CROSS

Matteo Cherubini 658274

Contents

1 Introduzione

Il documento è una relazione volta a spiegare i dettagli di progettazione, implementazione e utilizzo dei principali componenti del progetto di *Laboratorio III* **CROSS**. Lo sviluppo è avvenuto in ambiente **UNIX** e la programmazione in **Java 23**.

2 Scelte di progettazione

2.1 Client

Si è deciso di optare per un *Client* minimale, che gestisce solamente il *parsing* e la validazione dell'input dell'utente e il *parsing* delle risposte ricevute dal *Server*, sia sincrone (**TCP**) che asincrone (**UDP**, gestite dalla classe **NotificationHandler**). Tutta la logica, compresa la conversione dell'unità di misura dei dati degli ordini, avviene lato *Server*. Il *Client* legge le informazioni per la connessione con il *Server* dal file *client.properties*

2.2 Server

Il *Server* si occupa della gestione delle funzionalità del programma. Attende la ricezione di una richiesta da parte di ognuno dei *Client* connessi, esegue le operazioni richieste e invia una risposta. Le informazioni per gestire le **socket TCP e UDP** vengono lette dal file *server.properties*. Inoltre, gestisce l'*orderBook*, lo *storico degli ordini*, le *credenziali* degli utenti e i *log delle transazioni*. Tutti questi dati vengono salvati nei file JSON che si trovano nella cartella *Data/* e vengono gestiti dalle classi descritte di seguito

2.2.1 LoginHandler

Si occupa di gestire e validare le credenziali degli utenti e di renderle persistenti all'interno del file *credentials.json* tramite serializzazione di una lista di oggetti di tipo **Credentials**

2.2.2 Order

Struttura dati contenente le informazioni sugli ordini piazzati. Come da specifica, gestisce i dati degli ordini come migliaia di unità (**size**: 1 BTC = 1000, **price**: 10 USD = 10000). Il campo **price** di un ordine è il prezzo di **un BTC**, quindi il prezzo totale è dato dal prodotto della **size** dell'ordine per il relativo **price**

2.2.3 OrderRecord

Definisce l'elenco degli ordini di tipo **bid** e degli ordini di tipo **ask** per ogni sezione dell'*orderBook*. È implementata con una *HashMap* sincronizzata, in cui le chiavi sono **bid** e **ask** e i valori la lista degli ordini del tipo relativo

2.2.4 OrderBook

Definisce la struttura dell'*orderBook*, con un *OrderRecord* per gli ordini di tipo **limit** e uno per gli ordini di tipo **stop**. Questo oggetto viene serializzato e salvato nel file *orderBook.json*. Inoltre, gestisce in modo sincronizzato l'esecuzione di ordini di tipo **market**, l'inserimento o la cancellazione di ordini di tipo **stop**

e **limit** e il relativo **matching**. Quando completa la transazione di una coppia di ordini, si occupa di notificare gli utenti interessati (tramite la classe **NotificationHandler**) e di persistere i dati serializzando oggetti di tipo **Trade** nel file *ordersLog.json*

2.2.5 Trade

Definisce la struttura delle notifiche delle transazioni avvenute e dei relativi log all'interno del file *ordersLog.json*. A differenza delle strutture dati degli ordini, qui i campi **size** e **price** sono trattati come *double*. Questa scelta è stata presa per poter comunicare agli utenti i dati precisi delle transazioni, mantenendo la logica di calcolo dei valori corretti nel *Server*. Inoltre, consente di limitare la grandezza dei numeri inviati nei datagrammi **UDP** (il **price** di un **Trade** corrisponde al prezzo totale, quindi il prodotto tra **size** e **price** dell'ordine avrebbe facilmente superato il valore massimo consentito da un valore di tipo *integer*, dato il prezzo attuale dei BTC)

2.3 Messaggi

Di seguito una descrizione dei **protocolli** di comunicazione tra *Client* e *Server*. La comunicazione si basa sulla serializzazione JSON di oggetti contenuti nei relativi *package*

2.3.1 Client to Server

La struttura di ogni messaggio inviato dai *Client* è basata sulla serializzazione di una classe specifica, contenuta nel *package ClientToServer*. Questi oggetti vengono serializzati ed inviati nella socket **TCP** dai *Client* al *Server*, il quale li de-serializza e ne legge il contenuto

2.3.2 Server to Client

La struttura di ogni risposta **TCP** del *Server* (o messaggio asincrono **UDP**) verso un *Client* è basata su serializzazione di una classe specifica, contenuta nel *package ServerToClient*. In questo *package* è contenuta anche la classe che definisce la struttura dei messaggi contenenti lo storico dei prezzi di un certo mese, richiesto dal *Client* (l'unica struttura dati la cui specifica non era definita)

3 Schema dei Thread

3.1 Server

1. **Main**
   Thread principale che si occupa di inizializzare le strutture dati e il socket di comunicazione **TCP**. Rimane in ascolto di nuove connessioni e, una volta accettato un nuovo *Client*, lo salva in una lista **sincronizzata** e

sposta la sua logica di gestione in un thread **ClientHandler**, appartenente ad una **CachedThreadPool**. Questo thread terminerà quando si chiuderà l'applicazione

2. **ClientHandler**
   Thread che gestisce le operazioni di comunicazione con ogni *Client*. Si occupa di ascoltare le richieste, eseguire le operazioni (sincronizzandosi con gli altri thread) e rispondere al *Client*. Se il *Client* richiede un'operazione **getPriceHistory**, questo thread avvierà un'altra serie di thread per gestire la richiesta come descritto di seguito. Questo thread terminerà quando si interromperà la connessione con il relativo *Client*

