RELAZIONE PROGETTO

TOMMASO SOMIGLI
598357
VECCHIO ORDINAMENTO

INDICE

1. Sviluppo e caratteristiche tecniche

La realizzazione del progetto si è basata sul linguaggio di programmazione **Java** e sulla libreria **Gson**, impiegata per la gestione della serializzazione e deserializzazione dei dati relativi agli utenti e agli hotel del sistema Hotelier.
L'architettura del progetto si articola in due directory principali: **Client** e **Server**, che comunicano tra loro principalmente attraverso socket TCP.
Per la lettura degli stream di caratteri ci si avvale della classe **BufferedReader**, mentre la scrittura di tipi di dati primitivi in output viene effettuata tramite la classe **DataOutputStream**.
All'avvio, entrambe le classi leggono un rispettivo file di configurazione (.properties) contenente i parametri di input dell'applicazione, quali numeri di porta, indirizzi, timeout e altri.
Il file di configurazione è strutturato in coppie chiave-valore e in caso di lettura di una chiave non riconosciuta viene generata un'eccezione di tipo **RuntimeException**.

1.1 Ambiente di test

La fase di test del programma è stata condotta su un sistema operativo **macOS Sonoma 14.5**

2. Compilazione ed esecuzione

A. Tramite terminale:

- Posizionarsi all'interno della cartella Hotelier e digitare il seguente comando per la compilazione:
  - javac -cp "lib/*" -d . utility/*.java ClientMain.java ServerMain.java
- Posizionarsi all'interno della cartella Hotelier e digitare i seguenti comandi (in due terminali diversi) per l'esecuzione:
  - java -cp ".:lib/*" ServerMain
  - java -cp ".:lib/*" ClientMain

B. Tramite .jar

- Posizionarsi all'interno della cartella Hotelier e digitare i seguenti comandi (in due terminali diversi) per l'esecuzione:
  - java -jar server.jar
  - java -jar client.jar

3. Comandi

Una volta stabilita la connessione al server, il client permette all'utente di interagire con i servizi offerti. I comandi da impartire seguono una sintassi specifica:

- **Separatori:** Ogni comando deve essere separato dai suoi eventuali argomenti mediante uno spazio. Se presenti, gli argomenti devono essere separati tra loro da virgole.
- **Spaziatura:** È possibile utilizzare più spazi tra un argomento e l'altro, ma è importante rispettare le maiuscole e le minuscole.

3.1 Elenco dei comandi disponibili

Comando	Esempio
register <username>, <password>	register admin, admin
login <username>, <password>	login admin, admin
logout	logout
searchHotel <nome hotel>, <città>	searchHotel Hotel Firenze 1, Firenze

Comando	Esempio
searchAllHotels <città>	searchAllHotels Firenze
insertReview <nome hotel>, <città>, <voto globale>, <pulizia>, <posizione>, <servizio>, <qualità>	insertReview Hotel Firenze 1, Firenze, 5, 4, 5, 3, 4
showMyBadges	showMyBadges

4. Server

Il server è implementato utilizzando le librerie **Java I/O** e un **ThreadPool** per gestire le richieste in modo efficiente e concorrente.
La classe principale **ServerMain** gestisce l'inizializzazione e l'esecuzione del server.
Essa:

- Definisce le variabili per il file di configurazione e gli oggetti remoti.
- Istanzia due strutture dati fondamentali:
  - **allAccount:** un *CopyOnWriteArrayList* di Account che memorizza le informazioni sugli utenti registrati.
  - **allHotel**: un *CopyOnWriteArrayList* di Hotel che memorizza le informazioni sugli hotel gestiti.
- Crea un'istanza di **Jsonservices** e ne invoca i metodi per inizializzare le strutture dati.
- Mette a disposizione degli oggetti remoti metodi per la registrazione e la notifica.
- Gestisce le connessioni **TCP** con i client:
  - Assegna ogni socket a un ThreadPool per il parallelismo.
  - Delega la gestione del client alla classe **ClientHandler**, che estende Thread.

