UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Triennale in Informatica

CROSS: an exChange oRder bOokS Service

Relazione del progetto di LabIII

A.S. 2024-2025

A cura di Giulio Bartoloni

Matricola 654370

Indice

1 Scelte personali

Nel contesto del progetto, alcune decisioni sono state lasciate alla libera interpretazione. In questa sezione, esporrò le principali scelte effettuate, accompagnandole dalle motivazioni che le hanno guidate:

- **Politica di esecuzione degli ordini**: alcuni aspetti della politica dell'esecuzione degli ordini non sono stati specificati, al di là di alcuni dettagli e della time/price priority. Nell'approccio che ho adottato, il piazzamento di LimitOrder esegue subito i controlli per verificare se questo può essere soddisfatto o soddisfi altri ordini, provocando un'aggiornamento del prezzo al termine dell'algoritmo di match. Ad ogni aggiornamento del prezzo, gli stopOrder sono verificati ed eseguiti (NON PARZIALMENTE) secondo una priorità basata sul prezzo e sul timestamp. I marketOrder seguono un comportamento simile e vengono eseguiti istanteneamente. Nel processo di matching ho scelto di tenere in considerazione anche l'utente che ha piazzato l'ordine, evitando che un utente possa soddisfare i propri ordini precedenti, per non favorire manipolazioni del prezzo.

- **Notifiche**: l'invio delle notifiche per l'esecuzione degli ordini è stato progettato in modo tale da inviare notifiche singole per ciascun ordine completato, piuttosto che un'unica notifica per un insieme di ordini parziali.

- **Storico**: la logica relativa al salvataggio degli ordini eseguiti nello storico non era specificata, ho per questo deciso di memorizzare ogni esecuzione parziale, indicando il prezzo a cui è avvenuto il trade. Questa scelta permette di ottenere una rappresentazione accurata dei prezzi aggregati di mercato, riflettendo correttamente ogni singolo scambio di valuta.

2 Client e Server Threads

I thread lato client sono molto semplici:

- **main**: il thread del main ascolta i comandi inviati dall'utente tramite la linea di comando, prepara i messaggi, li invia al server, ascolta le risposte e le mostra all'utente.

- **NotificationListener**: questo thread rimane in ascolto di messaggi UDP sulla porta specificata nel file di configurazione. Quando riceve notifiche relative al completamento degli ordini, le mostra all'utente; mentre, quando riceve la notifica che il server è stato chiuso, termina il programma, evitando che l'utente debba inviare un messaggio per esserne informato.

I thread lato server sono invece:

- **main**: il thread del main fa da welcome thread. Questo dopo aver caricato tutti i file necessari e letto le configurazioni, rimane in attesa di nuove connessioni da instaurare. Quando questo avviene, crea un nuovo elemento nel threadPool dei ClientHandler.
- **clientHandler**: il ClientHandler, dopo aver inizializzato le variabili per la connessione, avvia un ciclo di ascolto. Ad ogni richiesta ricevuta, elabora la risposta corretta applicando la propria logica interna e invia la risposta al client. Questo thread fa parte del thread pool che gestisce tutte le connessioni attive dei client.
- **OrderBookCleanup**: questo thread viene eseguito in base a un intervallo di tempo configurato nel file di configurazione. Si occupa di rimuovere gli ordini completati dall'OrderBook e di eseguire il salvataggio periodico dei file della userMap e dello storicoOrdini.
- **InputListener**: questo semplice thread rimane in ascolto di comandi da terminale. Risponde al comando "shutdown" eseguendo una chiusura ordinata e salvando tutti i file necessari, e al comando "showOrders" (utile principalmente per il debugging e la gestione del server) per mostrare tutti gli ordini presenti nell'OrderBook.

3 Strutture dati

Dal lato client, non è stato necessario definire strutture dati particolari, ad eccezione dei vari tipi di richiesta e risposta.
Dal lato server, invece, oltre alla definizione dei tipi di richiesta e risposta, sono necessarie diverse strutture dati per garantire una gestione e funzionalità corrette:

- **crossOrderBook**: il crossOrderBook è definito come un Singleton che contiene il LimitOrderBook e lo StopOrderBook. Ciascuno dei due contiene un askOrderBook e bidOrderBook, separando i due lati per semplificare l'implementazione e migliorarne la leggibilità. Per supportare l'aggregazione degli ordini in priceRange, ciascun ask e bid OrderBook contengono un TreeSet (scelto per l'ordinamento sui prezzi) di aggregazioni di ordini con prezzo e size. Ogni aggregazione, naturalmente, contiene al suo interno tutti i singoli ordini, organizzati in un ulteriore TreeSet per semplificare l'ordinamento basato sul timestamp. Non sono state utilizzate ConcurrentCollections poiché i benefici di queste non sarebbero stati sfruttati. Ad esempio, il programma non avrebbe tratto vantaggio dal fine-grained locking, poiché l'esecuzione di un ordine richiede che l'intero orderBook sia bloccato, a causa di possibili incroci con StopOrder o altri ordini. Per evitare deadlock e interazioni indesiderate, l'uso di synchronized

sull'intero orderBook garantisce un'esecuzione corretta, senza compromettere eccessivamente l'efficienza.

