UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea in Informatica

Progetto: Winsome, a rewarding social media

Laboratorio di reti di calcolatori

Anno accademico: 2021/2022
Prof. Laura Ricci

Lorenzo Jori
607152 - Corso A

Introduzione:

Il progetto, che di seguito viene presentato nei dettagli, ha come idea di base quella di creare un social network dove gli utenti vengono premiati per i post che pubblicano o per le interazioni che hanno con essi (i così detti curatori, che non sono gli owner del post ma sono coloro che contribuiscono alla sua popolarità aggiungendo commenti, e valutazioni: like e dislike). Winsome è inspirato da un reale social network che si chiama Steemit, basato su blockchain. Le principali funzionalità tipiche dei social come il post, il like e il following sono tutte implementate da Winsome, a seguito la trattazione più dettagliata darà un'idea della implementazione di tali funzionalità. In aggiunta alle funzioni social è presente il sistema di ricompensa periodica per curatori ed autori di post.

Trattazione dettagliata

Struttura ed architettura generale:

L'architettura di base del progetto è quella del classico modello client-server multithread. Nella mia implementazione sia client che server sono entrambi processi multithread che comunicano tra di loro scambiandosi messaggi. (di seguito sarà fatta una trattazione più dettagliata dell'architettura).
Il modello scelto è stato quello multithread e non NIO con multiplexing perché ho trovato più consono utilizzare multithreading in quanto mi sento più confidente con il multithreading rispetto al multiplexing (nonostante effettivamente il multiplexing, possa offrire delle performance migliori in caso di un grosso quantitativo di thread attivi sul server).
Server e client interagiscono mediante connessione TCP, UDPmulticast (per le notifiche del calcolo delle ricompense) e RMI per l'invocazione dei metodi remoti per la registrazione e per il servizio di follow/list follower.
Di seguito è riportata una figura che spiega l'architettura generale del software.