Oltre al thread principale, sono presenti altri due thread:

- **Thread Ranking**:
  - Aggiorna periodicamente la classifica locale.
  - In caso di modifiche, notifica gli utenti interessati tramite:
    - **RMI Callback** per cambiamenti in classifica.
    - **Pacchetto UDP** a un gruppo multicast per cambiamenti solo al primo posto.
  - Il periodo di aggiornamento è definito nel file di configurazione.
- **Thread HotelJsonWriter**:
  - Serializza periodicamente le informazioni sugli hotel.
  - Il periodo di aggiornamento è definito nel file di configurazione.

5. Classi di supporto del Server

Directory Utility che contiene classi di supporto

5.1 Interfacce

- **ServerInterface:** Interfaccia remota che definisce i metodi per la registrazione e la de-registrazione al servizio di notifica.
- **Registration**: Interfaccia remota che definisce i metodi per la registrazione di un utente al servizio Hotelier.

Implementazioni delle interfacce:

- **RegisterRmi**:
  - Implementa l'interfaccia Registration.
  - Estende RemoteServer per il supporto RMI.
  - Contiene un'istanza di Jsonservices.

    - Chiama il metodo JsonAddUser e restituisce il messaggio ottenuto dalla sua invocazione.
- **RmiServerImpl**:
    - Implementa l'interfaccia ServerInterface.
    - Estende RemoteServer per il supporto RMI.
    - Gestisce la registrazione e la deregistrazione dei client per le callback.
    - Notifica i client di cambiamenti nella posizione degli hotel.
    - Fornisce accesso alle preferenze degli utenti.

  Variabili di istanza (RMIServerImpl):
    - *userPreferences*: Mappa che associa una città a una lista di account interessati a quella città.
    - *clients*: Mappa 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 (RMIServerImpl):
  - *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.

5.2 Serializzazione e deserializzazione di oggetti

Il sistema utilizza la serializzazione e la deserializzazione per rappresentare e ricostruire i dati relativi ad account, hotel e recensioni. Questo permette di scambiare informazioni in modo efficiente tra il server e i client.

Classi utilizzate:

- **Account**:
    - Contiene le informazioni relative agli utenti registrati, come username, password, numero di recensioni e città di interesse.
    - Il metodo *setBadge* assegna un badge all'utente in base al numero di recensioni.
    - I metodi *getter* e *setter* permettono di accedere e modificare le informazioni dell'account.
- **HotelServer**:
    - Contiene le informazioni relative agli hotel presenti sulla piattaforma, come nome, città, punteggio e data dell'ultima recensione.
    - I metodi getter e setter permettono di accedere e modificare le informazioni dell'hotel.
- **Review**:
    - Memorizza le informazioni relative a una recensione, come l'utente che l'ha scritta, l'hotel recensito, la città dell'hotel e i voti assegnati.
    - La classe possiede un unico costruttore per la creazione dell'oggetto recensione.
    -
- **LocalDateTimeAdapter**:
    - Questa classe non viene direttamente serializzata o deserializzata.
    - Il suo scopo è quello di personalizzare il comportamento di serializzazione e deserializzazione di oggetti *LocalDateTime* utilizzando la libreria *Gson*.
    - È importante sottolineare che la classe non gestisce la serializzazione diretta di variabili di tipo LocalDateTime ma interviene solo quando è necessario convertire oggetti di quel tipo in formato JSON o viceversa.

5.3 Thread

A. ClientHandler: gestione delle comunicazioni con i client

È un thread che gestisce la comunicazione con un singolo client connesso al server.
Viene creato e inizializzato nella classe **ServerMain** e gestito da un ThreadPool.
Esiste un'istanza di ClientHandler per ogni client connesso.

Il thread comunica con il client tramite due stream:
- Un stream di input *BufferedReader* per ricevere i comandi dal client.
- Uno stream di output *DataOutputStream* per inviare risposte al client.

Per eseguire le operazioni richieste dai client, il ClientHandler utilizza i metodi delle classi **JsonService**, **HotelService** e **AccountService**.
Le istanze di queste classi sono passate al costruttore del ClientHandler.
In caso di disconnessione del client, viene stampato un messaggio di errore.

