UNIVERSITÀ DI PISA

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

Corso 3° anno - 9 CFU

Relazione progetto Laboratorio III

Professore:
Prof. Laura Ricci

Autore:
Matteo Giuntoni

Anno Accademico 2023/2024

Contents

1 Istruzioni avvio

1.1 Compilazione

La fase di compilazione si divide in compilazione del server e compilazione del client. Può essere svolta da riga di comando eseguendo i normali comandi java scritti qui sotto.

Per la compilazione del Server eseguire il seguente comando:

```
$ javac -cp src:lib/gson.jar -d bin -source 1.8 -target 1.8
            src/ServerApp/*.java
            src/Framework/Database/*.java
            src/Framework/Server/*.java
            src/Framework/Notify/*.java
            src/Data/*.java
```

Questo comando compilerà tutti i file necessari per l'esecuzione del programma Server in formato .class, e li salverà nella cartella ./bin. Oltre a questo va a linkare la libreria gson.

Successivamente bisogna andare a compilare il Client eseguendo il seguente comando:

```
$ javac -cp src:lib/gson.jar -d bin -source 1.8 -target 1.8 -Xlint:-options
            src/ClientApp/*.java
            src/ClientApp/Commands/*.java
            src/Framework/Notify/*.java
            src/Data/*.java
```

Anche questa volta vengono generati tutti i file .class ed inseriti nella cartella ./bin, inoltre viene linkata anche qui la libreria gson, che è situata nella cartella lib.

1.2 Esecuzione

1.2.1 File sorgente

Se si sono compilati i file seguendo le procedure di descritta sopra si avrà una serie di file .class situati all'interno della cartella ./bin, divisi in varie cartelle.

Per eseguire il server usare il seguente comando:

```
$ java -cp bin:lib/gson.jar ServerApp.ServerMain
```

Per far in modo che il server parta correttamente deve essere presente un file .properties chiamato `config_server.properties` nella directory principale dove si esegue il comando di esecuzione.
Il suo interno deve avere il seguente formato:

```
server_port=8080
thread_num=10
data_dir=data
notify_port=8888
notify_address=239.0.0.1
data_save_timeout=5000
notify_timeout=5000
```

Se il file è presente e non ci sono problemi nell'avvio (che nel caso verranno visualizzati con un apposito errore). Durante l'avvio vengono stampate le seguenti scritte sul terminale che indicano l'inizializzazione di alcuni sistemi usati dal server e l'avvio della connessione TCP sulla porta definita (in questo caso la porta 8080).

```
[DATABASE] Inizialization completed.
[ROUTER] Inizialization completed.
[SERVER] Running server on 8080
```

Per il client il comando da eseguire è simile:

```
$ java -cp bin:lib/gson.jar ClientApp.ClientMain
```

Anche in questo caso deve essere presente un file .properties chiamato `config_client.properties` per il corretto avvio del client. il file deve avere il seguente formato.

```
tcp_address=0.0.0.0
tcp_port=8080
notify_port=8888
notify_address=239.0.0.1
notify_timeout=4000
```

Se il file è presente e non ci sono problemi nell'avvio (che nel caso verranno visualizzati con un apposito errore) sul terminale si visualizza la seguente schermata con i comandi possibili, dalla quale si potrà proseguire con l'utilizzo dell'applicazione Hotelier.

1.2.2 File .jar

Per eseguire invece i file .jar già esistenti sarà sufficiente eseguire i seguenti comandi.

```
$ java -jar server.jar
$ java -jar client.jar
```

Questo è possibile grazie alla presenza dei file .md che contengono già le informazioni sulla posizione della libreria gson. Anche in questo caso devono essere presenti i file di configurazioni specificati nella sezione sopra.

Ovviamente, come per l'avvio tramite file .class anche qui dovranno essere presenti i file .properties specificati nella sezione sopra, il `config_server.properties` ed il `config_client.properties` .

