Hotelier

Sandro Montesu

Contents

1 Introduzione e struttura del progetto

Il progetto è stato realizzato tramite l'utilizzo di **Java 17.0.1** e della libreria **Gson** per la gestione della serializzazione e deserializzazione delle informazioni relative agli utenti e agli hotel di Hotelier. Si struttura in due directory, **Client** e **Server**, i quali comunicano principalmente attraverso **socket TCP**. Le operazioni di lettura degli stream di caratteri sono gestite mediante **BufferedReader**, mentre per la scrittura di tipi di dati primitivi in output si fa uso di **DataOutputStream**. Entrambe le classi, all'avvio, leggono il rispettivo file di configurazione contenente i parametri di input dell'applicazione, quali numeri di porta, indirizzi, timeout, etc. Il file di configurazione è composto da coppie chiave-valore e se viene letta una chiave non attesa, viene generata un'eccezione di tipo *RuntimeException*. Il programma è stato testato su Windows 11 Home.

2 Hotelier Server

L'implementazione del Server fa uso delle funzionalità di **Java I/O** e di un **ThreadPool**. La classe primaria **ServerMain** assume un ruolo cruciale nell'organizzazione del programma: oltre alla definizione delle variabili pertinenti per il file di configurazione e gli oggetti remoti, istanzia due strutture dati fondamentali per il corretto funzionamento del servizio

La prima struttura, **allAccount**, è implementata come un **CopyOnWriteArrayList** di **Account**. Questa struttura memorizza tutte le informazioni relative agli utenti che si registrano al servizio. La seconda struttura, **allHotel**, è anch'essa implementata come un **CopyOnWriteArrayList** di **Hotel** e memorizza tutte le informazioni relative agli hotel gestiti dalla piattaforma.

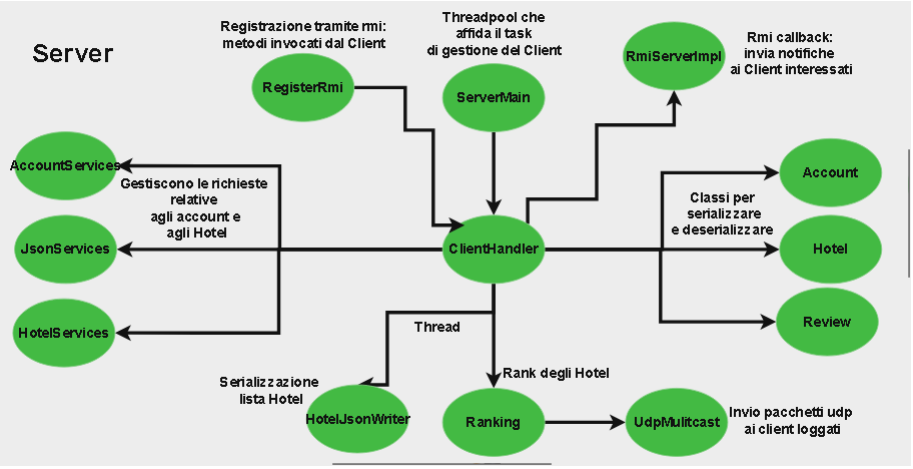
All'interno del metodo main, la classe ServerMain crea un'istanza di **Jsoservices** e chiama diversi metodi per inizializzare le strutture dati. Successivamente, il **ServerMain** mette a disposizione degli oggetti remoti, metodi per la registrazione e la notifica, che possono essere invocati dai client. All'interno di un ciclo, una volta stabilita una connessione **TCP** con il client, la socket viene assegnata a un **ThreadPool** per gestire le richieste dell'utente in modo efficiente e parallelo. Il **ThreadPool** quindi delega il compito di gestire il client alla classe **ClientHandler**, che estende la classe **Thread**.

Inoltre, all'interno del programma, sono presenti altri due thread:

- Il thread **Ranking** ha il compito di aggiornare periodicamente la classifica dei rank locali. In caso di cambiamenti nella classifica, viene utilizzato il servizio di notifica implementato tramite **RMI Callback** per informare gli utenti interessati. Se invece il cambiamento coinvolge solo il primo posto in classifica, tutti gli utenti loggati vengono aggiornati tramite l'invio di un pacchetto **UDP** a un gruppo **multicast**, al quale tutti gli utenti loggati sono iscritti. il periodo che intercorre tra un aggiornamento e il sucessivo è un parametro in input letto dal file di configurazione

- Il thread **HotelJsonWriter**, il quale si occupa di serializzare periodicamente le informazioni relative agli hotel. il periodo che intercorre tra un aggiornamento e il sucessivo è un parametro in input letto dal file di configurazione.

