Documentazione Cross

Martini Matteo 636694

21 marzo 2025

Indice

1 Scelte Progettuali

Il progetto è stato realizzato seguendo i principi SOLID per garantire modularità, scalabilità e manutenibilità

1.1 Organizzazione delle unità di codice

La suddivisione intensiva in package garantisce la modularità per le varie funzionalità e migliora la fruibilità del codice. Di seguito verranno elencati i package principali

- **Client**: contiene le task da eseguire lato Client e la classe che verrà istanziata per usufruire del servizio
- **Commands**: contiene tutti i comandi che possono essere utilizzati dagli utenti, insieme alla factory per istanziarli
- **Communication**: contiene i protocolli di comunicazione utilizzati insieme ai tipi di messaggio definiti per la comunicazione
- **Config**: contiene le classi per configurare Client e Server
- **Executables**: contiene ServerMain e ClientMain che eseguiranno rispettivamente Server e Client del servizio Cross
- **JsonAccessedData**: contiene tutti i vari file Json utilizzati per mantenere informazioni persistenti
- **Server**: contiene le task da eseguire lato Server e la classe che verrà istanziata per creare il Server centrale del servizio Cross
- **Utils**: contiene varie classi di utilità che semplificano l'esecuzione del codice

1.2 Comunicazione

La comunicazione definita nel package **Communication** contiene i due protocolli utilizzati:

- **TCP**: Protocollo maggiormente utilizzato per la comunicazione Client-Server mediante il quale il Client può inviare richieste al Server. La conessione viene instaurata all'avvio del Client e persiste fino alla chiusura dello stesso o di problemi che causano la chiusura del Server
- **UDP**: Protocollo utilizzato per notificare gli utenti della finalizzazione delle transazioni legate agli ordini da loro piazzati nell'orderbook

1.3 Gestione di Orderbook e Registrazioni

L'orderbook viene realizzato mediante un file Json diviso in due campi principali **askMap** e **bidMap** che contengono i relativi ordini Questa scelta consente di persistere i dati sugli ordini anche dopo la chiusura del Server, oltre a garantire una relativa semplicita nel caricamento in memoria degli ordini tramite una struttura dati apposita.
La gestione delle registrazioni è simile mediante un **"Userbook"** con una chiave usermap per rendere più agevole la traduzione da Json a struttura dati, inoltre per garantire maggiore sicurezza le password sono state criptate grazie alla libreria **BCrypt**.

1.4 Sincronizzazione

Per garantire consistenza in un ambiente concorrente vengono utilizzati metodi **synchronized** per proteggere sezioni critiche ed inoltre le varie strutture dati presenti nella collezione **java.util.concurrent**.

1.5 Design Pattern

Il progetto utilizza principalmente la **Simple Factory** per generare i comandi da inviare al Server. Questa generazione avviene sul Client sfruttando l'interfaccia **Values** la quale offre il metodo execute che il Server utilizzerà per eseguire correttamente i vari comandi disponibili. Inoltre questo approccio consente di mantenere invariato il formato di messaggi,**Message{Operation,Values}** che il Client invia al server e facilita l'aggiunta di comandi in quanto è sufficiente creare una nuova implementazione di Values.

2 Schema Generale dei Thread Attivati

Il progetto sfrutta il multithreading per gestire la comunicazione Client-Server in maniera efficiente, oltre all'elaborazione asincrona di attività interne

2.1 Thread Lato Client

Il client attiva ed usa 3 thread durante il normale funzionamento:

- **SenderThread**: utilizzato per inviare i comandi al server mediante connessione **TCP**.
- **ReceiverThread**: utilizzato per ricevere le risposte del server mediante connessione **TCP**.
- **UDPReceiverThread**: utilizzato per ricevere le notifiche legate alle transazioni che riguardano gli ordini piazzati dall'utente mediante connessione **UDP**.

I primi due thread vengono istanziati nella classe ClientClass e persistono le informazioni fino alla chiusura del socket, la scelta di avere due thread separati per invio e ricezione consente di ricevere in maniera asincrona le risposte del server, aiutando quindi a gestire i casi in cui si riscontrano dei problemi sul server richiedendo però l'utilizzo di una variabile per sincronizzare i due thread impedendo così di inviare al massimo 1 messaggio alla volta aspettando così la relativa risposta.
UDPReceiverThread viene istanziato subito dopo la ricezione del primissimo messaggio inviato dal server, che contiene le informazioni su porta e gruppo multicast al quale connettersi per ottenere informazioni sugli ordini. È stato scelto di utilizzare multicast per semplificare l'implementazione del paradigma Publish-Subscribe per gli ordini; un'altro vantaggio di multicast consiste nel poter assegnare ad ogni ordine dell'orderbook un gruppo multicast, vantaggio che in questo progetto non è stato sfruttato appieno in quanto non richiesto esplicitamente.