2 Server

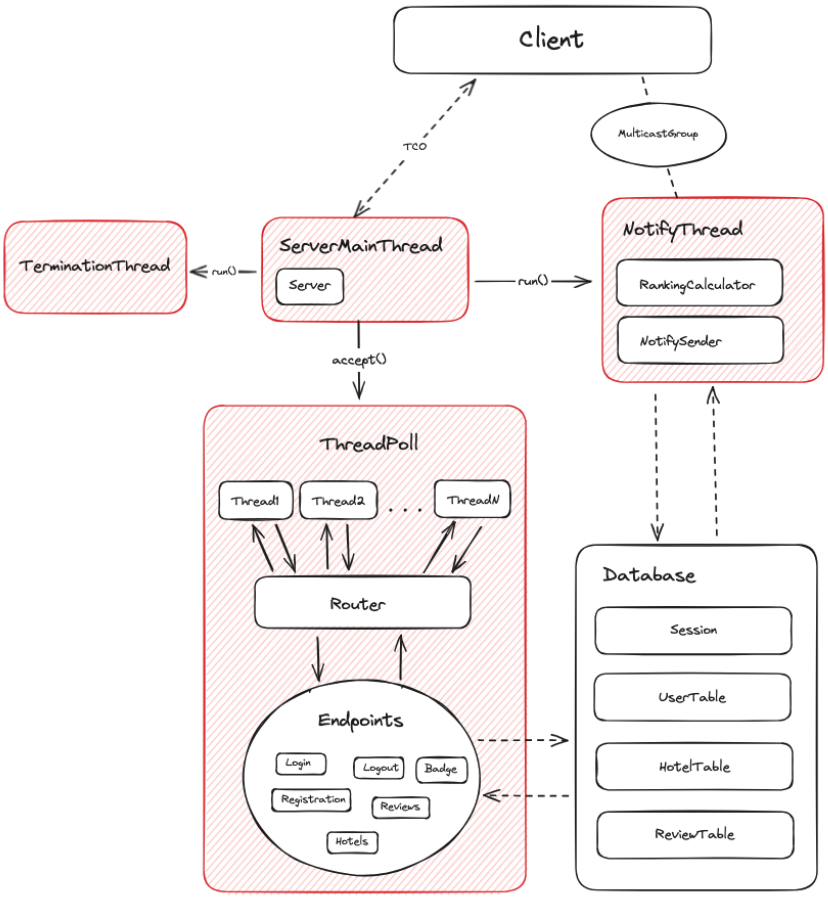

Figure 1: Architettura Server

Il **Server** è realizzato prendendo come esempio l'architettura di una REST API HTTP, implementa infatti una serie di endpoints dove al loro interno viene scritta la logica richiesta, tutto il sistema è statelass, infatti per avere dati persistenti si utilizza le funzionalità offerte dalla classe **Database**.

L'unica cosa che viene mantenuta fra le richieste è la connessione che rimane persistente, essa infatti rimane attiva per lo scambio di messaggi con un singolo Host. Il flusso di lavoro del server è il seguente:

1. **Server**. Il Server è attivo sul Thread principale, esso istanzia un **ThreadPoll** ed avvia una connessione TCP accettando gli host che intendono connettersi. Una volta individuata una connessione crea un oggetto **RequestHandler** e lo assegna ad thread nel **ThreadPoll**.

   Oltre a questo il server avvia due ulteriori threads, il primo si occupa dell'invio delle notifiche di aggiornamento del ranking degli Hotel, il secondo serve per intercettare il segnale SIGINT e salvare i dati sul database prima di spengere il server.

2. **RequestHandler**. Il RequestHandler si occupa di prendere le richieste ed inoltrarle al **Router**, successivamente prendere la risposta del **Router** ed inviarla al client. Le operazioni di di lettura dell'input e di scrittura dell'output avvengono tramite le classi `DataOutputStream` e `DataOutputStream` insieme al supporto delle classi **Request** e **Response** che ereditano a loro volta la classe Message.

   - **Request**. Classe che contiene URL, metodo e body salvato json formattato in una stringa tramite i metodi forniti dalla libreria esterna gson.
   - **Response**. Classe che contiene il codice di risposta ed il body di risposta, anche esso salvato come json formatto in una stringa con gson.

3. **Router**. Il router estrapola dalla richiesta l'URL e il metodo, verifica che esista un endpoint assegnato a quell'url con il metodo definito, e nel caso inoltra la richiesta a questo endpoint; successivamente prenderà la risposta dell'endpoint e la inoltrerà indietro al **RequestHandler** per essere spedita al client.

