PROGETTO LABORATORIO III

STEFANO VITALE

DESCRIZIONE PROGETTO:

Il progetto riguarda l'implementazione dell'order book, un servizio fondamentale nei mercati finanziari ed utilizzato per supportare lo scambio di asset (azioni, titoli, altri strumenti finanziari, criptomonete) in servizi di trading centralizzati. Un order book è un registro che contiene tutti gli ordini di acquisto e di vendita disponibili per un asset specifico riportando prezzi e volumi delle transazioni proposte da compratori e venditori, ed evidenziando, rispettivamente, il prezzo richiesto per la vendita (ask) ed il prezzo offerto per l'acquisto (bid). Queste informazioni sono fondamentali per valutare la dinamica di domanda e offerta per un asset e per eseguire scambi equi, efficienti e ordinati. In questo progetto ci focalizziamo sugli order books degli exchange centralizzati di criptovalute (ad esempio Binance, Coinbase, Kraken,...), ovvero piattaforme digitali che permettono agli utenti di acquistare, vendere, e scambiare criptovalute, facilitando le transazioni tra acquirenti e venditori di asset digitali come Bitcoin, Ethereum e altre criptovalute. In generale, un servizio di exchange di criptovalute gestisce in parallelo ordini di acquisto e di vendita di diverse "coppie di scambio", ovvero combinazioni di due diverse criptovalute o di una criptovaluta con una moneta fiat, ad esempio la coppia BTC/ETH, ovvero Bitcoin/Ether. Per semplicità in CROSS, considereremo solo scambi BTC/USD (Bitcoin/dollaro statunitense).

DESCRIZIONE CARTELLA PROGETTO

StefanoVitaleLab3 /
- lib / gson-2.10.1.jar (libreria gson usata)
- out /
  - client / (destinazione file .class lato client)
  - server / (destinazione file .class lato server)
- src /
  - client / (classi lato client)
  - server / (classi lato server)
  - util / (classi condivise)
- ConfClient.properties (configurazione esempio per client)
- ConfServer.properties (configurazione esempio per server)
- FastRun.bat (script per eseguire tutti i comandi di build)
- ManifestClient.txt (per creare i .jar)
- ManifestServer.txt (per creare i .jar)
- RelazioneProgetto.pdf

ESEMPIO CARTELLA "CLIENT_1"

Client_1 /
- lib / gson-2.10.1.jar (copiare dalla cartella progetto)
- Client.jar
- ConfClient.properties (configurazione specifica per il client_1)

ESEMPIO CARTELLA "SERVER"

Server /
- lib / gson-2.10.1.jar (copiare dalla cartella progetto)

Server.jar
ConfServer.properties (configurazione specifica per il server)

SCELTE ADOTTATE

1) **COSTRUZIONE E RICEZIONE MESSAGGI**: Per lo scambio di informazioni tra client e server i messaggi vengono convertiti in JSON sfruttando la **libreria Gson (Tree Model API).** Le principali operazioni sono:
    a. (**serializzazione**) *gson.toJson(jsonObject);*
    b. (**deserializzazione**) *JsonObject obj = gson.fromJson(messaggio, JsonObject.class);*
2) **RISPOSTA A "getPriceHistory":** utilizza oggetti Json (con Gson) per creare la risposta. Le transazioni vengono raggruppate per data attraverso una TreeMap per garantire che i record siano ordinati cronologicamente. Per convertire il Timestamp in GMT viene usato Instant e ZonedDateTime, assicurando che le operazioni vengano raggruppate correttamente per giorno, indipendentemente dalle impostazioni locali del server. All'utente viene inviato un JsonArray dove ogni elemento è un giorno con le proprietà date, open, close, max, min;
3) **COMUNICAZIONE PORTA UDP CLIENT:** Ogni volta che un utente effettua il login viene inviata al server anche la porta udp su cui il client è in ascolto per le notifiche.

SCHEMA GENERALE DEI THREAD ATTIVI

- **LATO SERVER:**
    - **MainServer:** Avvia il ServerSocket e crea un thread pool;
    - **ServerExecutor:** Per ogni nuova connessione TCP, viene avviato un thread (dal pool) che gestisce la comunicazione con il client;
- **LATO CLIENT:**
    - **MainClient:** Gestisce l'interazione con l'utente tramite una un menù di operazioni;
    - **UdpClient:** Avvia un thread separato, riname in ascolto per ricevere le notifiche asincrone UDP dal server, aggiornando l'utente in tempo reale sull'evasione degli ordini da lui effettuati;

STRUTTURE DATI PRINCIPALI

- **(OrderBook)** ConcurrentHashMap<Integer, Order>: Utilizzata per associare un identificativo univoco a ogni ordine e garantire accessi concorrenti;
- **(OrderBook)** PriorityQueue<LimitOrder>: Due code separate per gli ordini BID e ASK, ordinate secondo la time / price priority per garantire che l'ordine più vantaggioso (e più veccio a parità di prezzo) venga eseguito per primo;
- **(OrderBook)** ArrayList<StopOrder>: Utilizzata per memorizzare gli Stop Order in attesa di essere attivati;
- **(UserManager)** ConcurrentHashMap<String, User>: ottenere l'oggetto utente dall'username;
- **(UserManager)** ConcurrentHashMap<String, Integer>: ottenere la porta udp dall'username;
- **(UserManager)** ConcurrentHashMap<String,InetAddress>: ottenere l'indirizzo ip dall'username;