  Di seguito un'immagine rappresentante una struttura semplificata del Server:



.

2.1 Classi di supporto

- Interfacce:

  ServerInterface:

  Interfaccia remota che definisce i metodi per registrarsi e dere-gistrarsi al servizio di notifica.

  Registation:

  Interfaccia remota che definisce i metodi per registrare un utente a Hotelier.

- Implementazione Interfacce:

  RegisterRmi:

  Implementa l'interfaccia **Registation** e estende **RemoteServer** per supportare **RMI**, contiene un'istanza di **Jsonservices**, chiama il metodo **JsonAddUser** e restituisce il messaggio ottenuto dalla sua invocazione;

  RmiServerImpl:

  Implementa l'interfaccia **ServerInterface** e estende **RemoteServer** per supportare **RMI**. Gestisce la registrazione e la deregistrazione dei client per le callback, notifica i client di cambiamenti nella posizione degli hotel e fornisce accesso alle preferenze degli utenti.

  Variabili di istanza:

  - *userPreferences*: **Map** che associa una città a una lista di account interessati a quella città.
  - *clients*: **Map** che associa un account a un'interfaccia di notifica per le callback. Viene utilizzata per tenere traccia dei client registrati per le callback.
  - *accounts:* Lista di tutti gli account.

  Metodi:

  - *registerForCallback*: Questo metodo permette a un client di registrarsi per le callback.
  - *unregisterForCallback:* Questo metodo permette a un client di disiscriversi dalle callback.
  - *notifyHotelPositionChanged*: Questo metodo viene chiamato per notificare i client di un cambio di posizione di un hotel in una città.

- Serializzazione e Deserializzazione:

  Account:

  Si tratta di una classe utilizzata per serializzare e deserializzare le informazioni relative agli account, quali **username, password, numero di recensioni** e **città di interesse**. La funzione **setBadge** consente di impostare il badge dell'utente in base al numero di recensioni, mentre i metodi **getter** e **setter** permettono l'accesso alle diverse informazioni dell'account.

  Hotel:

  Si tratta di una classe utilizzata per serializzare e deserializzare le informazioni relative agli hotel, quali **nome, città, rate**, e **data dell'ultimo voto inserito**. I metodi **Getter** e **setter** consentono di accedere alle varie informazioni dell'hotel.

  Review:

  Questa classe presenta un solo costruttore e memorizza le informazioni relative a una recensione. Tali informazioni includono l'username dell'utente che ha effettuato la recensione, il nome dell'hotel recensito, la città dell'hotel e i voti inseriti.

  LocalDateTimeAdapter

  Questa classe non viene direttamente serializzata o deserializzata. Tuttavia, viene utilizzata per sovrascrivere il comportamento predefinito di serializzazione e deserializzazione della classe **Gson**. È importante notare che questa classe non gestisce direttamente la serializzazione di variabili di tipo **LocalDateTime**. L'override viene effettuato solo quando è necessario serializzare o deserializzare oggetti di quel tipo.

- Thread:

  ClientHandler:

  Thread creato e inizializzato nella classe **ServerMain**, gestito da un Threadpool, esistono tante istanze di questo Thread quanti sono gli utenti connessi al server. Il thread gestisce i client comunicando con ognuno di essi tramite uno stream di input **BufferedReader** e uno stream di output **DataOutputStream**. Per eseguire varie operazioni, si avvale dei metodi delle classi **JsonService, HotelService** e **AccountService**, di cui le istanze sono presenti nel costruttore. In caso il client venga disconnesso verrà stampato un messaggio di errore.

  All'interno del metodo **run()**, viene utilizzato uno switch per distinguere tra i diversi comandi inviati dal client. Alcuni comandi possono essere eseguiti solo da utenti loggati, quindi viene effettuato un controllo prima di eseguirli utilizzando if(utente==null).

  - *login:* Durante la fase di login, il Client invia il proprio nome utente e password. Il metodo di login della classe **AccountServices** verifica se l'utente è registrato e se la password inserita è corretta; in caso di errore, invia un messaggio di errore al client. Se il login ha successo, il thread controlla se l'elenco delle città di interesse dell'utente è vuoto e, in tal caso, richiede all'utente di inserirle.
  - *Favorite*: Questo campo gestisce l'inserimento delle città di interesse dell'utente. In caso di errore durante l'inserimento, l'utente dovrà ripetere la procedura di login.

