RELAZIONE PROGETTO WINSOME

Studente: Stefano Passanante

Matricola:597415

INTRODUZIONE

Descrizione del sistema

Il progetto consiste nello sviluppo di una rete sociale orientata ai contenuti (un “social media network”) ispirato a STEEMIT: una piattaforma social basata su blockchain la cui caratteristica più innovativa è quella di offrire una ricompensa agli utenti che pubblicano contenuti interessanti e a coloro che votano/commentano tali contenuti. Questa rete sociale identificata con il nome di WINSOME deve quindi permette la registrazione degli utenti. Ogni utente registrato e successivamente loggato, può accedere a vari comandi disponibili, es: follow di un utente, creazione di un post, vedere i post del proprio feed e altri comandi elencati nell’interfaccia utente. Per sviluppare il suddetto social network è stato utilizzato il paradigma client-server (richiesta-risposta) e la successiva implementazione delle due componenti principali: WInsomeClient e WinsomeServer

WinsomeServer

All’avvio controlla se c’è un file di backup da cui riprendere lo stato del sistema, gestisce la fase di registrazione tramite RMI, memorizza tutti gli utenti in una Map che associa alla stringa dell’username un’istanza della classe Utente e le loro relazioni di follow/followed in Set di stringhe, che contengono rispettivamente gli username degli utenti che li seguono e gli username degli utenti che essi seguono, avvia un thread per effettuare, periodicamente il calcolo delle ricompense e invia tramite multicast una notifica per avvisare dell’avvenuto calcolo, gestisce i contenuti prodotti dagli utenti (post, commenti, voti). Inoltre permette una conversione da wincoin (criptomoneta di Winsome) in bitcoin, esegue periodicamente il backup per maniere lo stato del sistema

WinsomeClient

Gestisce l’interazione con l’utente, tramite linea di comando, invia al server le operazioni richieste dall’utente e ne restituisce il risultato secondo, appunto, il paradigma client-server, partecipa anche ad un gruppo di multicast da cui riceve messaggi di avvenuto calcolo delle ricompense, espone anche un oggetto remoto, che servirà al server per segnalare nuovi follow o unfollow all’utente loggato (se è presente un utente che ha eseguito il login)

Scelta del protocollo di comunicazione

Per la comunicazioni di queste due componenti ho scelto il protocollo TCP, quindi un protocollo orientato alla connessione e affidabile. La prima cosa che il client fa al suo avvio, è quella di stabilire una connessione TCP con il server(connessione persistente fino a quando il processo WinsomeClient è attivo), quindi non apro una connessione nuova per ogni utente che fa un login sul processo WinsomeClient, ma avvio un’unica connessione all’apertura del client.Successivamente inizializzare i buffer di scrittura e di lettura. Solamente dopo l’esito positivo di questi passaggi un utente registrato può proseguire con il log-in e l’invio dei vari comandi. Ho preferito utilizzare il protocollo TCP invece che l’UDP, quindi a discapito delle prestazioni, per avere un’unica connessione affidabile per tutta la durata delle comunicazioni tra client e server, in modo tale che il protocollo stesso mi gestisse i casi di errore nella comunicazione: pacchetti corrotti, dispersi o fuori sequenza. Il server rimarrà sempre acceso per tutta la durate della sua “vita”, il client invece quando un utente vorrà chiudere la connessione con il server (per fare ciò l’utente digiterà il comando exit), invierà al server gli ultimi messaggi per avvertirlo della chiusura della connessione e in seguito, terminerà il processo.

SERVER

Strutture dati utilizzate

La struttura dati principale del server è la seguente:

static final private Map<String,Utente> utenti

Inizializzata tramite il metodo *synchronizedMap* della classe Collections in modo tale da avere i metodi delle Map utenti protetta da Lock, quindi thread-safe. Inoltre ad ogni thread che va ad utilizzare la Map utenti, non accede mai all'oggetto originale, ma alla Map restituita dalla classe Collections. In questo modo tutte le operazioni individuali sulla map eseguite da ogni thread, sono safe, e per operazioni composte, per renderle atomiche, vado ad utilizzare metodi synchronized o blocchi di codice synchronized.

Altra struttura dati molto importante per il server è stata:

static final private Set<String> utenti_loggati

Come per la *Map utenti* anche in questo caso utilizzo la classe Collections per creare una versione sincronizzata della *Set.* utenti_loggati mi contiene gli username, degli utenti che hanno effettuato il login, in modo tale da evitare che un utente già loggato con un client, possa fare il login con un altro client.