Gestione dei comandi:
All'interno del metodo run(), un'istruzione switch viene utilizzata 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**: L'utente invia il proprio nome utente e password. Il metodo login della classe **AccountServices** verifica se l'utente è registrato e se la password è corretta. In caso di errore, viene inviato 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 comando permette all'utente di inserire le città di suo interesse. In caso di errore durante l'inserimento, l'utente dovrà ripetere la procedura di login.
  La scelta delle città di interesse può essere effettuata correttamente una sola volta e non è possibile modificarla successivamente.

- **logout**: Se l'utente è già loggato, viene effettuato il logout.

- **searchHotel:** Questo comando è valido per tutti gli utenti. L'utente inserisce 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** contenente tutte le informazioni relative all'hotel.

- **searchAllHotels**: Questo comando è simile al 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. La lista è ordinata in base all'ultimo ranking locale effettuato, dove il primo hotel visualizzato sarà il primo della classifica.
  È importante notare che il thread responsabile dell'aggiornamento delle posizioni degli hotel nel ranking locale potrebbe non aver ancora eseguito l'ultimo aggiornamento. Allora, in questo caso, gli hotel verranno mostrati con i voti aggiornati, ma la posizione in cui verranno visualizzati sarà quella precedente.

- **insertReview**: Questo comando 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**: Questo comando è valido 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*"

B. HotelJsonWriter:

- Questo thread viene creato e inizializzato nella classe **JsonServices** tramite il metodo *HotelJsonWriter.*
- La sua funzione è quella di serializzare periodicamente le informazioni aggiornate della lista degli hotel in un file **JSON** chiamato **"Hotels.json"**.
- Per la serializzazione, vengono utilizzati i metodi della classe Gson.

C. Ranking:

- Questo thread viene creato e inizializzato nella classe **JsonServices** tramite il metodo *JsonHotel*.
- Nel costruttore del thread viene passata una *ConcurrentHashMap* chiamata *mapCity*, che mappa ogni città alla lista degli hotel corrispondenti.
- Il thread, implementato nel metodo *run()*, si occupa di calcolare il ranking locale per ogni città.
- Il 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 ranking.

Gestione della concorrenza e coerenza dei dati

- La concorrenza è gestita attraverso l'utilizzo di strutture dati **thread-safe** come *ConcurrentHashMap* e *CopyOnWriteArrayList.*
- 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 per Ranking

HotelComparator:

Questa classe definisce un unico metodo, compare, che restituisce una lista di hotel ordinati in base a diversi criteri:

1. **Voto globale** (rate): Gli hotel vengono ordinati in base al loro voto medio. In caso di parità, si passa al criterio successivo.
2. **Voti delle singole categorie** (ratings): Se il voto globale è uguale per più hotel, si confrontano i voti per le singole categorie (Pulizia, Posizione, Servizio, Qualità). L'ordine di priorità delle categorie è quello specificato nel codice.
3. **Numero di voti** (nOfVote): Se anche i voti delle singole categorie sono uguali, si considera il numero totale di voti ricevuti da ogni hotel.

4. **Data dell'ultimo voto** (lastVoteDate): In caso di parità anche nel numero di voti, si ordina in base alla data dell'ultimo voto ricevuto da ogni hotel. L'hotel con l'ultimo voto più recente viene posizionato prima.

Questo metodo di ordinamento tiene conto sia della qualità che della quantità delle recensioni. La qualità è rappresentata dai voti medi e per categoria, mentre la quantità è rappresentata dal numero totale di voti e dalla data dell'ultimo voto.

UdpMulticast

Questa classe contiene un unico metodo, *sendMulticast*, che si occupa di inviare un messaggio multicast UDP a tutti gli utenti loggati. Il messaggio notifica il nuovo hotel in prima posizione per ogni ranking locale. In questo modo, gli utenti vengono informati in tempo reale sugli aggiornamenti della classifica.

5.4 Servizi: gestione di file JSON

1. JsonServices

La classe JsonServices gestisce tutte le operazioni di lettura e scrittura da file JSON, in particolare con i file "account.json" e “Hotels.json".