    ATTENZIONE: La scelta delle città di interesse può essere effettuata correttamente una sola volta e non è possibile modificarla successivamente.
  - *logout*: Se l'utente risulta già loggato viene effettuata la fase di logout
  - *searchHotel*: Questa operazione è valida per tutti gli utenti. L'utente inserisce prima il nome dell'hotel che desidera cercare, seguito dalla città in cui si trova. Utilizzando i metodi di **HotelServices**, viene verificata l'esistenza della città e dell'hotel in quella città. Se la ricerca ha successo, viene inviata al client una stringa **JSON** ottenuta tramite la classe **Gson**, contenente tutte le informazioni relative all'hotel.
  - *searchAllHotels*: Operazione simile alla precedente. Il Client invia il nome della città di interesse e riceve una stringa **JSON** contenente la lista di tutti gli hotel presenti in quella città. Tale lista è ordinata in base all'ultimo ranking locale effettuato, dove il primo hotel visualizzato sarà il primo della classifica. Tuttavia, è importante notare che il thread responsabile dell'aggiornamento delle posizioni degli hotel nel ranking locale potrebbe non aver ancora eseguito l'ultimo aggiornamento. In questo caso, gli hotel verranno mostrati con i voti aggiornati, ma la posizione in cui verranno visualizzati sarà quella precedente.
  - *insertReview:* Questa operazione consente di aggiungere una recensione a un hotel specifico in una determinata città. L'utente invia prima il nome dell'hotel e successivamente la città in cui si trova. In seguito, inserisce il voto globale e un voto per le diverse categorie (Pulizia, Posizione, Servizio, Qualità), con valori compresi tra 0 e 5, inclusi gli estremi. Dopo aver effettuato i controlli necessari utilizzando la classe **HotelServices**, la recensione viene inserita e l'istanza della classe **Hotel** che rappresenta l'hotel in questione viene aggiornata.
  - *showMyBadges:* Questa operazione è valida solo per gli utenti loggati. In base al numero di recensioni inserite, è possibile ottenere un badge, rappresentato da una stringa. Utilizzando un metodo della classe **Account**, è possibile ottenere il badge corrispondente. Successivamente, viene effettuato un aggiornamento dell'istanza di quell'account. I badge disponibili sono i seguenti:
    - se l'utente non ha inserito recensioni verrà stampato *nessun badge*
    - se l'utente ha inserito una recensione il badge ottenuto sarà *Recensore*
    - se l'utente ha inserito due recensioni il badge ottenuto sarà *Recensore Esperto*
    - se l'utente ha inserito tre recensioni il badge ottenuto sarà *Contributore*
    - se l'utente ha inserito quattro recensioni il badge ottenuto sarà *Contributore esperto*
    - se l'utente ha inserito cinque recensioni il badge ottenuto sarà *Contributore Super*

HotelJsonWriter:

Thread creato e inizializzato nella classe **JsonServices** tramite il metodo **HotelJsonWriter**.

Si occupa di serializzare periodicamente le informazioni aggiornate della lista degli hotel sul file **JSON "Hotels.json"**, utilizzando i metodi della classe **Gson**.

Ranking:

Thread creato e inizializzato nella classe **JsonServices** tramite il metodo **JsonHotel**.

Nel costruttore, viene passata una **ConcurrentHashMap** chiamata **mapcity**, la quale mappa ogni città alla lista degli hotel corrispondenti. Il **thread**, implementato nel metodo **run()**, si occupa di calcolare il ranking locale per ogni città. Questo ranking viene determinato invocando il metodo **Compare** della classe **HotelComparator**. Se il ranking del primo hotel in una città cambia, il metodo **sendMulticast** della classe **UdpMulticast** viene chiamato per inviare un pacchetto **UDP** a tutti gli utenti loggati. In caso di cambiamenti nella classifica complessiva, il metodo **notifyHotelPositionChanged** della classe **RmiServerImpl** viene invocato per notificare i Client interessati. Il servizio di notifica invia un messaggio al Client per ogni Hotel che ha cambiato classifica nelle città di interesse indicate dal client stesso, indicando la nuova posizione nel rank.

La concorrenza è gestita attraverso l'utilizzo di strutture dati thread-safe come **ConcurrentHashMap** e **CopyOnWriteArrayList**. Inoltre, per garantire la coerenza dei dati, ogni campo della classe Hotel su cui più thread possono accedere è sincronizzato utilizzando il modificatore synchronized.