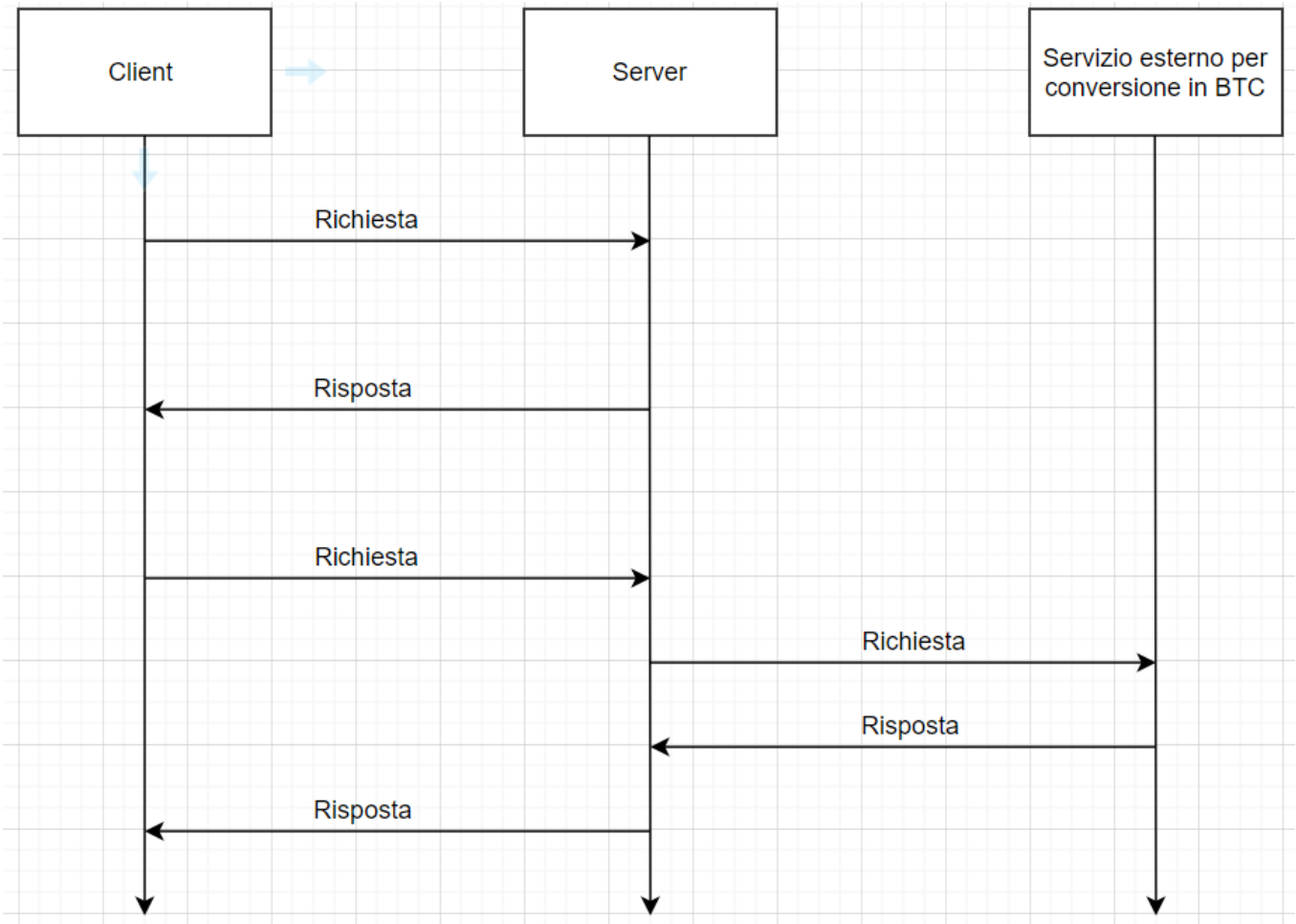

Figura 1 architettura generale del software

Come si può vedere dalla figura il server, qualora venga richiesto dal client, si connette ad un servizio esterno tramite URL, al che, il server invia una richiesta di tipo GET per ottenere il tasso di conversione dei winCoin in Bitcoin. (Il servizio a cui si connette il server non è niente di più di un semplice generatore di numeri casuali, esso genera un numero che poi verrà utilizzato per fare la conversione come sopra detto).

Specifica dei metodi implementati:

- Register(): metodo per la registrazione di un utente, implementato con RMI usando l'interfaccia Registration e implementata dalla classe WorkerServerRMI
- Login(): metodo per il login dell'utente, implementato nella classe WorkerServerTCP lato server. Lato client è un metodo statico che invia le richieste al server, compie l'hashing della password per non inviarla in chiaro al server e setta un flag di controllo per non permettere che un utente si logghi quando c'è già un altro utente loggato. Al momento del login, sempre lato client, viene fatta la registrazione dell'utente al servizio RMI per il

follow (aggiornamento lista follower). Il client al momento del login riceve dal server i riferimenti per mettersi in ascolto sulla multicastSocket.

- Logout(): il servizio di logout implementato nel metodo logout() non fa nient'altro se non che scollegare l'utente dal social
- ListUsers(): il servizio implementato dal seguente metodo è quello di ritornare la lista degli utenti che hanno almeno un tag in comune con il client richiedente il servizio
- ListFollowers(): ritorna la lista dei followers del client richiedente il servizio (implementato con RMI)