Costruttore: riceve alcune strutture dati condivise:

- **AllAccount**: un *CopyOnWriteArrayList* di oggetti Account che contiene le informazioni di tutti gli account registrati.
- **AllHotels**: un *CopyOnWriteArrayList* di oggetti Hotel che contiene le informazioni di tutti gli hotel presenti nel sistema.

L'istanza di questa classe viene creata nel metodo main e passata poi ai costruttori di tutte le altre classi che ne necessitano.

Metodi:

- *createNewFile*:
  - Controlla se il file "*account.json*" esiste. Se non esiste, lo crea.
  - Si presume che il file "*Hotels.json*" sia già presente.
  - Viene chiamato nel metodo main.
- *JsonHotelWriter*:
  - Crea un'istanza della classe **HotelWriter** e avvia un nuovo thread chiamando il metodo *start()*.
  - Viene chiamato nel metodo main.
- *JsonAccount*:
  - Dopo aver chiamato il metodo *createNewFile()* nel metodo main, viene invocato questo metodo.
  - Se il file "*account.json*" contiene informazioni sugli account, queste vengono deserializzate e inserite nella lista **AllAccount** di oggetti *Account*.
  - Il flag *logged* per ogni account viene impostato su *false*.
- *JsonHotel*:
  - Deserializza le informazioni presenti nel file "*Hotels.json*" e le memorizza nella lista **AllHotels** di oggetti *Hotel*.
  - Per semplificare le operazioni sugli hotel, crea una *ConcurrentHashMap* chiamata *mapCity*.
  - La mappa *mapCity* associa ogni città alla sua lista di hotel corrispondenti.
  - Le operazioni di ranking, inserimento recensioni e ricerca vengono eseguite su questa mappa, garantendo un accesso più efficiente alle risorse, specialmente per la lista degli hotel di una stessa città.
  - Imposta il valore l*astVoteDate* al tempo corrente (indicando che inizialmente questo campo coincide con la data di creazione della lista di hotel).
  - Avvia il thread responsabile del ranking degli hotel.

- *JsonAddUser*:
  - Metodo synchronized che gestisce la registrazione di un nuovo utente.
  - Viene invocato dalla classe **RegisterRmi**.
  - Controlla se lo username dell'utente esiste già.
  - Se non esiste, aggiunge un nuovo account alla lista **AllAccount** e aggiorna il file "*account.json*".
- *UpdateUser*:
  - Metodo synchronized che aggiorna le informazioni sul file "*account.json*".
- *jsonReview*:
  - Inizializza una lista di recensioni.
  - Se il file JSON "*Review.json*" è vuoto, la lista viene inizializzata come un nuovo *ArrayList*.
  - Se il file esiste, il suo contenuto viene deserializzato nella lista.
  - Questa operazione utilizza la classe **LocalDateTimeAdapter** per *LocalDateTime,* che sovrascrive i metodi di serializzazione e deserializzazione di Gson per consentire la corretta deserializzazione degli oggetti *LocalDateTime* presenti nelle recensioni.
- *jsonAddReview*:
  - Serializza la lista di recensioni.
  - Si basa sulla classe di supporto **LocalDateTimeAdapter** per sovrascrivere i metodi di serializzazione utilizzati dalla libreria Gson.
- *Getter*:
  - Esistono metodi *getAllAccount*, *getMapCity* e *getAccount* che restituiscono i rispettivi oggetti.

Funzioni:

- *addCity*:
  - Legge dal file "*listOfCity.txt*" e crea una lista di stringhe contenenti le città.
  - La lista viene utilizzata per creare la *ConcurrentHashMap mapCity*.

2. HotelServices

La classe HotelServices gestisce tutte le operazioni relative agli hotel che non coinvolgono direttamente il file JSON "*Hotels.json*". Questa classe contiene un'istanza della classe **JsonServices** per accedere alle strutture dati condivise.

Costruttore:

- Viene invocato il metodo j*sonReview* della classe **JsonServices** per inizializzare la lista delle recensioni.
- Questa lista è una struttura dati che memorizza tutte le recensioni relative agli hotel.

