Introduzione

Il progetto è stato diviso in tre packages: uno contenete il codice del client, uno il codice del server e l'ultimo contenente metodi "condivisi" accessibili da entrambi. Fa parte del progetto anche la cartella Recovery che contiene le informazioni necessarie per ripristinare lo stato del server a seguito di un riavvio.

All'interno della cartella Recovery sono presenti:

- Il Database relativo alle **credenziali utente** (*credentials.json*). Il file contiene un oggetto di tipo CredentialDB che a sua volta è costituito da una ConcurrentHashMap<String,String> che ha come chiavi della stessa i nomi degli Utenti (univoci) e la relativa password.
- Il Database relativo agli **utenti registrati** (*utentiRegistrati.json*). Il file contiene un oggetto di tipo UserDB che contiene una ConcurrentHashMap<String, ArrayList<String>> contente tutti gli utenti e i relativi followers. Contiene anche una ConcurrentHashMap<String, ArrayList<String>> che memorizza tutti i tag degli utenti.
- Il Database relativo ai **wallets** (*wallets.json*) che contiene un oggetto di tipo WalletDB che a sua volta contiene un ArrayList<Wallets> di oggetti di tipo Wallet.
- Il Database relativo ai **post** (*posts.json*) che contiene un oggetto di tipo PostDB che a sua volta è formato da una ConcurrentHashMap<Integer,Post> che contiene tutti i post degli utenti.

All'interno della cartella recovery sono già presenti dei file per testare il programma, ma l'eventualità in cui la cartella sia vuota è comunque gestita opportunamente.

Server

Il server, all'avvio, ripristina lo stato del sistema leggendo dalla cartella Recovery l'elenco degli utenti registrati, le credenziali, le informazioni dei vari post e i wallet salvandoli in quattro strutture dati separate: users, credentials, posts, e wallets che sono rispettivamente oggetti di tipo UsersDB, CredentialDB, PostDB e WalletDB.

Successivamente è implementata la pubblicazione dei metodi remoti tramite un'istanza di tipo RegistrationClass, e per quelli con callback con un'istanza ServerNotificationService, infine la connessione TCP viene implementata da un'istanza di MultiThreadedServer.

Client

Il client, all'avvio, inizializza una propria struttura dati locali che contiene l'elenco degli utenti registrati alla piattaforma.

Inoltre, inizializza una variabile username che conterrà il nome utente memorizzato previo login.

Successivamente vengono "prelevati" i metodi condivisi dal server, viene effettuata l'iscrizione al sistema di notifiche (condividendo la funzione di aggiornamento) e instaurata la connessione TCP col server.

Inoltre, viene attivato un Thread Deamon che si occuperà della recezione dei datagrammi multicast mandati dal Server.

Il client legge richieste da standard input, e a seconda della richiesta, decide se eseguirla in locale o farla gestire al server. L'operazione eseguita in locale è list followers che va a leggere la struttura dati locale dell'utente.

Tutte le altre operazioni sono gestite direttamente dal server, ad eccezione della funzione register, che va ad invocare il metodo remoto messo a disposizione dal server.

RMI- Registrazione alla piattaforma

La registrazione di un utente alla piattaforma è gestita tramite il meccanismo RMI. Il server dichiara un'istanza di RegistrationClass che chiama a sua volta il metodo start, il cui scopo è quello di "pubblicare" la funzione all'interno di un registro creato sulla porta 7777 (presa dal file di configurazione); il client inizialmente andrà a "prelevare" questo metodo remoto per poi invocarlo su richiesta.

La chiamata di esso, da parte del client, permette di aggiungere un nuovo utente alla piattaforma: sarà il server ad effettuare i controlli di esistenza e in caso aggiornare la struttura degli users.

Quindi qualsiasi modifica da parte degli utenti verrà effettuata direttamente alla struttura dati del server.

Successivamente la modifica verrà salvata in memoria persistente, tramite le funzioni writeOnDBWallet() (per aggiungere in memoria persistente un nuovo wallet per l'utente), writeOnDBCredentials() (per aggiungere in memoria persistente le credenziali del nuovo utente) e writeOnDBUsers() (per aggiornare in memoria persistente il database degli utenti).