- ListFollowing():
- FollowUser(): il servizio segue l'utente il cui username viene specificato da riga di comando (gli user name per scelta implementativa devono essere univoci)
- UnfollowUser(): il servizio offerto dal metodo permette al client di smettere di seguire un utente specificato da riga di comando
- ViewBlog(): il servizio offerto dal metodo permette di vedere la lista dei post creati dal client che richiede il comando
- CreatePost(): servizio che permette di creare un post. Il post creato viene aggiunto al blog e al feed degli utenti che il client segue
- ShowFeed(): servizio che mostra il feed del client che lo richiede
- ShowPost(): servizio che mostra un post specificando l'id del post da riga di comando
- DeletePost ():servizio che consente di eliminare un post dato il suo id da riga di comando. Elimina anche gli eventuali rewin dal blog degli utenti che hanno fatto il rewin
- RewinPost(): servizio che permette di condividere un post dato il suo id da riga di comando
- RatePost (): servizio che consente di valutare un post dato l'id da riga di comando e un voto: 1 se voto positivo, -1 se voto negativo
- AddCommet(): servizio che permette di aggiungere un commento ad un post. Se colui che vuole aggiungere il commento è il creatore del post, il post non viene commentato. Questo per far si che un utente non si commenti da solo per guadagnare wincoin
- GetWallet(): permette di vedere lo storico delle transazioni in wincoin
- GetWalletBitcoin(): permette di convertire il wallet in bitcoin collegandosi ad un servizio esterno alla URL: https://www.random.org/decimal-fractions/?num=1&dec=4&col=1&format=plain&rnd=new

