Matthew Bernardi, matricola 656365

CROSS

1. Interpretazione progetto

Nel sistema CROSS, abbiamo una serie di client che si connettono al server per svolgere delle operazioni, come login, registrazione, logout, e - parte fondamentale del progetto - inserire ordini di ask o bid nel sistema. Questi ordini possono essere di 3 tipi: market, limit o stop. Nella gestione di questi tipi di ordini, mi sento di dover spiegare la mia implementazione del progetto:

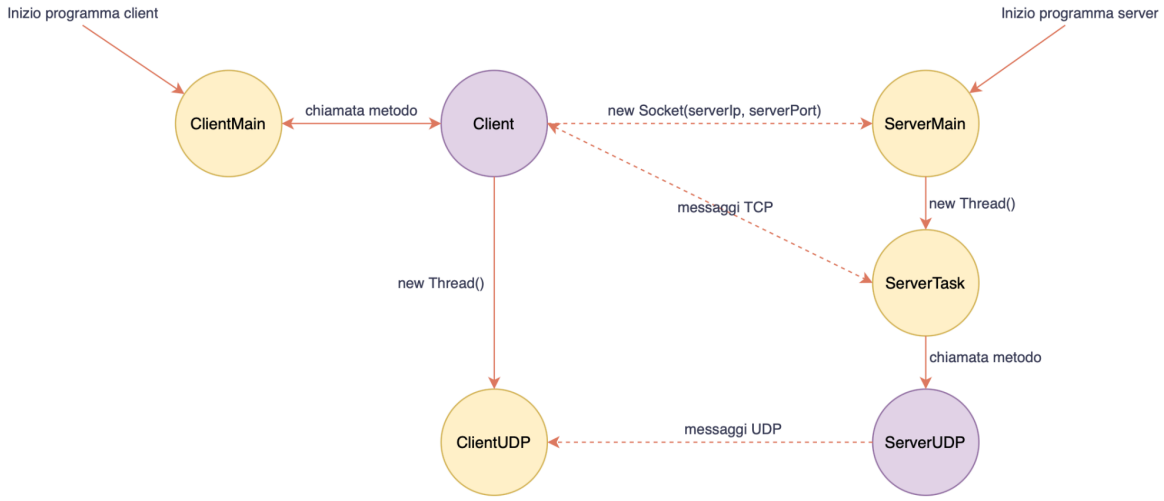
- ***Market order***
  I market order sono ordini che devono essere eseguiti immediatamente. Un market order viene evaso nel momento in cui, nel sistema, sono presenti degli ordini (non ancora evasi) che possono consentire all'algoritmo di far evadere l'ordine inserito. Ad esempio, se vogliamo inserire un ordine di tipo ask di dimensione n, perché tale ordine venga evaso correttamente c'è bisogno che, nella lista degli ordini di tipo bid, siano presenti almeno x ordini tali che la somma delle dimensioni sia maggiore o uguale a n. Se questa condizione non è verificata, l'ordine viene rifiutato, e si ritorna un codice di errore (-1) all'utente che ha richiesto l'inserimento.

- ***Limit order***
  I limit order sono ordini che non devono essere eseguiti immediatamente, anzi, possono essere eseguiti se ci sono le condizioni necessarie, ma possono anche essere inseriti nella struttura dati apposita in caso non siano eseguibili sul momento. Ad esempio, se un utente inserisce un limit order di ask (rispettivamente bid) di dimensione n e costo c, per poterlo evadere immediatamente c'è bisogno che esistano degli ordini di bid (rispettivamente ask) nella lista di ordini non ancora evasi tali che la dimensione sia complessivamente maggiore o uguale ad n e il costo di ogni ordine sia maggiore o uguale (rispettivamente minore o uguale) al costo c dell'ordine inserito. Altrimenti, il limit order viene inserito nella lista contenente gli ordini non ancora evasi dello stesso tipo.

- ***Stop order***
  Gli stop order sono ordini che devono essere gestiti come market order quando la loro condizione di stop risulta verificata. Quando un client vuole inserire uno stop order, viene prima verificata la condizione, e in caso questa sia vera, lo stop order viene eseguito come un market order (usando esattamente lo stesso metodo per evadere un market order), e questa operazione potrebbe fallire; altrimenti, se la condizione di stop non è verificata, inserisco lo stop order nella struttura dati contenente gli stop order da eseguire non appena si verifica la loro condizione. Questa struttura dati viene analizzata ogni volta che viene inserito - o evaso - un nuovo limit order e quando viene evaso un nuovo market order, perché questi sono gli unici casi in cui gli ordini attualmente presenti possono portare all'avverarsi delle condizioni per gli ordini di stop in attesa di evasione.
  Ad esempio, se un client inserisce un nuovo stop order ask di dimensione n e costo di stop c, se nella struttura che contiene gli ordini di bid (vedere sezione *Strutture dati usate e sincronizzazione*) il primo ordine ha costo minore o uguale a c, allora lo stop order viene eseguito con lo stesso algoritmo dei market orders, con possibile fallimento. Altrimenti, viene inserito nella coda degli stop order in attesa di evasione, che verrà ricontrollata alla prossima evasione di limit order o market order, oppure all'inserimento di un altro limit order. In tal caso, ho inserito come risposta al client {"orderId": -2}, per segnalare che l'ordine non è fallito ma non è nemmeno stato eseguito.