- Classi di supporto alla classe Ranking:

  HotelComparator:

  Questa classe presenta un unico metodo **compare**, il quale restituisce una lista di hotel ordinati. Gli hotel vengono ordinati in base al voto globale (**rate**), e in caso di parità di voto, vengono ordinati in base ai voti delle singole categorie (**ratings**). Se vi è parità nei voti delle singole categorie, gli hotel vengono ordinati in base al numero di voti (**nOfVote**). Infine, in caso di parità anche nel numero di voti, gli hotel vengono ordinati in base alla data dell'ultimo voto inserito (**lastVoteDate**). Questo ordinamento tiene conto sia della qualità che della quantità delle recensioni. Per quanto riguarda la qualità, vengono considerati i voti delle recensioni nelle diverse categorie. Per quanto riguarda la quantità, viene considerato il numero totale di recensioni. Tuttavia, per l'attualità, viene preso in considerazione solo l'ultimo voto inserito e non quelli precedenti.

  UdpMulticast:

  Questa classe contiene un unico metodo **sendMulticast**, il quale si occupa di inviare tramite **UDP multicast** un messaggio che indica il nuovo hotel in prima posizione per ogni rank locale a tutti gli utenti loggati.

- Services:

  JsonServices:

  La classe **JsonServices** si occupa di gestire tutte le operazioni che interagiscono coi file **JSON**, nello specifico con "**account.json**" e "**Hotels.json**" Il costruttore riceve alcune strutture dati condivise, tra cui **AllAccount**, un **CopyOnWriteArrayList** di **Account** che conterrà le informazioni di tutti gli account, e **AllHotels**, un **CopyOnWriteArrayList** di **Hotel** che conterrà le informazioni di tutti gli hotel. L'istanza di questa classe viene creata nel main e poi passata nel costruttore di tutte le altre classi che ne fanno utilizzo.

  Metodi:

  - *createNewFile:* Il metodo verifica l'esistenza del file "**account.json**" e, nel caso in cui non esista, lo crea. Si presuppone che il file "**Hotels.json**" sia già presente. Viene chiamato nel Main.
  - *JsonHotelWriter* Crea un'istanza della classe **HotelWriter** e avvia un nuovo Thread mediante l'invocazione del metodo **start()**. Viene chiamato nel Main.

- *JsonAccount*: Dopo aver chiamato il metodo **createNewFile()** nel Main, il seguente metodo viene invocato. Se nel file "**account.json**" sono presenti informazioni sugli account, queste informazioni vengono deserializzate e inserite nella lista di Account **AllAccount**, mentre il booleano che indica se l'utente è loggato viene impostato a **false**.
- *JsonHotel*: Questo metodo si occupa di deserializzare le informazioni presenti nel file "**Hotels.json**" e di memorizzarle all'interno della lista di Hotel denominata **AllHotels**. Per semplificare le operazioni relative agli hotel, crea una **ConcurrentHashMap** chiamata **mapcity**, la quale mappa ogni città alla rispettiva lista di hotel. Le operazioni di Ranking, di inserimento di una recensione e di ricerca vengono eseguite su questa mappa, che garantisce un sistema di accesso alle risorse più efficiente, soprattutto per quanto riguarda la lista degli hotel di una stessa città. Viene inoltre impostato il valore **lastVoteDate** al tempo corrente (indicando che questo campo sarà inizialmente uguale alla data della creazione della lista di Hotel) e viene avviato il thread responsabile di effettuare il ranking degli hotel.
- *JsonAddUser*: Questo metodo **synchronized** si occupa di gestire l'operazione di registrazione dell'utente ed è invocato dalla classe **RegisterRmi**. Verifica se lo username dell'utente esiste già; in caso contrario, aggiunge un nuovo account alla lista degli **Account** e aggiorna il file "**account.json**".
- *UpdateUser* Metodo **synchronized** che aggiorna le informazioni sul file "**account.json**"
- *jsonReview*: Questo metodo viene utilizzato per inizializzare una lista di recensioni. Se il file JSON "**Review.json**" è vuoto, la lista viene inizializzata come un nuovo **ArrayList**. In caso contrario, il contenuto del file "**Review.json**" viene deserializzato all'interno della lista. Questa operazione viene eseguita utilizzando la classe **LocalDateTimeAdapter** per **LocalDateTime**, che effettua l'overriding dei metodi di serializzazione e deserializzazione usati da **Gson** per consentire la corretta deserializzazione degli oggetti **LocalDateTime** presenti nelle recensioni.
- *jsonAddReview*: Questo metodo viene chiamato per serializzare la lista di recensioni e dipende dalla classe di supporto **LocalDateTimeAdapter** per sovrascrivere i metodi di serializzazione utilizzati dalla libreria Gson.
- *Getter* Esistono inoltre i metodi **getAllAccount, getMapCity e getAccount**, i quali restituiscono i rispettivi oggetti.

