminato (si esegue una `Thread.join()`)
3. **Salvataggio dei Dati**: Dopo la terminazione di `MainPool` nessun dato del server viene più modificato causa richieste client quindi si può iniziare il salvataggio dei dati, che per garantire la consistenza dei dati con il contesto del server deve avvenire nell'ordine specificato.
   - Ricalcolo del Rank con attesa di terminazione
   - Serializzazione di Utenti e Hotel
   - Serializzazione delle Recensioni con attesa di terminazione
   - Unione delle recensioni nel file principale
   - Graceful Shutdown di tutti i threadpool

3. Client

Il client dell'applicazione l'interfaccia utilizzata da un generico utente che intende connettersi al server. L'interfaccia è di tipo `Command Line`, utilizzando però sistemi che consentono la navigazione tramite `key-bindings` in contrapposizione all'invio diretto dei comandi. La `TUI` si compone di un menù principale che può essere navigato utilizzando le "frecce" e di tanti menù secondari, circa quanti sono le possibili richieste al server. La visualizzazione di Hotel e Gruppi di Hotel avviene a tutto schermo, mostrando un oggetto per volta.

Nel caso di gruppi di hotel si può navigare il gruppo con le frecce. Allo stesso modo si sono costruiti meccanismi di visualizzazione e inserimento di parametri, simili per tutte le operazioni, la realizzazione viene discussa nella sottosezione **Command Line Interface**.

Il Client inoltre implementa un thread che si comporta da `UDPListener` su un canale di multicast dove riceve delle notifiche dal server in casi specifici. La realizzazione è discussa nella sezione **UDP Receiver**.

3.1. Hotelier API

La classe `Hotelier API` costituisce il fondamento della comunicazione che avviene tra client e server. Si è deciso, di optare per la realizzazione di metodi semplici che il client può invocare passando i parametri della richiesta.

```java
HotelierAPI {
    String              ServerAddress;
    int                 ServerPort;
    Socket              socket;
    BufferedInputStream in;
    BufferedOutputStream out;
    Gson                gson;
    boolean             fetch_init;
    ByteBuffer          lengthBuffer;
}
```

Le fasi di istanziazione della richiesta, serializzazione e invio al server e ricezione, vengono gestiti internamente dalla classe. Questo approccio consente, oltre alla modularizzazione dei metodi, di raggiungere un certo livello di *separation of concerns* e *information hiding* dato che i metodi dell'API si pongono in mezzo alla comunicazione tra il client e server, nascondendo al client l'implementazione e le caratteristiche specifiche dei pacchetti e di come i dati vengono gestiti dal server.

Il client, utilizza il costruttore di `HotelierAPI` come entry point per connessione e comunicazione. Il costruttore prende i parametri `host` e `porta` forniti dal client, per la creazione di una socket che viene memorizzata dall'istanza. Inoltre vengono allocati una `flag` che viene utilizzata per la corretta sincronizzazione di una richiesta e un `GsonBuilder` che viene utilizzato per la serializzazione e deserializzazione dei pacchetti. Questo viene configurato con il `TypeAdapter` della classe `Response` per garantire una corretta deserializzazione (come gia discusso nella sezione **Parsing & Marshalling**).

L'API definisce anche una classe **APIException** che contiene un set di errori specifici sollevati dai metodi della classe **HotelierAPI**

3.1.1. Connessione

Per la comunicazione da client a server, si utilizza una socket, con letture e scritture standard (bloccanti). La socket viene creata dal metodo `HotelierAPI.connect()` che utilizza, l'host e porta dell'istanza.

Durante la fase di connessione vengono anche configurati i due capi di comunicazione della socket come un `InputReader()` e un `OutputWriter()`. A partire da questi vengono creati due `BufferedStreams` che rendono la comunicazione molto più efficiente introducendo dei buffer intermedi per la lettura e la scrittura di dati (non scrivendo direttamente sulla socket) effettuando quindi meno chiamate di sistema per le operazioni di networking.

