Struttura Generale del Progetto CROSS

1 Introduzione

Il progetto CROSS (*CROSS: an exChange oRder bOokS Service*) implementa un **order book** per la gestione di ordini di acquisto e vendita di Bitcoin (BTC) in USD su un exchange centralizzato. Il sistema è basato su un'architettura **client-server**, dove il server gestisce l'order book e le transazioni, mentre il client permette agli utenti di interagire con il sistema.

2 Struttura delle Cartelle

Il codice è organizzato nella cartella principale `CROSS`, che contiene:

- `cross.server`: Contiene il codice del server, inclusa la gestione degli ordini, utenti e connessioni.
- `cross.client`: Contiene il codice del client, che permette agli utenti di inviare ordini e ricevere notifiche.
- `config`: Contiene i file di configurazione per client e server.

3 Descrizione dei File Principali

3.1 Server

- `ServerMain.java`: Avvia il server e gestisce le connessioni.
- `Orderbook.java`: Coordina l'esecuzione degli ordini limite, di mercato e stop.
- `BuyStopOrderExecutor.java` / `SellStopOrderExecutor.java`: Gestiscono gli ordini di stop.
- `StoricoOrdiniHandler.java`: Memorizza e recupera la cronologia degli ordini eseguiti.
- `UserManager.java`: Gestisce utenti e autenticazione.
- `ConnectionHandler.java`: Gestisce le connessioni TCP con i client.
- `NotificationSender.java`: Invia notifiche sugli ordini eseguiti.

3.2 Client

- `ClientMain.java`: Avvia il client e gestisce la comunicazione con il server.
- `SocketTCP.java`: Gestisce la connessione TCP per inviare richieste al server.
- `ServerListener.java`: Riceve notifiche via TCP e UDP.
- `SyncConsole.java`: Sincronizza l'output su console.

3.3 Ordini e Trading

- `Order.java`: Classe astratta che rappresenta un ordine.
- `LimitOrder.java` / `MarketOrder.java` / `StopOrder.java`: Sottoclassi per ordini limite, di mercato e di stop.
- `Trade.java`: Rappresenta una transazione eseguita tra due ordini.

4 Gestione dei Thread e Thread Pool

Per garantire un'elaborazione efficiente e scalabile, sia il server che il client utilizzano diversi thread per gestire le operazioni in parallelo. L'uso di thread pool consente di ottimizzare le risorse disponibili e migliorare la reattività del sistema.

4.1 Thread Attivati nel Server

Il server avvia diversi thread per gestire le operazioni concorrenti:

- **Thread di gestione connessioni**: Ogni client connesso viene gestito da un thread dedicato tramite `ConnectionHandler.java`, che gestisce autenticazione e operazioni sugli ordini.
- **Thread per gli Stop Order**: `BuyStopOrderExecutor.java` e `SellStopOrderExecutor.java` sono due thread dedicati alla gestione degli ordini stop, monitorando continuamente le condizioni di attivazione.
- **Thread per il salvataggio dei dati**: Un thread schedulato esegue periodicamente il salvataggio dello storico degli ordini e degli utenti registrati, evitando perdita di dati in caso di arresto improvviso.

4.2 Thread Attivati nel Client

Anche il client utilizza thread per garantire un'interazione fluida con il server:

- **Thread di gestione della connessione TCP**: `SocketTCP.java` avvia un thread per inviare richieste e ricevere risposte dal server.
- **Thread di ascolto delle notifiche**: `ServerListener.java` avvia due thread separati:
  - Un thread per la ricezione delle risposte via TCP.
  - Un thread per la ricezione delle notifiche asincrone via UDP.