2.2 Thread Lato Server

Il Server utilizza un **FixedThreadPool** per gestire i vari client connessi che vengono rappresentati da una **GenericTask**, inoltre attiva i seguenti Thread:

- **UDPListner**: Thread che si occupa dell'invio delle notifiche secondo il modello Publish-Subscribe.
- **StopOrderChecker**: Thread che si occupa di controllare gli **StopOrder** piazzati dagli utenti ed eseguirli in caso lo stopprice lo consenta.
- **ClosingTask**: Thread che si occupa di svolgere la chiusura del server in sicurezza quando si riscontrano dei problemi sul server per i quali non si può garantire la fruizione del servizio di trading.

3 File Java

Il progetto sfrutta i java package per organizzare meglio i file Java contenenti le classi che verranno usate per fornire il servizio di trading; Di seguito verranno analizzati i vari package ed alcuni file tra i più importanti, motivando le scelte fatte.

3.1 Communication

Questo package contiene i protocolli di comunicazione utilizzati dal progetto, insieme ai tipi di messaggio che vengono scambiati tra Client e Server. I protocolli, come menzionato precedentemente, sono **UDP** e **TCP** che vengono realizzati implementando l'interfaccia **Protocol** che richiede un metodo rispettivamente per invio e ricezione di messaggi ed uno per la chiusura del protocollo; Inoltre **TCP** utilizza i metodi forniti da **Moshi** per realizzare uno scambio di messaggi in formato Json, sfruttando le varie implementazioni dell'interfaccia **Values**.

Moshi si è rivelato più facilmente manutenibile rispetto a Gson in quanto non perde le informazioni della sottoclasse di **Values** incapsulata nel messggio inviato/ricevuto.
Questo si è rivelato fondamentale in quanto ha permesso di mantenere la struttura dei messaggi richiesta dalla consegna. Questo si ottiene registrando tutti le Classi o sottoclassi che potrebbero essere passata nei Messaggi ad un builder di moshi tramite **JsonAdapterFactory**, come si può notare nella figura affianco.
Inoltre per finire la configurazione di Moshi è bastato creare un **JsonAdapter<Message>** che, similmente a Gson, si occuperà di serializzare/deserializzare il messaggio in formato Json.

```
protected Moshi moshi = new Moshi.Builder().add(PolymorphicJsonAdapterFactory.of(baseType:Message.class, labelKey:"type")
    .withSubtype(subtype:ServerMessage.class, label:"ServerMessage")
    .withSubtype(subtype:ClientMessage.class, label:"ClientMessage")
    .withSubtype(subtype:OrderResponseMessage.class, label:"Confirmation"))
    .add(PolymorphicJsonAdapterFactory.of(baseType:Values.class,labelKey:"type")
    .withSubtype(subtype:Limitorder.class, label:"limitorder")
    .withSubtype(subtype:MarketOrder.class,label:"marketorder")
    .withSubtype(subtype:StopOrder.class, label:"stoporder")
    .withSubtype(subtype:ShowOrderBook.class, label:"showorderbook")
    .withSubtype(subtype:ShowStopOrder.class, label:"showstoporder")
    .withSubtype(subtype:CancelOrder.class, label:"cancelorder")
    .withSubtype(subtype:Logout.class, label:"logout")
    .withSubtype(subtype:Register.class, label:"register")
    .withSubtype(subtype:Login.class, label:"login")
    .withSubtype(subtype:UpdateCredentials.class, label:"updatecredentials")
    .withSubtype(subtype:Help.class, label:"help")
    .withSubtype(subtype:getPriceHistory.class, label:"getpricehistory")
    .withSubtype(subtype:ErrorMessage.class, label:"errormessage")
    .withSubtype(subtype:Disconnect.class, label:"disconnect"))
    .add(PolymorphicJsonAdapterFactory.of(baseType:ZonedDateTime.class, labelKey:"GMT"))
```

Figura 1: Aggiunta delle sottoclassi di Values

3.2 Commands

Questo package contiene tutti i comandi che il Client può inviare al Server e la factory che si occupa di crearli. La creazione avviene lato Client per rispettare il formato dei messaggi da scambiare ed inoltre consente di alleggerire il dei gestori lato Server che si limiteranno ad usare il metodo execute fornito dall'interfaccia **Values**.
I comandi sono divisi in base al tipo di dato Json che richiedono per essere eseguiti:

- **Credentials**: comandi che aggiungono o modificano informazioni nell'**Userbook**
- **Order**: comandi che aggiungono o modificano informazioni nell'**Orderbook**
- **Internal**: comandi che non richiedono alcun tipo di Json e sono utilizzati per operazioni interne, ad esempio i messaggi d'errore.

