Documentazione Cross

Paola Cerminara 634260

13 Maggio 2025

1. Introduzione

La presente relazione descrive la struttura e l'organizzazione del progetto CROSS, un sistema client-server sviluppato in Java per implementare un servizio di orderBook per il trading di criptovalute. Il progetto segue le specifiche fornite nel documento "Progetto LAB2425Versione1.3.pdf" ed è stato strutturato in modo da garantire una separazione logica tra le varie componenti del sistema.

2. Scelte Progettuali

2.1 Struttura Organizzativa

Il progetto è organizzato in una struttura gerarchica di cartelle che rispecchia la natura dell'applicazione. Le componenti principali sono:

- Cartella *src*: include tutti i file sorgente Java organizzati in package logici:

1. Package *Eseguibili*

   Contiene le classi principali per l'esecuzione dell'applicazione

   1.1. Client: Implementa la logica lato client

      1.1.1. Printer.java: Garantisce la visualizzazione dei messaggi e delle informazioni all'utente

      1.1.2. ReceiverClient.java: Gestisce la ricezione di dati dal server sulla connessione TCP

      1.1.3. UDPReceiverClient.java: Gestisce la ricezione delle notifiche asincrone UDP

   1.2. Main: Contiene i punti di ingresso dell'applicazione

      1.2.1. MainClient.java: Classe principale per l'avvio del client

      1.2.2. MainServer.java: Classe principale per l'avvio del server

   1.3. Server: Implementa la logica lato server

      1.3.1. DailyParameters.java: Gestisce i parametri giornalieri per lo storico dei prezzi

      1.3.2. SockMapValue.java: Rappresenta i valori della mappa usata per mantenere i dati necessari per l'invio della notifica UDP degli utenti online

      1.3.3. TimeoutHandler.java: Gestisce i time-out di inattività delle connessioni

      1.3.4. Tupla.java: Rappresenta i valori della mappa usata per mantenere password e stato di login di ogni utente registrato

      1.3.5. Worker: Implementa il thread worker che gestisce le richieste del cliente

2. Package *GsonClasses*

   Contiene le classi per la serializzazione e deserializzazione JSON utilizzando la libreria Gson

   2.1. Commands: Classi che rappresentano i comandi inviati dal client

      2.1.1. GsonAskHistory.java: Comando per richiedere lo storico dei prezzi

      2.1.2. GsonCredentials.java: Comando per gestire l'aggiornamento delle credenziali

      2.1.3. GsonLimitStopOrder.java: Comando per inserire un LimitOrder o uno StopOrder

      2.1.4. GsonMarketOrder.java: Comando per inserire un MarketOrder

      2.1.5. GsonTrade.java: Rappresenta un trade letto dallo storico degli ordini

      2.1.6. GsonUser.java: Rappresenta un utente con username e password

   2.2. Responses: Classi che rappresentano le risposte inviate dal server

      2.2.1. GsonOrderBook.java: Risposta contenente l'orderBook o la lista degli StopOrders

      2.2.2. GsonResponse.java: Risposta generica del server

      2.2.3. GsonResponseOrder.java: Risposta specifica per operazioni sugli ordini

   2.3. GsonMessage.java: è la classe base per tutti i messaggi JSON

   2.4. Values.java: Gestisce i valori generici nei messaggi JSON

3. Package *JsonFile*

   Contiene i file JSON utilizzati per la persistenza dei dati

3.1. orderBook.json: Persistenza dell'orderBook
3.2. storicoOrdini.json: Persistenza dello storico degli ordini
3.3. userMap.json: Persistenza degli utenti registrati

4. Package *OrderBook*
   Contiene le classi che implementano la logica dell'orderBook
   4.1. BookValue.java: Rappresenta i campi della askMap o della bidMap
   4.2. OrderBook.java: Implementa la struttura dati dell'orderBook
   4.3. StopValue.java: Rappresenta i campi della coda degli StopOrders
   4.4. TradeUDP.java: Rappresenta il trade inviato al client tramite la notifica UDP
   4.5. UserBook.java: Rappresenta i campi delle liste di utenti che hanno piazzato un ordine di bid o ask

- Cartella bin: include i file (.class) generati dalla compilazione, mantenendo la stessa struttura delle cartelle del codice sorgente.
- **Cartella lib**: ospita la libreria gson-2.11.0.jar di Google, utilizzata per la gestione dei dati in formato JSON.
- **Cartella manifest**: include i file manifest che fungono da istruzioni di avvio per la JVM. In questa cartella sono presenti due file manifest distinti che puntano a entry point differenti: MainServer e MainClient. I manifest inoltre specificano la dipendenza esterna che l'applicazione deve caricare per funzionare correttamente: libreria gson-2.11.0.jar.

