Relazione per il Progetto LAB III

Premessa:

Avviso che purtroppo non sono riuscito a completare il progetto al 100%, non sono riuscito a implementare le notifiche, avevo progettato di creare un listNotify per ogni User e memorizzare all'interno tutte le notifiche che non sono raggiungibili nel momento in cui l'user si trova offline (l'attributo login = false, campo della classe User in UserManager.java), se invece si trovava online, le notifiche sarebbero state mandate tramite UDP. Come vedra' nel codice ci sono i metodi che lo implementano, peccato per la realizzazione.

---

1. Scelte Progettuali

Struttura Generale:

Il progetto presenta 4 files:

1. *mainServer.java* che gestisce la logica di assegnamento thread (1 per client) suddividendo le chiamate di metodi in base all'input de-serializzando i messaggi Json da parte del client. Gestisce con un switch le varie casistiche. Usera' i file 3, 4 per gestire i comandi con delle chiamate ai metodi

2. *mainClient.java* invece si mette in comunicazione con mainServer.java e stampa i comandi permessi dal servizio CROSS. Mette a lavoro 3 threads: uno (listenForNotifications) occupato a catturare le notifiche sincrone, l'altro (monitorSocket) che controlla se la connessione si e' interrotta, e un'altro thread si preoccupa del timeout del client aggiornando il "conto alla rovescia" ogni volta che l'utente agisce.

3. *OrderBook.java* implementa tutti i metodi necessari alla gestione degli scambi asset: in questa classe sono gestite tutti i metodi accessibili dopo il Login dell'utente

4. *userManager.java* implementa tutti i metodi necessari alla gestione degli utenti.

Panoramica generale:

- **Uso di un Server Multithread:** Il server utilizza un pool di thread («ThreadPool») per gestire più connessioni simultanee, garantendo scalabilità e prestazioni.
- **Strutture Condivise Thread-Safe:** Sono state adottate strutture come `ConcurrentHashMap` e liste sincronizzate per evitare condizioni di competizione («race conditions») tra i thread.
- **Protocollo di Comunicazione:** La comunicazione tra client e server avviene tramite socket TCP per i comandi principali, mentre le notifiche vengono inviate tramite UDP.
- **Formato JSON:** Gson è utilizzato per serializzare e deserializzare i dati scambiati tra client e server.

- **Persistenza Dati:** Gli utenti e gli ordini vengono salvati in file JSON per mantenere la persistenza tra diverse sessioni del server.
- **Gestione degli Ordini con `OrderBook`:** L'oggetto `OrderBook` si occupa di organizzare gli ordini secondo diversi criteri, supportando funzionalità come cancellazione, inserimento e aggiornamento degli ordini attivi, OrderBook inoltre gestisce il file ordersServer.json (localizzato nel path del main) che in caso non ci fosse viene creato in automatico, se c'e' invece crea il file.
- **Gestione Utenti con `UserManager`:** Centralizza la logica di registrazione, login e gestione delle notifiche per gli utenti, permettendo un accesso sicuro e sincronizzato alle strutture dati condivise. UserManager come OrderBook crea il file userServers.json che viene arricchito in base alle nuove registrazioni. Inoltre implementa dei metodi che permettono al main di fare il setup della struttura dati userDatabase, in modo da mantenere le informazioni sulle registrazioni passate.

---

2. Schema Generale dei Thread

Lato Server

- **Thread Principale:** Accetta nuove connessioni client e le assegna a thread del pool per l'elaborazione.
- **ThreadPool ClientHandler:** Gestisce le richieste dei client (login, registrazione, inserimento ordini, ecc.) e interagisce con le strutture condivise.
- **Thread di Notifica UDP:** Utilizzato per inviare notifiche agli utenti connessi.

Lato Client

- **Thread Principale:** Gestisce l'interazione con l'utente, inviando comandi al server e ricevendo risposte.
- **Thread UDP Listener:** Rimane in ascolto per le notifiche UDP inviate dal server.
- **Thread Monitor Socket:** Controlla lo stato del socket TCP per gestire eventuali chiusure impreviste.

---

3. Strutture Dati Utilizzate

Lato Server

- **`ConcurrentHashMap<User, List<Order>> userDatabase`:** Mappa gli utenti alle loro liste di ordini. Ogni utente è un oggetto `User`, che include informazioni come username, password, indirizzo IP, stato di login e una lista di notifiche. Ad ogni User c'e' associato la lista degli ordini che ha fatto.

- **`OrderBook`:** Una classe dedicata per gestire gli ordini attivi. Le sue funzionalità principali includono:

    - **Inserimento di Ordini:** Inserisce ordini di mercato o con limite, garantendo che siano posizionati correttamente nella lista degli ordini in base al prezzo o alla priorità temporale.
    - **Cancellazione:** Rimuove un ordine specifico dato il suo ID.
    - **Notifiche Associate:** Gestisce gli utenti da notificare per ogni ordine.
- **`List<Order>`:** Liste sincronizzate contenenti gli ordini degli utenti, utilizzate sia per gli ordini attivi sia per le notifiche.

Lato Client

- **`JsonObject`:** Utilizzato per creare richieste strutturate da inviare al server. Ogni comando del client viene inviato come un oggetto JSON contenente un campo `command` e i dati associati.

- **`DatagramPacket`:** Utilizzato per gestire la ricezione di notifiche UDP. Ogni pacchetto contiene il messaggio di notifica inviato dal server.

BookOrder

- **PriceSegment**: come chiave ha il prezzo, come valore una lista di ordini, ad ogni ordine ci sono associate tutte le informazioni perche' possa risalire all'User o altri User coinvolto/i nello scambio

---

4. Primitive di Sincronizzazione

Lato Server

- **`synchronized`:** Utilizzato in `UserManager` per garantire la consistenza durante l'accesso e la modifica degli utenti e delle notifiche associate, `ConcurrentHashMap` garantisce thread-safety per la gestione della mappa `userDatabase`, vengono `implementate` Liste sincronizzate per evitare conflitti durante l'accesso simultaneo agli ordini, inoltre la scrittura su file utilizza un metodo synchronized

Lato Client

Non è necessaria una sincronizzazione complessa, in quanto il client lavora principalmente in modalità sequenziale per le richieste al server. Tuttavia:

- **UDP Listener:** Usa il controllo dello stato dei thread per terminare correttamente il listener quando il socket viene chiuso.

---

5. Istruzioni per la Compilazione ed Esecuzione

Dipendenze

- **Librerie Esterne:** Gson (è necessario includere `gson-2.11.0.jar` nel classpath).
- **Java Development Kit (JDK):** Versione 8 o successiva.

Compilazione

1. Posizionarsi nella directory principale del progetto con tutti i file.

Eseguire il seguente comando per compilare il codice:
*javac -cp .:gson-2.11.0.jar *.java*

Esecuzione del Server

2. Avviare il server con il seguente comando, sempre all'interno della cartella:

java -cp .:gson-2.11.0.jar mainServer

Esecuzione del Client

Avviare il client con il seguente comando:
java -cp .:gson-2.11.0.jar mainClient