Funzioni:

- *addCity* La funzione addCity legge dal file "**listOfCity.txt**" e crea una lista di stringhe contenenti le città. Questa lista viene successivamente utilizzata per creare la **ConcurrentHashMap mapcity**

HotelServices:

La classe **HotelServices** gestisce tutte le operazioni relative agli hotel che non coinvolgono direttamente il file JSON "Hotels.json". All'interno di questa classe è presente un'istanza della classe **JsonServices**, che permette l'accesso alle strutture dati condivise. Nel suo costruttore, viene invocato il metodo **jsonReview** della classe **JsonServices**, il quale inizializza la lista delle recensioni. Questa lista è una struttura dati destinata a memorizzare tutte le recensioni relative agli hotel.

Metodi:

searchHotel: Metodo **synchronized** che si occupa di effettuare la ricerca dell'hotel specificato nella città specificata. La ricerca viene eseguita all'interno della **ConcurrentHashMap mapcity**. Se si verifica un errore di qualche tipo, ad esempio se la città selezionata non esiste, il metodo restituisce **null**. Altrimenti, ritorna l'istanza dell'hotel richiesto.

searchAllHotels: Metodo simile al precedente, restituisce la lista degli Hotel nella città indicata.

addReview: Questo metodo, marcato come **synchronized**, gestisce l'aggiunta di una recensione all'hotel selezionato. Calcola i voti delle diverse categorie come media aritmetica di tutti i voti, incrementa il numero di voti dell'hotel e il numero di recensioni dell'utente, e aggiorna le informazioni nell'istanza della classe **Hotel**, ottenuta tramite l'accesso alla **ConcurrentHashMap**

mapcity. Se l'operazione di inserimento della recensione ha successo, viene creata un'istanza della classe Review, che viene aggiunta alla lista di recensioni dell'hotel. Infine, utilizzando il metodo **jsonAddReview** presente nella classe **JsonServices**, la recensione viene serializzata nel file **Review.json**.

Funzioni:

- La funzione **changeIntCategory** è una funzione di supporto che converte l'indice del ciclo attuale nella chiave corrispondente nella HashMap.

AccountServices:

Questa classe si occupa di gestire le operazioni relative agli account che non interagiscono direttamente col file **JSON "account.json**".Contiene un'istanza della classe **JsonServices** per poter accedere alle strutture dati condivise.

Metodi:

- *login*: Metodo **synchronized**. Effettua la verifica dell'**username**, della **password** e del booleano **LoggedIn**. Setta **LoggedIn** a **true** se l'utente in questione non è già loggato in un'altra sessione. Questo perché solo un utente alla volta può accedere a un account, e non è consentito loggare contemporaneamente con lo stesso account su client diversi. Restituisce una stringa che indica se l'accesso è avvenuto con successo o se si è verificato un errore.
- *logout*: Metodo **synchronized**. Effettua il logout dell'utente e setta il booleano **LoggedIn** a **false**

2.1.1 Strutture Dati

Le strutture dati più usate sono:

- **CopyOneWriteArrayList <Account> allAccount** usata per memorizzare le informazioni degli Account registrati al servizio, viene usata inoltre per motivi di serializzazione e deserializzazione. Creata nella classe **ServerMain**, viene poi inizializzata nella classe **JsonServices** e resa disponibile tramite i metodi get di quest'ultima.
- **CopyOneWriteArrayList <Hotel> allHotel** usata per memorizzare le informazioni degli Hotel registrati al servizio, viene usata inoltre per motivi di serializzazione e deserializzazione. Creata nella classe **ServerMain**, viene poi inizializzata nella classe **JsonServices**.
- **ConcurrentHashMap<String>, CopyOnWriteArrayList<Hotel>**: Map concorrente che mappa ad ogni città la corrispondete lista di Hotel, usata per facilitare alcune operazioni quali ad esempio l'operazione di ranking e quella di ricerca degli hotel. viene creata e inizializzata nella classe **JsonServices** tramite il metodo **JsonHotel**.
- **Map<String, List<Account> userPreferences**: struttura dati utilizzata per gestire le **callback RMI**. Essa mappa ogni città a una lista di account interessati a quella città. Inizialmente è una mappa semplice, ma viene successivamente castata a **ConcurrentHashMap** per garantire la concorrenza. Questa mappa viene inizializzata nel costruttore della classe **RmiServerImpl** e viene aggiornata nei metodi **registerForCallback** e **unregisterForCallback**.
- **List<Review>**: struttura dati utilizzata per tener traccia delle recensioni inserite dagli utenti,
- **Map<Account, NotifyEventInterface> clients**: Struttura dati usata per gestire la **callback RMI**. Essa mappa ogni Account al rispettiva NotifyEventInterface. Inizialmente è una mappa semplice, ma viene successivamente castata a **ConcurrentHashMap** per garantire la concorrenza. Questa mappa viene inizializzata e aggiornata nei metodi **registerForCallback** e **unregisterForCallback**.