Oltre alle cartelle sopra descritte, nella directory principale, si trovano i file di configurazione client.properties e server.properties, ed i file ServerApp.jar e ClientApp.jar utilizzati per avviare l'applicazione.

2.2 Meccanismi di Comunicazione

Il progetto implementa due meccanismi di comunicazione tra client e server:

- TCP: Protocollo maggiormente utilizzato per la comunicazione Client-Server mediante il quale il Client può inviare richieste al Server. La connessione viene instaurata all'avvio del Client e persiste fino alla sua chiusura o in caso di chiusura del Server
- UDP: Protocollo utilizzato per notificare in modo asincrono gli utenti dell'esecuzione delle transazioni legate agli ordini da loro piazzati nell'orderBook

2.3 Gestione della sincronizzazione

Per gestire la concorrenza in un ambiente multithreading si è fatto uso di diverse strutture dati thread-safe, descritte di seguito:

- ConcurrentSkipListMap: struttura dati che implementa un'interfaccia Map ordinata basata su skip list. Offre eccellenti prestazioni per accessi concorrenti, garantendo operazioni di inserimento, rimozione e ricerca in tempo logaritmico. Nel progetto è utilizzata per organizzare le sezioni bid e ask dell'orderBook e per mantenere i dati relativi alla connessione UDP di ciascun client (socketMap).
- ConcurrentHashMap: struttura dati che implementa l'interfaccia Map ottimizzata per gestire accessi concorrenti. All'interno del progetto è utilizzata per memorizzare gli utenti registrati e le relative informazioni. Ogni chiave rappresenta l'username dell'utente, mentre il valore è un oggetto Tupla corrispondente alla coppia password-stato di login.
- ConcurrentLinkedQueue: struttura dati che implementa una coda non bloccante basata su una lista concatenata. Fornisce operazioni di inserimento e rimozione altamente efficienti che possono essere eseguite contemporaneamente da più thread senza necessità di sincronizzazione esplicita. Nel progetto viene impiegata per mantenere gli stopOrders inseriti dagli utenti e per la lista che tiene traccia di tutti i worker attivi.

3. Organizzazione dell'OrderBook

3.1 Descrizione della Struttura

L'orderBook è il componente fondamentale del sistema di trading implementato, che supporta diversi tipi di ordini e garantisce l'efficienza e la coerenza nelle operazioni. La sua implementazione si basa su più strutture dati:

1) askMap: è una mappa chiave-valore (ConcurrentSkipListMap) in cui la chiave è un intero che rappresenta il prezzo, mentre il valore è un oggetto BookValue. La mappa memorizza gli ordini di vendita ordinati in modo crescente rispetto al prezzo.
2) Spread: è un intero che rappresenta la differenza tra il miglior prezzo di acquisto (massimo della bidMap) e il miglior prezzo di vendita (minimo della askMap)
3) LastOrderID: è un contatore utilizzato per assegnare identificatori univoci agli ordini
4) StopOrders: è una coda (ConcurrentLinkedQueue) i cui elementi sono oggetti StopValue che descrivono gli stopOrders inseriti dai clienti
5) bidMap: è una mappa analoga alla askMap, ma dedicata agli ordini di acquisto. Gli ordini sono memorizzati in ordine decrescente di prezzo

Per motivi implementativi, è stato deciso di persistere i dati di askMap e bidMap nel file orderBook.json alla chiusura del server. Gli stopOrders non vengono salvati, quindi la coda viene azzerata a ogni avvio del server.
La classe BookValue rappresenta le informazioni aggregate per un dato prezzo, includendo:

- size: quantità totale degli ordini a quel prezzo;
- total: valore complessivo (prezzo × quantità);
- userList: coda di oggetti UserBook relativi agli ordini individuali.

Ogni oggetto UserBook descrive un singolo ordine e contiene:

- size: dimensione dell'ordine;
- username: utente che ha inserito l'ordine;
- orderID: identificatore univoco dell'ordine.