3.1.2. Socket: Lettura e Scrittura

I metodi di lettura e scrittura sugli streams, sono privati alla classe e vengono invocati solamente dai due metodi `sendRequest()` e `getResponse()` che sono privati a loro volta e vengono invocati internamente dalle chiamate API pubbliche.

La lettura restituisce in output la stringa letta (serializzata a `JSON` dal server) utilizzando un processo analogo alla lettura che avviene a lato server. Convenzionalmente, il server prima di inviare il messaggio invia la sua lunghezza che viene salvata nel campo `lengthBuffer` dell'istanza e successivamente dichiara un `byte[message_length]` e legge la stringa fino a riempire il vettore. Un ciclo di letture continua fino a che la somma dei valori di ritorno delle chiamate `read` non è uguale alla lunghezza del messaggio. Si restituisce una nuova stringa costruita a partire dal `byte[]`.

La scrittura è analoga, prendendo in input la stringa (richiesta serializzata), e scrivendo sul `BufferedStream`, la lunghezza della stringa e il messaggio. A scrittura completata si esegue il `flush` dello stream. Il metodo `write()` è di tipo `void` in quanto la scrittura ha successo, altrimenti genera un'eccezione.

La serializzazione e deserializzazione di richieste e risposte viene eseguita da due metodi privati `wrapper` (`sendRequest(), getResponse()`) che chiamano internamente la lettura e la scrittura sul canale di comunicazione.

3.1.3. API Response

I metodi forniti dall'API forniscono oggetti di tipo `APIResponse` simili a `Response` del package `packet`, che possono essere letti e interpretati dal client in maniera semplice utilizzando i metodi della classe.

Una risposta API contiene un campo di tipo `Status`, come valore di un'enum. In questo modo possiamo disaccoppiare gli `Status` e `Error` dal server con quelli che vengono restituiti al client. Infatti come si vede nella sezione (**Gestione Richieste**) molti metodi che implementa il server, richiedono un doppio scambio di comunicazioni, gestite internamente dall'API.

Il server richiede dati ulteriori per soddisfare la richiesta tramite `Status.AWAIT_INPUT` che viene letto solamente dal metodo API quindi non necessario al client.

Per questo si imposta una mappatura tra gli errori restituiti dal server e degli `Status` dell'API. La mappatura avviene tramite una `EnumMap` per ragioni di chiarezza del codice e di efficienza della struttura.

Oltre allo `Status` abbiamo anche i campi `Message` (mappatura da `Enum Status` a stringa) che associa una status phrase a ogni status e un campo data di tipo `Object` che nel caso di richiesta con `Status.OK` può contenere dei campi significativi. I tipi di dato che vengono gestiti sono `String`, `HotelDTO` e `HotelDTO[]`. Per il retreival con casting del campo data possono utilizzare dei getters appositi che restituiscono `null` nel caso di `ClassCastException`.

3.1.4. Metodi API

Vediamo quindi il funzionamento generico dei metodi API, illustrandone due, paralleli a qelli visti nel caso del server.

3.1.4.1. Risposta Registrazione

Il client invoca una registrazione tramite il metodo `APIResponse UserRegister()` specificando username e password in formato stringa. Questo è un metodo `wrapper` dato che, richieste di `Register,Login e FullLogout` vengono gestite dallo stesso metodo, dato che sono molto simili.

```java
private APIResponse HandleUserOperation(String username,
                                        String password,
                                        Method override){
        //early return
        if(username == null || username.trim().isEmpty())
            return new APIResponse(Status.INVALID_PARAMETER);
        //early return
        else if(password == null || password.trim().isEmpty())
            return new APIResponse(Status.INVALID_PARAMETER)
        //prima comunicazione
        Request request = new Request(override, username.trim());
        sendRequest(request);
        Response response = getResponse();
        //erly return
        if(response.getStatus() == Response.Status.FAILURE)
            return new APIResponse(response.getError());
        //recupera il salt;
        String salt = response.getData().toString();
        password = HashUtils.computeSHA256Hash(password, salt);
        //seconda comunicazione
        request.setData(password);
        sendRequest(request);
        response = getResponse();
        return new APIResponse(response.getError());
    }
```

