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 come si può notare nella fiigura 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.

3.2 Commands

3.3 Utils

3.4 JsonAccessedData

3.5 ClientClass

3.6 ServerClass

4 Primitive di Sincronizzazione