Nome: Emanuele
Cognome: Mori
Matricola: 638005

Laboratorio di Reti: HOTELIER

Progetto di fine corso A.A. 2023/24

Università di Pisa, Informatica

Emanuele Mori
638005

Nome: Emanuele
Cognome: Mori
Matricola: 638005

Visione generale

Il progetto è diviso in due package: Client e Server, due directory che contengono rispettivamente il codice sorgente relativo al client e al server di HOTELIER, insieme ai file di configurazione, librerie esterne e file in formato JSON.
All'interno di Client si trova:

- HOTELIERCustomerClientMain.java: file contenente il codice sorgente del client;
- Client.properties: file contenente i parametri per configurare il corretto funzionamento del client. I parametri sono:
    - Porta per contattare il server con connessione TCP;
    - Porta per contattare il server con connessione UDP;
    - Indirizzo per contattare il server con connessione UDP.

All'interno di Server si trova:

- HOTELIERServerMain.java: file contenente parte del codice sorgente del server, principalmente riguardo la connessione col client;
- User.java: file contenente parte del codice sorgente del server che si occupa della gestione dell'utente;
- Hotel.java: file contenente parte del codice sorgente del server che si occupa della gestione degli hotel;
- Citta.java: file contenente parte del codice sorgente del server che si occupa della gestione delle città;
- ServerTerminationHandler.java: file contenente il codice sorgente che si occupa della terminazione del server;
- Server.properties: file contenente i parametri per configurare il corretto funzionamento del client. I parametri sono:
    - Porta su cui mettersi in ascolto con la connessione TCP;
    - Tempo di attesa prima della chiusura del pool di thread;
    - Nome del file contenente la descrizione degli hotel;
    - Nome del file contenente le informazioni degli utenti già registrati;
    - Porta su cui mettersi in ascolto con connessione UDP;
    - Indirizzo su cui mettersi in ascolto con connessione UDP.

HOTELIERServerMain

Il server si divide in due parti principali: il main e Worker. Il main si occupa di inizializzare i parametri e creare un ServerSocket; la classe *Worker*, invece, si occupa di gestire la connessione TCP e UDP coi client. Per quanto riguarda la gestione degli hotel e degli utenti, saranno le classi *Hotel*, *Citta* e *User* all'interno dei rispettivi file ad occuparsene.

Main

Inizialmente il server legge i parametri di input dall'apposito file di configurazione testuale.
In seguito, viene inizializzata la socket. Il threadpool, invece, è dichiarato staticamente e ha il nucleo di dimensione pari al numero di core del dispositivo su cui il programma sta girando, e dimensione massima del pool pari a cinque volte tanto.
Successivamente viene definito quali azioni compiere in seguito a un'interruzione che provochi la terminazione del thread. Infatti, ogni volta che viene ricevuta una tale interruzione, è chiamata la classe *ServerTerminationHandler*, la quale si occupa di chiudere la connessione TCP coi client.

Nome: Emanuele
Cognome: Mori
Matricola: 638005

Dopo di ciò vengono recuperate le informazioni riguardo gli hotel e riguardo gli utenti tramite *Hotel.importadajson()* e *User.recupera_utenti()*. Successivamente viene lanciato un thread che si occupa di aggiornare i file JSON contenenti informazioni sugli hotel e sugli utenti ogni minuto. Per far lanciare il thread ho usato il metodo *scheduleWithFixedDelay()* della classe *Executors*. In tal modo il thread verrà attivato ogni minuto, garantendo che i file JSON siano sempre ben aggiornati.
In seguito a ciò, il server si mette in attesa di connessioni, e per ogni connessione con un client fa partire un Worker, che si occuperà di gestire la connessione.