Oltre alla classe WinsomeServer, ho dovuto definire la classe Utente e Post per contenere tutte le informazione riferenti agli utenti registrati e ai post che essi creano, quindi ho utilizzato altre strutture dati per queste due classi, che verrano definite in seguito nella sezione: **Descrizione risorse server.**

Architettura server

Per la realizzazione del server ho utilizzato Java I/O e un cachedThreadPool per la gestione dei client che si connettono ad esso. Ho deciso di utilizzare questo approccio in modo tale da dedicare ad ogni client attivo un thread che vada a risolvere le sue richieste. Quindi per ogni client che crea una connessione con il server, tramite il threadpool gli si assegna un thread, in questo modo il *mainthread* avrà un carico di lavoro inferiore, perché dovrà solamente occuparsi di stare in ascolto per ricevere i client che richiedono la connessione e ognuno di essi sarà affidato ad un thread che gestirà tutto il lavoro in maniera efficiente. Con questo approccio un client, con un'unica connessione potrebbe gestire più utenti (non in contemporanea). WinsomeServer oltre ad avviare un cachedThreaPool.WinsomeServer avvia anche altri 3 thread: *backup, guadagno, chiudi_server*. Il thread backup, sempre attivo per tutta la durata del server effettua il backup degli utenti registrati e di tutte le loro informazioni (follow, followed, rate, feed, blog). Il backup oltre ad eseguito in caso di chiusura istantanea del server, viene anche fatto periodicamente (periodo scelto nel file di inizializzazione). Il ripristino viene effettuato grazie alla libreria esterna jackson che permette di scrivere file con estensione .json. Quindi, o periodicamente o prima della chiusura del server il thread in questione salverà la struttura dati principale *Map<String,Utente> utenti* che tramite il metodo *valueToTree()* di objectMapper viene rappresentata in un JsonNode e salvata nel file *Winsome.json.* Ogni volta che il server viene riavviato, apre il file *Winsome.json* e se contiene informazioni, ripristina lo stato del sistema. Il thread guadagno, anche esso, sempre attivo dall'avvio del server, effettua periodicamente (periodo scelto nel file di inizializzazione del server) il calcolo delle ricompense per gli autori dei post e per i loro curatori (utenti che hanno commentato e/o votato i post) dopo aver effettuato il calcolo delle ricompense, aggiorna il Wallet degli utenti e invia tramite UDP multicast una

notifica a tutti gli utenti loggati e quindi registrati al gruppo mutlicast, per avvisarli dell'avvenuto calcolo. Il thread chiudi_server invece, rimane in attesa della stringa "exit", quando dal lato server verrà scritta da linea di comando, verrà effettuato e il backup e il processo server verrà brutalmente terminato. Spiegazione più approfondita nelle sezione: *Chiusura Server.* WinsomeServer estisce tramite RMI le registrazioni degli utenti, quindi esporta un oggetto che permette al client che lo richiede, di fare registrare l'utente a Winsome (Quindi come da specifica, l'iscrizione di un utente a Winsome non si basa sulla connessione TCP. Viene anche utilizzato RMI callback per notificare ad utente delle ricezione di un nuovo follow o della perdita di un follow. RMI e RMI callback verranno approfonditi nella sezione successiva.

RMI

Ho deciso di utilizzare due oggetti remoti separati: *register_rmi* e *notifica_rmi*. Il primo, implementa i metodi dell'interfaccia: *InterfacciaRemota,* il secondo implementa i metodi di *Interfaccia_Callback.* Ho deciso di procedere in questo senso, in modo tale da poter fare una giusta distinzione dei metodi che possono essere utilizzati per i due oggetti remoti. Inoltre *notifica_rmi* viene passato al thread che gestisce il client che ha effettuato la connessione, così, quando un utente inizierà a seguire un altro utente, utilizzando il metodo *segnala* dell'oggetto *notifica_rmi,* gli segnalerà l'evento e aggiornerà la sua struttura dati. *notifica_rmi* espone i metodi *registra_al_callback* e *unregister_al_callback* per permetta al client di registrarsi e ricevere le notifiche.