La funzione utilizza il parametro `Method override` differenziare tra i 3 wrapper. La struttura è semplice. Si effettuano controlli su validità di username e password e si invia una prima richiesta al server invocando `sendRequest()`. Nel caso la risposta del server sia un errore, allora abbiamo un early return del metodo specificando utilizzando l'errore del server per ricavare lo status del metodo. Altrimenti si effettua una seconda richiesta, calcolando l'Hash della password concatenata al salt ricevuto dal server alla richiesta precedente, tramite il `package HashUtil` contenente un insieme di metodi che utilizzano la classe `MessageDigest` per eseguire operazioni crittografiche. La risposta del server viene restituita automaticamente senza controlli sullo stato dato che una risposta con `Status.SUCCESS` contiene il campo `Error` mappato a `Error.NO_ERR` che viene mappato allo status API di `Status.OK`.

3.1.4.2. Risposta Gruppo Hotel

La richiesta di un gruppo di Hotel può essere inviata con i metodi `HotelsFetch(String)/HotelsFetch()`. Questa richiesta è particolare, dato che gli Hotel vengono richiesti e restituiti dal server in batches, per rappresentare la volontà del client di ottenere un batch della stessa città invocando `HotelsFetch()` senza parametri. Nel caso invece si voglia richiedere il primo batch, si deve specificare la città.

```java
public APIResponse HotelsFetch(String City) throws CommunicationException,
                                                    ResponseParsingException,
                                                    NullPointerException
    {   if(City != null) fetch_init = true;
        Request request   = new Request(Method.SEARCH_ALL, City);
        sendRequest(request);
        Response response = getResponse();
        APIResponse apiResponse = new APIResponse(Status.OK);
        Response.Status res = response.getStatus();
        //early return
        if(res == Response.Status.FAILURE){
           apiResponse.setStatus(response.getError());
           return apiResponse;
        }
        else if(res == Response.Status.SUCCESS){
            fetch_init = false;
            apiResponse.setStatus(Status.FETCH_DONE);
        }
        else    apiResponse.setStatus(Status.FETCH_LEFT);
        apiResponse.setData(response.getData());
        return  apiResponse;
    }
```

```java
/*Il metodo può essere chiamato solo se il metodo precedentemente invocato è HotelsFetch(City)*/

    public APIResponse HotelsFetch() throws CommunicationException,
                                            ResponseParsingException,
                                            InvalidMethodInvocation {
        if(!fetch_init) throw new InvalidMethodInvocation();
        return HotelsFetch(null);
    }
```

Il metodo `HotelsFetch()` è invocabile successivamente ad almeno un invocazione di `HotelsFetch(String)` (altrimenti la risposta sarebbe inconsistente). Questo è controllato da dalla flag `fetch_init` che viene settata a `true` la prima volta che il metodo viene invocato, e viene risettata a false nel momento in cui la trasmissione del gruppo termina. In questo caso gli `Status` della risposta sono specifici e notificano al client se può ricevere altri batches di Hotel.

