Relazione Sintetica del Sistema CROSS

Reti e Laboratorio: Modulo Laboratorio 3 - Progetto di Fine Corso A.A. 2024/25 - Corsi A - Nuovo Ordinamento

Yuchen Cao
Matricola: 616906

1 Schema generale dei thread attivati

Lato server

- **Main thread - `ServerMain`**
  È il primo thread attivo all'avvio del server.
- **Thread pool TCP - `executor`**
  Creato con `Executors.newCachedThreadPool()` dentro `ServerMain`, viene utilizzato per gestire le connessioni TCP in entrata da parte dei client. Ogni volta che un client si connette, viene assegnato un thread del pool per gestire la comunicazione. Il pool:
  - riutilizza i thread già creati, se disponibili;
  - termina i thread inattivi per più di 60 secondi.
- **Thread UDP listener - `udpThread`**
  Un singolo thread creato in `ServerMain` esegue la classe Runnable `UdpListenerServer`, che resta in ascolto su una porta UDP (configurato come 54321). Quando un client effettua il login correttamente, invia un pacchetto UDP contenente il proprio `username`; questo thread registra l'associazione `username - indirizzo UDP` per le notifiche future. È un daemon thread, quindi termina automaticamente alla chiusura del server.
- **Thread pool per notifiche UDP - `udpExecutor`**
  Un secondo thread pool creato in `UdpNotifier` con `Executors.newCachedThreadPool()` viene usato da `UdpNotifier` per inviare notifiche asincrone ai client. Il modello è non bloccante e garantisce l'invio parallelo a più utenti.

Lato client

- **Main thread - `ClientMain`**
  È il primo thread attivo all'avvio del client.
- **Thread UDP listener - `udpThread`**
  Dopo il login, il client invia un pacchetto UDP contenente il proprio `username` al server per registrare l'indirizzo UDP. Successivamente, viene creato un nuovo thread `udpThread` che esegue la classe Runnable `UdpReceiverClient`, il quale:
  - resta in ascolto sul socket UDP;
  - riceve notifiche asincrone inviate dal server;
  - funziona in parallelo al thread principale senza bloccarlo.

2 Definizione delle strutture dati utilizzate

In questo progetto, lato **server** e **client** condividono una serie di strutture dati comuni, utilizzate per la comunicazione e la gestione delle operazioni. Inoltre, ogni lato definisce strutture dedicate al proprio ambito operativo. Di seguito viene fornita una panoramica delle principali.

Strutture dati comuni

Queste strutture vengono serializzate/deserializzate in formato JSON e scambiate tra client e server tramite TCP e UDP (solo `ClosedTradesNotification`). Rappresentano richieste, risposte, dati storici e ordini.

- **Entità dominio**:
  - `Order`: Classe astratta con sottoclassi `LimitOrder`, `MarketOrder`, `StopOrder`.
  - `User`: Rappresenta un utente con nome, password e lo stato di loggedin.
  - `DailyPriceData`: Classe aggregatore di prezzi giornalieri per operazione `getPriceHistory`.
- **Richieste**:
  - `RegisterRequest`, `UpdateCredentialsRequest`, `LoginRequest`
  - `LogoutRequest`, `InsertLimitOrderRequest`, `InsertMarketOrderRequest`, `InsertStopOrderRequest`, `CancelOrderRequest`, `GetPriceHistoryRequest`
- **Risposte**:
  - `OperationResponse1`: Risposta generica con codice e messaggio.
  - `OperationResponse2`: Risposta con ordine ID o -1.
  - `GetPriceHistoryResponse`: Risposta contenente dati aggregati giornalieri.
  - `ClosedTradesNotification`: Notifica asincrona di trade avvenuti, inviata via UDP.