RMI Callbacks- Aggiornamento strutture dati

Per prima cosa il server pubblica, mediante NotificationClass.start, i metodi che permettono all'utente di iscriversi e cancellarsi al sistema di notifica. Questo metodo ritorna un'istanza di ServerNotificationClass, classe vera e propria che implementa sia le funzioni di iscrizione e cancellazione alle callback sia la procedura di notifica update.

Dall'altra parte, il client, ha il compito di recuperare i metodi pubblicati dal server, e a sua volta di pubblicare il metodo update che il server invocherà per mandare la notifica ad un singolo client.

In breve, quando il server invocherà la funzione update, per ogni client iscritto verrà invocata la funzione copyUserDB, che ha il compito di prendere come parametro la struttura dati aggiornata e sostituirla alla struttura dati locale del client: in questo modo ogni client avrà una struttura dati sempre aggiornata.

La funzione update verrà invocata dal server ogni qual volta viene modificata la struttura dati users, in particolare quando un utente segue o smette di seguire un altro utente.

La funzione Update viene chiamata quando viene eseguito il login, in questo modo, il Client riceve la struttura data aggiornata non appena si connette al servizio.

Grazie alla struttura dati sempre aggiornata tramite le callbacks, l'utente può eseguire in locale la richiesta di visualizzare la lista di tutti gli utenti che lo seguono in locale senza doverla inoltrare al server.

TCP- Gestione richieste

Un server TCP dev'essere in grado di gestire richieste provenienti da più client contemporaneamente, ho deciso, quindi, di implementarlo Multithreaded gestendo opportunamente la concorrenza alle strutture dati condivise dai vari Threads: users, wallets, post e credentials .

Per far sì che il server sia in grado di gestire i Threads in modo autonomo ho implementato un Threadpool.

Sulla connessione TCP avviene lo scambio principale di messaggi tra client e server: il client si collega alla porta 55555 (precedentemente scritta sul file di config) su cui è in ascolto il server e la comunicazione avviene secondo il paradigma richiesta-risposta.

Ogni qual volta che il server accetta una richiesta di connessione da un client, viene creato un Task di tipo RequestHandler e passato in gestione al Threadpool.

I Threads del Threadpool hanno il compito di leggere la richiesta del client, eseguirla e infine comunicare l'esito: per ogni richiesta vengono effettuati dei controlli sull'esistenza dell'oggetto richiesto e di accesso da parte dell'utente a quell'oggetto, ad esempio se un utente vuole eliminare un post è necessario che esso esista.

Ogni funzione richiesta dal client ha un proprio handler all'interno del Task, nel caso in cui il server riceva una funzione sconosciuta restituirà un messaggio con l'elenco delle funzioni disponibili.

Ogni qual volta si effettuano modifiche allo stato del sistema (nuovo utente registrato, nuovo post creato, …), queste modifiche vengono salvate in memoria persistente.

UDP Multicast – Rewarding lato Server

Il sistema calcola le ricompense passato un determinato tempo indicato nel file di config come Update Rewarding.

Il Thread salva in una struttura temporanea tutti i post. Dopo aver salvato la struttura dati contenente tutti i post, il Thread va in stato di sleep per il tempo indicato.

Dopo aver esaurito il timer, la funzione prende come parametro la struttura dati attuale e calcola la differenza tra commenti e gli elementi nelle liste upVote e downVote per ricavare il guadagno di ogni utente.

Dopo aver calcolato il guadagno, il Thread scrive le modifiche sulla memoria permanente e manda un messaggio multicast a tutti i client iscritti al gruppo.

UDP Multicast – Rewarding lato Client

All'avvio, Il Client crea un Oggetto di tipo MulticastSocket che permetterà al Client di ricevere messaggi Multicast dal Server.

Questo Oggetto (ms) verrà passato come parametro al costruttore di una classe MulticastingTsk che implementa Runnable.

Dopo aver creato questo Task, il Client avvierà un Thread che lo andrà ad eseguire.

Inoltre, il Thread in questione verrà impostato come Thread Deamon, in modo tale da poter interrrompere il Thread principale senza che la JVM attenda la chiusura di questo Thread.

Il Thread che si occupa dell'esecuzione del Task MulticastingTsk gestisce la recezione dei messaggi Multicast inviati dal Server.

Non appena viene ricevuto un messaggio il Thread restituisce in output un avviso "[E' stato aggiornato il saldo sui Wallet]".

Concorrenza

Ho deciso di creare delle classi, che all'interno avessero delle strutture dati, che riuscissero a gestire la concorrenza.

