

Reti e Laboratorio: Modulo Laboratorio 3

CROSS: an exChange oRder bOOkS Service

Progetto di Fine Corso A.A. 2024/25

Corsi A – B

Progetto e Relazione a cura di Vicarelli Tommaso – Matricola 638912

DESCRIZIONE GENERALE

Il progetto CROSS implementa un servizio di order book, componente essenziale nei mercati finanziari e nei servizi di trading centralizzati. L'order book regista tutti gli ordini di acquisto (bid) e di vendita (ask) relativi a un determinato asset, mostrando prezzi e volumi e consentendo così di analizzare la domanda e l'offerta e di eseguire scambi in modo equo ed efficiente. Nel contesto del progetto, l'attenzione è rivolta agli exchange centralizzati di criptovalute (come Binance o Coinbase), piattaforme che gestiscono le transazioni tra utenti. Per semplicità, CROSS si concentra sulla gestione degli ordini relativi alla coppia di scambio BTC/USD (Bitcoin/Dollaro statunitense).

STRUTTURAZIONE E ORGANIZZAZIONE DEI FILE

Il progetto è organizzato in una struttura gerarchica di cartelle che rispecchia la natura dell'applicazione.

1. “/lib”: contiene la libreria gson-2.10.1.jar di Google, utilizzata per la gestione dei dati in formato JSON
2. “/src/main/java”: contiene tutti i file sorgente Java organizzati in package:

2.1. Package “**Eseguibili**”: contiene le classi principali per l'esecuzione dell'applicazione

2.1.1. Package “**Client**”: implementa la logica lato client

- 2.1.1.1. Printer.java: responsabile della stampa asincrona dei messaggi sulla console
- 2.1.1.2. Receiver.java: thread che gestisce la ricezione dei messaggi TCP ricevuti dal server
- 2.1.1.3. ReceiverUDP.java: thread che gestisce la ricezione dei messaggi UDP ricevuti dal server

2.1.2. Package “**Main**”: contiene le classi con i metodi main per l'avvio delle applicazioni

- 2.1.2.1. ClientMain.java: classe principale dell'applicazione client
- 2.1.2.2. ServerMain.java: classe principale dell'applicazione server

2.1.3. Package “**Server**”: implementa la logica lato server

- 2.1.3.1. SocketUDPValue.java: classe di utilità che rappresenta indirizzo IP e porta UDP di un singolo client
- 2.1.3.2. TimeoutHandler.java: thread che monitora le attività e l'eventuale time-out del client associato ad un worker
- 2.1.3.3. Tupla.java: classe di utilità per memorizzare la coppia (password, isLoggedIn) associata ad ogni username
- 2.1.3.4. Worker.java: classe che implementa il thread worker che gestisce le richieste del client a lui associato

2.1.4. Package “**GsonClasses**”: contiene le classi per la serializzazione/deserializzazione JSON secondo le specifiche del protocollo

- 2.1.4.1. GsonMess.java: classe generica di ricezione di un messaggio dal client
- 2.1.4.2. Values.java: classe astratta per i valori nei messaggi ricevuti
- 2.1.4.3. Package “**Commands**”: sottoclassi concrete di Values per i diversi tipi di comando
 - 2.1.4.3.1. GsonCancelOrder.java: comando richiesta cancellazione ordine
 - 2.1.4.3.2. GsonLimitStopOrder.java: comando inserimento LimitOrder o StopOrder
 - 2.1.4.3.3. GsonMarketOrder.java: comando richiesta MarketOrder
 - 2.1.4.3.4. GsonPriceHistory.java: comando richiesta storico dei prezzi
 - 2.1.4.3.5. GsonTrade.java: rappresenta un trade eseguito per la persistenza nello storicoOrdini.json

