Tesi progetto laboratorio 3

Università di Pisa

Leonardo Sinceri

2024/2025

Contents

1 Introduzione

Il progetto *CROSS* (exChange oRder bOokS Service) ha l'obiettivo di sviluppare un sistema per la gestione degli order book in un ambiente di trading centralizzato di criptovalute. Gli order book sono fondamentali nei mercati finanziari, poiché permettono di visualizzare le transazioni di acquisto e vendita in corso, gestendo gli ordini di acquisto (bid) e di vendita (ask) di un determinato asset. Il progetto si concentra sulla creazione di un servizio che simula il funzionamento di una piattaforma di exchange per criptovalute, ma per semplicità, in questo progetto si gestiscono solo ordini di acquisto e vendita di Bitcoin (BTC) contro il dollaro statunitense (USD).

Gli utenti possono interagire con il sistema per inserire, modificare e cancellare ordini, monitorando le transazioni in tempo reale grazie alle notifiche. Il servizio è strutturato su un'architettura client-server, dove il client gestisce l'interazione con l'utente attraverso un'interfaccia a riga di comando (CLI), mentre il server è responsabile della gestione dell'order book, dell'esecuzione degli ordini market, limit e stop, nonché della persistenza delle informazioni. Il sistema si avvale di tecniche di programmazione concorrente, utilizzando il pooling dei thread per garantire la gestione di più richieste simultanee da parte degli utenti.

2 Struttura del sistema

Nel progetto CROSS, il sistema è organizzato in tre moduli principali:

- Modulo Client
- Modulo Server
- Modulo Shared (condiviso tra client e server)

Ogni modulo è progettato per separare la logica e semplificare la gestione del sistema. Alcuni componenti sono presenti sia nel Client che nel Server, avendo però parametri diversi.

2.1 ClientSettings e ServerSettings

Le classi `ClientSettings` e `ServerSettings` gestiscono le configurazioni condivise tra client e server, con funzionalità specifiche per ciascun modulo. Entrambe centralizzano la configurazione per ridurre la duplicazione e facilitare l'aggiornamento delle impostazioni. I dati sono caricati e salvati nei file `client.properties` e `server.properties`.

2.2 GlobalData

La classe GlobalData è fondamentale per i moduli client e server, raccogliendo variabili e funzioni statiche accessibili da tutte le altre componenti. Fornisce una visione d'insieme dell'applicazione, semplificando l'accesso e l'aggiornamento delle informazioni globali.

2.3 Modulo Client

Il modulo Client gestisce la comunicazione con il server e l'interazione con l'utente. È composto da:

Main Punto di ingresso che inizializza le componenti, gestisce le connessioni e le risorse, e chiude correttamente il client.

RequestHandler Gestisce l'invio delle richieste al server e la ricezione delle risposte.

2.4 Modulo Server

Il modulo Server gestisce le richieste dai client, elaborandole e restituendo risposte. La struttura include:

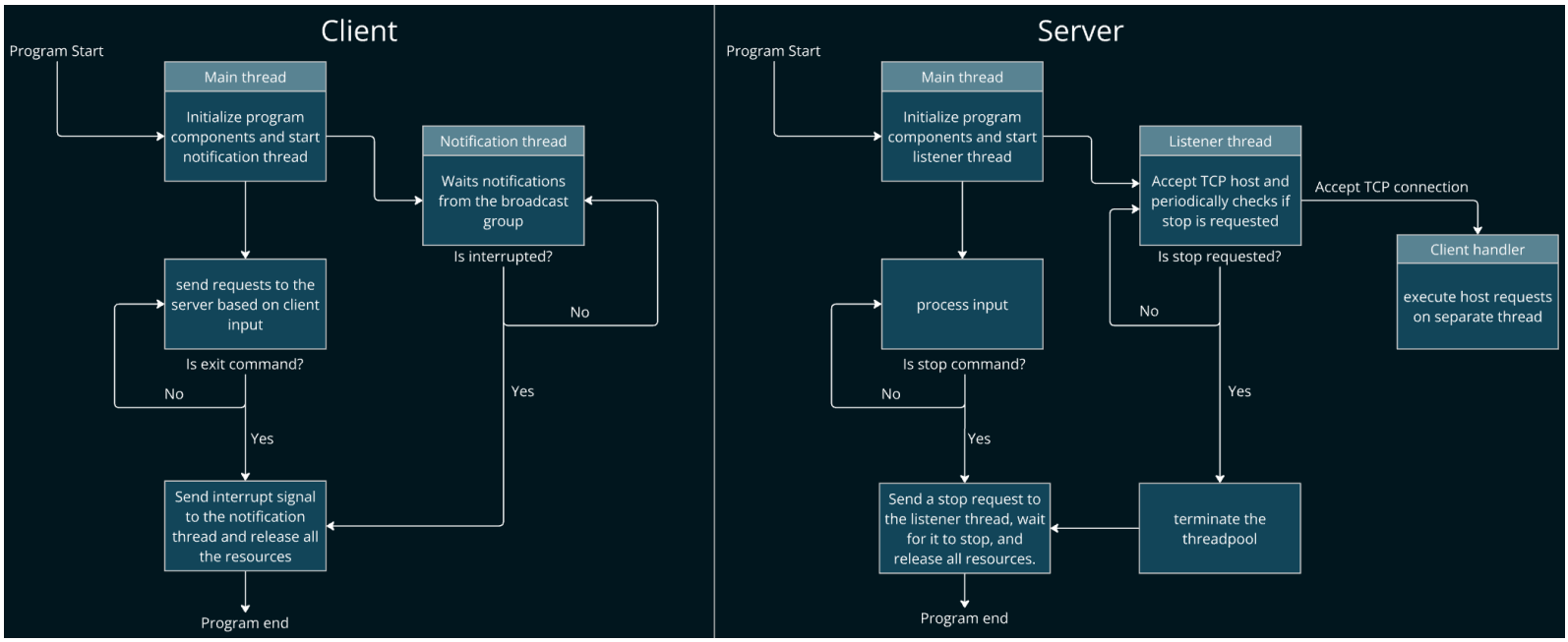
Main Avvia il sistema, carica i dati e gestisce la chiusura del server.