2.1.2 Concorrenza

La concorrenza è gestita in modo sicuro attraverso l'uso di metodi **synchronized** per la maggior parte dei metodi presenti nelle classi **JsonServices, HotelServices** e **AccountServices**. Inoltre, le strutture dati condivise utilizzate da più thread contemporaneamente sono strutture concorrenti come **ConcurrentHashMap** e **CopyOnWriteArrayList**. Inoltre, i metodi **get** e **set** di alcune variabili presenti nelle classi **Account** e **Hotel** sono anch'essi sincronizzati per garantire un maggiore livello di concorrenza e prevenire eventuali problemi di accesso concorrente.

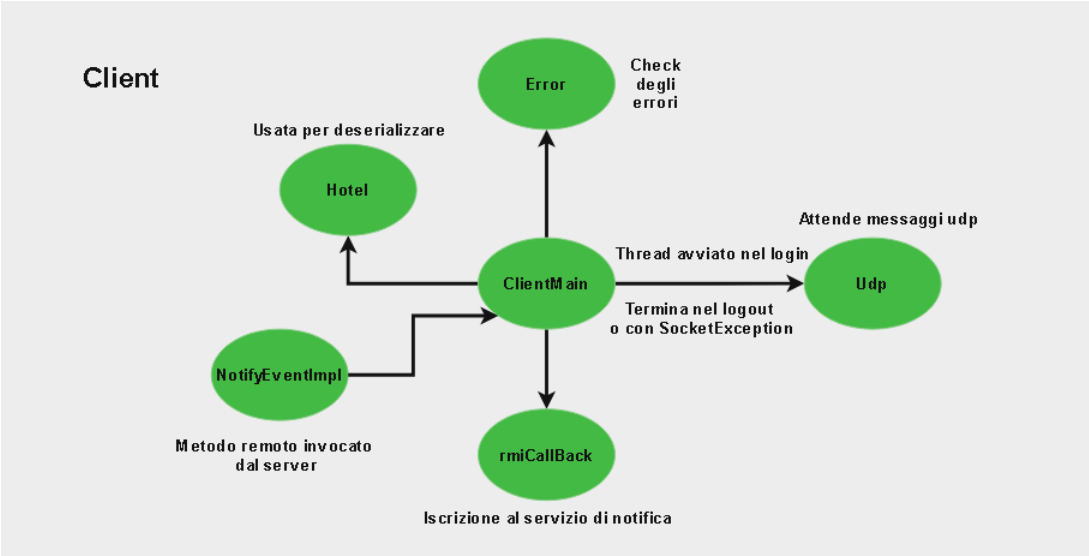
3 Hotelier Client

La classe principale è **ClientMain**, la quale, dopo aver letto i parametri di input dal file di configurazione (indirizzo, porta del server...), stabilisce una connessione **TCP** con il server e gestisce l'interazione tra l'utente e il server. La comunicazione avverrà tramite l'utilizzo di un **BufferedReader** per lo stream di input e di un **DataOutputStream** per lo stream di output.

L'input da tastiera viene suddiviso in un array di stringhe utilizzando il metodo **split**, separando le parole mediante uno spazio per distinguere il comando dagli argomenti.

Il comando viene poi filtrato attraverso uno switch, dove gli argomenti, se presenti, vengono ulteriormente suddivisi mediante il metodo **split** in stringhe separate da una virgola. La classe utilizza il metodo **Check** della classe di supporto **Error** per verificare che i comandi e gli argomenti siano scritti correttamente. La classe verifica se un utente è già loggato utilizzando il metodo **getIsLogged** della classe **Login**, inizialmente settato a **false**. In caso il client venga disconnesso dal server verrà stampato un messaggio di errore.

Di seguito un'immagine rappresentante una struttura semplificata del CLient:



Comandi:

- *register*: Dopo aver verificato le varie condizioni (come utente non loggato, assenza di errori e numero corretto di argomenti), viene effettuata la registrazione dell'utente tramite la funzione **rmi**, che a sua volta utilizza il metodo remoto per la registrazione.

  Successivamente, si passa alla fase di login, durante la quale l'utente comunica con il client tramite stream di input e output. Il client invia una stringa contenente comando e argomenti tramite il metodo **writeBytes**, e il server deve separare nuovamente la stringa ricevuta.

