Reti e Laboratorio: Modulo Laboratorio 3 CROSS: an exChange oRder bOokS Service
Progetto di Fine Corso A.A. 2024/25

Carlo Tarabbo - n.m. 654342

Versione 1.1

Scelte personali

Rappresentazione dell'order book

La classe utilizza tre collezioni principali per rappresentare i dati:

1. `askOrders`: Un `TreeSet` che contiene gli ordini di vendita (ask), ordinati in modo crescente per prezzo, utilizzando un comparatore personalizzato.
2. `bidOrders`: Un `TreeSet` che contiene gli ordini di acquisto (bid), ordinati in modo decrescente per prezzo, sempre grazie a un comparatore personalizzato.
3. `stopOrders`: Una `ArrayList` per gestire ordini di tipo stop, mantenendo una gestione lineare e semplice per questa categoria.

L'uso di `TreeSet` garantisce un ordinamento dinamico degli elementi, fondamentale per accedere rapidamente agli ordini con i prezzi più alti o più bassi. Inoltre, l'uso di comparator personalizzati per `askOrders` e `bidOrders` facilita la distinzione tra l'ordinamento rischiesto per ordini di acquisto e vendita.

La classe `OrderBookEntry` rappresenta una singola voce nel libro ordini e funge da contenitore per gli ordini con lo stesso prezzo, utilizza una `ArrayList<Order.LimitOrder>` per mantenere una reference agli ordini che lo compongono, necessario per l'invio di notifiche.

Il metodo `addOrder` permette di aggiungere un nuovo limitOrder all'orderBook, creando una entry nuova o aggiornando una con lo stesso prezzo.

msgtype per la comunicazione TCP

Oltre alle specifiche fornite per i campi dei messaggi di richiesta e risposta fornite dal testo, si introduce un campo `msgtype` utilizzato da client e server per permettere un corretto handling delle diverse richiesta, che portano ad un cambiamento dei campi del messaggio.

Le due funzioni `handleMessage` e `createMessage` , rispettivamente nel server e client, permettono di creare messaggi con campi diversi a seconda del tipo.

Limit Orders

I limit orders, sottoclasse che estende un `Order` generico, sono gestiti in questo modo:

- Al momento della ricezione da parte del server di una richiesta contenente un limit order da processare, questo viene processato come un market order, rispettando però il limitprice indicato

come restrizione.

- Se il market order corrispondente viene accettato, viene salvato nello storico delle transazioni come limit order e gli utenti online che ne hanno fatto parte ricevono una notifica tramite protocollo UDP
- Nel caso in cui non si possa soddisfare la richiesta immediatamente, questa viene salvata nell'order book e verra' consultata per soddisfare prossimi ordini di qualsiasi tipo

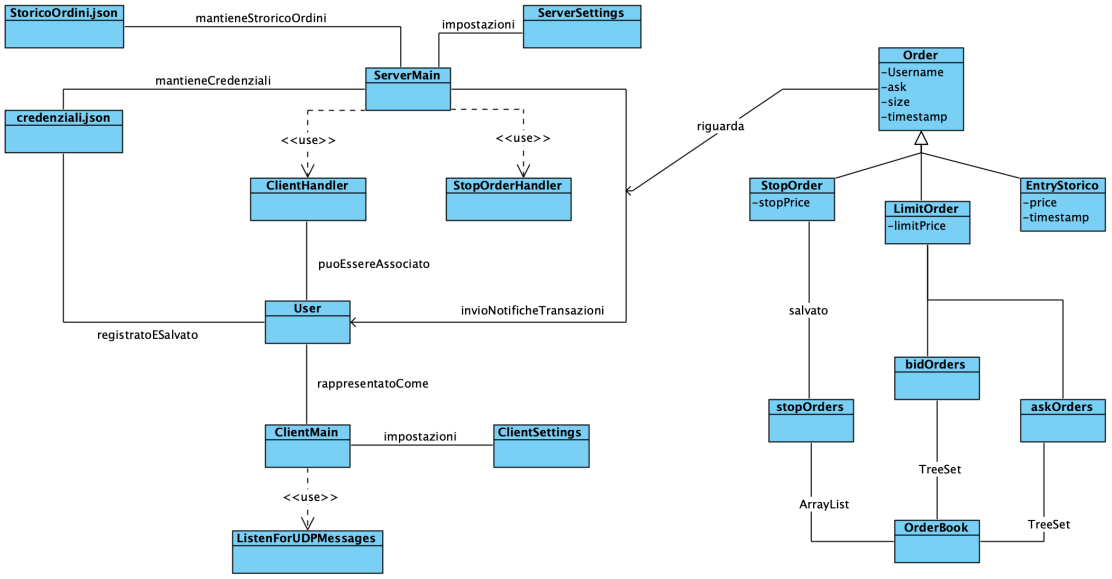
Price History

La funzione `getPriceHistory`, come le altre funzioni di handling delle richieste, riceve un messaggio JSON che contiene il mese e l'anno da cui l'utente è interessato a ricevere delle informazioni.

Vengono quindi calcolati i limiti temporali del mese e filtrate le transazioni rilevanti, per poi raggrupparle per giorno e calcolare le statistiche di interesse.

Il metodo restituisce i risultati in formato JSON, o un errore se non ci sono transazioni nel periodo richiesto. Il formato in cui vengono inviate le notifiche è inteso come una classica risposta di una api, quindi non e' formattata per essere letta direttamente dall'utente.

Schema generale del sistema



Lato Server - `ServerMain.java`

Contiene 3 classi per il funzionamento del server.

