Relazione Hotelier

Francesca Pinna

May 17, 2024

Relazione progetto di Laboratorio di reti per l'appello di Maggio 2024 di Francesca Pinna, matricola 601433.

1 Utilizzare Hotelier

1.1 Compilazione

Il client può essere compilato con il comando:

```
javac -d out ClientMain.java
```

Il server può essere compilato con il comando:

```
javac -cp gson-2.10.1.jar -sourcepath . -d out ServerMain.java
```

Assicurarsi che il file jar della libreria gson si trovi nella cartella in cui si esegue la compilazione. La compilazione è testata sulla versione 22.0.1 di Java e potrebbe non funzionare su versioni precedenti.

1.2 Esecuzione

I programmi sono pronti per essere eseguiti con i comandi:

```
java -jar hotelierServer.jar
java -jar hotelier_client.jar
```

Se, dopo aver effettuato la compilazione, si vuole eseguire dai file .class, bisogna assicurarsi che siano nella stessa directory dei file di configurazione `client.properties` e `server.properties`. Si può eseguire il programma con i comandi:

```
java ServerMain
java ClientMain
```

1.3 Testing

Per effettuare il testing automatico di Hotelier, è possibile utilizzare un file di input già scritto.

Per iniziare il testing del client, modificare i parametri di configurazione nel file client.properties, impostando `testing=true` e `testfile=test.txt` o il nome del file di testing desiderato. Il client leggerà le righe del file di testing ad un intervallo specificato dal parametro `seconds`.

Per testare il server, è possibile effettuare il backup in memoria permanente delle strutture dati su un file diverso da quello di lettura, in modo da ricominciare da uno stato iniziale senza recensioni e senza utenti. Per farlo, impostare i parametri `hotelsbackup`, `hotelfile`, `usersbackup`, `usersfile`, sui nomi dei file desiderati su cui scrivere e leggere rispettivamente i dati degli hotel e i dati degli utenti. A questo punto invocare normalmente gli eseguibili come da punto 1.2 Delle versioni "pulite" del file hotels.json e users.json si possono trovare nella cartella `clean_files`.

E' possibile scegliere il livello di dettaglio del logging. Se il logging è spento, stampa solo informazioni sulle connessioni TCP e sul calcolo delle classifiche locali. Altrimenti, è possibile scegliere un livello di dettaglio del logging attraverso il parametro `loglevel` che può andare da SEVERE a FINEST. Il livello SEVERE stampa informazioni solo sugli errori, il livello INFO stampa informazioni sulla maggior parte delle operazioni, e il livello finest stampa informazioni di debug sulla maggior parte delle funzioni di Hotelier.

1.4 Utilizzo

Il server può essere acceso e non richiede ulteriori interazioni. Non si spegne automaticamente, perchè entra in un loop infinito in cui attende connessioni con i client.

Il client, se non è in modalità di testing, legge righe da linea di comando, e le invia al server. Come previsto, le funzioni implementate sono:

- register(username, password) per registrare un nuovo utente.
- login(username, password) per effettuare un login come utente preesistente.
- logout(username) per effettuare il logout dall'account specificato.
- searchHotel(nomeHotel, città) per stampare un singolo hotel.
- searchAllHotels(città) per stampare gli hotel di una città in ordine di ranking.
- insertReview(nomeHotel, nomeCittà, GlobalScore, [ ] singleScores) per inserire una recensione
- showMyBadges() per vedere il proprio badge più recente.

Alcuni esempi di comandi accettabili sono:

```
register(giovanna, password1234)
login(giovanna, password1234)
searchAllHotels(Firenze)
searchHotel(Hotel Firenze 1)
insertReview(Hotel Firenze 2, Firenze, 5, [0, 2, 3, 4])
showMyBadges()
logout(giovanna)
```

Il client termina se riceve il comando exit. Mentre si è collegati, è possibile ricevere notifiche ogni volta che cambia il ranking locale di una qualche città.