Metodi:

- *searchHotel*:
  - Metodo synchronized che ricerca un hotel specifico in una città specificata.
  - La ricerca viene eseguita all'interno della *ConcurrentHashMap mapCity*.
  - Se si verifica un errore (ad esempio, la città non esiste), il metodo restituisce *null*.
  - In caso contrario, restituisce l'istanza dell'hotel richiesto.
- *searchAllHotels*:
  - Metodo simile al precedente, ma restituisce la lista di tutti gli hotel in una città specificata.
- *addReview*:
  - Metodo synchronized che gestisce l'aggiunta di una recensione a un 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.
  - Aggiorna le informazioni nell'istanza della classe Hotel ottenuta dalla *ConcurrentHashMap mapCity.*
  - Se l'inserimento della recensione ha successo:

- Crea un'istanza della classe **Review**.
- Aggiunge la recensione alla lista di recensioni dell'hotel.
- Serializza la recensione nel file "*Review.json*" utilizzando il metodo *jsonAddReview* della classe **JsonServices**.

Funzioni:

- *changeIntCategory*:
  - Funzione di supporto che converte l'indice del ciclo attuale nella chiave corrispondente nella *HashMap*.

3. AccountServices

La classe AccountServices gestisce le operazioni relative agli account che non interagiscono direttamente con il file JSON "*account.json*". Contiene un'istanza della classe **JsonServices** per accedere alle strutture dati condivise.

Metodi:

- *login*:
  - Metodo synchronized che verifica l'username, la password e il flag *LoggedIn*.
  - Imposta *LoggedIn* su true se l'utente non è già loggato in un'altra sessione.
  - Questo perché solo un utente alla volta può accedere a un account e non è consentito il login contemporaneo sullo stesso account da client diversi.
  - Restituisce una stringa che indica se l'accesso è avvenuto con successo o se si è verificato un errore.
- *logout*:
  - Metodo synchronized che effettua il logout dell'utente e imposta il flag *LoggedIn* su false.

5.5 Strutture dati utilizzate

Il sistema utilizza diverse strutture dati per memorizzare e gestire le informazioni:

1. CopyOnWriteArrayList:

- *allAccount (Account)*:
  - Memorizza le informazioni di tutti gli account registrati nel servizio.
  - Viene utilizzata anche per la serializzazione e deserializzazione dei dati degli account.
  - Creata nella classe **ServerMain** e inizializzata nella classe **JsonServices**.
  - Accessibile tramite i metodi *get* della classe **JsonServices**.
- *allHotels (Hotel)*:
  - Memorizza le informazioni di tutti gli hotel registrati nel servizio.
  - Viene utilizzata anche per la serializzazione e deserializzazione dei dati degli hotel.
  - Creata nella classe **ServerMain** e inizializzata nella classe **JsonServices**.

2. ConcurrentHashMap:

- *mapCity (String, CopyOnWriteArrayList<Hotel>)*:
  - Mappa concorrente che associa ogni città alla sua lista di hotel corrispondenti.
  - Facilita operazioni come il ranking e la ricerca degli hotel.
  - Creata e inizializzata nella classe **JsonServices** tramite il metodo *JsonHotel*.

3. Mappe per la gestione di callback RMI:

- *userPreferences* (String, List<Account>):
  - Memorizza le preferenze degli utenti per le città.
  - Mappa ogni città a una lista di account interessati a ricevere notifiche su di essa.
  - Inizialmente una mappa semplice, poi castata a ConcurrentHashMap per la concorrenza.
  - Inizializzata nel costruttore della classe RmiServerImpl e aggiornata nei metodi registerForCallback e unregisterForCallback.

- *clients* (Account, NotifyEventInterface):
    - Associa ogni account alla sua interfaccia NotifyEventInterface per le callback RMI.
    - Inizialmente una mappa semplice, poi castata a ConcurrentHashMap per la concorrenza.
    - Inizializzata e aggiornata nei metodi registerForCallback e unregisterForCallback.