Le strutture dati che rappresentano i file Json verranno poi successivamente aggiunte al comando nel **GenericTask**, lato Server, prima di essere eseguito.
L'unica eccezione è **getPriceHistory** che richiede lo storico degli ordini per il suo funzionamento però tale file è richeisto nella configurazione del Server in quanto viene usato come risorsa a sola lettura.
Questa scelta di utilizzare sottoclassi di Values e una **SimpleFactory** semplifica l'aggiunta di nuovi comandi in quanto è sufficiente:

1. **Creare** un classe Java che implementi Values
2. **Aggiungere** alla factory la creazione della nuova classe
3. **Aggiungere** nel builder di moshi la sottoclasse appena creata

3.3 Utils

Questo package contiene le varie classi utilizzate per semplificare la manipolazione dei file Json, le eccezzioni create per gestire i parametri errati nella creazione dei comandi mediante factory ed una cache per il mantenimento in memoria degli ordini evasi durante l'esecuzione di un ordine.

3.4 JsonAccessedData

Questo package contiene le classi che rappresentano in memoria i file Json, ed i file Json veri e propri ad eccezzione dello storico ordini come discusso precedentemente.

3.4.1 Orderbook

L'orderbook è il cuore del servizio offerto in quanto contiene i limitorder piazzati dagli utenti e deve essere sempre acceduto per ogni ordine. È suddiviso in **askMap** e **bidMap** dove verranno memorizzati i relativi ordini, mentre gli StopOrder verranno memorizzati in una **ConcurrentLinkedQueue**.

3.4.2 Userbook

L'userbook contiene le informazioni sugli account degli utenti e viene acceduto per poter iniziare ad usare il servizio di trading. È costituito da una **userMap** che contiene tutti gli utenti, questo è stato fatto per facilitare lettura e scrittura in memoria tramite **Moshi**.

3.4.3 Storico Ordini

Lo storico ordini sebbene venga trattato come risorsa a sola lettura presenta una struttura singolare che andremo ad analizzare.
Viene caricato in memoria come **TreeMap<Integer,TreeMap<DayTime,Trade>>**, dove la chiave Integer della prima TreeMap rappresenta l'anno in cui è avvenuta la transazione, mentre la TreeMap interna ha come chiave la classe **Daytime** che contiene informazioni su ora giorno e mese in cui è avvenuta la transazione, mentre il valore è rappresentato dal **Trade** ossia la transazione effettiva. **DailyTradeStats** è la classe che viene usata per rappresentare i seguenti parametri registrati giorno per giorno:

- Prezzo di apertura
- Prezzo di chiusura
- Minimo prezzo di una transazione di tipo Ask
- Minimo prezzo di una transazione di tipo Bid
- Massimo prezzo di una transazione di tipo Ask
- Massimo prezzo di una transazione di tipo Bid

Questo per consentire una più agevole esecuzione del comando **getPriceHistory**.

3.5 Client

La classe ClientClass, contenuta nel package Client, si occupa di comunicare col server sfruttando il **multithreading** per poter riceve richieste TCP **asincrone** dal Server, oltre a predisporre un Thread di ricezione UDP per essere notificato sullo stato delle transazioni. Questa è la classe che viene istanziata dal ClientMain per realizzare il client del servizio offerto.

3.6 Server

La classe ServerClass, contenuta nel package Server, si occupa di creare il ServerSocket sul quale accetterà le connessioni dei client e che verranno gestite mediante **GenericTask** che verrà passata al threadpool.
GenericTask si occupa quindi di comunicare direttamente col Client e di eseguire i comandi da lui richiesti.

4 Primitive di Sincronizzazione

Sfruttando il multithreading è necessario controllare che le sezioni critiche non vengano compromesse da accessi simultanei da parte di più thread, per fare questo sono state utilizate varie primitive di sincronizzazione.

4.1 synchronized

4.2 volatile

4.3 ConcurrentHashMap

4.4 ConcurrentSkipListMap

4.5 ConcurrentLinkedQueue

5 Utilizzare il Progetto

5.1 Requisiti di Sistema

5.2 Struttura del Progetto

5.3 Esecuzione del Progetto