4. **Endpoint**. Un endpoint prende come input una variabile di tipo **Request** e ha come output un tipo **Response**. Implementa al suo interno un o più metodi fra POST, GET, PATCH e DELETE.
   All'interno del metodo si eseguono le operazioni richieste, per mantenere dati persistenti si può utilizzare le funzionalità offerte dal **Database**.
   Oltre a questo fornisce dei metodi per andare a creare, controllare e eliminare una sessione, dato un certo identificatore, queste funzionalità sono utilizzate per il controllo dell'autenticazione.

5. **Database**. Il Database è una collezione di **Table** che possono essere aggiunte prima dell'inizializzazione del Database. Esso fornisce una serie di operazioni sulle tabelle come insert, delete, select e sort. Il database ha due metodi che devono essere chiamati obbligatoriamente per far si che il suo funzionamento sia corretto.

   - **Inizialize**. Questo metodo va chiamato dopo aver aggiunto le tabella al database e serve per recuperare i dati da eventuali file .json se esistenti. Dopo la chiamata di questo metodo non sarà più possibile aggiungere tabelle.
   - **Shutdown**. Questo metodo va invece chiamato quando si vuole salvare tutte le tabelle nei corrispettivi file .json, nel caso di questa app viene chiamato prima dello spegnimento.

2.1 Thread attivi

Sono attivi 2 threads di default più N threads gestiti dal ThreadPoll (il numero N viene definito come paramentro all'interno del file `config_server.json`). Il ThreadPoll è di tipo `FixedThreadPool` .

- **ServerMainThread**. Thread principale che avvia il socket TCP, riceve le richieste di connessione e le inoltra ai vari Thread del ThreadPool. Oltre a questo inizialmente avvia **NotifyThread** e **TerminationThread**.
- **ThreadPoll**. Un java ThreadPoll della tipologia `FixedThreadPool` che gestisce N threads, alla quale verranno assegnate a ciascuno una eventuale richiesta di connessione. All'interno di ogni thread viene lanciato un RequestHandler che funziona come spiegato sopra.
- **NotifyThread**. Questo Thread viene fatto partire dal ServerThreadMain durante l'avvio. Esso contiene all'interno le funzionalità per il calcolo del ranking degli Hotel. Al suo interno viene inoltre definito un NotifySender che si occupa di mandare un messaggio su un MulticastGroup quando il ranking di una città cambia.
- **TerminationThread**. Non è un vero e proprio thread attivo, in quanto funge da Shutdown Hook, ha lo scopo di entrare in funzione alla chiusura del server, principalmente quando si chiude inviando un signal di tipo SIGINT, una volta attivo va a chiudere in maniera controllata tutte le risorse lasciate aperte a RunTime, andando chiudere NotifyThread e salvando tutti i dati nei file tramite `Database.shutdown()` .

2.2 Primitive di sincronizzazione

La gestione della sincronizzazione e dei dati è delegata interamente al Database, in particolare alla classe **Table** che usa un blocco **syncronized** sul `ArrayList` che contiene i dati della tabella. Questo blocco è inserito nelle operazioni di insert, delete, select e sort per far in modo che solo un thread alla volta possa accedere o modificare la lista.

```
private ArrayList<T> elements;

public boolean insert(T element) {
    synchronized(this.elements) {
        // Operation insert ...
    }
}

public boolean delete(Predicate<T> predicate) {
    synchronized(this.elements) {
        // Operation delete ...
    }
}

public ArrayList<T> select(Predicate<T> predicate) {
    synchronized(this.elements) {
       // Operation select ...
    }
}

public void sort(Comparator<T> compare) {
    synchronized(this.elements) {
        // Operation sort ...
    }
}
```

Oltre a questo abbiamo la tabella degli Hotel che utilizza metodi syncronized per l'aggiornamento dei rate. Questo fa in modo che i campi rate e ratings non vengano letti mentre vengono aggiornati.

```
public class Hotel {
    // Altri attributi che vengono solamente letti quindi non
    // necessitano di sincronizzazione

    private Map<String, Float> ratings;
    private ArrayList<String> services;

    public synchronized float getRate() { return this.rate; }

    public synchronized void setRate(float rate) { /*...*/ }

    public synchronized Map<String, Float> getRatings() { return this.ratings; }

    private synchronized void setRatings(String name, float rate) { /*...*/ }
}
```