4. Altre strutture dati:
- *List<Review>*:
    - Mantiene traccia di tutte le recensioni inserite dagli utenti.

5.6 gestione della concorrenza

La concorrenza all'interno del sistema è gestita in modo sicuro utilizzando diverse strategie:

1. **Metodi synchronized**:
   - La maggior parte dei metodi nelle classi **JsonServices, HotelServices** e **AccountServices** sono synchronized.
   - Questo garantisce che solo un thread alla volta possa accedere ed eseguire un metodo critico, prevenendo conflitti di dati e garantendo la coerenza delle informazioni.

2. Strutture dati concorrenti:
- Le strutture dati condivise da più thread contemporaneamente sono di tipo concorrente, come *ConcurrentHashMap* e *CopyOnWriteArrayList*.
- Queste strutture dati sono progettate per gestire l'accesso simultaneo da parte di più thread in modo efficiente e sicuro, evitando problemi di concorrenza e garantendo l'integrità dei dati.

3. Sincronizzazione di accessi ad attributi:
- I metodi *get* e *set* di alcune variabili nelle classi **Account** e **Hotel** sono anch'essi synchronized.
- Questo garantisce che l'accesso e la modifica degli attributi di un oggetto avvengano in modo atomico, prevenendo accessi concorrenti non sicuri e garantendo la coerenza dello stato dell'oggetto.

6. Client

La classe principale ClientMain gestisce l'interazione tra l'utente e il server.

Funzionamento:
1. **Lettura configurazione:**
    - Legge i parametri di input dal file di configurazione (indirizzo server, porta, ecc.).
2. **Connessione al server:**
    - Stabilisce una connessione TCP con il server utilizzando i parametri letti.
3. **Gestione interazione:**
    - Gestisce l'interazione tra l'utente e il server tramite:
        - Input da tastiera:
            - Legge l'input dell'utente dalla tastiera.
            - Suddivide l'input in un array di stringhe usando il metodo *split*.
            - Separa le parole utilizzando lo spazio per distinguere il comando dagli argomenti.
        - Switch sui comandi:
            - Filtra il comando utilizzando uno switch.
            - Se presenti, gli argomenti vengono ulteriormente suddivisi in stringhe separate da una virgola usando il metodo *split*.
        - Verifica comandi e argomenti:
            - Utilizza il metodo *Check* della classe di supporto **Error** per verificare la correttezza di comandi e argomenti.
        - Controllo login:

        - Verifica se l'utente è già loggato utilizzando il metodo *getIsLogged* della classe **Login**(inizialmente impostato su false).
    - Gestione disconnessione:
        - In caso di disconnessione dal server, stampa un messaggio di errore.

Comunicazione:
- La comunicazione con il server avviene tramite:
    - *BufferedReader*: per lo stream di input (ricezione di dati dal server).
    - *DataOutputStream*: per lo stream di output (invio di dati al server).

Comandi:
1. **Registrazione** (*register*):
    - Verifica prerequisiti:
        - Utente non loggato
        - Nessun errore
        - Numero corretto di argomenti
    - Effettua la registrazione:
        - Chiamata **RMI** alla funzione *rmi*
        - Utilizzo del metodo remoto *reg* per la registrazione
    - Fase di login:
        - Comunicazione tramite stream di input e output
        - Invio di stringa comando/argomenti con *writeBytes*
        - Separazione stringa ricevuta sul server
    - Successo login:
        - Inserimento città di interesse
        - Iscrizione al servizio di notifica **RmiCallback** (*callBackreg*)
        - Avvio thread per ricevere messaggi **UDP** (*start*)
    - Errore login:
        - Logout automatico
        - Ripetizione procedura login
        - Rinserimento città di interesse
    - Registrazione completata:
        - Iscrizione al servizio di notifica
        - Thread in attesa di messaggi UDP

2. **Login:**
    - Fase simile alla registrazione:
        - Esclusa la fase di registrazione
        - Recupero città di interesse già inserite (se presenti)

3. **Logout:**
    - Verifica utente loggato:
        - Invio comando al server
        - Impostazione *IsLogged* a false in caso di successo
    - Join thread UDP:
        - Chiamata *UnregisterCallback* di **RmiCallback**