3. **PriceHistory**
   Una serie di massimo **4 thread** alla volta, i quali vengono avviati al fine di comunicare in modo efficiente i dati storici di prezzo dei BTC al *Client*. Questi thread si suddividono il carico inviando ognuno i dati di al massimo 8 giorni, mentre i restanti vengono processati (*4 thread * 8 giorni ognuno, sufficiente ad inviare i dati di 31 giorni*). Quando un thread viene avviato, prende il **lock esclusivo** della **stream output TCP** e inizia ad inviare il proprio buffer. Nel frattempo, vengono preparati i dati dei successivi 8 giorni. Al termine di ogni elaborazione, un nuovo thread verrà messo in coda, pronto ad inviare i propri dati non appena la **lock** viene liberata. Questa scelta è stata presa in combinazione con la decisione di leggere il file *storicoOrdini.json*, date le sue dimensioni, in modalità **streaming**, quindi non caricandone tutto il contenuto in memoria. Questo file contiene esclusivamente informazioni sui prezzi per i mesi **09/2024** e **10/2024**.

3.2 Client

1. **Main**
   Thread principale che si occupa di inizializzare le strutture dati e il socket di comunicazione **TCP**, gestire la logica di inserimento dell'input e di leggere la risposta inviata dal *Server*. Questo thread terminerà quando si chiuderà l'applicazione o tramite il comando **exit**

2. **NotificationHandler**
   Thread che si occupa di ricevere in modo asincrono i datagrammi **UDP** inviati dal *Server*. Questi datagrammi contengono le informazioni delle transazioni appena eseguite. La porta di ricezione dei datagrammi viene decisa in modo arbitrario, e viene comunicata al *Server* nel primo messaggio dello stream **TCP**. Questo thread terminerà insieme al thread **Main**

4 Sincronizzazione

La gestione della concorrenza tra thread avviene tutta lato *Server*

- **ClientList**:
  Lista dei *Client* connessi al *Server*. Implementato con una **synchronizedList** che garantisce atomicità nella rimozione, da parte del thread che lo gestisce, di un *Client* disconnesso. Inoltre, questa lista viene sincronizzata all'interno di un blocco **synchronized** quando viene iterata per trovare la socket **UDP** degli utenti loggati a cui inviare notifiche di transazioni avvenute

- **LoginHandler**:
  La classe che si occupa di gestire le credenziali degli utenti è sincronizzata sulla classe stessa ad ogni accesso. Dato che tutte le operazioni offerte da questa classe richiedono di iterare sulla lista delle credenziali, ogni volta che un thread di gestione di un *Client* interagisce con una di queste funzioni, acquisisce il **lock** della classe statica e lo libera alla fine dell'operazione

- **OrderBook**:
  Tutta la struttura dati OrderBook, e conseguentemente OrderRecord, che si occupa di gestire l'orderBook è sincronizzata sull'oggetto stesso. È imperativo garantire la **sequenzialità** e l'**atomicità** delle operazioni sugli ordini effettuate dai thread di gestione dei *Client*. Quindi ogni thread deve poter accedere all'*orderBook*, e alle conseguenti strutture dati, in modo esclusivo sia in scrittura che in lettura. Inoltre, la *HashMap* degli *OrderRecord*, che mette in relazione la lista degli ordini con la loro tipologia, è di tipo **ConcurrentHashMap**, che garantisce sincronizzazione nell'utilizzo della *HashMap* per operazioni atomiche

5 Compilazione e utilizzo

Il progetto utilizza la libreria esterna **GSON** presente all'interno della cartella *src/*, quindi va inclusa durante la compilazione e l'esecuzione come mostrato di seguito

5.1 Compilazione

- Spostarsi nella cartella con i file sorgente: **cd src/**
- Compilazione *Server*: **javac -classpath gson.jar -sourcepath . Server/ServerMain.java**
- Compilazione *Client*: **javac -classpath gson.jar -sourcepath . Client/ClientMain.java**

5.2 Avvio tramite class files

- Spostarsi nella cartella con i file sorgente: **cd src/**
- Avvio *Server*: **java -cp :gson.jar Server.ServerMain**
- Avvio *Client*: **java -cp :gson.jar Client.ClientMain**

5.3 Avvio tramite jar file

- Spostarsi nella cartella con i file jar: **cd jar/**
- Avvio *Server*: **java -jar Server.jar**
- Avvio *Client*: **java -jar Client.jar**

5.4 Elenco comandi eseguibili

- **register(name, password)**: registrazione dell'utente
- **login(name, password)**: login dell'utente
- **logout()**: logout dell'utente
- **insertLimitOrder(ask/bid, size, price)**: creazione limit order
- **insertStopOrder(ask/bid, size, stopPrice)**: creazione stop order
- **insertMarketOrder(ask/bid, size)**: creazione market order
- **cancelOrder(orderId)**: cancellazione ordine piazzato in precedenza
- **getPriceHistory(monthNumber)**: storico dei prezzi di un certo mese
- **exit()**: chiusura applicazione client

Nota: è possibile inserire ordini di vendita o di acquisto indipendentemente dalla quantità di BTC o di USD posseduti. In altre parole, ai fini del progetto, non viene tenuta traccia del capitale (BTC o USD) degli utenti al momento dell'inserimento di un ordine, ma viene data la libertà di poter inserire un numero arbitrario di ordini e di osservare come vengono gestiti dal sistema, rispettando le specifiche e gli algoritmi richiesti.