Università di Pisa

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

CROSS: an exChange oRder bOokS Service

Laboratorio III

Professore:
Prof. Laura Ricci

Autore:
Birindelli Leonardo

Anno Accademico 2024/2025

1 Spiegazione scelte adottate a livello di codice

1.1 Client

Nella realizzazione dell'interfaccia del client ho pensato di creare due distinte interfacce interattive : la prima è per interfacciare l'utente che non ha ancora provveduto ad accedere al proprio profilo, essa mette a disposizione tra cui la registrazione, la richiesta di accesso nel servizio, il cambio delle credenziali (ovviamente se il profilo è presente nella base di dati lato server) e per concludere la possibilità di terminare l'applicazione client.

La seconda interfaccia è quella che si presenta all'utente solo dopo aver effettuato l'autenticazione di un profilo registrato, in essa si da la possibilità all'utente di effettuare operazioni di trading permettendo l'inserimento di ordini di acquisto e vendita (tra cui gli ordini a mercato, ordini limite e ordini di stop), la cancellazione di uno specifico ordine non ancora evaso e la possibilità di richiedere l'elenco dei dati storici degli ordini effettuati in un dato mese e anno. Inoltre l'utente ha la possibilità di cambiare il profilo (tornando così alla schermata iniziale) e di chiudere l'applicazione client.

La decisione di separare le due interfacce è basata sulla necessità di mostrare una esplicita separazione tra gli utenti autenticati e quelli non garantendo anche una maggiore sicurezza nel caso in cui un utente non autorizzato provi ad accedere al servizio. In entrambe le implementazione ho deciso di creare un interfaccia basata sull'inserimento di numeri corrispondenti alle operazioni che compaiono a terminale dove l'utente inserisce a tastiera l'azione che vuole effettuare e il sistema la svolge, tale approccio fornisce una interfaccia basilare e guidata all'utente permettendo così da ridurre al minimo i possibili errori di input che l'utente può inserire anche erroneamente.

Perlopiù il client gestisce lo spegnimento dell'applicativo permettendo così chiudere opportunamente il socket di comunicazione con il server al ricevimento un interruzione utente da linea di comando (`crtl+c`).

Inoltre l'applicativo client è munito di un sistema di gestione degli errori che permette di gestire eventuali errori di comunicazione con il server e di input errati da parte dell'utente: sia gli errori che le risposte con successo inviate dal server vengono stampate a schermo per notificare lo stato dell'operazione effettuata. In conclusione, il client implementa la ricezione asincrona di notifiche attraverso un thread indipendente che si occupa di ricevere le notifiche di evasione degli ordini dell'utente autenticato e di stamparle a terminale.

1.2 Server

Il server è progettato in modo tale che una volta caricati i parametri di configurazione dal file "`server.properties`" e le caricate le opportune strutture dati (la lista degli utenti registrati e l'order book) con le informazioni presenti nei vari file in formato JSON presenti nella cartella `data`, il server si mette in ascolto su una porta TCP per la ricezione delle richieste dei client. Al momento della ricezione di una richiesta, il server crea un thread indipendente per gestire la richiesta del client. Questo thread si occupa di adempire all'operazione richiesta del client e di inviare la risposta corrispondente.

Il server è inoltre dotato di un meccanismo di timeout per la gestione delle sessioni utente, infatti se un client con utente autenticato non interagisce con il server per un tempo superiore a quello impostato nel file di configurazione, il server chiude termina la sessione utente e il client deve riautenticarsi per poter effettuare nuove operazioni. Perlopiù, il server instaura una procedura di un meccanismo di notifica best-efford per gli ordini evasi di un utente autenticato.

Nel server vengono utilizzati diversi tipi di oggetti ognuno dedicato ad una specifica funzionalità in modo tale da separare le responsabilità e rendere il codice più leggibile e manutenibile. Ad esempio ho deciso di creare un oggetto `SessionManager` per gestire le sessioni utente, un oggetto `OrderBook` per gestire l'ordine book con le varie strutture dati, un oggetto `UDPNotifier` per gestire l'invio delle notifiche di evasione degli ordini. Ho anche realizzato una gerarchia di classi, la cui classe madre `Order`, per gestire i vari tipi di ordini che possono essere inseriti nell'ordine book (ad esempio `MarketOrder`,

`LimitOrder`, `StopOrder`). In fine, ho deciso di utilizzare un scheduled executor per procedere alla persistenza dei dati in modo periodico per le stesse motivazioni della thread pool del client così da sfruttare le peculiarità della struttura come la gestione interna delle eccezzioni a livello del thread, ma anche per rendere il tutto più leggibile.

1.3 Schema delle richieste client-server

Ho avuto la necessità di aggiungere ulteriori campi alle richieste JSON inviate sia dal client che dal server per poter salvare informazioni aggiuntive che mi permettessero di gestire al meglio le richieste e le risposte (come ad esempio l'identificativo dell'utente, il tempo massimo della sessione per un utente ...), nel codice commentato ho riportato per ognuna delle operazioni le varie richieste JSON client-server.