Listener Accetta le connessioni dai client e crea un ClientHandler per ciascun client.

ClientHandler Gestisce le richieste di un singolo client in un thread separato.

3 Implementazione

3.1 Flow generale del Client e Server



Molte delle strutture e delle logiche di implementazione, in particolare per quanto riguarda le request e le response, sono state direttamente ispirate dalle specifiche fornite nel testo d'esame.

3.2 Request e Response

La classe `Request` e le sue sottoclassi sono state costruite seguendo il modello descritto nel testo, in cui ogni richiesta è costituita da due principali componenti:

operation Un tipo enume che rappresenta l'operazione che il sistema deve eseguire (ad esempio: registrazione, login, ordini, ecc.).

values un oggetto che contiene i dati specifici per l'operazione, che vengono serializzati nel formato JSON. Le sottoclassi di `Request` implementano la logica per serializzare e deserializzare i dati specifici in questo campo.

Analogamente, le classi di `Response` sono state progettate per gestire la struttura comune delle risposte nel sistema, che vengono inviate dal server al client. La classe `Response` fornisce una struttura di base per la serializzazione e deserializzazione delle risposte in formato JSON. Due classi principali che si specializzano da essa sono `SimpleResponse` e `OrderResponse`, ciascuna con un comportamento specifico. La prima è una risposta che rappresenta un codice numerico e un messaggio di errore e la seconda è una risposta specifica

che rappresenta la conferma di un ordine, restituendo un identificatore dell'ordine (o -1 se non è andato a buon fine).

La risposta alla richiesta `getPriceHistory` è un lista di oggetti, ciascuno contenente le proprietà relative al giorno e ai prezzi di apertura, chiusura, minimo e massimo. Il messaggio incapsula la lista in formato json.

3.3 Price history collection

La gestione della storico dei prezzi (price history) è stata implementata utilizzando una struttura dati complessa che sfrutta una combinazione di `TreeSet` e `Tuple` per organizzare i dati in modo efficiente e per permettere la ricerca ottimizzata (`TreeSet<Tuple<Long, TreeSet<Tuple<Long, List<HistoryRecord>>>>>`). La struttura dati è composta da più livelli, e ogni livello è progettato per riflettere una gerarchia temporale, dal più generale al più dettagliato. La struttura complessiva appare come segue:

```
[
    2024/09 : [
        2024/09/01 : [
            { price record 1 },
            { price record 2 },
            ...
        ],
        2024/09/02 : [
            { price record 1 },
            { price record 2 },
            ...
        ],
    ...
]
```

La struttura utilizza dei `TreeSet` per mantenere i dati ordinati in modo naturale in base alla data (anno, mese, giorno). La struttura permette di eseguire operazioni come l'inserimento e la ricerca in modo efficiente grazie all'ordinamento e alla ricerca binaria garantiti dei `TreeSet`. Inoltre permette di salvare e caricare i record in modo ordinato nel file senza dover ordinare la struttura ad ogni salvataggio (i salvataggi sono periodici).