Nota: Per il servizio di login ho aggiunto, come scelta implementativa, quella di mandare al sever un idClient (implementato come un numero casuale) nel momento in cui richiede il login. In questo modo posso avviare più processi client e fare il login con lo stesso user. Grazie a questo id salvo sul server una lista di client loggati, al momento che uno dei client duplicati richiede il logout riesco a identificarlo grazie all'id e quindi a scollegare il corretto processo client che ha richiesto il logout. [questa scelta è stata ispirata dai moderni social che ti permettono di essere loggato sia da telefono che da pc; quindi, ho pensato di emulare questa cosa offrendo l'implementazione di cui sopra].

Dettagli architetturali:

Le classi principali del mio softwer sono:

- **Client.java:** implementa i metodi richiesti dalla specifica del progetto
- **Server.java:** implementa il server, che si occupa di accettare connessioni dai client e avviare i thread necessari al progetto
- **WorkerServerTCP.java:** classe che implementa i task che i thread del threadpool devono compiere, al suo interno vi è un ciclo while che contiene una serie di if per identificare il task che viene richiesto dall'utente. Una volta identificato il task da compiere viene invocato il corrispettivo metodo che implementa il servizio richiesto.
- **RevenueWorker.java:** implementa il task del calcolo delle ricompense, viene svolto periodicamente da un thread ogni tre minuti. Thread avviato indipendentemente dal pool. Esso invia anche la notifica al gruppo di multicast.
- **JsonThread.java:** implementa il task del salvataggio dello stato del server su file json, viene svolto periodicamente da un thread ogni cinque minuti. Thread avviato indipendentemente dal pool (quindi ad esso esterno).
- **ListenerForNotification.java:** implementa l'ascolto sulla multicastSocket
- **WorkerServerRMI.java:** implementa la registrazione di un utente sul social tramite chiamata a metodo remoto usando: java RMI.
- **User.java:** la classe User mi serve per incapsulare le informazioni di un utente. Essa contiene anche la lista dei post che un utente ha creato (può sembrare una ridondanza con la classe blog, ma nel blog ci finiscono anche i post condivisi dall'utente mentre nella lista della classe User ci finiscono solo i post che l'utente ha creato col comando "post")
- **Post.java:** la classe post invece mi serve per incapsulare le informazioni relative ad un post come ad esempio: il creatore, l'id, il numero di valutazioni, il titolo e il contenuto del post.

Client:

Il client, così come il server, sono entrambi processi multithread. I thread del client sono rispettivamente: il thread main che si occupa di inoltrare le richieste al server e ricevere le risposte tramite connessione TCP (o invocazione di metodi remoti e callback RMI nel caso del servizio di registrazione e di follow); il thread listenerForNotification che si mette in ascolto sulla multicast socket, aspetta l'arrivo di un messaggio dal server (quando sono state calcolate le ricompense) e lo stampa a video. [di seguito la figura 2 spiega la struttura dei thread del client]. Ho scelto di fare un thread dedicato all'ascolto sulla multicastSocket perché in questo modo il client è sempre in ascolto sulla multicastSocket in attesa di notifiche fa parte del server. Il processo client si occupa di prendere l'input da tastiera dell'utente (Istanziando e usando un oggetto della classe Scanner) e, nel metodo main, compie un ciclo "while(true)" in attesa dei comandi inseriti dall'utente, l'interpreta e inoltra le richieste

al server. Nel ciclo while sono presenti una serie di “if” per identificare il comando inserito, nel loro corpo vi è la chiamata a dei metodi statici che si occupano di inoltrare richieste e ricevere risposte.