In particolare:

Nella classe CredentialDB la struttura dati usata per gestire la memorizzazione è una ConcurrentHashMap, che per implementazione garantisce l'accesso alla struttura in maniera Thread safe.

Allo stesso modo, nella classe PostDB, ho deciso di creare una ConcurrentHashMap che gestisca i post in maniera concorrente.

Inoltre, ho deciso di implementare il numero del post (idPost) come un AtomicInteger, in modo tale da avere operazioni atomiche, quali l'incremento e l'accesso, cosicché sia gestita in maniera corretta la concorrenza.

Per quanto riguarda la classe Register, ho deciso di usare blocchi synchronized sulle strutture dati in modo tale da acquisire la lock implicita dell'oggetto e lavorare su di esso, in questo caso aggiungere un Utente.

Ho voluto utilizzare blocchi dove possibile per aumentare la velocità di esecuzione e non bloccare tutte le risorse.

Grazie a questa implementazione ogni thread per accedere alle relative strutture deve prima acquisire la lock implicita dell'oggetto.

Nella Classe WalletDB, invece, ho usato blocchi synchronized per garantire il corretto accesso concorrente all'ArrayList all'interno della classe.

Nella Classe UserDB, ho scelto di utilizzare una ConcurrentHashMap per la memorizzazione dei tag e dei followers degli utenti.

In questo modo, per operazioni atomiche posso supporre che la concorrenza venga gestita implicitamente dalla struttura dati scelta.

Per concludere, nella classe Request Handler, ovvero il task passato per la creazione di una connessione, ho usato blocchi synchronized per avere un'esecuzione Thread safe in tutti i metodi della classe.

Quando un Thread, quindi, vuole modificare una risorsa, visto che essa è stata passata per riferimento, deve acquisire la lock implicita dell'oggetto.

Infine, il campo volatile sulla dichiarazione delle risorse è necessario per assicurarsi che più thread vedano sempre il valore più recente, anche quando il sistema di cache o le ottimizzazioni del compilatore sono al lavoro.

Istruzioni

Librerie Esterne Utilizzate (presenti nella cartella lib):

- jackson-annotation-2.9.7.jar
- jackson-core-2.9.7.jar
- jackson-databind-2.9.7.jar

Compilazione: javac ./client/*.java ./common/*.java ./server/*.java

Esecuzione Server: java server.ServerMainClass

Esecuzione Client: java client.ClientMainClass

Operazioni

register: username password: per registrare un utente;

login: username password: per eseguire il login;

logout: username: per eseguire il logout;

list users: per mostrare tutti gli Utenti con un almeno un Tag uguale al tuo;

list followers: per mostrare i propri followers;

list following: per mostrare chi segui;

follow user *IDutente*: per seguire un utente;

unfollow user *IDutente*: per smettere di seguire un utente;

view blog: per mostrare il tuo blog;

post *"Title" "Content"*: per postare sul tuo blog;

show post *IDpost*: per mostrare un post;

show feed: per mostrare i post di tutti gli utenti che segui;

delete post *IDpost*: per eliminare un post che hai creato;