3.2 Gestione degli Ordini

L'orderBook supporta tre categorie di ordini:

1. LimitOrder: ordini che vengono eseguiti solo ad un prezzo specifico o uno migliore. Se non eseguibili immediatamente rimangono nell'orderBook. Per la loro gestione vengono implementati i metodi tryBidOrder e tryAskOrder, che processano rispettivamente gli ordini di acquisto e di vendita, cercando corrispondenze nella mappa di tipo opposto. Qualora l'ordine non venga soddisfatto completamente, la parte residua viene caricata nella mappa corrispondente tramite i metodi loadBidOrder o loadAskOrder.
2. MarketOrder: ordini che vengono eseguiti immediatamente al miglior prezzo di mercato. Il metodo tryMarketOrder ne gestisce l'elaborazione, controllando la disponibilità di liquidità nell'orderBook e abbinando l'ordine fino al suo completamento. In caso di esecuzione parziale o fallimento, il sistema notifica l'utente con un messaggio di errore.
3. StopOrder: ordini che vengono attivati solo quando il prezzo di mercato raggiunge lo stopPrice, dopodiché vengono eseguiti come MarketOrder. Sono immediatamente aggiunti alla coda dedicata e gestiti dal metodo checkStopOrders, che viene invocato ogni volta che l'orderBook o la coda degli stopOrders subisce modifiche. Quando la condizione sul prezzo è soddisfatta, l'ordine viene trasformato ed eseguito tramite tryMarketOrder.

3.3 Algoritmo di matching

Il cuore dell'orderBook è l'algoritmo di matching, che abbina gli ordini di acquisto e vendita. Questo è implementato nel metodo tryMatch, il quale opera secondo la seguente logica:

a) Itera sulla lista di ordini della controparte (bid per un ordine di ask e viceversa)
b) Per ogni ordine, verifica che l'utente sia diverso (per evitare self-trading)
c) Confronta le dimensioni degli ordini ed esegue uno dei seguenti casi:
    - Se l'ordine della controparte è più grande, l'ordine dell'utente viene completato e l'ordine della controparte viene ridotto
    - Se l'ordine dell'utente è più grande, l'ordine della controparte viene completato e rimosso, e l'ordine dell'utente viene ridotto
    - Se gli ordini hanno la stessa dimensioni, entrambi vengono completati e l'ordine della controparte viene rimosso

Dopo ogni match, gli utenti coinvolti vengono notificati via UDP con i dettagli della transazione. Il metodo restituisce la quantità residua dell'ordine originale, nel caso in cui non sia stato possibile eseguirlo completamente.

3.4 Funzionalità Aggiuntive

L'orderBook include funzionalità avanzate per la gestione degli ordini, tra cui la cancellazione, l'aggiornamento dello stato e la notifica agli utenti.
La cancellazione degli ordini è gestita tramite il metodo cancelOrder, che consente a un utente di rimuovere un ordine solo se ne è il proprietario. L'ordine può essere rimosso solo se è ancora presente nell'orderBook oppure nella coda degli stopOrders.
Il mantenimento della coerenza dei dati è affidato al metodo updateOrderBook, che aggiorna dinamicamente le informazioni relative a size, total e spread.
Per quanto riguarda le notifiche, il metodo notifyUser invia notifiche UDP agli utenti i cui ordini sono stati eseguiti. Il metodo esegue le seguenti operazioni in sequenza:

- Esegue una scansione della socketMap per trovare la porta e l'indirizzo associati all'utente
- Controlla se l'utente è online verificando che siano stati trovati sia l'indirizzo che la porta
- Se l'utente non è online, la notifica non viene inviata poiché non si hanno le informazioni di destinazione
- Se l'utente è online:
    - Crea un oggetto TradeUDP contenente tutti i dettagli dell'ordine
    - Serializza l'oggetto in formato JSON usando la libreria Gson
    - Converte la stringa JSON in un array di byte
    - Crea un pacchetto UDP con i dati e le informazioni di destinazione
    - Invia il pacchetto attraverso un socket UDP

4. Panoramica dei Thread Avviati

Per garantire reattività ed efficienza, il progetto adotta un'architettura multithread sia nella gestione delle connessioni Client-Server sia nell'esecuzione asincrona delle operazioni interne.