L’unica struttura dati che il client utilizza è una LinkedBlockingQueue che viene usata per mantenere localmente nel client gli username dei follower che ha un utente del social. Per terminare il processo client ho scelto di usare semplicemente “ctrl + c”, in quanto inserire una sequenza di caratteri, da tastiera, per la terminazione mi sembrava ridondante.

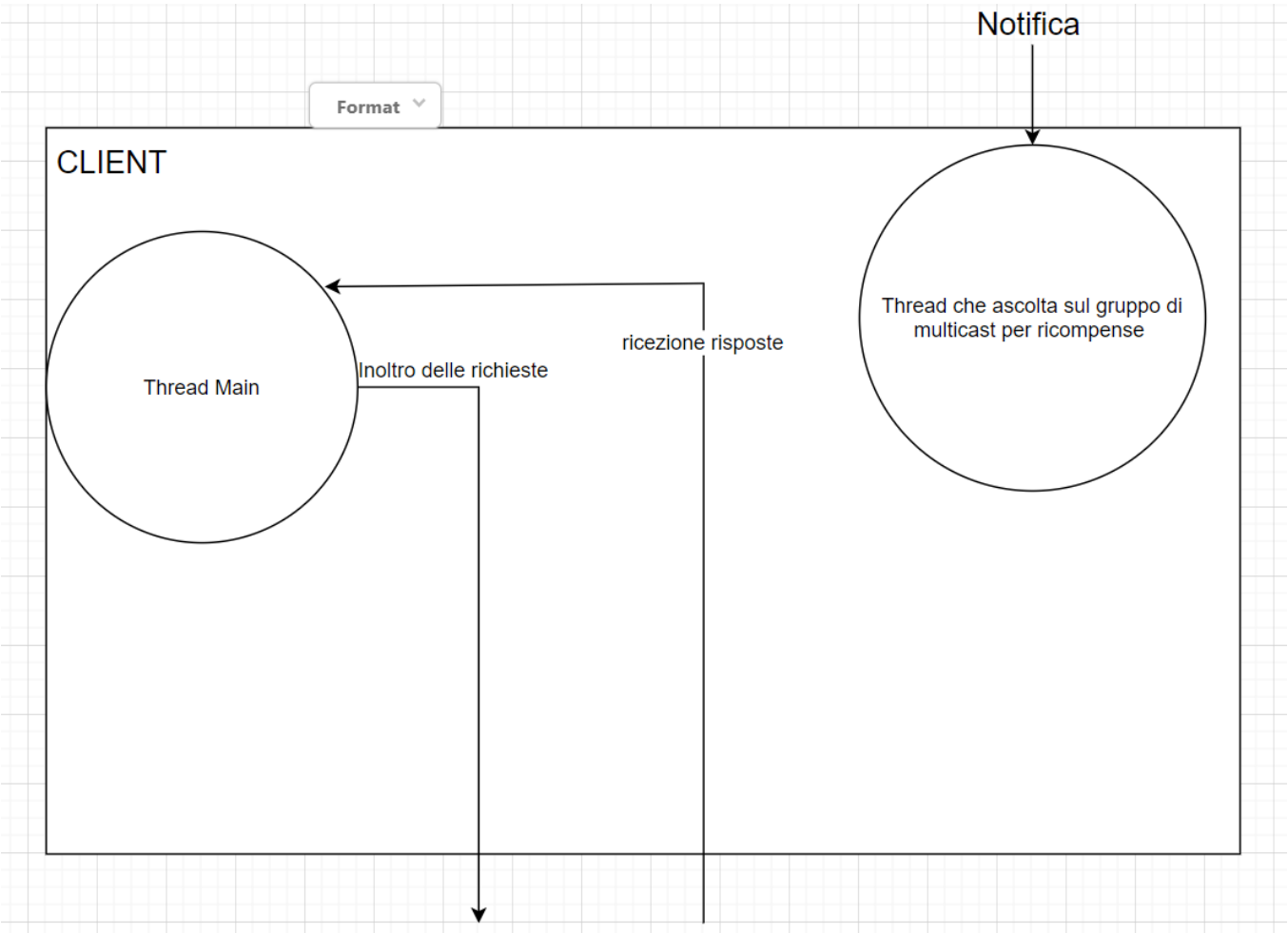

Figura 2 Client thread architecture

Server:

Il server ha un'architettura un po' più complessa in quanto sono coinvolti più thread e persino un threadPool. Anch'esso esegue un ciclo "while(true)" dove attende le richieste di connessione da parte dei client. In dettaglio: il server costruisce un listening socket invocando il costruttore della classe ServerSocket e nel ciclo while si mette in ascolto di connessioni da parte dei client invocando il metodo bloccante "serverSocket.accept()", che sta in ascolto e in attesa finché una connessione non viene stabilita. La porta per questo servizio è fornita mediante file di configurazione. Il server attiva un thread per connessione accettata. Per implementare il multithreading, oltre a far partire singolarmente thread tramite il metodo thread.start(), ho scelto di usare un threadPool: la scelta è stata quella di usare un cachedThreadPool perché mi sembrava un buon compromesso per le seguenti caratteristiche: riutilizzo dei thread (visto che viene attivato un thread a connessione, il riuso dei thread, non più occupati da alcun task, aiuta a non avere un sovraccarico in termini di CPU e memoria), il fatto che comunque i task richiesti non sono poi di così gran dimensione e nessun limite alla dimensione del pool fa sì che un client avrà, con una buona probabilità, un thread a disposizione. Gli altri thread esterni al threadPool sono rispettivamente: il thread che si occupa di calcolare periodicamente le ricompense, aggiornare i wallet e comunicare sulla multicastSocket la notifica dell'avvenuto calcolo e il thread che si occupa periodicamente di salvare lo stato del server su dei file json. Vengono salvate le informazioni relative agli utenti e ai post. Ho scelto di fare thread separati per il calcolo delle ricompense e per il salvataggio dello stato del server perché essendo task che devono svolgersi di continuo, periodicamente, trovavo più giusto implementarli come thread assestanti, piuttosto che facenti parte del thread pool.

[Di seguito la figura 3 spiega la struttura dei thread del server].