Infine, quando si esegue un qualsiasi ordine che può essere evaso grazie all'uso di più ordini in attesa (e.g. quando un market order ask può essere evaso usando m ordini presenti nella struttura dati con gli ordini di bid in attesa di evasione), allora l'ordine evaso viene diviso in "chunk", ognuno venduto al prezzo corrispondente all'ordine in attesa con cui fa match (e.g. se abbiamo un market order di ask di dimensione 10 e abbiamo 2 limit order di tipo bid di dimensione 5 ciascuno e di costo 9000 e 8000, il market order viene diviso in due chunk, di dimensione 5, ma costi 9000 e 8000). Questo vale per tutti i tipi di ordini.

2. Schema thread usati



Nella figura sopra, viene riportato uno schema dei thread che vengono eseguiti: colorati in giallo ci sono i thread, mentre in viola istanze che sono eseguite all'interno dei thread.
Nella parte di sinistra troviamo l'applicazione lato client, che viene eseguita lanciando il thread principale ClientMain, che comunica con l'oggetto di classe Client (non thread). Questo oggetto si occupa di comunicare con il server per scambiare i messaggi TCP su cui si basa il progetto. Inoltre, questo lancia un altro thread, il ClientUDP, che gestisce l'arrivo di notifiche UDP dal server.
Nella parte di destra invece si trova l'applicazione server, che inizia con l'avvio del thread ServerMain, che gestisce l'arrivo di nuove connessioni TCP con una socket di benvenuto. Quando un client si connette, si crea un thread ServerTask che si occupa di gestire la connessione TCP col client tramite i messaggi json specificati (login, register, inserimento di ordini, …). Inoltre, ServerMain istanzia un oggetto (non un thread) di tipo ServerUDP che produce e invia messaggi UDP ai client. Questo oggetto è condiviso tra tutti i thread ServerTask e viene usato da questi ultimi per l'invio di notifiche.

3. Strutture dati usate e sincronizzazione

Le strutture principali usate sono le seguenti:
Per quanto riguarda l'applicazione ***lato client***, non ci sono strutture dati per memorizzare informazioni, solo primitive per la comunicazione con l'utente via riga di comando e per la comunicazione via TCP e UDP col server.
Per l'applicazione ***lato server***, abbiamo le seguenti strutture dati:

- **`LinkedBlockingQueue<Runnable> workingQueue`**
  Insieme di task che la thread pool deve eseguire (i task rappresentano istanze della classe ServerTask).

- **`ConcurrentLinkedQueue<User> users`**
  Questa struttura dati viene usata per memorizzare gli utenti registrati nel sistema CROSS. L'uso di una coda mi assicura la dimensione unbounded (fino al massimo valore rappresentabile dagli interi) e anche l'atomicità delle singole operazioni sulla coda.

- **`ConcurrentLinkedQueue<Order> issuedOrders`**
  Questa struttura dati memorizza al suo interno gli ordini che sono stati evasi in precedenza e gli ordini storici letti dal file apposito.

Le prossime due strutture dati sono di tipo ConcurrentSkipListSet in quanto questa classe consente di mantenere i dati ordinati secondo un certo comparatore.

- **`ConcurrentSkipListSet<Order> askOrders`**
  Con questa struttura dati, memorizzo gli ordini attualmente non evasi, in particolare quelli di tipo ask. Gli ordini di ask presenti sono quindi ordinati per costo crescente, e a parità di costo, per timestamp crescente.

- **`ConcurrentSkipListSet<Order> bidOrders`**
  Con questa struttura dati, memorizzo gli ordini attualmente non evasi, in particolare quelli di tipo bid. Gli ordini di bid presenti sono quindi ordinati per costo decrescente, e a parità di costo, per timestamp crescente.

- **`ConcurrentLinkedQueue<Order> stopOrders`**
  In questa lista, memorizzo invece gli stop order le cui condizioni di stop non erano soddisfatte. Questa lista verrà eventualmente aggiornata rimuovendo gli ordini le cui condizioni di stop saranno soddisfatte.