1. ServerMain contiene i parametri globali statici utilizzati dal main thread e dai diversi clients collegati.
2. I client handlers sono thread associati ad ogni nuovo client che si collega alla socket ed ha il compito di gestire la comunicazione con esso.
3. Lo stop order handler si occupa di controllare periodicamente l'order book per processare gli stop order che hanno superato la soglia indicata nel loro *stopPrice*

Il server è diviso in diversi threads che vengono gestiti da una `cachedThreadPool`. Il thread principale si occupa di aprire le socket per la comunicazione con i thread tramite protocollo TCP e UDP, carica in

memoria lo storico delle transazioni passate per poterle analizzare e carica le credenziali degli utenti fino ad ora registrati. Entra poi in un loop accettando le richieste di collegamento dei clients ed associandone un thread per la gestione.

I `clientHandler` threads gestiscono la comunicazione con i client, accettando, processando e inoltrando alle strutture dati interessate le richieste fatte dagli utenti.
Ogni thread ha un client associato, che una volta ricevuta una richiesta corretta di login, viene associato ad un account utente, sbloccando la possibilità di interagire con l'order book e fornire *Orders*. Un thread, e quindi un client, possono quindi fornire richieste sotto account (*User*) differenti nel caso di diversi login/logout; solamente l'ultimo utente associato ad un thread potrà ricevere notifiche al completamento di una transazione, data l'impossibilità di comunicare con un utente non collegato al server. Tutte queste informazioni sono gestite dalla classe `User` e salvate nella `concurrentHashMap clientsUsernames`

Lo *stopOrdersHandler,* unico thread, ha il compito di controllare periodicamente se ci sono *stopOrders* che hanno superato (in positivo o negativo) la soglia di prezzo indicata; e nel caso viene passato un market order associato tramite `tryMarketOrder`. Se questo "marketOrder" e' accettato lo stopOrder associato viene eliminato dalla lista dell'order book e inserito nello storico, altrimenti rimane in attesa.

Lato Client

Il client e' diviso in due threads:

1. un main thread per l'interazione con l'utente, la creazione di richieste da inviare al server e la stampa delle risposte ricevute
2. un thread dedicato alla ricezione e stampa dei messaggi asincroni tramite protocollo UDP al completamento di un ordine

Comunicazione UDP

La comunicazione tramite UDP viene implementata in questo modo:

- All'avvio del server viene aperta una `DatagramSocket udpSocket` senza specificare la porta, lasciando la scelta alla JVM.
- Al collegamento con il server il client apre la sua socket e invia come primo messaggio al server la porta su cui si e' collegata tramite la connessione TCP instaurata precedentemente.
- Il thread *clientHandler* che sta gestendo il client per prima cosa legge il messaggio contenente la porta inviata dal client e la associa alla variabile locale `udpPort` ed ad uno *User* al momento del suo login. Questa porta verrà poi utilizzata quando necessario dalla funzione `notifyUser(String username, String message)`, che andando a cercare nelle informazioni degli utenti collegati usando lo username come chiave di ricerca, la troverà con il resto dei dati di quell'utente, se loggato

Strutture dati utilizzate

- `ConcurrentHashMap<String,User>` per i dati degli utenti registrati al servizio.
- OrderBook per rappresentare l'order book, che e' composto da:
  - `TreeSet<OrderBookEntry>` per mantenere ordinate e facili da interrogare i limitOrder in attesa, divisi per ask e bid dati i diversi ordinamenti richiesti
  - `ArrayList<Order.StopOrder>` per mantenere la lista di stopOrdini in attesa , ordinata quando richiesta per recuperare gli ordini più vecchi

- `credentialsFileLock`, `storicoTransazioniLock` e `stopOrdersLock` per la sincronizzazione delle strutture non thread-safe
- `OrderBookEntry` rappresenta una entry all'interno dell'order book e contiene tutti i *limitOrders* con uno stesso prezzo, indicandone la quantità e valore totale, oltre a mantenere una lista degli ordini che la compongono

Sincronizzazione

L'esecuzione garantisce un comportamento thread-safe utilizzando synchronized e strutture dati thread-safe di libreria.

Le struttura che possono essere utilizzate da più threads sono:

- il JsonArray contenente lo storico delle transazioni, di cui e' garantita la mutua esclusione in scrittura tramite synchronized e `storicoTransazioniLock`
- La hashMap contenente i dati degli utenti, thread-safe perchè dichiarata come *ConcurrentHashMap*
- La scrittura su file di aggiornamenti delle credenziali degli utenti e' resa un mutua esclusione con `credentialsFileLock`
- La scrittura sull'order book di nuovi *limitOrders* è garantita in mutua esclusione da `orderBookLock`

Istruzioni per la compilazione ed esecuzione

Struttura del progetto

```
.
├── MANIFESTCLIENT.MF
├── MANIFESTSERVER.MF
└── src
    ├── ClientMain.class
    ├── ClientMain.java
    ├── Order.java
    ├── OrderBook.java
    ├── ServerMain.java
    ├── Settings
    │   ├── ClientSettings.java
    │   └── ServerSettings.java
    ├── User.java
    ├── gson-2.11.0.jar
    └── json
        ├── credenziali.json
        └── storicoOrdini.json
```

Server

```
Compilazione
javac -cp src/gson-2.11.0.jar -d out $(find . -name "*.java") //lista di tutti i f

Creazione file jar
jar cfm ServerMain.jar MANIFESTSERVER.MF -C out .
```

```
Esecuzione
java -jar ServerMain.jar
```

Client

```
Compilazione
javac -cp src/gson-2.11.0.jar -d out $(find . -name "*.java") //lista di tutti i f

Creazione file jar
jar cfm ClientMain.jar MANIFESTCLIENT.MF -C out .

Esecuzione
java -jar ClientMain.jar
```

Librerie esterne utilizzate

- gson-2.11.0.jar per la gestione di file e oggetti di tipo JSON