1.5 Parametri di configurazione

I parametri di configurazione del client, tenuti nel file `client.properties`, sono:

- **hostname**: nome dell'host su cui risiede il server, default localhost.
- **port**: porta di ascolto del server, default 12000.
- **bufSize**: dimensione dei buffer usati per lo scambio di messaggi client-server
- **exitMessage**: il messaggio di terminazione del client, default exit.
- **testing**: vero o falso, indica se il client deve essere eseguito nella modalità di testing.
- **test**: nome del file di testing, da cui vengono letti comandi da inviare al server.
- **seconds**: tempo che intercorre tra la lettura di una riga del file di testing e un'altra.
- **multicastAddress**: indirizzo su cui vengono inviati i datagrammi tcp in multicasting.

I parametri di configurazione del server, tenuti nel file `server.properties` sono i seguenti:

- **port**: Porta di ascolto
- **bufsize**: dimensione del buffer di lettura e scrittura.
- **exitMessage**: messaggio che indica la chiusura di un canale.
- **logLevel**: livello di verbosità delle stampe relative al funzionamento del server, utili per il debug, vanno da OFF a FINEST.
- **reviewTimeDifference**: tempo in giorni tra una recensione che un utente può lasciare allo stesso hotel.

- **interval**: tempo in secondi tra un aggiornamento dei ranking locali e l'altro. Determina anche la frequenza dei backups in memoria temporanea.
- **hotelFile**: file di lettura della lista degli hotel.
- **hotelbackup**: file di scrittura della lista degli hotel.
- **usersfile**: file di lettura dell'insieme degli utenti.
- **usersbackup**: file di scrittura dell'insieme degli utenti.
- **multicastAddress**: indirizzo su cui vengono inviati i datagrammi tcp in multicasting.

2 Scelte implementative

La mia versione di hotelier è personalizzabile: è possibile scegliere l'intervallo di tempo tra un calcolo classifiche e l'altro e quanto spesso è possibile lasciare recensioni.

2.1 Algoritmo di ranking

L'algoritmo per il calcolo del punteggio dell'hotel è diviso in due fasi. Nella prima fase avviene il calcolo di un punteggio totale che tiene in considerazione le medie delle valutazioni dell'utente, il numero di recensioni e la data dell'ultima recensione. Nella seconda fase avviene l'ordinamento degli hotels sulla base del punteggio totale. Se questo è uguale si usa come discriminante il numero di recensioni, infine si sceglie l'hotel la cui ultima recensione è più recente.

Il punteggio totale ha un valore compreso tra gli zero e i 50 punti, di cui:

- dieci punti dipendono dalla media delle valutazioni generali,
- venti punti dipendono dalle medie delle valutazioni specifiche(pulizia, qualità, servizi, posizione),
- dieci punti dipendono dalla quantità di recensioni,
- dieci punti dipendono dal livello di attualità.

Il punteggio di attualità decresce in modo logaritmico quando aumenta la distanza in giorni dall'ultima recensione. Il punteggio di quantità cresce linearmente fino a quattro recensioni, poi logaritmicamente, in modo che non si possa ottenere dieci avendo meno di trecento recensioni. Questo algoritmo cerca di essere il più possibile semplice e abbastanza veloce da essere chiamato ogni pochi minuti o anche secondi.

2.2 Gestione connessioni

Il server effettua multiplexing dei canali mediante NIO. Si mette in attesa di connessioni dai clienti, e utilizza un selettore per la selezione dei canali. Tutte le operazioni legate alle connessioni sono effettuate sul thread principale in modo sequenziale.

Il server riceve una richiesta sottoforma di bytes, ne effettua la conversione in stringa e il parsing, esegue le operazioni richieste, che siano legate all'account, ricerca di hotel, inserimento di recensioni o visione dei badges, e invia un messaggio di risposta. Poi si mette nuovamente in attesa.

3 Strutture dati

Il lato client è molto minimale dal punto di vista delle strutture dati. Gestisce solo invio e ricezione di messaggi, oltre alla lettura dal file di configurazione.

Il lato server gestisce la lista degli hotel e l'insieme degli utenti, che a loro volta sono composte da oggetti complessi.

