Relazione del Progetto CROSS

Alberto Fresu
a.fresu1@studenti.unipi.it
582314

Abstract

Relazione del progetto `CROSS` in cui sono state prese diverse decisioni architetturali per garantire efficienza, modularità e scalabilità del sistema. Di seguito si espongono le principali scelte effettuate

1 Struttura del Progetto 1
2 Gestione della Concorrenza 2
2.1 Sincronizzazione 2
3 Protocollo di Comunicazione 2
4 Schema Generale dei Thread 2
4.1 Lato Server 2
4.2 Lato Client 3
5 Strutture Dati Utilizzate 3
5.1 Lato Server 3
6 Gestione dello storico prezzi e serializzazione ordini 3
6.1 Gestione dello storico prezzi 3
6.2 Serializzazione e deserializzazione ordini 3
7 Controlli aggiuntivi 4
7.1 Riconnessione del client 4
7.2 Vincoli sulle credenziali di accesso 4
7.3 Comando di terminazione del client 4
8 Istruzioni per la Compilazione ed Esecuzione . . . 4
8.1 Dipendenze Esterne 4
8.2 Compilazione del Progetto su Windows 4
8.3 Esecuzione del Server 5
8.4 Esecuzione del Client 5
8.5 Compilazione del Progetto su Linux 5
8.6 Esecuzione del Server 5
8.7 Esecuzione del Client 5

1 Struttura del Progetto

Il progetto è stato suddiviso in cartelle di lavoro, ciascuna delle quali contiene un insieme di classi che aiuta a comprendere il progetto e separa i concetti, in maniera da facilitare la risoluzione di eventuali problemi e rendere semplice e veloce la manutenibiltà. Di seguito si presentano le cartelle di lavoro ed il loro utilizzo nel progetto.

Server	In questa cartella si gestisce la comunicazione server e l'invio di notifiche verso i client.
Client	Come nella cartella precedente, si fanno uso di classi per la gestione dell'invio delle richieste verso il server e la ricezione delle notifiche asincrone.
Common	All'interno della quale sono presenti le classi comuni alla parte client e alla parte server. Come ad esempio la classe **JsonParsing** utile per il parsing dei messaggi scambiati in formato **JSON** e viceversa.
Order	Come si evince dal nome questa cartella contiene tutte le classi per la gestione dei diversi tipi di ordine dell'OrderBook.
User	All'interno si trovano le classi per il salvataggio e la gestione degli utenti collegati al server.

RMI Creata per separare concettualmente l'utilizzo del metodo **Remote Method Invocation**, usato per la registrazione dell'utente.

Config Usata per contere i **file di configurazione** `.cfg`. I parametri di configurazione (porte, indirizzi, numero di thread da usare) vengono letti da questi file, evitando richieste interattive all'utente. Questo grazie alla classe a comune `ConfigReader`.

Resources Contiene i file utilizzati per tenere traccia degli ordini evasi oltre a persistere le informazioni degli utenti registrati.

Le cartelle `Config` e `Resources`, si trovano fuori dalla cartella dei sorgenti in quanto sono parametri di configurazione.

2 Gestione della Concorrenza

- **Implementazione parte TCP**: Per quanto riguarda la parte Server, è stato utilizzato un modello basato su **NIO (Non-blocking I/O)** per la gestione efficiente delle connessioni multiple. Ogni messaggio ricevuto dai Client viene sottomesso ad un pool di thread che elabora la richiesta e invia la risposta al corrispettivo richiedente, grazie all'utilizzo di **Selector** e **Channels**. Il processo Client è stato creato per l'invio di richieste e la ricezione delle stesse da un server non bloccante.
- **Implementazione parte UDP**: Nella parte Server, ogni notifica viene sottomessa ad una **Chached Thread Pool** ed inviata al client. Nella parte Client, è stato usato una **Fixed Thread Pool** dove per ogni Client, vengono sottomessi due task. Il primo, si occupa di gestire la parte della notifica UDP asincrona, utilizzando un approccio non bloccante. Il secondo, si occupa dello ricezione di notifiche tramite **Multicast**.

2.1 Sincronizzazione

I metodi che che consentono l'accesso alle strutture dati in maniera concorrente o che scrivono su file condivisi, sono stati dichiarati **synchronized** acquisendo quindi lock implicite. Per gestire l'atomicità degli identificativi degli ordini è stato implementato un generatore di ID incrementale tramite l'uso della classe **Atomic Long**, ogni qualvolta si crea un ordine viene genearato un ID.
In particolare all'interno della classe **OrderBook**, posizionata dentro la cartella Order il quale gestisce la sottomissione ed il processamento degli ordini, i metodi che processano gli ordini e che li salvano sono stati dichiarati sincronizzati. Anche nella classe **UserDatabase** i metodi che si occupano della persistenza degli utenti sono dichiarati synchronized.