Worker

Il Worker si occupa soltanto della comunicazione fra client e server: si mette in attesa di un intero dal server, il quale specifica il tipo di operazione richiesta. In seguito, accetta i parametri in input necessari e sfrutterà la classe User o Hotel per ottenere la risposta adeguata, che procede immediatamente a spedire come risposta. Ho scelto di fare due file separati per la gestione dell'utente e dell'hotel in modo da rendere più comprensibile il codice. Spesso la risposta è binaria: 1 se l'operazione è andata a buon fine, 0 altrimenti.
Vi sono tre casi particolari casi in cui la risposta non è binaria:

- L'utente vuole inserire una recensione. Il worker chiama *hotel.inserisci_recensione()*, che classe restituisce un array di booleani di dimensione due. Il primo elemento indica se l'inserimento della recensione ha avuto luogo o meno. Il secondo elemento indica se l'inserimento della recensione ha fatto in modo che l'hotel in prima posizione del ranking sia cambiato. In tal caso, sarà aggiornato a riguardo ogni utente attualmente loggato.
- L'utente vuole ricercare i dati di tutti gli hotel di quella città o di un hotel in particolare. In tal caso il metodo *toString()* restituisce una rappresentazione in formato stringa di un hotel o della lista degli hotel di una città. Di conseguenza il server chiama la funzione *sendtoClient()* per spedire la stringa ricevuta al client.
- L'utente vuole fare login. In tal caso il metodo *login()* restituisce un'istanza di *User*.

Per quanto riguarda i datagrammi UDP da spedire agli utenti loggati, ogni qual volta ciò sia necessario è chiamata *inviamessaggioUDP()*. Questa funzione, in primo luogo, crea un oggetto *MulticastSocket* associato a una porta specifica. Successivamente, ottiene l'indirizzo IP del gruppo multicast specificato. Il messaggio da inviare viene convertito in un array di byte. Quindi, crea un pacchetto *DatagramPacket* contenente il messaggio, la lunghezza del messaggio, l'indirizzo IP del gruppo multicast e la porta. Infine, il pacchetto viene inviato tramite il metodo *send()* dell'oggetto *MulticastSocket*, dopodiché viene chiuso il socket per liberare le risorse utilizzate.
Se invece la registrazione desiderata dall'utente è andata a buon fine, viene aggiornato il file JSON contenente le informazioni degli utenti.
Infine, per quanto riguarda la connessione TCP e l'invio di lunghe sequenze di caratteri, *sendtoClient()* risolve il problema di far ricevere una grande stringa al client mandando prima la dimensione della stringa e poi la stringa stessa, linea per linea.

User

La classe *User* si occupa di gestire tutte le operazioni riguardo l'utente: registrazione, login, logout ed esibizione del distintivo.
Ad ogni nuova connessione viene associata un'istanza di *User*. I metodi *register*, *login* e *logout* sono statici. Il metodo *register* aggiunge alla lista *listautenti* un nuovo utente. Il metodo *login* restituisce un'istanza corrispondente al profilo a cui si vuole accedere e *null* se le credenziali non sono nella lista.
Per quanto riguarda i metodi *register* e *login*, ho deciso di non salvare in nessun caso la password. In base a quanto visto nel corso di Crittografia, ho ritenuto opportuno definire una funzione hash one

way trap door *hash_one_way*, e salvare in memoria la coppia del risultato di *hash_one_way* applicata alla concatenazione della password a un seme casuale, e il seme stesso. Per questo ho definito una classe *Couple*, contenente i due valori elencati poc'anzi.
Analizziamo ora come opera la funzione *hash_one_way()*:
per prima cosa crea un'istanza di *MessageDigest* per l'algoritmo SHA-256, responsabile di eseguire l'hashing, poi ottiene i byte corrispondenti alla stringa ricevuta in input. In seguito, applica la funzione SHA-256 all'array di byte ottenuto poc'anzi. Successivamente converte l'array di byte risultante prima in una rappresentazione esadecimale e poi in una stringa, che sarà l'output della funzione.
Infine, ogni volta che l'utente inserisce una recensione verrà chiamato il metodo *inseriscirecensione()*, il quale si occupa di aumentare il livello dell'utente ogni cinque recensioni.
A ogni richiesta di login, viene scorsa tutta lista di *User*. Quando si trova un'occorrenza di *User* con l'username uguale alla stringa ricevuta dal client, allora viene chiamata *hash_one_way(password+seme casuale)* e viene confrontato il risultato con *res*. In caso di uguaglianza l'utente è loggato, altrimenti l'operazione non ha avuto successo. Il seme casuale e *res* sono recuperati da coppia, la password viene spedita in chiaro dal client.
Infine, il metodo *recupera_utenti()* si occupa di recuperare le informazioni degli utenti già loggati dal file JSON. D'altra parte, il metodo *aggiorna_utenti()* ha la funzione di aggiornare il file JSON.
L'elenco degli utenti è salvato all'interno di un *ArrayList*. Ho scelto di non usare una lista o una coda thread safe, perché avrei dovuto fare diverse operazioni con la lista all'interno di svariate funzioni, rendendo la sincronizzazione sulla lista inevitabile e di conseguenza una collezione thread-safe avrebbe rallentato solamente gli accessi.