4.3 Scelta dei Thread Pool

L'uso di thread pool consente una gestione più efficiente delle risorse, evitando la creazione e distruzione continua di thread. Nel progetto sono stati utilizzati:

- **CachedThreadPool** (`ServerMain.java`): Viene utilizzato per la gestione delle connessioni con i client. Questo tipo di thread pool crea nuovi thread su richiesta e riutilizza quelli esistenti quando diventano disponibili, adattandosi dinamicamente al carico del sistema.
- **FixedThreadPool** (`BuyStopOrderExecutor.java` e `SellStopOrderExecutor.java`): Due thread fissi dedicati esclusivamente alla gestione degli ordini stop, evitando sovraccarichi e garantendo un'elaborazione costante.
- **ScheduledThreadPool** (`StoricoOrdiniHandler.java` e `UserManager.java`): Utilizzato per eseguire operazioni periodiche come il salvataggio automatico dello storico ordini e degli utenti, migliorando l'affidabilità del sistema.
- **FixedThreadPool** nel client (`ClientMain.java`): Il client utilizza un thread pool con due thread fissi per la gestione della comunicazione TCP e UDP, garantendo la ricezione delle notifiche in tempo reale senza bloccare l'interfaccia utente.

5 Strutture Dati e Persistenza

5.1 Strutture Dati per l'Order Book

- **ConcurrentSkipListMap(Integer, ConcurrentLinkedQueue(LimitOrder))**: Permette la gestione degli ordini per prezzo garantendo la priorità price/time.
- **ConcurrentLinkedQueue(LimitOrder)**: Ogni prezzo nella mappa ha associata una coda che mantiene l'ordine temporale degli ordini.

5.2 Strutture Dati per la Gestione degli Utenti

- **ConcurrentHashMap(String, User)**: Garantisce l'accesso sicuro e veloce agli utenti.
- **ConcurrentHashMap(String, Boolean)**: Traccia gli utenti online in modo efficiente.

5.3 Strutture Dati per la Memorizzazione degli Ordini Passati

- **ConcurrentLinkedDeque(Trade)**: Utilizzata per mantenere lo storico delle transazioni in ordine temporale con accesso rapido agli ultimi ordini eseguiti.

5.4 File di Persistenza

I dati vengono salvati in formato JSON per garantire la persistenza:

- `storicoOrdini.json`: Contiene la cronologia delle transazioni eseguite.
- `users.json`: Memorizza gli utenti registrati.
- `orderbook.json`: Contiene gli ordini limite attivi.
- `buyStopOrders.json` / `sellStopOrders.json`: Salvano gli ordini stop.

6 Scelte a Libera Interpretazione

- Quando uno **Stop Order** viene scartato, il proprietario riceve una notifica sotto forma di trade con il suo ID e valori di size e prezzo impostati a -1.
- Se il **client viene chiuso per timeout** (`setSOtimeout`) o il **server viene terminato**, il client riceve un messaggio con codice `999` e chiude l'esecuzione.
- Per la gestione delle **socket UDP**, il client comunica al server la porta su cui è in ascolto. Il server associa all'ordine l'**indirizzo IP** e la **porta UDP** del proprietario, garantendo notifiche efficienti.

7 Primitive di Sincronizzazione

- **Lock Espliciti**: Usati in `Orderbook.java` e `StoricoOrdiniHandler.java` per evitare inconsistenze.
- **Semafori**: Utilizzati in `BuyStopOrderExecutor.java` e `SellStopOrderExecutor.java` per attivare i thread solo quando ci sono nuovi ordini quindi la coda non è vuota.
- **Variabili Atomiche**: `AtomicInteger orderId e price` per la gestione degli ID degli ordini e per la gestione del prezzo corrente e `AtomicBoolean running` per controllare lo stato del server.

8 Istruzioni per la Compilazione ed Esecuzione

Su una finestra avviare il server con:

```
$ java -jar ServerMain.jar
```

Su un'altra finestra avviare il server:

```
$ java -jar ClientMain.jar
```

Esempio di interazione del client:

```
Client running...
Connecting to server...
> Connessione TCP avviata.
> Inserisci operazione (help per vedere le operazioni disponibili):
help
> Operazioni disponibili:
> login
> register
```

```
> updateCredentials
> getCurrentPrice
> exit
```

```
> Inserisci operazione (help per vedere le operazioni disponibili):
login
> Inserisci username:
davide
> Inserisci password:
davide11
[TCP] OK
```

Dopo essersi loggati potrò usare gli altri comandi

```
> insertLimitOrder
> Inserisci tipo (ask/bid):
bid
> Inserisci quantita':
0.457
> inserisci il prezzo:
15555554.678
[TCP] L'ID del tuo ordine e' 82045

> logout
[TCP] OK
```