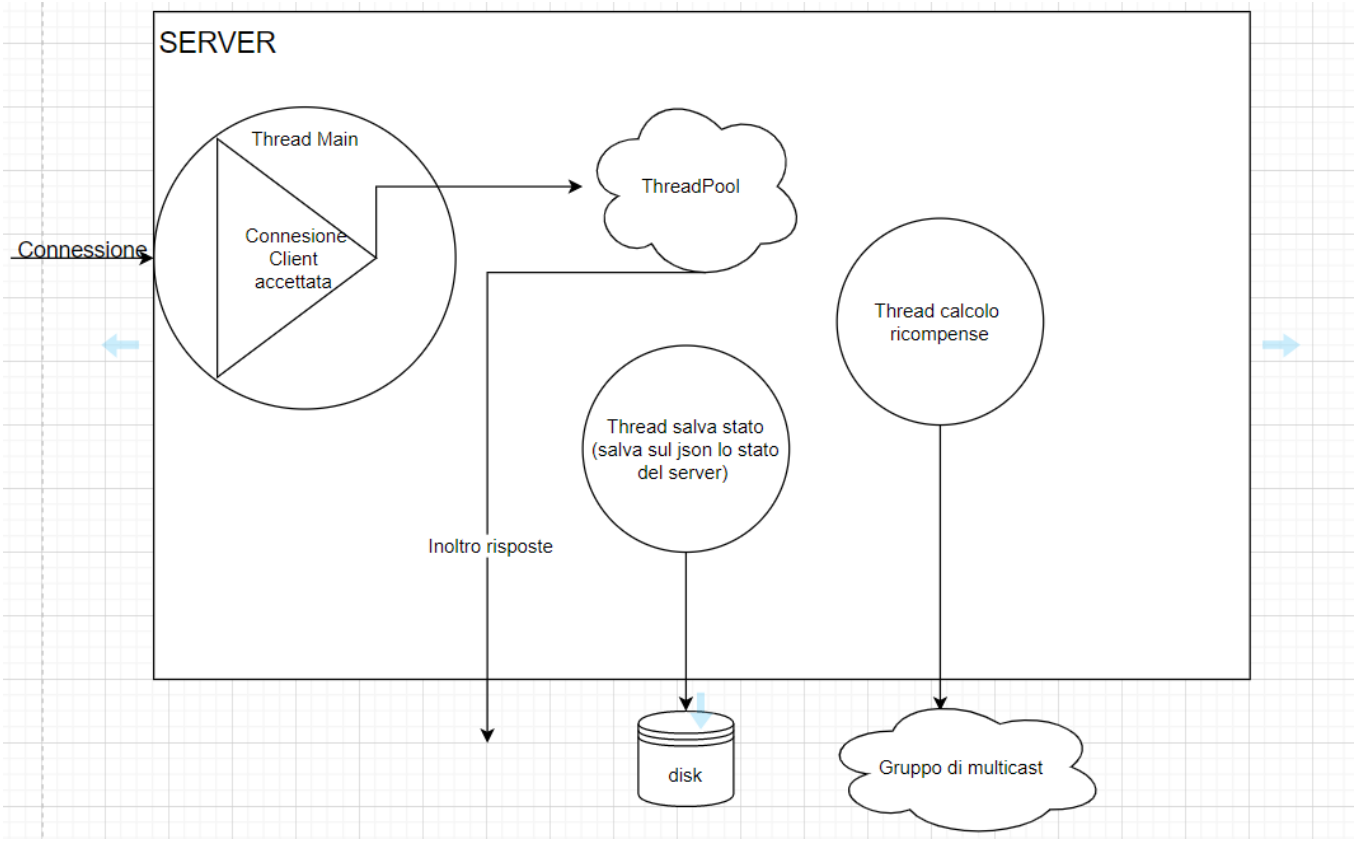

Figura 3 Server thread architecture

Strutture dati lato server:

Le strutture dati che il server utilizza per l'implementazione del social sono essenzialmente delle ConcurrentHashMap che sono definite su tipi non primitivi, quindi su classi definite da me, come la classe User, la classe Post e la classe Blog. L'utilizzo di queste strutture dati concorrenti mi permette di avere accesso ad esse in modo thread safe e visto l'architettura multithread del server ho pensato che potessero essere una buona soluzione per garantire una gestione corretta della concorrenza. La scelta di design di usare delle hashMap mi porta ad avere una frammentazione dello stato del social che comporta, in alcuni casi come ad esempio: il ripristino dello stato dopo l'interruzione del processo server, che la complessità algoritmica aumenti ad ordini superiori ad O(n). Le mappe:

- usersProfile <User, Blog>: contiene profili e blog degli utenti
- posts <User, Post>: contiene gli utenti e i post
- wallet <String, Wallet>: contiene le transazioni associate agli utenti
- socialNetwork <Pair<User, Blog>, Home>: definisce il social in termini di feed, blog e utenti. Tutto in un'unica mappa. [nota: la classe Home, sarebbe il feed. L'ho chiamata Home per avere io più chiarezza in fase d'implementazione].

essendo definite non su tipi primitivi (a parte la mappa wallet) mi hanno causato difficoltà e problemi in fase di serializzazione e deserializzazione per salvare lo stato del server. Ho dovuto costruire la stringa json "a mano" (cioè ho dovuto, scorrere le mappe per ogni entry e costruire la stringa, aggiungendo persino le parentesi quadre) per la mappa userProfile e per la mappa post. Questo errore di design è dovuto al fatto che non avevo preso in considerazione, durante il design del progetto, che Gson prova a risolvere ricorsivamente i riferimenti alle classi e ciò può causare stackOverFlow in casi di risoluzioni ricorsive circolari, dovute a definizioni circolari, che causano la creazione di file json di grandi dimensioni.

