Corso di Laurea in Informatica

PROGETTO ESAME CORSO LABORATORIO III

”CROSS: an exChange oRder bOokS Service” online orders book scritto in Java, Multi-threaded, CLI, client/server, TCP & UDP sockets

Professoressa:
Laura Emilia Maria Ricci

Candidato:
Giulio Nisi

ANNO ACCADEMICO 2024/2025

Indice

Note

Si consiglia la lettura del codice in caso di necessitá, é molto commentato, risulta piú chiaro di tale relazione riassuntiva. Esso é scritto in inglese per essere compreso potenzialmente da un pubblico piú ampio e pubblicato su GitHub. Il codice é stato sviluppato su macOS (Sequoia 15.4.1) (Darwin Kernel Version 24.4.0) (arm64), ma supporta tutte le principali tipologie di macchine con Java. La versione di Java utilizzata é la Java JDK 21 LTS (long time support).

”Write Once, Run Anywhere.”

Sun Microsystem, about the new Java programming language.

1. Scelte implementative e funzionalitá aggiuntive

- La risposta alla richiesta API GetPriceHistory segue il formato JSON:

```
{
    "priceHistory": [
        {
            "dayGMT": "2025-01-01 00:00:00 GMT",
            "high": 100,
            "low": 90,
            "open": 95,
            "close": 98
        }, ...
    ]
}
```

  Si ha un array dove ogni elemento contiene le statistiche di un giorno del mese. L'autenticazione é stata volutamente resa non necessaria per questa funzione. Il sistema elabora lo storico basandosi solo sui market order bid per congruenza, contenuti nel file detabase degli ordini. Il file "storicoOrdini.json" puó esser adoperato come demo.

- Tutte le risposte (ma anche le richieste) JSON alle richieste API seguono esattamente lo schema previsto nell'assignment, senza la presenza di campi aggiuntivi. Quindi per gli ordini, in caso di errore, viene ritornato -1 come ID in tutte le situazioni, senza specificare il motivo del fallimento, perché questo avrebbe richiesto un valore aggiuntivo nella risposta.

- Le richieste e le risposte API hanno delle loro classi che fungono da wrapper degli oggetti del progetto generali. Questo rende tutto piú pedante, rispetto all'invio diretto degli oggetti Java con JSON, ma permette maggiore controllo e flessibilitá nello scegliere cosa inviare / ricevere e in che formato (con quali campi). Quindi, ad esempio, nel client, la creazione di un ordine segue il flusso: stringa del comando -> oggetto Java ordine -> oggetto Java CreateRequest -> oggetto Java Request -> stringa JSON. Nel server si segue il flusso inverso,

passando comunque dagli oggetti Request per eseguire il parsing: stringa JSON della richiesta -> oggetto Java Request -> oggetto Java CreateRequest -> oggetto Java ordine. L'ordine sará eseguito e poi il server risponderá con il flusso: oggetto Java ordine -> oggetto Java ExecutionResponse -> oggetto Java Response -> stringa JSON risposta.

- Sul file database degli ordini vengono salvati, oltre agli ordini limit e market, anche gli stop. Questi peró, non hanno un corrispondente ordine market salvato sul file, ma lo hanno in memoria in struttura dati. Dopo l'esecuzione di un ordine stop, sul file database degli ordini ci sará solo questo, in memoria invece, ci sará questo ed in aggiunta il corrispondente ordine market in cui lo stop si é trasformato.

- Gli ordini stop sul file database vengono salvati con il prezzo dell'ordine market che li ha eseguiti, mentre nella struttura dati in memoria, mantengono il prezzo della richiesta.

- Gli ordini scritti sul file database hanno come quantitá il valore per il quale sono stati eseguiti, mentre i loro corrispondenti in memoria, nella struttura dati, hanno il valore rimanente da eseguire.

- Per la scrittura dei files database per ordini ed utenti, é stata creata una classe FileHandler, che appende ordini ed utenti in fondo ai corrispettivi files senza riscrivere completamente tutto il contenuto JSON dei files. Questo é molto efficiente in caso di database grandi, perché riduce il tempo richiesto dalle costose operazione di I/O. Poiché per gli utenti deve esser possibile modificare la password, in questo caso, la riga dell'utente nel file viene sostituita con spazi per cancellarla e l'utente con la password aggiornata scritto al termine del file.

- In caso di logout, non arrivano notifiche di ordini eseguiti.

- Per gli ID degli ordini é stata creata una classe ad-hoc, UniqueNumber, che crea dei numeri univoci.

- Presenza del comando (e richiesta API) "exit()" per uscire gracefully.

- Codice commentato secondo lo standard Javadoc.

- Supporto parziale al multimercato. Le classi nel sono state strutturate per la futura possibilitá di creare piú mercati oltre a BTC/USD, con valute di propria scelta crypto e fiat (es: ETH/EUR) ed il relativo order book. Parziale, perché per evitare di allontanarsi troppo dalle specifiche richieste le API e database supportano attualmente solo il mercato deafult che é BTC/USD.