3.1 Hotel

Le informazioni sugli hotel sono tenute in una lista sincronizzata di oggetti Hotel ordinata per città, in ordine alfabetico, e poi per ranking locale.

Un oggetto hotel contiene

- informazioni di base come nome, descrizione ecc.,
- la media delle valutazioni generali,
- un oggetto ratings con le medie delle valutazioni singole,
- punti rilevanza (che dipendono dalla data della valutazione più recente),
- punteggio generale per la classifica locale
- la posizione in classifica dell'hotel
- la data dell'ultima recensione.

La classe ratings contiene i punteggi relativi a pulizia, posizione, servizi e rapporto qualità-prezzo. Ha un suo metodo `toString()` che ne permette la stampa formattata.

La classe Hotel ha i propri metodi che permettono di aggiornare i punteggi, calcolare i punti classifica, stampare una rappresentazione, e due metodi di paragone per l'ordinamento, uno per città ed uno per punteggi relativi alla classifica.

3.2 Users

La classe Users rappresenta la lista degli utenti. Ha come attributo una `ConcurrentHashMap`, che mette in relazione gli username agli oggetti User. Permette di accedere alle informazioni di ogni utente a partire dagli username velocemente.

Un oggetto User rappresenta un utente. Contiene username, una password crittata con salt, il numero di recensioni effettuate e una lista di Review. Un oggetto Review rappresenta una recensione, e contiene il nome e la città dell'hotel su cui è stata scritta, il punteggio generale, un oggetto ratings, e la data della recensione.

La classe Users implementa i seguenti metodi che riguardano le operazioni sugli utenti:

- **toJson** che genera una rappresentazione della mappa degli utenti
- **fromJson** che genera la mappa degli utenti da una rappresentazione json
- **register** che inserisce un nuovo utente nella mappa.
- **login** che verifica se la password inserita è corretta.
- **getSalt** che genera un numero casuale necessario per crittare le password.
- **hashPassword** restituisce un hash della password in chiaro.
- **addReview** che aggiunge una recensione alla lista delle recensioni di un utente.
- **getBadge** che calcola un badge sulla base del numero di recensioni scritte da un utente.
- **checkLastReview** che verifica se la data dell'ultima recensione ad un certo hotel è maggiore del periodo che deve intercorrere perchè un utente recensisca lo stesso hotel.

Ho scelto una ConcurrentHashMap per evitare problemi nel momento del backup degli utenti in memoria permanente.

3.3 Session

E' un oggetto che tiene lo stato di una connessione TCP tra client e server. Contiene dati relativi alla trasmissione del messaggio come numero di byte letti, dimensione del messaggio, buffer. Contiene anche il numero del canale (che permette di distringuere i canali nel debugging) e lo stato di login rappresentato tramite username. Se lo username è `null`, allora non è stato effettuato il login. Altrimenti il valore del campo username è il nome utente dell'utente attivo sul canale.

4 Threads e concorrenza

Nel client, esistono due threads. Il thread principale si occupa di ricevere ed inviare messaggi tramite connessione TCP, mentre il thread secondario riceve messaggi in multicasting tramite datagrammi UDP. In questo caso, i due thread non agiscono su strutture dati condivise, quindi non necessitano di sincronizzazione. Le loro attività sono indipendenti.

Anche nel server ci sono due threads. Il thread principale gestisce le connessioni TCP tramite Socket Multiplexing utilizzando un selettore, ed effettua le operazioni legate alle richieste ricevute dal client. Il secondo thread effettua periodicamente il calcolo dei ranking locali e i backup su memoria permanente.

La lista degli hotel è una syncronizedList della libreria Collection, quindi è thread-safe. Anche la ConcurrentHashMap che rappresenta gli utenti è thread-safe, e permette letture concorrenti. Il calcolo dei rankings, che coinvolge l'ordinamento degli hotel e la modifica degli stessi, è contenuto in un blocco syncronized sulla lista degli hotel, siccome itera sul contenuto della lista.

Il thread secondario del server si occupa anche di inviare notifiche quando cambia il ranking locale di una città, che fa tramite multicasting di datagrammi UDP. Questa operazione è indipendente dal comportamento del thread principale.