- 2.1.4.3.6. GsonUpdateCredentials: comando aggiornamento credenziali
- 2.1.4.3.7. GsonUser.java: comando registrazione e login di un'utente
- 2.1.4.4. Package “**Responses**”: classi per le risposte da parte del server al client
 - 2.1.4.4.1. GsonResponse.java: rappresenta una risposta generica
 - 2.1.4.4.2. GsonResponseOrder.java: rappresenta una risposta ad un ordine
- 2.2. Package “**JsonFile**”: contiene i file JSON per la persistenza dei dati nel sistema
 - 2.2.1. orderBook.json: persiste lo stato corrente dell'order book
 - 2.2.2. storicoOrdini.json: persiste tutti i trade eseguiti per analisi storiche
 - 2.2.3. userMap.json: persiste la mappa degli utenti registrati
- 2.3. Package “**OrderBook**”: contiene le classi per la gestione dell'order book e delle operazioni di trading
 - 2.3.1. DayPriceData.java: classe di utilità per aggregare dati di prezzo giornalieri durante l'analisi dello storico degli ordini
 - 2.3.2. OrderBook.java: classe che implementa la struttura dati dell'order book
 - 2.3.3. OrderValue.java: classe che rappresenta gli ordini aggregati per uno specifico livello di prezzo nell'order book
 - 2.3.4. StopValue.java: classe che rappresenta un ordine di tipo stopOrder nella stopQueue
 - 2.3.5. TradeNotifyUDP.java: classe utilizzata per le notifiche UDP inviate al client
 - 2.3.6. UserValue.java: classe che rappresenta un singolo ordine di un'utente all'interno di un OrderValue
- 2.4. Package “**Varie**”: contiene classi varie non categorizzabili nei precedenti package
 - 2.4.1. Ansi.java: contiene le costanti dei codici ANSI che permettono colorazione e formattazione del testo nella console.
- 2.5. “**client.properties**”: contiene le informazioni per la configurazione del client
- 2.6. “**server.properties**”: contiene le informazioni per la configurazione del server
- 3. “**/target**”: contiene i file .class generati dalla compilazione con Maven e i file JAR delle applicazioni client e server
- 4. “**dependency-reduced-pom.xml**”: versione ridotta del pom.xml
- 5. “**pom.xml**”: contiene le informazioni e le dipendenze del progetto per la compilazione tramite Maven

PROTOCOLLI DI COMUNICAZIONE

Il progetto implementa due protocolli di comunicazione:

1. **TCP**: utilizzato per la quasi totalità del progetto durante la comunicazione client-server i quali si scambiano messaggi di richiesta/risposta secondo il corretto formato JSON richiesto. La connessione viene instaurata all'avvio del client e viene mantenuta fino ad una sua richiesta di chiusura oppure in caso di richiesta di terminazione dal server, indipendentemente se un client ha effettuato login o no.
2. **UDP**: utilizzato per notificare in modo asincrono gli user interessati dalle esecuzioni, in modo parziale o totale, dei propri ordini inseriti precedentemente nell'order book.

ARCHITETTURA DEL SERVER

Il server implementa un'architettura multithread per gestire le connessioni con i client e le operazioni di inserimento/evasione degli ordini. Il processo iniziale (ServerMain) ha il compito di avviare il servizio ed inizializzare le strutture dati condivise, caricando quelle necessarie e disponibili dalla memoria, rispettivamente dai file json e dal file di configurazione. Si occupa, inoltre, di creare un thread pool e di accettare le richieste di connessione TCP in ingresso, avviando per ognuna di esse un thread Worker. In caso di chiusura, si occupa di gestirla tramite un'handler dedicato.

Vediamo ora nello specifico i thread che compongono il server:

- **ShutdownHook**: si occupa della chiusura del servizio, interrompendo l'accettazione di nuove connessioni. Una volta ricevuto il segnale, segnala ai worker attivi di terminare le loro operazioni e termina il thread pool.

- **Worker:** gestisce l'intero ciclo di vita della comunicazione con un singolo client sulla connessione TCP persistente. Ogni worker prende in input tutte le informazioni sulle connessioni e le strutture già istanziate dal ServerMain, come userMap e orderBook. Possiede una propria struttura sharedState, condivisa con il TimeoutHandler, che memorizza le flag di stato e il timestamp dell'ultima attività del client. Si occupa di:
 - Gestire le richieste di autenticazione
 - Ricevere e deserializzare i comandi JSON inviati dal client, controllando la semantica
 - Interagire con le strutture dati condivise in modalità thread-safe per eseguire le operazioni (inserimento, matching e cancellazione)
 - Inviare risposte al client
 - Verificare ogni volta che viene inserito un nuovo ordine se ci sono stopOrder da eseguire
- **TimeoutHandler:** associato ad un worker, si occupa di gestire la politica di logout automatico per inattività, monitorando il tempo trascorso dall'ultima operazione ricevuta dal client associato. Se l'intervallo supera la soglia predefinita (memorizzata e letta dal file di configurazione per una facile modifica), l'handler forza la chiusura della connessione TCP, notificando il client ed effettuando il logout dell'utente, così da liberare risorse sul server.

ARCHITETTURA DEL CLIENT

Il client, come il server, implementa un'architettura multithread per gestire contemporaneamente, l'interazione con l'utente, la ricezione di risposte sincrone dal server tramite TCP e la ricezione di notifiche asincrone tramite UDP, garantendo un'output ordinato e ben formattato sulla console.