- Avanzata coordinazione e gestione dello STDIN e STDOUT della CLI del client. L'input dell'utente é gestito e coordinato con l'output (le risposte e notifiche ricevute dal server), quest'ultimo viene asincronamente stampato senza interrompere la digitazione dell'utente. Implementato mediante la libreria per la creazione di CLI avanzate in Java, denominata JLine.

2. Threads

Il server utilizza:

- MainThread: Creato dall'esecuzione di MainServer.java. Svolge viarie funzioni, tra cui, avvia il server con i suoi socket, carica gli utenti dal file database, carica gli ordini dall'altro file database, avvia gli altri thread. Dopo il bootstrapping si arresta.
- AcceptThread: Accetta connessioni TCP socket dai client e crea un ClientThread per ciascuno di essi, sottomettendolo ad un cached thread pool executor.
- ClientThread: Gestisce il suo corrispondente client. Elabora e risponde alle sue richieste.
- NotificationRegisterThread: Aspetta, riceve e memorizza indirizzi e porte da parte dei client. Questi dati saranno poi utilizzati (non da questo thread) per notificare i client con aggiornamenti sui loro ordini eseguiti mediante messaggi UDP.
- StopOrdersExecutorThread: Esegue gli ordini stop, giá convertiti in market precedentemente quando triggered dal ClientThread ed aggiunti ad una lista, da cui vengono prelevati dallo StopOrdersExecutorThread con sincronizzazione e coordinazione.

Il client utilizza:

- ClientCLIThread: Aspetta e riceve l'input dell'utente da command line. Lo trasforma in richiesta API JSON e lo invia al server.
- ResponsesThread: Riceve e stampa le risposte alla richieste, ricevute dal server, da socket TCP.
- NotificationsThread: Riceve e stampa le notifiche sugli ordini eseguiti, ricevute dal server, da socket UDP.

3. Strutture dati

Il server utilizza:

- LinkedList<Socket>: Per lista TCP socket degli utenti loggati. Nella classe User.
- LinkedList<InetSocketAddress>: Per lista dati (ip e porta) a cui notificare ordini eseguiti con UDP. Nella classe User.
- TreeSet<User>: Per memorizzare tutti gli utenti del database, con comparazione su username stringa. Nella classe Users.
- TreeMap<String, InetSocketAddress>: Per associare una stringa rappresentante un socket TCP (formato ip:porta) (non si poteva creare una TreeMap con chiavi direttamente dei Socket, perché questi non implementano comparatori) ai dati UDP a cui inviare notifiche. Per inviare notifiche solo ad utenti loggati. Nella classe Server.
- TreeSet<Order>: Per memorizzare gli ordini, comparazione fatta su ID. Nella classe Orders.
- LinkedList<Order>: Per memorizzare gli ordini duplicati. Infatti avevo assunto che non ci potessero essere ordini con ID duplicati, ma leggendo il file storico, mi sono accorto dopo, che vi erano, quindi ho aggiunto questa seconda struttura dati. Nella classe Orders.
- TreeMap<SpecificPrice, LimitBookLine<LimitOrder>>: Associa ad un prezzo una linea di tipo limit. L'insieme di queste creano il limit order book. Nella classe OrderBook. Scelta per avere accesso ad un prezzo in $O(log(n))$ sfruttando l'ordinamento dei prezzi. In ogni linea c'é la lista degli ordini di quel livello, citata sotto.
- TreeMap<SpecificPrice, LimitBookLine<StopOrder>>: Come sopra ma per lo stop order book.
- LinkedList<OrderBook>: Mantiene la lista degli OrderBook esistenti, per eventuale (non richiesto e parziale) supporto a piú mercati. Nella classe OrderBook.

- LinkedList<GenericOrder>: Mantiene la lista degli ordini su una specifica linea di un order book. Nella classe OrderBookLine. GenericOrder é un tipo generico che permette l'utilizzo sia per limit che per stop order book. Scelta per aggiungere ed estrarre ordini (creazione ed esecuzione) in $O(1)$. Questo NON vale per la cancellazione di ordini, che deve scorrere in $O(n)$ tutta la lista, ma ho assunto fosse un'operazione minoritaria rispetto all'inserimento ed esecuzione degli ordini.

Il client utilizza:

- LinkedList<>: Temporanee per l'elaborazione e il parsing dei comandi, di stringhe ed oggetti. Nelle classi ClientCLIThread e CLientCLICommandParser.