3 Protocollo di Comunicazione

Il client e il server scambiano messaggi in formato **JSON**, garantendo interoperabilità e facilità di parsing grazie all'utilizzo della libreria **gson** che viene utilizzata soprattutto nella classe **JsonParsing**.

4 Schema Generale dei Thread

4.1 Lato Server

- **Thread Principale**: Si occupa di accettare nuove connessioni e di attivare i thread di elaborazione dei messaggi.
- **Thread di Elaborazione**: un thread viene assegnato a ogni richiesta di elaborazione dei Client, questo chiama a seconda del comando ricevuto le classi per la gestione del messaggio come ad esempio **UserDatabase** o **OrderBook** .
- **Thread per le Notifiche UDP**: quando un ordine viene evaso o quando si raggiunge una certa soglia di prezzo, la classe OrderBook richiama la classe **NotificationSender** che si occupa di gestire l'invio delle notifiche UDP, sottomettendo ad una **Cached Thread Pool** la notifica da inviare ai client connessi.

- **Thread Schedulato**: aggiunto per monitorare gli utenti inattivi, dopo un certo intervallo di tempo.

4.2 Lato Client

- **Thread Principale**: Gestisce l'interfaccia utente e l'invio delle richieste al server, che a seconda del comando (solo se conosciuto) viene inviato. Inoltre, avvia una **Fixed Thread Pool** per la ricezione delle notifice.
- **Thread di Ascolto Notifiche**: Una volta che il thread principale sottomette i thread di ascolto, questi utilizzano due classi in particolare. La prima per la ricezione delle notifiche asincrone, **NotificationRecevierUDP**, usando **DatagramChannel** per gestire le notifiche in maniera non bloccante. La seconda si appoggia ad un **MulticastManager** che apre un socket e lo utilizza per la ricezione in multicast nella classe **NotificationReceiverMulticastUDP**.

5 Strutture Dati Utilizzate

5.1 Lato Server

- `ConcurrentHashMap<K, V>`: Per la gestione dei dati degli utenti con accesso concorrente e la relativa gestione dei canali associati. Usata anche per la gestione degli ordini attivi.
- `BlockingQueue<T>`: Per la gestione delle richieste in coda in attesa di elaborazione.
- `PriorityQueue<T>`: Per gestire l'ordinamento e le code degli ordini emessi.
- `List<T>`: Per tenere traccia dei prezzi giornalieri e per caricare lo storico degli ordini.
- `HashMap<K,V>`: Per la gestione degli ordini per giorno.

6 Gestione dello storico prezzi e serializzazione ordini

6.1 Gestione dello storico prezzi

Il metodo `getPriceHistory(int month, int year)` recupera lo storico dei prezzi degli ordini eseguiti in un dato mese e anno. L'implementazione segue questi passi:

- **Caricamento e filtraggio**: Gli ordini vengono recuperati e filtrati per mese e anno tramite il timestamp convertito in `ZonedDateTime`.
- **Raggruppamento per giorno**: Gli ordini sono organizzati in una mappa `ordersByDay`, con chiave la data e valore la lista di ordini eseguiti.
- **Calcolo delle statistiche**: Per ogni giorno vengono determinati prezzo di apertura, chiusura, massimo e minimo.
- **Restituzione del risultato**: I dati elaborati sono incapsulati in un oggetto `Response`, contenente una lista di `DayPriceData` o un messaggio d'errore se non sono disponibili dati.

Questa implementazione offre un'elaborazione efficiente dello storico prezzi, utile per l'analisi delle variazioni giornaliere.

6.2 Serializzazione e deserializzazione ordini

La classe `OrderAdapter` gestisce la conversione degli oggetti `Order` in formato JSON utilizzando Gson.

- **Deserializzazione**: Il metodo `deserialize` verifica la presenza di campi essenziali (`type`, `orderId`, `size`, `price`, `timestamp`), generando errori in caso di valori nulli o mancanti.
- **Serializzazione**: Il metodo `serialize` converte un oggetto `Order` in JSON, escludendo dati non necessari come `UserSession`.

Questa soluzione garantisce un formato dati compatto e sicuro per la comunicazione tra client e server.