4.1 Thread Lato Client

Il client implementa un'architettura multi-thread per gestire in modo efficiente le comunicazioni con il server e l'interazione con l'utente. Questo approccio consente di separare le diverse responsabilità dell'applicazione, permettendo operazioni concorrenti.
Il thread principale, MainClient, si occupa di gestire l'interfaccia utente a riga di comando e di inviare comandi al server. Al suo avvio, il Main crea tre nuovi thread:

- ReceiverClient: gestisce la ricezione dei messaggi TCP dal server. In particolare, esso si occupa di leggere continuamente dallo stream di input, parsare i messaggi JSON ricevuti, aggiornare le variabili condivise e mostrare all'utente i messaggi ricevuti
- UDPReceiverClient: gestisce la ricezione di messaggi UDP asincroni dal server. In particolare, esso si occupa di ascoltare continuamente sulla porta UDP, deserializzare i messaggi JSON in oggetti TradeUDP e notificare l'utente sullo stato degli ordini processati
- Printer: è un thread daemon che gestisce la stampa asincrona dei messaggi sulla console. Questo thread mantiene una coda ordinata di messaggi da visualizzare e li stampa in modo da evitare l'interruzione dell'input durante la visualizzazione dei messaggi.

Per coordinare l'esecuzione concorrente, il client impiega diversi meccanismi di sincronizzazione:

- BlockingQueue, utilizzata nella classe Printer per gestire i messaggi in modo asincrono
- Interrupt, utilizzato per terminare i thread in modo controllato
- Classe SharedData, utilizzata per condividere risorse tra thread. Le variabili contenute sono prevalentemente di tipo AtomicBoolean in modo da garantire l'accesso atomico e thread-safe.

4.2 Thread Lato Server

Il server implementa un'architettura multi-thread per gestire le connessioni client e le operazioni di trading in modo efficiente. Il thread principale (MainServer) ha il compito di avviare il servizio ed inizializzare le strutture dati condivise prelevando quelle necessarie dalla memoria. Il thread inoltre crea un CachedThreadPool ed accetta le connessioni TCP in ingresso avviando un thread Worker per ogni cliente che si connette. Infine, gestisce lo shutdown controllato del sistema tramite un hook di terminazione.

- Worker: gestisce un singolo client connesso ed è responsabile per:
    - Gestire la comunicazione TCP bidirezionale con il client
    - Elaborare i comandi JSON ricevuti
    - Avviare un TimeoutHandler per monitorare l'attività del client
    - Eseguire operazioni sull'orderBook in risposta ai comandi del client
    - Aggiornare le strutture dati condivise (userMap, orderBook, socketMap)
- TimeoutHandler: è un thread avviato da ogni Worker, che monitora l'attività del client associato. Il thread fa un controllo ogni 5 secondi e si preoccupa di:
    - Chiudere le connessioni dopo 10 minuti di inattività
    - Gestire le eccezioni per gli utenti che attendono uno stopOrder
    - Sincronizzarsi con l'orderBook per monitorare gli stopOrders
- ShutdownHook: è un thread speciale aggiunto come ShutdownHook per consentire la terminazione del servizio in maniera sicura in qualsiasi momento. Il thread viene attivato quando il server riceve un segnale di terminazione. Prima di chiudere, notifica tutti i clienti connessi della chiusura imminente e chiude correttamente i socket e il pool di thread.

Le istanze Worker condividono la userMap, che tiene traccia di tutti i clienti registrati con la relativa password e lo stato di login, l'orderBook e la socketMap che tiene traccia dei dati per la comunicazione UDP di ogni utente loggato. Inoltre, i Worker si sincronizzano con l'orderBook e si aggiornano reciprocamente quando avvengono cambiamenti rilevanti. Infine, ciascun TimeoutHandler comunica con il rispettivo Worker tramite uno stato condiviso (SharedState).

Questo tipo di architettura consente la gestione simultanea di più client garantendo la sincronizzazione delle operazioni sull'orderBook.

5. Esecuzione del Progetto

Il progetto include le componenti Client e Server, che devono essere avviati separatamente.

5.1 Avvio del Server e del Client

Per l'avvio del server è necessario posizionarsi nella directory Cross ed eseguire il comando:

java -jar ServerApp.jar

Per l'avvio del client è necessario posizionarsi nella directory Cross ed eseguire il comando:

java -jar ClientApp.jar

5.2 Utenti e Ordini esistenti

Il progetto viene fornito con dei file .json contenenti dati di esempio utili per testare il sistema.

UserMap.json contiene i seguenti utenti registrati:

- Username: mattia, Password: psw
- Username: matteo, Password: psw
- Username: andrea, Password: psw
- Username: sara, Password: psw

OrderBook.json contiene i seguenti ordini registrati:

ASK (vendita)			BID (acquisto)		
Prezzo	Quantità	Utente	Prezzo	Quantità	Utente
20.000	3	mattia	10.000	1	sara