2.3 Algoritmo di ranking

L'algoritmo di ranking si basa sull'utilizza della **Media Bayesiana** che permette di assegnare un punteggio che tenga in considerazione sia della media delle recensione ma anche della quantità per ciascuno Hotel.

$$Rank = \frac{(C \times M) + (R \times v)}{M + v}$$

Dove abbiamo che:

- C è la media globale di tutte le recensione.
- M è il numero minimo di recensione per considerare affidabile la media specifica. In questo caso M è uguale al numero di recensioni medio per gli hotel.
- R è la media delle recensioni per un hotel specifico.
- v è il numero di recensioni dell'oggetto specifico.

Oltre a questo però viene aggiunto un calcolo per il rate del singolo hotel (R) ponderato e che tiene in considerazione il passare del tempo, infatti abbiamo che.

$$R = \frac{1}{n \cdot totalWeight} \cdot \sum_{i=1}^{n} \frac{rate_i}{2} + \frac{position_i + cleaning_i + service_i + price_i}{8} \cdot timeWeight_i \tag{1}$$

Il $timeWeight$ è un valore che si ricava andando a prendere il tempo trascorso dalla creazione dell'ultima recensione ad oggi (rappresentato in giorni) ed inserendolo in una funzione del tipo $f(x) = e^{-\lambda x}$ dove $\lambda = 0.1$; questa funzione permette di far diminuire il valore di una recensione con il passare del tempo.

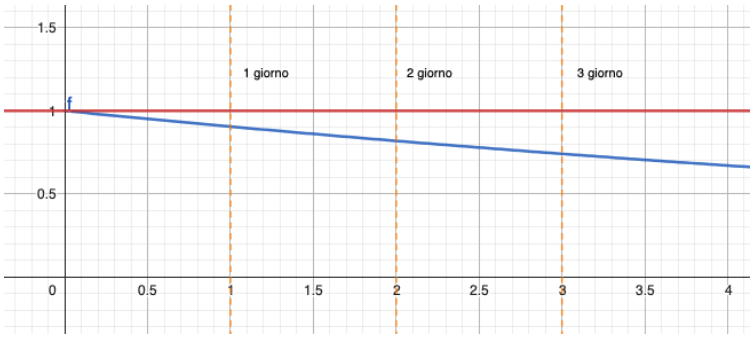

Figure 2: Funzione $e^{-\lambda x}$ con $\lambda = 0.1$

Note. Il paramento λ può essere aggiustato per modificare la caduta della validità della recensione in base al periodo di tempo che si vuole considerare. Il valore $\lambda = 0.1$ è stato scelto per praticità.

3 Client

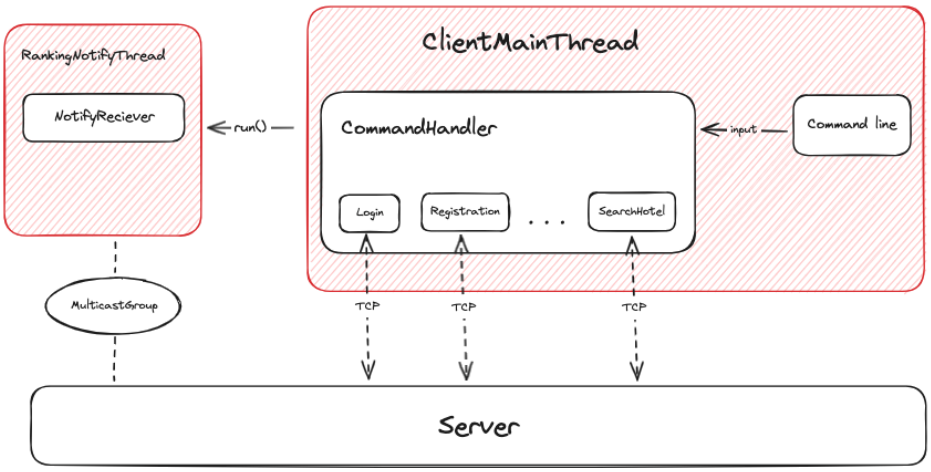

Figure 3: Struttura client

3.1 Thread attivi

All'interno del client sono attivi solamente 2 thread. Che rimangono fissi per tutta il tempo in cui l'applicazione client rimane attiva.