PRIMITIVE DI SINCRONIZZAZIONE

1) Le operazioni sull'OrderBook (matching, add Order, remove Order) sono gestite tramite blocchi synchronized per garantire la coerenza dei dati in presenza di accessi concorrenti;

2) Un oggetto lock (**MainServer.transactionFileLock**) viene utilizzato per sincronizzare l'accesso al file JSON delle transazioni, evitando corruzioni e conflitti di accesso;

GESTIONE DEI FILE DI CONFIGURAZIONE

La classe **LoaderConfig** è responsabile dell'importazione dei parametri dai file di configurazione (ConfServer.properties e ConfClient.properties). Questi file contengono i parametri di input essenziali (numeri di porta, indirizzi, timeout, ecc.) e vengono caricati a runtime per consentire la flessibilità necessaria in fase di esecuzione. La cartella in cui viene cercato il file di configurazione è la stessa dove è contenuto il file .jar del Client / Server. La cartella del progetto contiene un file .properties di esempio sia per Client sia per Server, dove i parametri hanno già un valore di esempio.

CONFIGURAZIONE SERVER (ConfServer.properties)

- **PORT:** Definisce la porta su cui il ServerSocket ascolta; *esempio: 12345;*
- **CORE_POOL_SIZE e MAX_POOL_SIZE:** Questi parametri controllano il numero minimo e massimo di thread nel pool per il multithreading; *esempio: core = 4, max = 10;*
- **EXECUTOR_INACTIVITY_TIME_MILLISECONDS:** Specifica il tempo massimo di inattività per i thread extra nel pool; *esempio: 350000;*
- **CLIENT_TIMEOUT_MILLISECONDS:** Imposta il timeout di disconnessione per un client; *esempio: 300000;*
- **EXECUTOR_TIMEOUT_MILLISECONDS:** Il timeout per la terminazione dell'executor dopo la richiesta di shutdown; *esempio: 400000;*
- **TRANSACTION_FILE:** percorso assoluto o relativo del file che conterrà i record relativi alle transazioni evase. Se relativo, verrà inserito a partire dalla cartella del jar*; esempio: transactions.json;*
- **UDP_TIMEOUT_CLOSE_MILLISECONDS:** usato in un thread.sleep per dare il tempo al socket udp di inviare il pacchetto prima di chiudersi; *esempio: 1000;*
- **DIM_QUEUE_TASK:** imposta una dimensione massima al buffer dei task in attesa; *esempio: 50;*

CONFIGURAZIONE CLIENT (ConfClient.properties)

- **SERVER_ADDRESS e SERVER_PORT:** Dati per connettersi al server; *esempio: 12345;*
- **UDP_PORT:** Specifica la porta su cui il client ascolta le notifiche UDP; *esempio: 54321;*
- **RECONNECTION_DELAY_MILLISECONDS:** Imposta il ritardo prima di tentare una riconnessione in caso di interruzione; *esempio: 5000;*

REQUISITI / INFO FILE / NOTE

- *Versione JAVA: JDK 8*
- *Sistema Operativo: Windows (Testato su windows 10)*
- *Libreia Gson* (Progetto testato con gson-2.10.1.jar, già inclusa nella cartella /lib);
- Ogni file jar eventualmente spostato dalla cartella del progetto deve essere in grado di leggere il percorso lib/gson-2.10.1.jar per recuperare la libreria;
- *File configurazione* client / server nella stessa cartella dei file .jar (Due file di esempio inclusi nella cartella principale del progetto);
- *In caso di modifica dei Manifest, assicurarsi che il file termini con una riga vuota;*

OPERAZIONI UTENTE (LATO CLIENT)

Il client presenta all'utente un menù testuale di operazioni, ciascuna identificata da un numero. Per selezionare l'operazione, l'utente deve digitare il numero corrispondente.Il menù è continuamente riproposto finché l'utente non decide di chiudere il programma. Durante l'intera esecuzione, il client mantiene la connessione con il server e, in caso di disconnessione imprevista, tentarà automaticamente una riconnessione dopo un intervallo configurabile. La voce [0] permette di termina il client in modo ordinato e chiudere la connessione con il server.

COMANDI CREAZIONE FILE .CLASS (da eseguire nella cartella principale del progetto)

- ***LATO SERVER***: *javac -d out/server -cp lib/gson-2.10.1.jar src/server/*.java src/util/*.java*
- ***LATO CLIENT***: *javac -d out/client -cp lib/gson-2.10.1.jar src/client/*.java src/util/*.java*

COMANDI CREAZIONE FILE .JAR (da eseguire nella cartella principale del progetto)

- ***LATO SERVER***: *jar cfm Server.jar ManifestServer.txt -C out/server . (il punto finale è necessario)*
- ***LATO CLIENT***: *jar cfm Client.jar ManifestClient.txt -C out/client . (il punto finale è necessario)*

COMANDI ESECUZIONE FILE .JAR (da eseguire nelle rispettive con jar + conf + lib)

- ***LATO SERVER***: java -jar Server.jar
- ***LATO CLIENT***: java -jar Client.jar