4. Sincronizzazione e coordinazione

Sono state utilizzate le seguenti primitive di sincronizzazione:

- Monitor / Lock implicite su oggetto su intero metodo: La maggioranza delle sincronizzazioni usano questo approccio. Sono state impiegate strutture dati non thread-safe, ma il loro accesso é sempre stato mediato da metodi sincronizzati quando necessario.
- Monitor / Lock implicite su oggetto su snippets di codice: Qualche volta, ad esempio nella sincronizzazione per l'uso del buffer contenente lo STDIN dell'utente, la sincronizzazione é inserita solo in parti di codice con blocchi ad-hoc sull'oggetto buffer.
- Class level lock: Per alcuni metodi statici, con funzionamento analogo alle precedenti, ma bloccanti a livello di intera classe e non del singolo oggetto / istanza.
- Wait e NotifyAll: ClientThread (n threads) ed il StopExecutorThread (1 thread) si sincronizzano con la lock implicita sulla lista degli ordini stop, ma si coordinano grazie a wait() e notifyAll() sempre sul medesimo oggetto. Le condizioni sono state testate in while per evitare spurius wake up.

5. Sorgente, librerie, compilazione, esecuzione, utilizzo, tests, script automatico

5.1 Sorgente

I files sorgenti sono contenuti in ”CROSS/src/”. Qui vi sono 3 files main:

- MainClient: Per avviare il client.
- MainServer: Per avviare il server.
- MainTests: Per eseguire alcuni tests.

Inoltre, vi é la cartella ”cross”, che rappresenta il package con all’interno tutte le varie classi sviluppate, suddivise in sottocartelle per area tematica.

5.2 Librerie

Le librerie sfruttate sono contenute in ”CROSS/lb/” e sono:

- JLine: Per poter leggere l’input dell’utente e stampare risposte e notifiche dal server in maniera asincrona, mantenendo congruenza grafica utilizzata nella CLI del client.
- GSon: Libreria di Google per il parsing e la creazione di JSON da oggetti Java, da utilizzare nelle comunicazioni API tra client e server.

5.3 Compilazione ed esecuzione manuale

Il progetto é stato sviluppato con JAVA 21 LTS. Una volta procuratosi tale versione ed aperto un terminale nella root del progetto si puó compilare con ”javac -cp ’CROSS/lib/*’ -d ’./bin’ filejava1.java filejava2.java ...” riportando tutti i files Java del package cross ed i main. Su Windows

sostituire gli ’ che delimitano i paths con ” ed i / con \. Allo stesso modo per eseguire, ad esempio il server, usare ”java -cp ’./bin:./CROSS/lib/*’ MainServer”. Su Windows sostituire gli ’ che delimitano i paths con ” ed i / con \ e il : con ;.

5.4 Compilazione ed esecuzione tramite script

Si puó avvalersi anche (sperando che funzioni, ma l’ho testato sia su Mac che Windows) di uno script automatico per compilazione ed esecuzione. Questo si occupa di scaricare anche la versione di Java 21 LTS corrispondente al sistema operativo ed architettura in uso. Per sfruttarlo recarsi con un terminale nella cartella ”./Java” ed eseguire python (o python3) compile.py. La sua creazione é motivata dal fatto che volessi un sistema che mi reperisse la versione di Java corretta in base al PC che stavo utilizzando e facesse funzionare tutto out of the box.

5.5 Esecuzione tramite jars precompilati

Aprire un terminale nella cartella root del progetto. Eseguire, ad esempio per il server, ”java -jar CROSS/dist/server.jar”.

5.6 Utilizzo

Per l’utilizzo del client la sintassi dei comandi é esattamente quella richiesta dall’assignment. Non utilizzare il file ”storicoOrdini.json” come database degli ordini e contemporaneamente eseguire ordini, i quali sarebbero scritti lí. Il file ”storicoOrdini.json” puó esser usato solo per richieste GetPriceHistory, in sola lettura, eseguendo degli ordini invece si andrebbe a scriverlo. Per eseguire ordini utilizzare il file ”orders.json” che dovrebbe inizialmente essere copia del file ”defaultOrders.json”, verranno scritti in ”orders.json”. Questo perché, come accennato nelle scelte implementative, il progetto utilizza la classe FileHandler per, durante l’append di un nuovo ordine eseguito sul file database, andare a scriverlo al termine, senza peró riscrivere tutto il JSON sul file. Questo é possibile solo con il formato JSON di ”defaultOrders.json”, per come sono disposte le virgole e le parentesi,

che storicoOrdini.json ha in modo diverso. Il server ed il client devono essere eseguiti dalla root del progetto perché possano trovare i files di configurazione da cui leggere i parametri.

5.7 Tests

Ci sono due tipi di tests. I primi sono in ”CROSS/src/MainTests.java” e sono per eseguire alcuni tests sulle classi e funzionalitá piú complesse, senza l’impiego di server e client. I secondi sono contenuti nel file ”README_TESTS.txt” nella root del progetto. Questi sono una lista di tests manuali con comandi da copia-incollare nel client, dopo aver avviato il server.