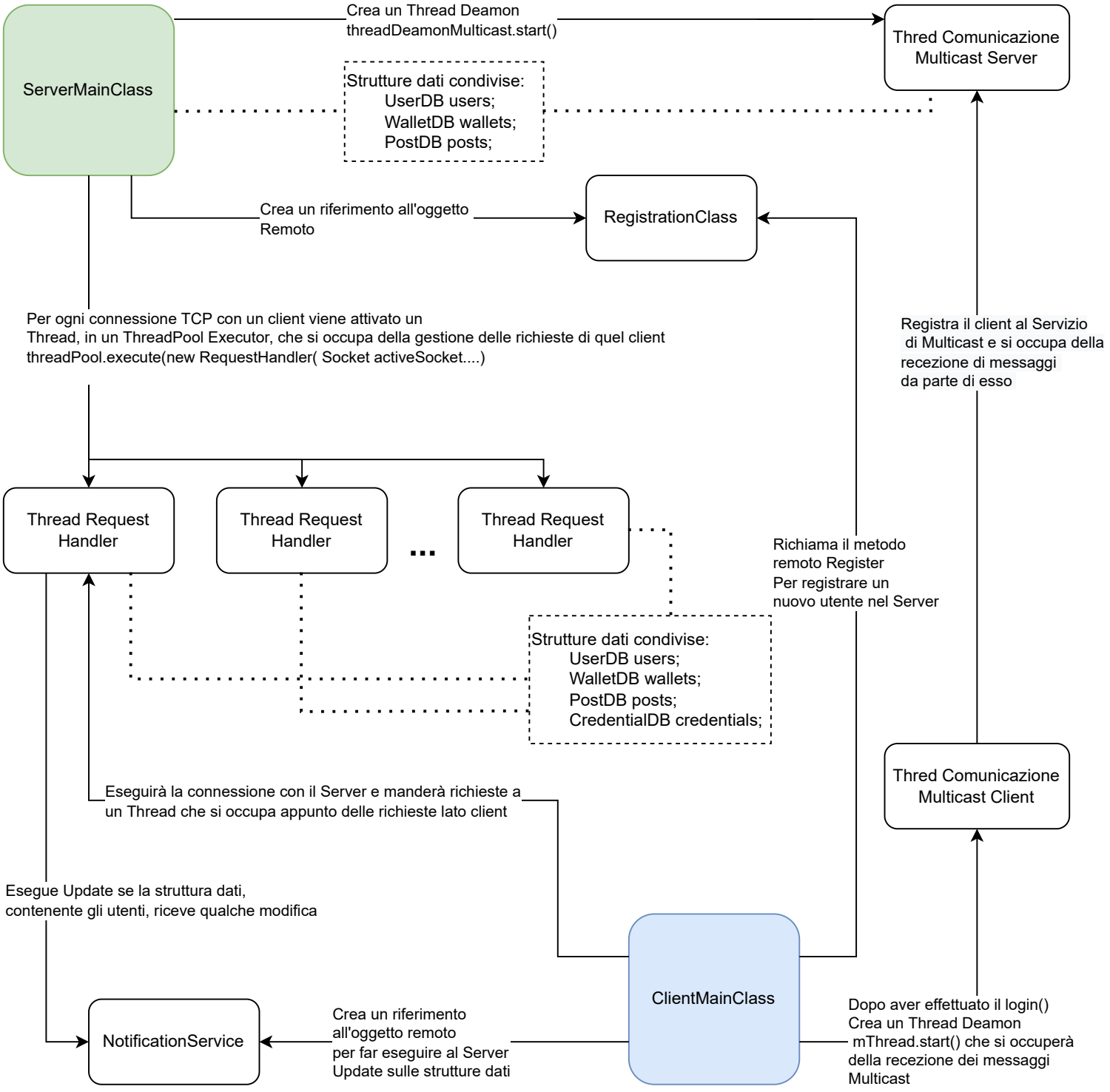
rewin *IDpost*: per inoltrare un post di un altro utente;

rate *IDpost* +1/-1: per votare un post di un altro utente;

comment *IDpost comment*: per commentare un post di un altro utente;

wallet: per vedere il saldo del tuo Wallet;

wallet btc: per vedere il saldo del tuo Wallet in BTC;


ServerMainClass
Crea un Thread Deamon
threadDeamonMulticast.start()
Thred Comunicazione
Multicast Server
Strutture dati condivise:
UserDB users;
WalletDB wallets;
PostDB posts;
Crea un riferimento all'oggetto
Remoto
RegistrationClass
Per ogni connessione TCP con un client viene attivato un
Thread, in un ThreadPool Executor, che si occupa della gestione delle richieste di quel client
threadPool.execute(new RequestHandler(Socket activeSocket....)
Registra il client al Servizio
di Multicast e si occupa della
recezione di messaggi
da parte di esso
Thread Request
Handler
Thread Request
Handler
...
Thread Request
Handler
Richiama il metodo
remoto Register
Per registrare un
nuovo utente nel Server
Strutture dati condivise:
UserDB users;
WalletDB wallets;
PostDB posts;
CredentialDB credentials;
Thred Comunicazione
Multicast Client
Eseguirà la connessione con il Server e manderà richieste a
un Thread che si occupa appunto delle richieste lato client
Esegue Update se la struttura dati,
contenente gli utenti, riceve qualche modifica
ClientMainClass
NotificationService
Crea un riferimento
all'oggetto remoto
per far eseguire al Server
Update sulle strutture dati
Dopo aver effettuato il login()
Crea un Thread Deamon
mThread.start() che si occuperà
della recezione dei messaggi
Multicast