Lato server

- **`ServerConfig`**
  Carica i parametri da file di configurazione (porte, buffer, timeout ecc.) all'avvio del server.
- **`OrderBook`**
  Contiene la logica principale di gestione del order book e del matching tra ordini. È implementata come singleton ed è interamente protetta da un `ReentrantLock` globale con politica `fair` per garantire l'accesso ordinato da parte di più thread. Mantiene le seguenti strutture dati e operazioni principali offerte:
  - `bidOrders`, `askOrders`: code prioritarie di ordini limite (`LimitOrder`) per lato acquisto e vendita, ordinate per prezzo e timestamp.
  - `bidStopOrders`, `askStopOrders`: code di ordini stop (`StopOrder`) per attivazione condizionata.
  - `activeOrders`: mappa concorrente di tutti gli ordini limite e stop, indicizzati per ID.
  - `addLimitOrder()`, `addMarketOrder()`, `addStopOrder()`: inserisce un ordine, esegue il matching e attiva ordini stop se necessario.
  - `checkTriggeredStopOrders()`: controlla se ci sono ordini stop attivabili in base ai prezzi attuali, li converte in ordini market e esegue il matching.
  - `removeOrderFromBook()`: rimuove un ordine (se ancora presente nelle rispettive code).
  - `load()`, `persist()`: recupera o persiste lo stato dell'intero `OrderBook` da/per file JSON.
  - `notifyUsers()`: invia notifiche UDP asincrone agli utenti coinvolti in un trade, tramite `UdpNotifier`.

La persistenza avviene immediatamente dopo ogni modifica allo stato, garantendo robustezza in caso di crash o riavvio del server.

- **`OrderHistory`**
  Registra tutti gli ordini ricevuti dal sistema, inclusi quelli `MarketOrder` rifiutati, utilizzando una `LinkedHashMap` sincronizzata che mantiene l'ordine di inserimento. Ogni ordine viene immediatamente salvato su file JSON tramite il metodo `persist()` al momento della registrazione con `addOrder()`. All'avvio del server, lo storico può essere ricostruito tramite il metodo `load()`.

- **`TradeHistory`**
  Gestisce la registrazione permanente delle transazioni concluse (`TradeInfo`) utilizzando una `LinkedBlockingQueue` come struttura interna. Ogni volta che nuovi trade vengono aggiunti tramite `addTrades()`, la coda viene aggiornata e il contenuto viene immediatamente salvato su file JSON tramite `persist()`. All'avvio del server, lo storico può essere ricaricato da file tramite il metodo `load()`.

- **`OrderIdGenerator`**
  Generatore di ID univoci per ordini, basato su un contatore incrementale thread-safe `AtomicInteger`.

- **`RegisteredUsers`**
  Contiene le informazioni di utenti registrate, con operazioni di lettura/scrittura thread-safe.

- **`UdpListenerServer`**
  Thread UDP in ascolto per la registrazione degli indirizzi UDP associati agli utenti autenticati.

- **`UdpNotifier`**
  Componente che invia notifiche UDP asincrone ai client, utilizzando un `ExecutorService` dedicato.

- **`UserUdpRegistry`**
  Mappa gli `username` agli indirizzi UDP client (`InetSocketAddress`) registrati.

Lato client

- **`ClientConfig`**
  Carica configurazioni (host, porte, buffer) da file esterno all'avvio del client.

- **`UdpReceiverClient`**
  Thread dedicato alla ricezione di notifiche UDP asincrone inviate dal server.

3 Primitive di sincronizzazione utilizzate

- `OrderBook` utilizza un `ReentrantLock` globale con politica `fair=true` per proteggere tutte le operazioni critiche (inserimento, matching, rimozione, persistenza, attivazione di stop order). Questa scelta garantisce accesso esclusivo e ordinato (FIFO) tra thread concorrenti, fondamentale in un sistema di trading.

- `OrderHistory` si basa su una `LinkedHashMap` sincronizzata tramite `Collections.synchronizedMap` per garantire accesso thread-safe. L'aggiunta degli ordini tramite `addOrder()` è protetta da `synchronized`, mentre la lettura è sicura per accessi singoli.