- **userMap**: la UserMap contiene tutti gli utenti registrati alla piattaforma. Utilizza una ConcurrentCollection per supportare il fine-grained locking e garantire la massima efficienza. Viene salvata periodicamente nel file specificato e conserva anche le informazioni sugli indirizzi e le porte degli utenti connessi, necessari per gestire il sistema di notifiche relative alla disconnessione e al completamento degli ordini.
- **orderHistory**: questa struttura dati, anch'essa ConcurrentCollection, contiene lo storico degli ordini (anche parzialmente) eseguiti. Viene utilizzata per la costruzione degli OHLC e, come la userMap, viene salvata periodicamente e al momento della chiusura.

4 Primitive di sincronizzazione

Oltre al fine-grained locking delle ConcurrentCollections, sono utilizzati diversi blocchi di synchronized per evitare interazioni indesiderate. Questi vengono principalmente impiegati per la lettura e scrittura dalle socket, da System.in, System.out, System.err e sul crossOrderBook.

5 Istruzioni per l'esecuzione

Per la compilazione del codice non sono necessarie particolari attenzioni, oltre all'utilizzo della libreria gson, il cui file jar è presente insieme al codice sorgente.
L'esecuzione non richiede parametri aggiuntivi, in quanto tutto viene letto dai file di configurazione forniti:

- **clientConfig.properties**: questo file di config controlla tre parametri:
  - server.host: contiene l'hostname del server con cui tenterà di stabilire una connessione.
  - server.port: contiene la porta del server con cui tenterà di stabilire una connessione.
  - notificatio.port: contiene la porta del client che verrà utilizzata per ricevere le notifiche dal server.
- **serverConfig.properties**: questo file controlla tutti i parametri per il server:
  - server.port: permette di indicare la porta che il server utilizzerà per accettare le connessioni.
  - session.timeout: specifica il timeout in ms, causerà le disconnessioni per inattività.

  - server.corepoolsize: indica la dimensione del core della pool per i thread che gestiscono i client.
  - server.maximumpoolsize: indica la dimensione massima della pool per i thread che gestiscono i client.
  - cleanup.timer: specifica ogni quanti secondi sarà eseguita la routine di cleanup.
  - usermap.filename: indica il nome del file per la userMap.
  - storicoordini.filename: indica il nome del file per lo storico degli ordini.
  - pendingorders.filename: indica il nome del file per gli ordini dell'orderBook da caricare.

Oltre a questo, i comandi utilizzabili dal client sono visualizzabili tramite il comando "help" che indicherà tutti i parametri necessari, mentre lato server sono, come già precedentemente detto, disponibili i comandi di "shutdown" e "showOrders".

6 Packages

Per la realizzazione del progetto, oltre a quelli del client e server, ho organizzato il codice in diversi pacchetti:

- **clientFunctions.Requests**: contiene tutti i tipi di richieste che il client può effettuare.
- **serverFunctions.Responses**: contiene tutti i tipi di risposte che il server può inviare.
- **serverDataStructures**: contiene diverse strutture dati, con alcune che vengono utilizzate sia dal client che dal server.

7 Aggiunte

Oltre a quanto specificato nel testo del progetto, ho implementato alcune funzionalità che ritenevo utili:

- **persistenza degli ordini irrisolti**: sebbene non fosse previsto, ho implementato la persistenza degli ordini di tipo limit e stop piazzati ma non ancora risolti al momento della chiusura del server. Questa funzionalità consente di mantenere uno stato più consistente degli ordini tra diverse esecuzioni del server. A differenza della userMap e dello storicoOrdini, gli ordini irrisolti non vengono salvati periodicamente, ma esclusivamente allo shutdown del server.
- **interazioni con il server**: i comandi utilizzabili con il server non erano richiesti ma, soprattutto nel caso dello shutdown, sono fondamentali per verificare la corretta esecuzione lato server.

- **GetOrders**: questo comando aggiuntivo per il client consente di ottenere tutti gli ordini dell'utente loggato attualmente presenti nell'OrderBook. Può essere particolarmente utile per visualizzare gli orderId dei propri ordini, ad esempio, per una eventuale cancelOrder, o più in generale per tenere traccia dei propri ordini.

- **GetOrderBook**: il comando getOrderBook consente al client di visualizzare l'intero orderBook, insieme ad alcune informazioni come lo spread e l'ultimo prezzo di mercato. Questa funzionalità può essere utile per ottimizzare il piazzamento degli ordini al fine di massimizzare il profitto e contribuisce alla funzionalità complessiva del programma.