2 Schema thread attivati

2.1 Client

I thread attivati dal client sono due:

- il primo thread è il thread principale che si occupa di gestire l'interfaccia utente, di inviare e stampare a schermo le risposte del server;
- il secondo thread è un thread indipendente gestito all'interno di una thread pool contente un solo thread che si occupa di ricevere le notifiche di evasione degli ordini dell'utente autenticato.
- il terzo thread è un thread indipendente e viene attivato solo al momento dell'interruzione dell'utente da linea di comando (`ctrl+c`) e si occupa di chiudere il socket TCP di comunicazione con il server e quello UDP per la ricezione delle notifiche di evasione degli ordini.

2.2 Server

Innanzitutto, il server fa uso di una thread pool di un numero variabile di thread che viene determinato al tempo di esecuzione in base al numero di core della macchina su cui viene eseguito il server così da permettere di poter accettare e gestire più connessione possibili. Questa thread pool viene utilizzata per gestire le richieste dei client. Ogni volta che il server riceve una richiesta da un client, attiva un thread indipendente per gestire la richiesta. Inoltre abbiamo attivi nel server:

- il thread principale che si occupa di accettare le connessioni dei client e di richiedere alla thread pool di gestire la richiesta del client;
- un thread indipendente per gestire le sessioni utente gestito da uno scheduled executor che si occupa di controllare lo stato delle sessioni utente e di chiudere le sessioni utente che sono scadute;
- un thread indipendente per la persistenza dei dati gestito da uno scheduled executor che si occupa di salvare i dati in modo periodico gestito da uno scheduled executor;
- un thread indipendente per la chiusura del server quando avviene una interruzione utente da linea di comando (`ctrl+c`) che si occupa di chiudere i socket di comunicazione con i client, le thread pool attive e di terminare il server.

3 Strutture Dati

3.1 Client

Lato client non sono state utilizzate strutture dati particolari in quanto la comunicazione con il server avviene tramite l'invio di stringhe contenenti le operazioni da effettuare e i dati necessari per effettuare tali operazioni. L'uniche informazioni che faccio memorizzare al client sono quelle legate alla sessione utente tra cui il nome dell'utente attualmente autenticato "`usernameLoggedIn`", il tempo massimo di sessione utente imposta nella configurazione del server "`maxLoginTime`" (passato al momento

dell'autenticazione dell'utente con il server) e il timestamp di stato della sessione utente memorizzata lato server "`userSessionTimestamp`" (la quale viene continuamente aggiornata ogni volta che il client interagisce con il server).

3.2 Server

Le strutture dati concorrenti che ho utilizzato sono le seguenti:

- `ConcurrentHashMap` per la gestione degli utenti registrati, della sessione utenti e delle notifiche di evasione degli ordini così da garantire la concorrenza tra i thread che gestiscono le richieste dei client;
- `ConcurrentSkipListMap` per la gestione dell'ordine book così da garantire la concorrenza e permettendomi di avere una struttura dati ordinata con a disposizione i metodi di ricerca e cancellazione in tempo logaritmico.
- `ConcurrentLinkedQueue` per la memorizzazione degli ordini nei diversi registri.

4 Primitive di Sincronizzazione

4.1 Client

Non utilizzo primitive di sincronizzazione dato che il client non ha bisogno di gestire concorrenza tra thread visto che ognuno di essi ha un compito ben definito e non interferisce con gli altri thread attivati dal client.

4.2 Server

Ho utilizzato il sistema delle lock implicite in java tramite 'synchronized' nei metodi in cui invoco funzioni che non garantiscono la concorrenza (come ad esempio 'putAll' della 'ConcurrentHashMap') così da assicurare l'accesso e modifica sicura alle strutture dati. Altre primitive di sincronizzazione sono già garantite dai tipi di strutture dati utilizzate. Tuttavia, ho aggiunto blocchi 'synchronized' dove le operazioni sulle strutture dati non erano completamente garantite.

5 Istruzioni di Compilazione ed Esecuzione

Il progetto è stato sviluppato in Java 11 e per la compilazione e l'esecuzione del progetto è necessario avere installato Java 11 o versioni successive e la versione utilizzata della libreria Gson per la gestione dei file JSON è la 2.11.0

5.1 Compilazione

Per compilare il progetto è necessario eseguire i seguenti comandi quando si è locati nella cartella root del progetto:

```
javac -d bin -cp "lib/*" $(find src/main/java -name "*.java")
```

Per l'esecuzione:

```
java -cp "bin;lib/*" com.crossserver.CrossServerMain
```

In presenza del file JAR, per eseguire bisogna :

5.2 Esecuzione del Client

```
java -jar crossclient.jar
```

Parametri: Configurazioni lette da `client.properties`.

5.3 Esecuzione del Server

```
java -jar crossserver.jar
```

Parametri: Configurazioni lette da `server.properties`.
Mentre i dati degli utenti e dell'ordine book sono caricati dai file JSON presenti nella cartella `data`.