- **ClientMainThread**. Questo è il thread principale che si occupa di avviare la connessione al server, prendere l'input da tastiera e processarlo facendo eseguire il comando richiesto, oppure segnalando un errore.
- **RankingNotifyThread**. Questo thread viene avviato all'avvio dell'applicazione, esso contiene al suo interno un NotifyReciever il quale si occupa di controllare quando ci sono aggiornamenti sul raking degli utenti da parte del server e stamparli a schermo.

  Questo thread è sempre operativo, ma il MulticastSocket effettua `joinGroup()` solo quando ci si autentica con il server, mentre esegue il `leaveGroup()` quando si chiama il logout.

3.2 Comandi

Indichiamo con "NO accesso" quando per eseguire un comando non bisogna essere loggati, con "SI accesso" quando è necessario che si sia loggati mentre con "accesso OPZIONALE" che si può eseguire il comandi sia loggati che non.

- `login` (NO accesso). Effettuare il login con un account già esistente.
  Parametri di **input**:
  - `username` deve essere un username esistente
  - `password` deve essere la password corretta associata allo username

  Possibili messaggi di **output**:
  - **Successo**. Login avvenuto con successo con creazione sessione.
    `[Ok] Login sucessful with Admin`

  - **Errore username**. Username inserito errato.
    `[Error] User doesn't exists`
  - **Errore password**. Password associata allo username sbagliata.
    `[Error] Password isn't correct`
  - **Accout in uso**. Se l'account con cui si sta cercando di fare login è già utilizzato (esiste già una sessione del database associata a questo username) viene comunicato.
    `[Error] Problem session creation, probably someone is just logged with this account`
  - **Errore già loggato**. Se si è già fatto il login bisogna prima effettuare un logout in caso contrario restituisce questo errore.
    `[Error] User alredy logged. Execute 'logout' first to login with new user`

- `registration` (NO accesso). Registrare un nuovo account ed effettuare l'accesso con questo, inserendo un nuovo username univoco, ed una password.
  Parametri di **input**:
  - `username` deve essere un username esistente
  - `password` deve essere una password che rispetta le seguenti direttive:
    * Contenere almeno una lettera maiuscola.
    * Contenere almeno una lettera minuscola.
    * Contenere almeno un numero.
    * Essere lunga almeno 8 caratteri.
    * Contenere almeno un carattere speciale fra: @ # $ % & + =

  Possibili messaggi di **output**:
  - **Successo**. Registrazione avvenuto con successo con creazione sessione.
    `[Ok] Registration was successful with Admin`
  - **Errore username**. Username inserito già presente nel database.
    `[Error] A user with this username already exists`
  - **Errore password**. Password non soddisfa i requisiti richiesti.
    `[Error] Password not valid (it must contains a capital letter, a number, a special character and it must be at least 8 characters long)`
  - **Errore già loggato**. Se si è già fatto il login bisogna prima effettuare un logout, in caso contrario restituisce questo errore.
    `[Error] User alredy logged. Execute 'logout' first to login with new user`

- `logout` (SI accesso). Esegue il logout rimuovendo le informazioni riguardate l'utente con cui si era fatto l'accesso dal client ed impedendo l'accesso ai comandi che richiedevano il login.
  Parametri di **input**:
  - Nessuno

  Possibili messaggi di **output**:
  - **Successo**. Logout avvenuto con successo con cancellazione sessione.
    `[Ok] Logout successful`
  - **Errore**. Non si è loggati.
    `[Error] User isn't logged in`

- `searchHotel` (accesso OPZIONALE). Inserendo città e nome si cerca un determinato Hotel, il risultato sono i dati dell'Hotel in caso esista, o un messaggio d'errore in caso contrario.
  Parametri di **input**:
  - `Città` deve essere una città esistente.
  - `Nome` deve essere un nome di un hotel esistente.