Per quanto concerne la sincronizzazione per l'accesso concorrente, ho semplicemente usato dei blocchi synchronized nelle situazioni in cui andavo a modificare o a leggere in una struttura condivisa, e anche nella scrittura sui file di ordini evasi e nuovi utenti.

4. Compilazione e informazioni sull'uso

Il progetto è diviso in due cartelle, `CROSS` e `config`. In `config` sono presenti i file di configurazione e quelli per gli ordini evasi e gli utenti registrati, oltre che gli script per l'esecuzione; mentre in `CROSS` abbiamo due cartelle: `src`, che contiene i file `.java`, il codice sorgente, e `lib`, che contiene le librerie usate.
Per compilare ed eseguire applicazioni server e client in modo separato, ho creato due script in bash per svolgere tutto il lavoro, e questi comprendono l'inclusione della libreria .jar usata (libreria gson per la gestione dei formati json). Gli script si chiamano, con poca fantasia, `server.sh` e `client.sh`.
Per configurare in modo particolare client e server, basta andare a modificare i file `userConfig.json` e `serverConfig.json`, che contengono le opzioni di configurazione delle due applicazioni, come tempi di timeout, porte su cui rimanere in ascolto e a cui connettersi, rispettivamente per server e client.
I file `issuedOrders.json` e `users.json` servono invece per la persistenza degli ordini evasi e gli utenti registrati nel sistema. Per mostrare la forma con cui vengono salvati gli utenti nel file, viene lasciato un esempio con l'utente "prova".

Il programma lato server stampa semplicemente informazioni circa la connessione o disconnessione di utenti e segnala la presenza di errori

dovuti ad eccezioni. Non ci sono parti interattive, è pensato semplicemente per essere eseguito senza supervisione umana.

Il programma lato client invece richiede interazione con l'utente tramite l'inserimento di codici associati ad azioni e informazioni sull'operazione da svolgere. Inoltre, a seguito dell'esecuzione di ogni azione, viene stampato un messaggio che inizia con `[RESPONSE]` che indica la risposta del server all'azione svolta per avere un feedback sull'effettività di quest'ultima; mentre le notifiche vengono stampate a schermo con un tag `[NOTIFICATION]`.
In una prima fase, viene richiesto di scegliere se svolgere l'operazione di login oppure quella di registrazione al sistema, e in seguito di inserire le credenziali per la relativa operazione (e.g. in un primo accesso, l'utente potrebbe scegliere il login e inserire come username "prova" e come password "prova" per accedere; altrimenti, potrebbe registrarsi con un proprio username e una propria password che, se ammissibili dal server, verranno salvati nel file degli utenti).
In seguito, la schermata principale del progetto lato client si presenta di fronte, proponendo l'azione da svolgere. Inserendo il codice relativo all'azione desiderata, si inseriscono i dati necessari al completamento e si ottiene la risposta:

- `Insert limit order` prevede che si inseriscano, in ordine, tipo dell'ordine (ask o bid), dimensione dell'ordine e prezzo limite.
- `Insert market order` prevede che si inseriscano, in ordine, tipo (ask o bid) e dimensione dell'ordine.
- `Insert stop order` prevede, ancora, l'inserimento del tipo dell'ordine, la sua dimensione e il prezzo di stop.
- `Cancel order` prevede che si inserisca l'id dell'ordine da cancellare. Ovviamente, come scritto nella specifica, l'ordine da cancellare non deve essere né già (completamente) evaso né posseduto da un altro utente (inteso come account). Da notare che, una volta che si arresta il programma server, tutti gli ordini non ancora evasi vengono persi.
- `Get price history` richiede il mese di cui ottenere le informazioni nel formato MMMYYYY, ad esempio Sep2024 o Oct2024 (il mese deve essere espresso con le prime 3 lettere del

nome in inglese, e la prima lettera maiuscola). In risposta, si visualizzerà, per ogni giorno del mese, il prezzo di apertura, chiusura, massimo e minimo. Per i giorni in cui non sono stati evasi ordini, il valore stampato sarà “`none`” per ogni campo.

- `Logout` prevede l’uscita dall’account, ma non la terminazione della comunicazione col server. Svolgendo il logout, un altro client potrà connettersi con l’account usato finora.

Una volta svolto il logout, visualizziamo una nuova schermata, che presenta 3 azioni da eseguire, ancora come menù a indici:

- `Update credentials` consente di modificare la password dell’account di cui si è appena fatto il logout.
- `Login` ci consente di svolgere di nuovo il login. Se il login è andato a buon fine, ricomparirà la schermata principale vista prima del logout.
- Selezionando -1, il client si disconnette dalla connessione TCP col server, e il programma termina.