4. **Ricerca Hotel** (*searchHotel*):
    - Inserimento nome hotel e città:
        - Separazione con virgola
    - Ricerca con successo:
        - Ricezione stringa JSON con informazioni hotel (*readLine*)
        - Deserializzazione JSON in oggetto Hotel (libreria Gson)
        - Stampa informazioni di interesse (*printAll*)
    - Gestione errori:
        - Blocco try-catch per la deserializzazione
        - Lettura messaggio di errore nella sezione catch

5. **Ricerca Tutti Hotel** (*searchAllHotels*):

- Inserimento solo nome città:
- Ricerca con successo:
    - Ricezione lista hotel nella città specificata

6. **Inserimento Recensione** (*insertReview*):
   - Utenti loggati solamente:
   - Inserimento dati recensione:
     - Nome hotel
     - Città
     - Voto globale
     - Voti categorie (Pulizia, Posizione, Servizio, Qualità)

7. **Visualizza Badge** (*showMyBadges*):
   - Utenti loggati solamente:
   - Visualizzazione stringa rappresentante il badge utente

Funzioni:
- *readClientConfig*: Lettura parametri di input dal file di configurazione
- *rmi*: Esecuzione chiamata RMI per interagire con oggetto remoto "register-rmi"
    - Utilizzo del metodo remoto reg per la registrazione

7. Classi di supporto del Client

1. **Error:**
   - Ffornisce un metodo per la verifica dei comandi e degli argomenti inseriti dall'utente.
   - Il metodo *Check* accetta come input il comando e i suoi argomenti (array di stringhe).
   - Utilizza un'istruzione switch per identificare il comando e applicare le opportune verifiche:
     - Presenza di argomenti vuoti
     - Numero corretto di argomenti
     - Validità dei valori numerici
   - In caso di errore:
     - Restituisce true e stampa un messaggio indicante il tipo di errore.
   - In caso di successo:
     - Restituisce false.

2. **Hotel:**
   - Serve per deserializzare una stringa JSON in un oggetto Hotel.
   - Rispetto alla classe **HotelServer** utilizzata nel server, questa presenta un metodo aggiuntivo: *printAll*.
   - Il metodo *printAll* permette di stampare solo le informazioni rilevanti dell'hotel (ad esempio, nome, città, voto).

3. **Login:**
   - Gestisce lo stato di login dell'utente.
   - Contiene due metodi:
     - *getIsLogged*: verifica se l'utente è loggato.
     - *setIsLogged*: imposta lo stato della variabile di login.

4. **rmiCallback:**
   - Gestisce la registrazione e la deregistrazione degli utenti per le notifiche via RMI.
   - Il costruttore:
     - Ottiene il registro RMI tramite *LocateRegistry.getRegistry*.
     - Cerca il server RMI nel registro utilizzando il nome specificato.
     - Assegna il server RMI alla variabile server.
   - Il metodo *callBackReg*:
     - Registra un oggetto remoto di callback per un utente specifico presso il server RMI.
   - Il metodo *callBackUnreg*:
     - Deregistra un utente dalle callback presso il server RMI.

5. **HotelRankUpdate:**
   - Serve per rappresentare un aggiornamento del ranking di un hotel.
   - Contiene un unico costruttore che accetta i seguenti parametri:
     - nomeHotel: Nome dell'hotel
     - citta: Città in cui si trova l'hotel
     - posizionePrecedente: Posizione precedente dell'hotel nel ranking
     - posizioneAggiornata: Nuova posizione dell'hotel nel ranking

7.1 Interfacce

1. **ServerInterface:**
   - Questa interfaccia definisce due metodi:
     - *registerForCallback(String city, NotifyEventInterface callback)*: Permette ai client di registrarsi per ricevere notifiche su una specifica città.
     - *unregisterForCallback(String city, NotifyEventInterface callback)*: Permette ai client di cancellarsi dalle notifiche su una specifica città.
   - Estende l'interfaccia Remote per abilitare la comunicazione RMI.
   - L'implementazione di questa interfaccia si trova nel server.