  Possibili messaggi di **output**:
  - **Successo**. Restituisce i dati dell'hotel richiesto.

```
======================================
Name: Hotel Aosta 1
City: Aosta
Description: Un ridente hotel a Aosta, in Via della gioia, 25
Phone: 347-4453634
Services: TV in camera, Palestra, Cancellazione gratuita,
Rate: 1.0
- cleaning: 5.0
- position: 1.0
- services: 5.0
- quality: 0.0

>
```

  - **Errore ricerca**. Se dato nome e città non viene trovato nessun hotel si restituisce un messaggio di errore
    `[Error] Hotel doen't exsits`

- `searchAllHotels` (accesso OPZIONALE). Si cercano tutti gli Hotel di una determinata città, ordinati per ranking.
  Parametri di **input**:
  - `Città` deve essere una città esistente.

  Possibili messaggi di **output**:
  - **Successo**. Restituisce i dati degli hotel presenti nella città richiesta in ordine di ranking locale.

```
======================================
Name: Hotel Aosta 1
City: Aosta
Description: Un ridente hotel a Aosta, in Via della gioia, 25
Phone: 347-4453634
Services: TV in camera, Palestra, Cancellazione gratuita,
Rate: 1.0
- cleaning: 5.0
- position: 1.0
- services: 5.0
- quality: 0.0

>searchAllHotels
city: Aosta

======================================
Name: Hotel Aosta 2
City: Aosta
Description: Un ridente hotel a Aosta, in Via della libertà, 96
Phone: 385-8015757
Services: Cancellazione gratuita, Wi-Fi, Piscina, Paga in struttura, Frigo in camera, TV in camera,
Rate: 2.0
- cleaning: 2.0
- position: 2.0
- services: 2.0
- quality: 0.0


======================================
Name: Hotel Aosta 1
City: Aosta
Description: Un ridente hotel a Aosta, in Via della gioia, 25
Phone: 347-4453634
```

  - **Errore città**. Se non esistono hotel nella città richiesta (perché si può aver sbagliato a selezionare il nome della città) si otterrà un messaggio.
    `[Error] Hotel doen't exist`

- `insertReview` (SI accesso). Si inserisce una recensione andando a seleziona nome e città di un hotel ed inserendo sia il rate globale che i rate individuali dei servizi.
  Parametri di **input**:

  - `Città` deve essere una città esistente.
  - `Nome` deve essere un nome di un hotel esistente.
  - `Rate` indica il rate generale dell'hotel.
  - `Position rate` indica il rate relativo alla posizione dell'hotel.
  - `Cleaning rate` indica il rate relativo alla pulizia dell'hotel.
  - `Service rate` indica il rate relativo al servizio dell'hotel.
  - `Price rate` indica il rate relativo al prezzo dell'hotel.

  Possibili messaggi di **output**:

  - **Successo**. Logout avvenuto con successo con cancellazione sessione.
    `[Ok] Review added`
  - **Errore dati hotel**. Se non esiste nessun hotel nella città inserita con il nome inserita si otterrà questo errore.
    `[Error] Hotel doesn't exits`
  - **Errore valore rate**. Il valore dei rate da inserire deve essere compreso fra 0 e 5 (compresi estremi), in caso contrario si otterrà questo errore.
    `[Error] Values in rates must be between 0 and 5`
  - **Errore non loggato**. Se non si è loggati non si può effettuare recensioni e si otterà questo messaggio.
    `[Error] You must be logged`

- `showBadge` (SI accesso). Mostra il badge più alto ottenuto dall'utente che ha effettuato il login. Il badge è relativo alla quantità di recensione che sono state inserite dall'utente.

  - Nessuno

  Possibili messaggi di **output**:

  - **Successo**. Si ottiene la stringa che identifica il proprio badge fra le seguenti opzioni.
    * `Recensore` si ottiene con 0 recensioni date (base).
    * `Recensore esperto` si ottiene con 1 recensione.
    * `Contributore` si ottiene con 2 recensioni.
    * `Contributore esperto` si ottiene con 3 recensioni.
    * `Contributore Super` si ottiene con 4 o più recensioni.
  - **Errore non loggato**. Se non si è loggati non si può visualizzare nessun dato.
    `[Error] You must be logged`

- `help` (accesso OPZIONALE). Mostra la lista di tutti i comandi disponibili.
  Parametri di **input**:
  - Nessuno

  Possibili messaggi di **output**: L'unica opzione è che viene stampato a schermo una lista con descrizione dei vari comandi che possono essere eseguiti.