  Se il login ha successo, l'utente deve inserire le città di interesse per potersi iscrivere a un servizio di notifica tramite **RmiCallback**. In caso di errore in questa fase, l'utente viene sloggato e deve ripetere la procedura di login, indicando nuovamente le città di interesse.

  Se tutte le fasi precedenti sono terminate con successo, l'utente viene iscritto al servizio di notifica utilizzando il metodo **callBackreg** della classe **RmiCallback**, e viene avviato un thread in attesa di ricevere messaggi **UDP** utilizzando il metodo **start**.

- *login*: questa fase è identica alla precedente, ad eccezione della fase di registrazione. Se l'utente ha già inserito correttamente le città di interesse in precedenza, non sarà necessario richiedere nuovamente questa informazione.

- *logout*: Durante la fase di logout, viene verificato che l'utente non sia già loggato. Successivamente, viene inviato il comando al server. In caso di successo, viene impostato il booleano **IsLogged** a **false**. Viene poi eseguito un **join** sul thread **udp** e infine viene chiamato il metodo **UnregisterCallback** della classe **RmiCallback**.

- *searchHotel*: Questa fase gestisce la richiesta delle informazioni relative a un hotel in una specifica città. L'utente deve inserire prima il nome dell'hotel e successivamente la città in cui si trova, separati da una virgola. Se la ricerca ha successo, grazie al metodo **readLine**, sarà possibile ricevere una stringa **JSON** contenente le informazioni sull'hotel in questione. Utilizzando la libreria esterna **Gson**, è possibile deserializzare la stringa in un'istanza della classe **Hotel**. Successivamente, mediante l'uso del metodo **printAll**, è possibile stampare solo le informazioni di interesse. La fase di deserializzazione è incapsulata all'interno di un blocco **try-catch**. Questo perché la stringa ricevuta potrebbe contenere un messaggio di errore. In tal caso, il messaggio verrà letto nella sezione **catch**.

- *searchAllHotels*: Questa fase è simile alla precedente, ma l'utente deve inserire solo il nome della città di interesse. Se la ricerca ha successo, riceverà la lista degli hotel presenti in quella città.

- *insertReview*: Questa fase è riservata agli utenti loggati e consente di inserire una recensione per un hotel specifico. L'utente deve inserire nella stessa riga il nome dell'hotel, seguito dalla città in cui si trova, il voto globale e i voti per le singole categorie (Pulizia, Posizione, Servizio, Qualità), eventualmente racchiusi tra parentesi tonde.

- *showMyBadges* Fase riservata agli utenti loggati, permette di visualizzare la Stringa rappresentante il proprio Badge.

- *help* Tramite il comando help sarà possibile visualizzare tutti i possibili comandi; digitando help-comando sarà possibile visualizzare la sintassi del comando e alcune informazioni relative ad esso.

Funzioni:

- *readClientConfig*: si occupa di leggere i parametri in input dal file di configurazione

- *rmi*: Questa funzione si occupa di eseguire una chiamata **RMI** per interagire con un oggetto remoto chiamato "**register-rmi**", utilizza il metodo remoto **reg** per effetuare la registrazione.

3.1 Classi di Supporto

- Classi di Supporto a ClientMain:

  Error:

  La classe **Error** contiene un unico metodo chiamato **Check**, il quale accetta il comando e i suoi argomenti come input. Gli argomenti sono passati come un array di stringhe. Utilizzando un'istruzione switch, il metodo identifica il comando e esegue il controllo degli errori, come la presenza di argomenti vuoti o il numero corretto di argomenti, nonché la validità dei valori numerici. In caso di errore, il metodo restituisce **true** e stampa un messaggio che indica il tipo di errore. Altrimenti, restituisce **false**.

  Hotel:

  La classe è utilizzata per deserializzare una stringa **JSON**. Rispetto alla classe usata nel server, questa classe presenta il metodo **printAll**, il quale consente di stampare solo le informazioni rilevanti dell'hotel.

Login:

Questa classe contiene solo due metodi: **getIsLogged** e **setIsLogged**, utilizzati per verificare se l'utente è loggato e impostare lo stato della variabile.

rmiCallback:

All'interno del costruttore, viene ottenuto il registro **RMI** tramite **LocateRegistry.getRegistry**. Successivamente, viene effettuata la ricerca del server **RMI** nel registro tramite il nome specificato. Infine, il server viene assegnato alla variabile **server**.

Nel Metodo **callBackreg** registra un oggetto remoto di callback per un utente specifico.

Nel metodo **callBackUnreg** deregistra un utente dalle callback presso il server **RMI**.