Hotel

Ogni istanza di Hotel contiene tutte le recensioni che ha ricevuto sottoforma di *List*, oltre naturalmente al suo nome e la città in cui si trova (sottoforma di istanza di *Citta*).
La classe Hotel si occupa di salvare correttamente una recensione, di importare la lista degli hotel da un file JSON, aggiornarlo regolarmente e di fornire una rappresentazione sottoforma di stringa di sé stesso.
Di fondamentale importanza è il metodo *BuildHotel()*. Quando il worker riceve il nome dell'hotel in formato stringa dal client, chiama *BuildHotel()*. Quest'ultimo si occupa di scorrere la lista degli hotel di una determinata città, e restituisce l'istanza corrispondente a cui il client si riferiva. Se invece non è stata trovata nessun'istanza dell'hotel cercato dall'utente, *BuildHotel()* restituisce *null*. Ciò è importante affinché il worker possa comunicare al server l'inesistenza dell'hotel richiesto. Si ha un procedimento identico per la ricerca di una città.
Con gli hotel e le liste di hotel per ogni città vengono effettuati due tipi di operazioni principali:

- Lettura: tramite *toString()*, *hotelincittatoString()* e *aggiorna_hotel()*;
- Scrittura: tramite *importadajson()*, *inserisci_recensione()* e *sorthotelincitta()*.

Dunque, per garantire che più thread possano leggere simultaneamente e un solo thread possa scrivere ho deciso di implementare un monitor. Prima di chiamare uno dei metodi elencati poc'anzi viene sempre acquisito il permesso di leggere o scrivere, e una volta terminato il metodo questo permesso viene rilasciato. I thread si sincronizzano su un'istanza di *Object()* che svolge il ruolo di variabile di condizione. Ho tenuto conto del numero di lettori e scrittori attivi e in attesa. Ho deciso di ricorrere a un monitor perché i metodi synchronized hanno effetto solo sull'istanza di un oggetto, e non su una lista di oggetti. Avrei potuto sincronizzare i thread sulla lista di hotel della rispettiva città su cui si tenta di fare una recensione, ma ho considerato che con un monitor più thread possono effettuare operazioni di lettura simultaneamente e che raramente gli utenti hanno intenzione di inserire recensioni; quindi, inserire una recensione alla volta è un buon compromesso con la possibilità di effettuare varie letture in parallelo.