L'API implementa anche un metodo `HotelsFetchAll(String City)` che invoca al suo interno i metodi descritti sopra, restituendo in un'unica invocazione l'intero gruppo di Hotel.

```java
public APIResponse HotelsFetchAll(String, City)
    throws  CommunicationException,
            ResponseParsingException,
            NullPointerException
    {   APIResponse response = HotelsFetch(City);
        if(response.getStatus() == Status.NO_SUCH_CITY) return response;
        ArrayList<HotelDTO> hotels = new ArrayList<HotelDTO>();
        boolean success = false;
        try{    while(response.getStatus() == Status.FETCH_LEFT){
                  for(HotelDTO hotel : response.getHotelList()){
                      hotels.add(hotel);
                  }
                  response = HotelsFetch();
            }
            if(response.getStatus() == Status.FETCH_DONE){
                for(HotelDTO hotel : response.getHotelList()){
                    hotels.add(hotel);
                }
                success = true;
            }
        } catch(Exception e){
            e.printStackTrace();
        }
        return new APIResponse(success ?    Status.FETCH_DONE :
                                            Status.FETCH_PARTIAL,
                                            hotels.toArray(new HotelDTO[0]));
    }
```

`HotelFetchAll(String)` restituisce gli stessi status delle procedure descritte sopra, inoltre può restituire `Status.FETCH_PARTIAL` se c'è stato un errore server durante una delle richieste e il gruppo che allega nel campo data non è completo.

Nonostante il metodo risulti implementato, il client non lo utilizza mai, dato che non si presterebbe bene alle modalità di visualizzazione.

3.2. UDPListener

La classe `UDPListener` estende `Thread` restituendo thread che possono essere schedulati tramite il metodo dell'istanza `.startUDPListening()`.

L'implementazione del thread è stata impegnativa, dato che ci si deve scontrare con il fatto che è impossibile restituire a schermo qualsiasi tipo di output, dato che la `TUI` costantemente effettua la `clear` del terminale andando a riscriverlo. Per questo la strategia è stata quella di creare un file di log che viene eventualmente scritto dal thread al verificarsi di eccezioni. Nel caso allo spegnimento del client, questo non è mai stato scritto (si utilizza una flag) allora questo viene automaticamente cancellato.

Il funzionamento del thread è abbastanza semplice in quanto questo si limita a raccogliere messaggi inviati sul canale di multicast e appenderli in una `LinkedBlockingQueue<String>` per cui il client può estrarre dalla coda e stamparli, senza bisogno di meccanismi di sincronizzazione tra i due thread (oltre a quelli nativi della coda).

Dato che da specifiche, le notifiche dovevano essere mandate solamente a utenti loggati, abbiamo due metodi sull'istanza: `start/stopUDPListening()` fanno partire il thread e lo fanno terminare. Tecnicamente il thread fa partire se stesso, dato che dopo la creazione della lista effettua `this.start()`. La terminazione viene invece controllata tramite una flag `AtomicBoolean`.

Alla terminazione del Client, viene invocato un `ShutdownHook` che si occupa di effettuare pulizie che esegue la `.join` sull' `UDPListener` in modo da garantire un corretto spegnimento senza causare dati inconsistenti nel file di log.

3.3. Interfaccia TUI

La realizzazione dell'interfaccia è stata possile grazie alla libreria `jline` di cui si allega il file `.jar` nel file `.zip`. La libreria infatti offre l'astrazione dei Terminali oltre che a metodi per la lettura di singoli caratteri inseriti configurando il terminale in `rawMode()`. Il funzionamento generale della `TUI` è abbastanza semplice. Si parte dalla definizione di un' `Enum` di opzioni, che rappresentano i pulsanti del menù principale.

Si è deciso, di utilizzare due varianti di menù: una per utenti standard e l'altra per gli utenti loggati (che esclude opzioni di registrazione e evidenzia logout per esempio). Il menù principale viene stampato a schermo nel momento in cui si preme un nuovo carattere. In base al carattere vengono evidenziate parti diverse del menù. Ogni volta che il menù viene stampato, si utilizzano delle `Ansi Escape Sequences` che permettono un controllo sul formatting del terminale. Le sequenze di escape principali utilizzate sono contenute nella classe Ansi che le memorizza come variabili statiche di tipo stringa.

Per l'esecuzione delle operazioni avviene una mappatura tra l'`Enum` delle opzioni e delle istanze di `Supplier<Runnable>`. Infatti le singole operazioni implementano `Runnable` e sono sottoclassi della classe `OptionHandler` che si occupa di definire metodi utilizzati da tutti i runnable.