4 Sincronizzazione

Per gestire lo scenario in cui più thread potrebbero cercare di accedere o modificare le collezioni dati, vengono utilizzati i blocchi di sincronizzazione su tutta la collezione per ogni operazione che legge o modifica i dati. L'uso della parola chiave `synchronized` assicura che tutte queste operazioni siano eseguite in modo sicuro in un ambiente multi-thread, evitando race conditions che potrebbero compromettere l'integrità dei dati, facendo si che solo un solo thread alla volta possa leggere o modificare i dati contemporaneamente.

History record collection Per le operazioni di aggiunta, lettura e salvataggio, è stato utilizzato il blocco `synchronized` sull'intera collezione. Se la lettura/scrittura della history diventasse un bottlenck c'è la possibilità di sincronizzare solo il secondo livello della struttura, quindi sincronizzare su ogni mese diverso, invece dell'intera collezione.

User e User collection Nello `User` le funzioni `MatchPassword` e `TryUpdatePassword` vanno a leggere e/o scrivere la risorsa condivisa password. Il blocco di sincronizzazione è applicato sull'oggetto `User` stesso. La `UserCollection` contiene le collezioni `_registered` (per gli utenti registrati) e `_connected` (per gli utenti connessi) e sono implementate come `ConcurrentHashMap`, che permette operazioni sicure in ambienti multi-threaded. Queste mappe non richiedono una sincronizzazione esplicita per operazioni come `putIfAbsent` o `remove`, poiché `ConcurrentHashMap` gestisce la concorrenza internamente. Solo nell'aggiunta e nel salvataggio degli utenti registrati la collezione è sincronizzata dentro un blocco.

Client orders Per monitorare gli ordini dell'utente connesso da quell'host, gli ID vengono salvati (quando un ordine viene inserito correttamente), rimossi (quando un ordine viene annullato correttamente) o letti (dal thread delle notifiche) da un `HashSet`, sempre all'interno di un blocco sincronizzato sulla collezione.

Listener Vengono utilizzati due AtomicBoolean per monitorare lo stato del listener (isRunning) e gestire la sua terminazione (isStopRequested). Le variabili di tipo Atomic (in questo caso AtomicBoolean) consentono di utilizzare operazioni thread-safe senza la necessità di sincronizzazione esplicita, in quanto sono già sincronizzate internamente.

Order book Il sistema è composto da quattro `PriorityQueue`: due per gli ordini limit (uno per gli ask e uno per i bid) e due per gli ordini stop (anch'essi separati in ask e bid). I market order, infatti, vengono eseguiti immediatamente e non necessitano di essere memorizzati per un'esecuzione futura. Le due collezioni per gli ordini limit e per quelli stop separano i tipi ask da quelli bid. Gli ordini ask vengono ordinati in ordine crescente di prezzo, mentre gli ordini bid in ordine decrescente. In caso di parità di prezzo, gli ordini vengono ordinati in base al timestamp e, infine, all'ID (univoco per ogni ordine).

Conclusioni Usare un database invece delle collezioni nel programma offre numerosi vantaggi, tra cui una gestione più efficiente dei dati, la possibilità di lavorare con grandi quantità di informazioni senza compromettere le prestazioni e una maggiore sicurezza grazie alla possibilità di fare backup e recupero. Inoltre, i database permettono operazioni avanzate come query, filtri e ordinamenti, che non sarebbero facilmente realizzabili con le sole collezioni in memoria.

le

5 Guida alla compilazione e l'esecuzione del Progetto

Di seguito sono riportate le istruzioni dettagliate su come compilare il progetto e avviare correttamente i moduli client e server. Prima di compilare il progetto, assicurati di avere la struttura delle cartelle correttamente organizzata. Il progetto dovrebbe essere suddiviso in tre moduli principali:

Shared Contiene le librerie esterne e il codice condiviso tra client e server.

Client Contiene il codice del client.

Server Contiene il codice del server.

Inoltre, assicurati di avere la libreria Gson (versione 2.11.0) inclusa nella cartella `Shared/Libraries`, poiché viene utilizzata per la gestione del formato JSON.

Dalla cartella principale del progetto (dove si trova Compile.bat) eseguire lo script `Compile.bat`. Per avviare il client o il server, sempre dalla cartella principale, utilizza lo script `StartClient.bat` `StartClient.bat`.

Assicurati che i file di configurazione `client.properties` e `server.properties` siano configurati correttamente. Questi file definiscono parametri come le porte di comunicazione, gli indirizzi del server, e altre configurazioni necessarie per la connessione tra il client e il server. I file di proprietà sono copiati automaticamente nella cartella `Build/` durante la compilazione. Tuttavia, se necessiti di modifiche, puoi farlo prima o dopo la compilazione, ed è necessario riavviare il server e il client per applicare le modifiche.

All'interno del client e del server, il comando `help` stamperà la lista dei comandi e i loro argomenti.