Laboratorio III a.a. 2024/2025

Candidato: Benetti Alberto
Matricola 660317

Struttura progetto

I files .java necessari per l'esecuzione del progetto sono divisi in 4 packages:

- *crossClient* ⇒ contiene i file sorgenti per l'esecuzione del client, in paticolare ha ClientMain.java che rappresenta il main del client e ClientCROSS.java che formula le richieste da inviare poi al server
- *crossServer* ⇒ contiene i sorgenti per eseguire il lato server dell'applicazione, in particolare ServerMain.java, il quale rimane in attesa di client che cercano di connettersi al servizio e, una volta ricevuti genera un'istanza di CrossExecutor per comunicare con il thread in un altro thread
- *orderTypes* ⇒ contiene le classi necessarie per definire gli ordini e l'OrderBook, necessario per memorizzare gli ordini non ancora evasi. Questo package viene utilizzato dal server per la gestione degli ordini
- *requestMessages* ⇒ contiene le classi necessarie per inviare e ricevere oggetti Json che rappresentano richieste (dal client per il server) e risposte (dal server verso il client)

Le due classi principali del progetto sono **ClientMain.java** e **ServerMain.java**
In più nel progetto è presente la cartella ***configs***, la quale contiene i file di configurazione del server e del client e la cartella ***libs*** contenente la libreria GSON

Strutture dati utilizzate

Dal lato del server sono state utilizzate le seguenti strutture:

- *CopyOnWriteArrayList<OrdineEvaso> storedOrders (storicoOrdini nell'OrderBook)* utilizzata per contenere soltanto gli ordini che sono stati memorizzati nel file json contenente lo storico degli ordini evasi.
- *ConcurrentHashMap* ⇒ questa è stata utilizzata in varie occasioni e con tipi diversi:
    - *<String,String> users* ⇒ utilizzata per mantenere come chiave lo username di ogni utente che si è registrato e come valore associato la relativa password
    - *<String, UserInfo> loggedInUsers* ⇒ per tenere traccia, in ogni momento, degli utenti che hanno eseguito l'accesso. Questa struttura dati ha come chiave sempre lo username dell'utente a cui è riferita ma vi associa un'istanza di UserInfo al fine di mantenere l'InetAddress di quell'utente e la porta sul quale questo si è messo in ascolto di eventuali messaggi udp di conferma in caso di ordine evaso.
- *ConcurrentLinkedQueue<OrdineEvaso> ordiniEvasi* ⇒ utilizzata per memorizzare tutti gli ordini che risultano evasi ma che non sono ancora stati memorizzati all'interno del file Json persistente
- PriorityBlockingQueue ⇒ questa struttura dati è risultata fondamentale per la gestione dell'OrderBook, in quanto è stata utilizzata per tutti gli ordini da mantenere nell'order book non ancora evasi. È stato utilizzato questo oggetto in quanto era

necessario mantenere delle liste ordinate in base al prezzo specificato con gli ordini. In particolare si hanno:

- le code di <LimitOrder>, per gli ordini di tipo Limit di vendita (caso di codaLimitAsk) e per quelli di acquisto (caso di codaLimitBid)
- code di StopOrder,per gli ordini di tipo Stop ricevuti come vendita (caso di codaStopAsk) e per quelli di acquisto (caso di codaStopBid)

Logica Server

Il server, una volta avviato ricava dal file serverConfig.properties le informazioni relative al numero di porta al quale si deve mettere in ascolto per futuri client, i path necessari per risalire ai file storicoOrdini.json e users.json ed il timeout per il quale, nel caso in cui non riceva richieste di connessione per quel determinato tempo, termini la sua esecuzione.
Da storicoOrdini.json ricava tutti gli ordini passati salvati in memoria persistente e schedula un thread con scheduleWithFixedDelay il quale, ogni 30 secondi salva in memoria persistente i nuovi ordini che sono stati evasi.
Lo stesso viene effettuato con il file contenente gli utenti registrati e le relative password.
In più, viene definito un Thread Pool di tipo Cached che ha il compito di eseguire le istanze di CROSSExecutor, delle quali ne viene generata una per ogni client che instaura una connessione con il server

Logica Client

Il client invece riceve input da riga di comando e in base all'opzione scelta dall'utente chiama metodi di ClientCROSS, il quale lavora come una sorta di handler per formulare i json da inviare al server.
All'inizio della sua esecuzione avvia un DatagramSocket sulla porta indicata dall'utente nel file clientConfig.properties (se libera, altrimenti ne sceglie una lui), passa questo numero di porta al server e avvia un thread che si mette in ascolto di eventuali messaggi UDP ricevuti che indicherebbero ordini evasi con successo.

Thread-Safety

Per garantire una corretta sincronizzazione nei thread del server sono utilizzate strutture dati che garantiscano thread safety, come la ConcurrentHashMap, PriorityBlockingQueue, CopyOnWriteArrayList e la ConcurrentLinkedQueue e, in più, alcuni metodi dell'Order Book sono definiti come synchronized, al fine di garantire una maggior sicurezza nell'inserimento di ordini.

Istruzioni per la compilazione

Grazie al Makefile che è stato generato è possibile compilare tutto grazie al comando **make all**.
Se invece vogliamo soltanto ripulire i file e le directory date dalla compilazione è presente il comando **make clean**
Infine per eseguire il server è necessario il comando **make run-server**, mentre per eseguire il client è necessario il comando **make run-client**