- Thread:

  Udp:

  Thread creato e istanziato nella Classe ClientMain, viene avviato mediante il metodo **start** solo dopo la fase di login. Nel metodo **run** crea una socket **UDP** e si unisce a un gruppo **multicast**. Successivamente, all'interno di un ciclo while verifica se l'utente è loggato utilizzando il metodo **getIsLogged** della classe **Login**. Se l'utente è loggato, attende di ricevere un messaggio dal server. La ricezione è bloccante, quindi è impostato un timeout. Se il messaggio non viene ricevuto entro il timeout, viene generata un'eccezione, gestita tramite un blocco **try-catch**. Quando l'utente non è più loggato, il thread esce dal ciclo, si disiscrive dal gruppo multicast e termina.

- Interfacce:

  ServerInteface:

  Questa **Interface** contiene due metodi: **registerForCallback** e **unRegisterForCallback**, entrambi utilizzati per registrare e deregistrare client per le callback. Inoltre, estende l'interfaccia **Remote**. L'implementazione di questa **Interface** si trova nel server.

  Registration:

  Questa **Interface** contiene il metodo **reg**, utilizzato per registrare un Account al servizio; inoltre estende l'interfaccia **Remote**. l'implentazione si trova nel server.

  NotifyEventInterface:

  Questa interfaccia contiene il metodo **NotifyEvent**, utilizzato per notificare il client di un cambiamento nel ranking locale delle città di suo interesse. Quando viene chiamato questo metodo, il server invia al client le informazioni sul cambiamento nel ranking locale, consentendo al client di visualizzare i cambiamenti avvenuti. Inoltre estende l'interfaccia **Remote**.

- Implementazione interfacce:

  NotifyEventImpl:

  Questa classe contiene un singolo metodo invocato dal server tramite RMI. Il metodo **NotifyEvent** crea un'istanza di tipo **HotelRankUpdate**, che viene aggiunta a una lista per tener traccia dei rank modificati. Successivamente, il metodo stampa gli hotel che hanno cambiato posizione, indicando la loro posizione precedente e quella attuale.

- Classe di supporto per i rank Modificati:

  HotelRankUpdate:

  Questa classe presenta unicamente un costruttore, il quale accetta i parametri necessari per salvare le informazioni rilevanti degli hotel. In particolare, le informazioni salvate includono il nome dell'hotel, la città, la posizione precedente nel Rank e la posizione aggiornata.

3.1.1 Strutture Dati

L'unica struttura dati degna di nota è **List<HotelRankUpdate> positionHotel** una lista che viene usata per tener traccia delle informazioni dei rank modificati

4 Servizi di Notifica

All'interno di Hotelier sono presenti due sistemi di notifica, entrambi attivati al momento del login e disattivati dopo il logout. Il primo sistema, effettuato mediante **UDP**, invia messaggi a tutti gli utenti loggati quando l'hotel in prima posizione di un qualsiasi ranking locale cambia. I messaggi vengono ricevuti dal client tramite la classe **Udp**. Il secondo sistema avviene attraverso l'utilizzo della **RMI callback**, dove il client viene notificato per ogni cambiamento di posizione di ogni hotel nelle città di interesse.

5 Sintassi dei comandi

All'avvio del client, se la connessione al server avviene con successo, verrà richiesto all'utente di interagire col servizio. Ogni comando dovrà essere separato dai suoi argomenti mediante uno spazio. Gli argomenti, se più di uno, dovranno essere separati da virgole. È possibile aggiungere più spazi tra un argomento e l'altro, ma è necessario rispettare le maiuscole e le minuscole. Di seguito sono elencati i comandi disponibili insieme alla loro sintassi:

- register username,password

```
register Mario, Rossi
```

- login username,password

```
login Mario, Rossi
```

- logout

```
logout
```

- searchHotel NomeHotel, Città

```
searchHotel Hotel Napoli 1, Napoli
```

- searchAllHotels Città

```
searchAllHotels Napoli
```

- insertReview NomeHotel,Città, Voto Globale, (Voto per ogni singola categoria) Le categorie in questione sono Pulizia, Posizione, Servizio, Qualità in questo ordine. Ciascun voto deve essere compreso tra 0 e 5.

```
insertReview Hotel Napoli 1, Napoli, 5,(4,3,2,1)
```

- showMyBadges

```
showMyBadges
```

6 Compilazione

I seguenti comandi devono essere effettuati posizionandosi all'interno della cartella Client e della cartella Server (il file gson-2.10,1.jar si trova sia nella cartella Client che nella cartella Server).

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

Altrimenti si possono utilizzare i rispettivi jar presenti nelle rispettive cartele digitando i seguendi due comandi:

```
java -jar Client.jar
```

```
java -jar Server.jar
```