7 Controlli aggiuntivi

7.1 Riconnessione del client

Quando un Client viene avviato da un utente può verificarsi che il Server non sia disponibile. Per ovviare alla possbilità che magari il Server si accenda in un breve lasso di tempo, il Client tenta ogni 5 secondi, per un massimo di 3 tentativi, di connettersi per poi terminare il processo se non si è verificata nessuna connessione.

7.2 Vincoli sulle credenziali di accesso

Se un utente vuole registrarsi deve immettere una password valida, cioè contenente:

- almeno una lunghezza di 8 caratteri
- lettere maiuscole e minuscole, ad esempio: `aBcD`
- numeri, ad esempio: `1234`
- caratteri speciali, ad esempio: `@*!()`

7.3 Comando di terminazione del client

Grazie al comando `exit` l'utente può terminare la sessione con il Server e chiudere il Client.

8 Istruzioni per la Compilazione ed Esecuzione

8.1 Dipendenze Esterne

- Java 17 o superiore
- Librerie esterne: `Gson` per la serializzazione JSON, `java.nio` per la gestione dei canali non bloccanti

8.2 Compilazione del Progetto su Windows

Per automatizzare la creazione dei file jar, la compilazoine e l'esecuzione del progetto, vengono forniti i due file

```
build_run_client.bat build_run_server.bat
```

che possono essere esguiti su due terminali diversi per la prima esecuzione. Se si vuole rieseguire il codice in un secondo momento basta digitare i comandi descritti nella sezione **8.3** e **8.4**. Se ci dovessero essere problemi con l'automatizzazione di seguito sono forniti i comandi passo per passo.

1. Posizionarsi nella directory principale del progetto su due terminali diversi.
2. Con il primo comando è possibile creare un file di elenco con tutti i sorgenti, e con il secondo compilarli tutti:

   ```
   dir /b /s src\*.java > sources.txt
   javac -cp lib\gson-2.8.9.jar -d out @sources.txt
   ```

3. Creare i file JAR eseguibili:

   ```
   jar --create --file client.jar --main-class=client.ClientMain -C out .
   jar --create --file server.jar --main-class=server.ServerMain -C out .
   ```

4. Dato che le cartelle config e resources si trovano nella directory principale è possibile aggiornare il jar con queste configurazioni:

```
jar --update --file client.jar -C config client.cfg -C resources .
jar --update --file server.jar -C config server.cfg -C resources .
```

8.3 Esecuzione del Server

```
java -cp server.jar;lib\gson-2.8.9.jar server.ServerMain
```

8.4 Esecuzione del Client

```
java -cp client.jar;lib\gson-2.8.9.jar client.ClientMain
```

8.5 Compilazione del Progetto su Linux

Come per Windows se si vuole automatizzare il primo avvio digitare su due terminali diversi;

```
.\build_run_client.sh   .\build_run_server.sh
```

Per rieseguire il codice in un secondo momento basta digitare i comandi descritti nella sezione **8.6** e **8.7**. Se non si dispongono dei permessi necessari eseguire i comandi:

```
chmod +x build_run_server.sh
chmod +x build_run_client.sh
```

e riprovare l'esecuzione. Di seguito vengono forniti i comandi passo per passo per una compilazione ed esecuzione guidata.

1. Posizionarsi nella directory principale del progetto su due terminali diversi.
2. Con il primo comando è possibile creare un file di elenco con tutti i sorgenti, e con il secondo compilarli tutti:

   ```
   find src -name "*.java" > sources.txt
   javac -cp "lib/gson-2.8.9.jar" -d out @sources.txt
   ```

   nota: se la cartella out non esiste, crearla con il comando:

   ```
   mkdir out
   ```

3. Creare i file JAR eseguibili:

   ```
   jar --create --file client.jar --main-class=client.ClientMain -C out .
   jar --create --file server.jar --main-class=server.ServerMain -C out .
   ```

4. Dato che le cartelle config e resources si trovano nella directory principale è possibile aggiornare il jar con queste configurazioni:

   ```
   jar --update --file client.jar -C config client.cfg -C resources .
   jar --update --file server.jar -C config server.cfg -C resources .
   ```

8.6 Esecuzione del Server

```
java -cp server.jar:lib\gson-2.8.9.jar server.ServerMain
```

nota: in linux per eseguire il programma nel comando appena citato bisogna inserire : al contrario di Windows dove si utilizza ;

8.7 Esecuzione del Client

```
java -cp client.jar:lib\gson-2.8.9.jar client.ClientMain
```