Il processo principale (ClientMain) avvia l'applicazione stabilendo una connessione TCP con il server e avviando i thread ausiliari. Vediamoli ora nello specifico:

- **ClientMain:** si occupa di gestire l'interfaccia a riga di comando CLI.
 - Legge i comandi inseriti dall'utente, assicurandosi tramite una regex che il comando rispetti la corretta sintassi (il controllo semantico poi spetterà al server).
 - Serializza le richieste nel formato json specificato
 - Invia al server le richieste attraverso la connessione TCP persistente
- **Receiver:** si occupa di stare in ascolto sulla connessione TCP. Riceve le risposte, le deserializza e in base a ciò effettua determinate azioni. In caso di stampa, passa la risposta al thread Printer che si occuperà di farla visualizzare sulla console dell'utente.
- **ReceiverUDP:** si occupa di stare in ascolto sulla porta UDP dedicata, il cui numero viene inviato al client tramite TCP una volta effettuato il corretto login. È responsabile di ricevere le notifiche asincrone da parte del server, di deserializzarle e di passarle al thread Printer per la stampa.
- **Printer:** si occupa di prelevare i singoli messaggi dalla coda di stampa, riempita man mano da tutti i thread dell'applicazione client, così da stamparli sulla console in modo ordinato di ricezione e senza sovrapposizioni tra loro. Il risultato è una stampa thread-safe e ordinata che garantisce una facile leggibilità dell'interfaccia utente. Anche gli eventuali messaggi di errore vengono stampati da lui.
- **ShutdownHook:** si occupa di effettuare una corretta terminazione del client in caso di ricezione di comando CTRL-C, inviando il comando "logout()" al server e gestendola come una terminazione volontaria.

Il ClientMain, in caso di ricezione di risposta positiva dal comando "logout()" si occupa di terminare tutti gli altri thread attivi.

Ogni client possiede una struttura condivisa tra tutti i thread che memorizza flag di stato dell'applicazione e la porta UDP per ricevere le notifiche asincrone.

SINCRONIZZAZIONE E CONCORRENZA

Sfruttando il multithreading è necessario controllare che non ci siano accessi a sezioni critiche da parte di più thread contemporaneamente. Per questo sono state utilizzate apposite strutture dati thread-safe e primitive di sincronizzazione:

- **ConcurrentHashMap**: struttura dati concorrente che implementa l'interfaccia Map, permettendo a più thread di leggere e scrivere contemporaneamente senza bloccare l'intera struttura, usando meccanismi di locking a grana fine e operazioni atomiche. Nel progetto viene utilizzata per memorizzare la userMap, dove la chiave corrisponde all'username, mentre il valore è un'oggetto di tipo Tupla contenente la coppia {password,isLogged}
- **ConcurrentSkipListMap**: struttura dati concorrente che fornisce una versione thread-safe della SkipList. È una mappa ordinata che mantiene le chiavi in ordine crescente, supporta accessi e modifiche concorrenti senza bisogno di sincronizzazione esterna. Nel progetto viene usata per implementare la socketMapUDP che contiene le associazioni utente -> indirizzo/porta UDP, nell'askMap e nella bidMap (in quest'ultima in ordine decrescente) dell'order book.
- **ConcurrentLinkedQueue**: struttura dati non bloccante che rispetta l'ordinamento FIFO, qualità necessaria per la gestione della stopQueue, garantendo la corretta priorità degli ordini. Nel progetto, oltre per la stopQueue, viene utilizzata per memorizzare i worker assegnati ad un client.
- **synchronized**: è una primitiva di sincronizzazione utilizzata per proteggere sezioni critiche di codice, consentendo l'accesso ad un solo thread alla volta. Viene impiegata principalmente nelle funzioni che agiscono sull'order book.
- **volatile**: è una primitiva di sincronizzazione che garantisce la visibilità immediata delle modifiche a variabili condivise tra thread. Nel progetto viene usata nel thread Printer del client, così da bloccare velocemente la stampa in caso di terminazione dell'applicazione.

ESECUZIONE DEL PROGETTO

Il progetto include due componenti principali, ovvero Server e Client, che devono essere avviati separatamente.

Prerequisiti:

- ambiente Java JDK/JRE versione 8.0 o superiore

Per avviare il SERVER

1. disporsi sulla directory "cross"
2. eseguire il comando "java -jar target/cross-server.jar"

Per avviare il CLIENT

1. disporsi sulla directory "cross"
2. eseguire il comando "java -jar target/cross-client.jar"

Compilazione da Sorgente (opzionale)

Il progetto è configurato per essere compilato tramite Maven.

ATTENZIONE: questa procedura è necessaria solo se sono state apportate modifiche al codice sorgente! Nella directory cross del progetto sono stati lasciati apposta i file di compilazione e delle dipendenze che sfrutta Maven.

Per compilare:

1. disporsi sulla directory cross
2. digitare il comando "mvn clean package" per compilare
3. una volta terminato il processo, si potrà eseguire secondo le precedenti istruzioni di avvio