- `TradeHistory` utilizza una `LinkedBlockingQueue` come struttura dati thread-safe per memorizzare i trade. Tuttavia, le operazioni composite (come l'aggiunta in blocco e il salvataggio) sono protette da blocchi `synchronized` per garantire coerenza e atomicità.
- `OrderIdGenerator` utilizza un `AtomicInteger` per generare ID univoci in modo non bloccante. Il metodo `getNextOrderId()` è `synchronized` per garantire l'atomicità tra la generazione e la persistenza su file.
- `RegisteredUsers` gestisce l'insieme degli utenti registrati tramite una `ConcurrentHashMap`, che consente letture e scritture concorrenti in modo efficiente. Il metodo `persistUsers()` è
  `synchronized` per evitare conflitti durante l'accesso al file JSON, mentre l'aggiunta di nuovi utenti avviene tramite l'operazione atomica `putIfAbsent()`.
- `UserUdpRegistry` utilizza una `ConcurrentHashMap` per mappare ogni `username` al corrispondente indirizzo UDP. L'accesso è completamente thread-safe e non necessita di sincronizzazione esplicita, poiché sia `put()` che `get()` sono operazioni atomiche in `ConcurrentHashMap`.
- `ExecutorService` viene utilizzato sia per la gestione delle connessioni TCP che per l'invio asincrono delle notifiche UDP. L'uso di un thread pool consente il riutilizzo dei thread esistenti e il controllo centralizzato del carico, evitando problemi legati alla creazione incontrollata di nuovi thread.

4 Istruzioni per la compilazione ed esecuzione del progetto

Questa sezione descrive in modo dettagliato come compilare ed eseguire correttamente il progetto da linea di comando, senza l'uso di ambienti di sviluppo (IDE). Tutte le operazioni sono state testate su sistemi Unix-like (Linux/macOS). In ambienti Windows, è necessario sostituire i separatori di classpath `:` con `;`.

Compilazione

Tutti i codici sorgenti si trovano nella cartella `src/final_project/`. Per compilare i file `.java` e produrre i `.class` nella cartella `bin/`, eseguire il seguente comando:

```
javac -cp src/gson-2.10.1.jar -d bin src/final_project/*.java
```

Creazione dei file .jar eseguibili

Per creare i file JAR eseguibili, assicurarsi di avere i seguenti manifest file:

`manifest_server.txt`, contenuto di seguito

```
Main-Class: final_project.ServerMain
Class-Path: src/gson-2.10.1.jar
```

`manifest_client.txt`, contenuto di seguito

```
Main-Class: final_project.ClientMain
Class-Path: src/gson-2.10.1.jar
```

Successivamente, creare il `server.jar` con il comando:

```
jar cfm server.jar manifest_server.txt -C bin final_project
```

creare il `client.jar` con il comando:

```
jar cfm client.jar manifest_client.txt -C bin final_project
```

Esecuzione

Per avviare le due applicazioni, è sufficiente eseguire i seguenti comandi:

per avviare il client end: `java -jar server.jar`

per avviare il client end: `java -jar client.jar`

Assicurarsi che il file `gson-2.10.1.jar` si trovi nella cartella `src/`, in modo che sia correttamente incluso dal `Class-Path` definito nei manifest.

Librerie esterne

Il progetto utilizza la libreria open source **Gson** (versione 2.10.1) per la serializzazione e deserializzazione JSON. Essa è già inclusa nel progetto nella cartella `src/`.

File di configurazione

I file di configurazione `Config_Client.properties` e `Config_Server.properties` devono trovarsi nella stessa directory dei file `server.jar` e `client.jar`. Essi contengono i parametri come indirizzo IP, porte TCP/UDP, buffer size, ecc.

Sintassi dei comandi per lato client

Il client fornisce un'interfaccia interattiva basata su menu, che guida l'utente nell'esecuzione delle principali operazioni. Al lancio dell'applicazione, viene presentata una schermata con le seguenti opzioni numerate:

- `1. Register` – per registrare un nuovo utente
- `2. Update Credentials` – per aggiornare le credenziali di accesso
- `3. Login` – per accedere e avviare la sessione interattiva.

L'utente deve semplicemente inserire il numero corrispondente all'opzione desiderata. Dopo aver selezionato l'opzione, il sistema richiederà automaticamente i parametri necessari (ad esempio nome utente, vecchia e nuova password, ecc.) in modo guidato. Non è quindi necessario conoscere né ricordare una sintassi specifica: tutte le interazioni avvengono tramite istruzioni testuali a schermo. Dopo il login, si accede automaticamente alla seconda modalità interattiva che consente

il logout, l'inserimento di ordini, la cancellazione e la richiesta dello storico dei prezzi, sempre seguendo le istruzioni proposte dal sistema.

Nota finale

Il progetto è stato progettato per essere eseguibile completamente da linea di comando e non richiede alcuna installazione aggiuntiva oltre a una JDK 11+.