Nome: Emanuele
Cognome: Mori
Matricola: 638005

Quando viene salvata una recensione, non è salvato solamente il punteggio. Per poi andare ad aggiornare il ranking degli hotel è necessario salvare da chi le recensioni sono state fatte e quando. Perciò ho definito una classe *Nupla*, che contiene al suo interno punteggio, data della recensione e l'utente che l'ha scritta. L'algoritmo di ordinamento, infatti, per prima cosa confronta il punteggio. A parità di punteggio viene confrontata la quantità di recensioni che i due hotel hanno, e a pari quantità viene confrontata l'attualità delle recensioni. I punteggi e le date di registrazione delle recensioni sono ottenuti da metodi di *Hotel* che si occupano di fare la media dei valori salvati nella lista del punteggio complessivo (Global Score). Gli altri punteggi sono ritenuti irrilevanti. Ogni volta che una recensione viene effettuata con successo, viene chiamato *aggiorna_hotel()* per aggiornare il rispettivo file JSON con la nuova recensione.
L'elenco di tutti gli hotel è mantenuto grazie ad una *ArrayList*, e ogni hotel ha una *LinkedList* di punteggi per ogni categoria. Ho scelto di non adottare classi thread-safe, perché avrebbero introdotto un overhead inutile dato che la thread-safety è già garantita dai monitor.
Per quanto riguarda l'importazione dell'elenco degli hotel dal file JSON, la funzione *importadajson()* segue il seguente schema: innanzitutto importa da JSON le variabili che si trovano nell'elenco, e poi scorre di nuovo l'array per aggiungere ad ogni hotel le variabili che non si trovavano nel file, come la rispettiva istanza di *Citta* e le liste di ogni punteggio. Poi scorre le città e in base ai punteggi riordina i ranking di ogni città. Infine, la funzione *aggiorna_hotel()* si limita a scorrere l'array contenente l'elenco di tutti gli hotel e trascrivere ogni istanza nel file JSON.

Città

Ho deciso di chiamare la classe *Citta* e non *Città* perché in alcuni host il nome di una classe in formato non UTF-8 potrebbe dare errore.
Per quanto riguarda la classe *Citta*, quest'ultima si occupa principalmente della gestione del ranking degli hotel. Per questo motivo ha al suo interno la classe *LocalComparator*, che si occupa di confrontare due istanze di *Hotel* secondo i criteri elencati poc'anzi. Il metodo *sorthotelcitta()* chiama il metodo *sort()* di un'istanza appena creata di *LocalComparator()* per ordinare gli hotel di una città. Oltre a ciò, prima dell'ordinamento si salva l'hotel in prima posizione e lo confronta con l'hotel in prima posizione dopo l'ordinamento. Se l'hotel in questione è cambiato, allora restituisce *true*, altrimenti *false*. In tal modo il metodo *inserisci_recensione()* potrà notificare il server che l'hotel in prima posizione è cambiato, affinché gli utenti loggati vengano notificati di tale cambiamento.
Citta contiene anche un metodo *BuildCitta()*, il quale segue lo stesso paradigma enunciato sopra per *BuildHotel()*.
Ogni città è memorizzata in una *ArrayList*. Non vi è bisogno di una classe che sia thread-safe, perché solamente un thread effettua scritture sulla lista di città.
Ogni città contiene al suo interno una *LinkedList* di Hotel. In questo caso non c'è bisogno di ricorrere a strutture dati thread-safe, dal momento che può succedere che più thread operino sulle liste delle città, ma ogni operazione di lettura o scrittura è autorizzata da un monitor e di conseguenza ogni accesso concorrente è corretto.

HOTELIERClientMain

Il client svolge un insieme di funzioni molto ristretto; è il server a occuparsi delle richieste. Il client, dunque, si limita a ricevere da terminale gli input, valutare la loro correttezza, spedirli al server e stampare su terminale l'output ricevuto. Il client è composto da due thread: uno si occupa di gestire la connessione TCP e l'altro si occupa di gestire la connessione UDP.