2. **Registration:**
   - Questa interfaccia definisce un metodo:
     - *reg(Account account)*: Permette la registrazione di un nuovo account al servizio.
   - Estende l'interfaccia Remote per abilitare la comunicazione RMI.
   - L'implementazione di questa interfaccia si trova nel server.

3. **NotifyEventInterface:**
   - Questa interfaccia definisce un metodo:
     - *notifyEvent(HotelRankUpdate update)*: Notifica al client un cambiamento nel ranking locale delle città di suo interesse.
     - Il metodo riceve un oggetto **HotelRankUpdate** che contiene le informazioni sui cambiamenti di ranking.
   - Estende l'interfaccia Remote per abilitare la comunicazione RMI.

   Implementazione:

   NotifyEventImpl:
   - Questa classe implementa l'interfaccia **NotifyEventInterface**.
   - Contiene un metodo:
     - *notifyEvent(HotelRankUpdate update)*:
       - Crea un'istanza di **HotelRankUpdate** e la aggiunge a una lista per tenere traccia degli aggiornamenti.
       - Stampa a console gli hotel che hanno cambiato posizione, indicando la loro posizione precedente e quella attuale.

7.2 Thread

UDP:
- Il thread UDP viene creato e istanziato nella classe ClientMain.
- Viene avviato tramite il metodo start solo dopo la fase di login dell'utente.

Funzionamento:
1. Creazione socket UDP e join al gruppo multicast:
   - Nel metodo *run*, il thread crea una **socket UDP**.
   - Si unisce a un gruppo **multicast** specifico per ricevere messaggi dal server.

2. Ciclo di ricezione messaggi:

- All'interno di un ciclo while, il thread 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 viene impostato un timeout.
  - Se il messaggio non viene ricevuto entro il timeout:
    - Viene generata un'eccezione.
    - L'eccezione viene gestita tramite un blocco try-catch.
  - Se il messaggio viene ricevuto correttamente:
    - Il messaggio viene elaborato e gestito in base al suo contenuto.
- Se l'utente non è più loggato:
  - Il thread esce dal ciclo.

3. Disconnessione e terminazione:
   - Il thread si disiscrive dal gruppo multicast.
   - Il thread termina la sua esecuzione.

7.3 Strutture dati utilizzate

L'unica struttura dati utilizzata in questo contesto è una lista denominata **positionHotel** con le seguenti caratteristiche:

- Tipo: List<HotelRankUpdate>
- Scopo: Memorizzare le informazioni relative agli aggiornamenti del ranking degli hotel.
- Funzionamento:
  - La lista contiene oggetti di tipo **HotelRankUpdate**, ognuno dei quali rappresenta un singolo aggiornamento del ranking.
  - Ogni oggetto **HotelRankUpdate** memorizza le informazioni relative a un hotel.

8. Servizi di notifiche

Hotelier dispone di due sistemi di notifica distinti, entrambi attivi durante il login dell'utente e disattivati al logout.

1. **Notifiche UDP:**
   - Funzionamento:
     - Invia messaggi a tutti gli utenti loggati quando l'hotel in prima posizione di un qualsiasi ranking locale subisce una variazione.
     - I messaggi vengono trasmessi tramite UDP, un protocollo di comunicazione efficiente per la trasmissione di dati in broadcast.
     - Il client riceve i messaggi attraverso la classe **Udp**.
   - Scopo:
     - Informare tempestivamente tutti gli utenti connessi di un cambiamento significativo nel ranking locale degli hotel.
     - Utile per gli utenti che desiderano essere informati su qualsiasi variazione nella classifica degli hotel, indipendentemente dalle loro città di interesse specifiche.

2. **Notifiche RMI Callback:**
   - Funzionamento:
     - Utilizza la tecnologia RMI (Remote Method Invocation) per notificare individualmente ogni utente su ogni cambiamento di posizione di un hotel nelle città di suo interesse.
     - Il client riceve le notifiche tramite un meccanismo di callback, garantendo una comunicazione diretta e personalizzata.
   - Scopo:
     - Fornire agli utenti notifiche mirate e pertinenti alle loro specifiche preferenze.

- Utile per gli utenti che desiderano essere informati solo su variazioni relative agli hotel nelle città che seguono attivamente.