Ogni Option Handler, contiene un `event loop` principale con cui si utilizza un `KeyBindingReader` per la lettura dei singoli "key-Strokes". Ogni carattere determina un'azione specifica, ad esempio le frecce vengono utilizzate per evidenziare un'altra opzione del menù. L'invio invece viene utilizzato per selezionare. I campi del menù possono essere campi testuali che è possibile scrivere (ad esempio il menù di login permette di scrivere i campi username e password). Esiste sempre un bottone che permette di eseguire l'azione desiderata.

Compilare un menù e selezionare il pulsante per performare l'azione agisce da trigger, per una chiamata API al server, per cui in base alla risposta si può avere uno dei seguenti risultati:

- **Transizione**
  - a una schermata di visualizzazione informazioni (Ricerca Hotel e Gruppi)
  - a menù secondario per inserimento di più dati (Inserimento Recensione)
  - a Menù principale (in caso di operazione completata).
- **Visualizzazione**
  - di messaggio informativo (e.g errore) nella schermata corrente

4. Compilazione, Esecuzione, Parametri di Configurazione

Si pone particolare attenzione sul fatto che il sistema non è portabile su piattaforme diverse da UNIX o McOS. La compilazione e l'esecuzione può essere eseguita tramite un `makefile` che definisce diversi metodi utili.

```bash
make all            #compila Server.jar & Client.jar
make Server.jar
make Client.jar
make runServer      #Esegue il file Server.jar nella finestra corrente
make runClient      #Esegue il file Client.jar nella finestra corrente
make runBoth        #Apre due nuovi terminali e esegue i due programmi
make extractClient  #Estrae Client.jar in extract/client
make extractServer  #Estrae Server.jar in extract/server
make clear          #Rimuove files e directories non necessari
```

```properties
#[SERVER] Default path: config/server.properties
# parametri striga di file di salvataggio ...
pool_maxsize     = 10000
pool_core_size   = 0
pool_keep_alive  = 60000     # (millis)   KeepAlive per thread inattivi
pool_queue_size  = 10000     # Lunghezza  coda task MainPool
pool_await       = 500       # (millis)   Attesa massima di dei Pool
file_coresize    = 3         # CoreSize   per Threadpool dei File Handlers
save_init        = 1000      # (millis)   Intervallo iniziale schedule save
save_delay       = 1000      # (millis)   Delay schedule save
rank_init        = 1000      # (millis)   Intervallo iniziale schedule rank
rank_delay       = 1000      # (millis)   Delay rank
max_dump         = 500       # Max Dump recensioni per esecuzione
exp_multiplier   = 0.1       # [Rank-Update] moltiplicatore dell'esperienza
time_decay       = 0.1       # [Rank-Update] moltiplicatore del  tempo
port             = 7284      # porta del server
multicast-addr   = 239.0.0.1 # indirizzo di multicast
multicast-port   = 9556      # porta di multicast
max_batch_size   = 2         # [Request-Hotels] lunghezza dei chunks
buffer_pool_size = 100       # [BufferPool] dimensione del pool
alloc_threshold  = 100       # (Bytes) [BufferPool] limite dimensione ottimale
add_exp          = 100       # aggiunta punti fissa [da commentare se random]
add_exp_inf      = 10        # limite inferiore aggiunta punti randomica
add_exp_sup      = 100       # limite superiore aggiunta punti randomica
salt_length      = 10        # lunghezza predefinita del salt
name_regex       = ^[A-Za-z0-9_-]+$ # regex per validità username
```

```properties
#[CLIENT MAIN] Default path: config/client.properties
udp_addr      = 239.0.0.1
udp_port      = 9556
# path del file di log di UDPListener
udp_log_path  = data/udp.log
host          = localhost
port          = 7284
```