Main

Nome: Emanuele
Cognome: Mori
Matricola: 638005

Innanzitutto, il client si occupa di configurare porte e indirizzi da client.properties. In seguito a ciò instaura una connessione col server. Successivamente viene fatto in modo che in caso di interruzione le connessioni TCP e UDP col server vengano terminate e anche che il client non dia errore quando il thread principale rimane in attesa di un intero da terminale, e il client prema invio.
Dopo aver configurato il numero di porta da un file specifico, il client instaura una connessione col server. È necessario che anche il client tenga in memoria lo stato dell'utente (se è loggato o no) affinché autorizzi solamente determinate operazioni: sarebbe assurdo che l'utente faccia il logout ancora prima di fare il login. Perciò, ogni volta che l'utente sceglie quale operazione effettuare, è invocato il metodo *check()*, che si occupa di verificare se l'operazione è consentita. Per memorizzare lo stato dell'utente, ho creato la variabile *stato*: quest'ultima vale *true* se l'utente è loggato e *false* altrimenti. All'utente è chiesto di digitare -1 per uscire o un intero compreso fra 1 e 7, corrispondente al tipo di operazione che vuole effettuare. Di conseguenza, in base all'intero ricevuto vengono richiesti i parametri appropriati e poi viene chiamata la funzione appropriata. Quest'ultima si occupa di spedire i dati al server, ricevere l'output ed eventualmente decodificarlo.
In caso l'utente voglia terminare la connessione, può semplicemente digitare -1. In tal caso, -1 è spedito al server e quest'ultimo chiude la connessione. D'altra parte, nel caso in cui l'utente digiti ^C o vi sia un errore, è invocato l'handler del segnale specificato inizialmente, il quale si occupa di chiudere la connessione.
Nel caso in cui il login abbia avuto successo, allora è invocato il metodo *RicevitoreUDP.start()*, che attiva il thread che si occupa di gestire la connessione UDP col server. È necessario mandare un'interruzione a *RicevitoreUDP* per farlo terminale quando il logout ha avuto successo, perché se fosse il server a farlo terminare allora farebbe terminare tutti i *RicevitoreUDP* in contatto col server. Per quanto riguarda il thread *RicevitoreUDP*, poiché la *receive()* è bloccante, senza nessun meccanismo particolare il processo del client rimarrebbe attivo anche quando l'utente ha digitato -1 per terminare la sessione. Per questa ragione ho deciso di impostare un timer di 1 secondo per la *receive()*, in modo che ogni secondo verifichi se l'utente ha deciso di terminare la sessione.
Per quanto riguarda la connessione TCP, ho creato una funzione per ogni operazione possibile. Ogni funzione inizialmente spedisce un intero che determina il tipo di operazione che il client intende effettuare, e poi i rispettivi parametri. Se la risposta è booleana si mette in attesa di 0 o 1. Se viene ricevuto 1 stampa su schermo che l'operazione richiesta è stata effettuata con successo, altrimenti avverte l'utente che vi è stato un errore. Se la risposta invece è testuale, dopo aver specificato il tipo di operazione e spedito i parametri opportuni, è invocato il metodo *leggihotel()*. Quest'ultimo prima riceve un intero che specifica quanti caratteri il server sta per spedire, e poi la stringa stessa. Ho pensato che tale meccanismo fosse indispensabile in quanto una lettura singola non è in grado di ricevere la descrizione di tutti gli hotel di una città, e mettersi in attesa con la *read()* avrebbe bloccato il client nel momento in cui il server avesse esaurito i caratteri da spedire.
Per quanto riguarda l'inserimento delle recensioni, ho definito due funzioni: *insertReview()* e *tryInsertReview()*. Ho adottato questa soluzione per fare in modo che vi siano due interazioni col server: una per verificare se l'hotel da recensire esista davvero ed una per eventualmente salvare la recensione. *tryInsertReview()*, infatti, prima spedisce un intero al server per fare capire che l'utente vuole inserire una recensione, e successivamente il nome della città e dell'hotel. Il server verifica se tale hotel esiste davvero e notifica il client. In caso affermativo, viene invocata *insertReview()* che riceve i punti da terminale e li spedisce al server.

Manuale

Per compilare il Server di HOTELIER è sufficiente navigare le cartelle del progetto, entrare nella cartella Server e da terminale digitare il seguente comando:

Nome: Emanuele
Cognome: Mori
Matricola: 638005

javac -cp gson-2.10.jar:. HOTELIERServerMain.java User.java Hotel.java ServerTerminationHandler.java Citta.java

Per mandare in esecuzione il server digitare:

java -cp gson-2.10.jar:. HOTELIERServerMain.java User.java Hotel.java ServerTerminationHandler.java Citta.java

Per compilare il Client di HOTELIER è sufficiente navigare le cartelle del progetto, entrare nella cartella Client e da terminale digitare il seguente comando:

javac HOTELIERCustomerClientMain.java

Per eseguire il programma client digitare:

java HOTELIERCustomerClientMain.java

Da lato client, per interagire col sistema è necessario seguite le istruzioni ricevute da terminale.