Note:

Credo sia giusto menzionare il fatto che nella classe Post e nella classe User ho dovuto fare l'override del metodo equals fornito da java nella classe Object. Questa necessità è nata dal fatto che uso il metodo "remove(<argomento>)" di java, per rimuovere i post che un utente vuole eliminare e per rimuovere gli utenti che smettono di seguire qualcuno dalla lista dei follower. Le strutture dati che contengono i follower e i post contengono oggetti di tipo Post e di tipo User. Il metodo remove(<argomento>) prende l'argomento e lo confronta usando il metodo equals() della classe Object. Dunque, quando andavo a chiamare la remove il confronto veniva fatto sugli oggetti e non su idPost e username (ciò portava alla non rimozione dalla struttura dati, perché non veniva confrontato l'oggetto passato ma una copia di esso che quindi risultava diversa dall'oggetto da rimuovere). Al che ho dovuto fare l'override in modo tale che il metodo confrontasse i post in base all'id e gli utenti in base allo username per trovare l'istanza (della classe post e della classe user) da rimuovere dalle strutture dati.

Inoltre, l'incremento del contatore delle valutazioni di un post viene fatto chiamando un metodo synchronized, definito nella classe post ed invocato dal thread che si occupa del calcolo periodico delle ricompense.

Per concludere all'interno del mio codice si può notare una ridondanza sui post cioè: io salvo i post in due liste, una definita nella classe Blog e una definita nella classe User. il motivo di questa scelta è che la lista della classe User contiene solo i post che vengono creati da quel particolare utente. Mentre nella lista della classe Blog ci finiscono tutti i post anche quelli che vengono condivisi (usando il comando rewin). Questa ridondanza dell'informazione è quindi usata per tenere traccia di quali sono i post creati da un particolare utente, in modo tale da avere frammentazione sui post così che sia più facile non perdere informazioni su di essi.

File .jar: non sono riuscito a creare i file jar in quanto l'IDE utilizzato (intellij) ha delle impostazioni che non sono in grado di cambiare che non mi permettono di creare i file .jar in quanto mi compare la notifica che il codice potrebbe creare file non sicuri. Ho provato a seguire la procedura per crearli

manualmente ma non sono riuscito nella creazione del file “manifest”. File che serve per capire quali classi contengono il metodo main al loro interno.

Contenuto file di configurazione:

Client:

Per il client il file di configurazione che viene letto all’avvio del processo contiene le seguenti informazioni:

```
#CLIENT_CONFIG_FILE

# porta TCP
TCPPORT=5454

# porta RMI
RMIREGISTRATIONSERVICE=5458

# porta RMI per il servizio di follow
RMIFOLLOWSERVICE=5896

# indirizzo InetAddress
INDIRIZZOINETADDRESS=localhost

# hostRMI
HOSTRMI=localhost
```

Server:

Per il server il file di configurazione che, anch’esso, viene letto all’avvio del processo; contiene le seguenti informazioni:

```
#SERVER_CONFIG_FILE

# porta TCP del server:
TCPPORT=5454

# Indirizzo di multicast:
```

```
MULTICAST=230.0.0.0

# Porta di multicast
MCASTPORT=4446

# porta del registry RMI per servizio di registrazione:
REGISTRYPORTREGISTRATION=5458


# porta registry per servizio di following:
REGISTRYFOLLOWINGPORT=5896

# Tempo di attesa per il calcolo delle ricompense
TIMETOWAIT=200000
```

Compilazione ed esecuzione:

Per compilare da terminale il progetto basta inserire il comando: javac -cp ".;./lib/gson-2.8.9.jar" *.java

Per eseguire il processo server il comando è il seguente: java -cp ".;./lib/gson-2.8.9.jar" Server

Per eseguire il processo client il comando è il seguente: java Client