Descrizione risorse server

Per gestire tutte le informazioni necessarie per il funzionamento del social network Winsome, ho dovuto definire due ulteriori classi: *Utente e Post*. Il WinsomeServer utilizza i metodi implementati dal due classi, per avere le istanze dei vari utenti e dei loro post, e per eseguire le richieste che arrivano dai client. Utente implementa metodi necessari per aggiungere un follow, eliminarlo, creare post, aggiornare feed e blog di ogni utente, necessari per il giusto funzionamento del server, invece la classe Post implementa metodi per aggiungere dei voti ai post o per aggiungere commenti. Le principali strutture dati utilizzate in Utente, sono: *private final Set<String> followers, private final Set<String> follow, private final Set<Post> blog, private final Set<Post> blog. Post invece utilizza:* una *Map<String,List<String>> commenti* per tenere traccia degli username degli utenti e dei loro commenti al post in questione. Oltre a questa Map, tiene traccia dei voti positivi e negativi al post, tramite dei Set<String> like, Set<String> disdice che mantengo gli username degli utenti che hanno votato "+1"(like) o "-1"(dislike).

Gestione della concorrenza

Come detto in precedenza il server utilizza un cachedThreadPool per gestire i client. Dato il possibile avvio di vari thread che lavorano in contemporanea, è indispensabile una gestione dalla concorrenza efficace. Per fare ciò oltre ad aver utilizzato come detto i precedenza strutture dati sincronizzate fornite dai metodi di Collections, ogni metodo di Utente e Post a cui accede WinsomeServer è sincronizzato. In questo modo non devo aggiungere manualmente delle lock per avere una mutua esclusione, ma il tramite il metodo synchronized Java associa a ciascun oggetto una lock implicita e una coda associata tale lock. Quindi, quando il metodo synchronized viene invocato, il metodo tenta di acquisire la lock intrinseca associata all'oggetto su cui esso è invocato. Così facendo, la gestione della concorrenza è garantita, essendo gestita ad un livello di astrazione più alto.

Chiusura server

WinsomeServer termina non appena si digita da linea di comando “exit”. Non ho effettuato un chiusura “pulita” quindi interrompendo ogni thread attivo, eliminando gli oggetti esporti e chiudendo la connessione, perché, dato che un server deve rimanere sempre attivo, ho inteso la chiusura del server, come un rimedio ad un problema al suo interno. Terminando il processo in maniera bruta con un *System.exit(0)* si evita la propagazione dell’errore che ci potrebbe essere e si chiude istantaneamente, dopo aver effettuato il backup

CLIENT

Architettura client

Come da specifica la sua struttura è quella di un “thinclient”, invia solamente richieste al server e ne stampa le risposte, tranne che per l’operazione di *list followers* in cui, non si invia nessuna richiesta al server ma viene gestita dal client stesso, grazie alla struttura dati: *private final static Set<String> followers* che nel momento del login effettuato con esito positivo, recupera lo stato della collezione ricevendo dal server i followers che l’utente ha acquisito fin in quel momento, da li in poi gli aggiornamenti a followers verranno fatti localmente, grazi all’oggetto remoto callback che riceve le notifiche e aggiorna followers. Poi fa un controllo sugli errori nei comandi scritti dall’utente per evitare di inviare al server comandi che possono creare errore. WinsomeClient inoltre utilizza il thread notifica_ricompensa che viene avviato quando un utente esegue il login correttamente, una volta avviato si registra ad un gruppo multicast per ricevere messaggi di notifica per un nuovo calcolo delle ricompense avvenuto. Con *notifca_ricompensa* ho esteso la classe Thread, così da poter fare l’ovverride del metodo interrupt e implementare un nuovo metodo: stop_notifche che viene utilizzato per togliere il client dal gruppo multicast a cui si era registrato. Ho fatto l’override del metodo interrupt così che, quando viene chiamato oltre a notificare la richiesta di terminazione del thread, chiude anche la socket per l’UDP multicast. quando l’utente effettua un logout: si resetta la struttura dati *followers,* si chiama il metodo stop_notifica del thread notifica_ricompensa per non ricevere più le notifiche del calcolo delle ricompense e invia al server la richiesta di logout. Da notare, come detto in precedenza, che per ogni client viene istaurata una sola connessione TCP con il server, anche quando un utente effettua il logout il client rimane connesso, in attesa di un altro utente che effettui il login. La connessione con il server viene chiusa quanto un utente digita da linea di comando “exit”.

Terminazione client

Quando un utente digiterà da linea di comando “exit”, WinsomeClient andrà chiamerà il metodo interrupt del thread notifica_ricompensa, rimuoverà l’oggetto remoto precedentemente esportato e invia un messaggio di “saluto” al server per avvertirlo della chiusura della connessione, dopo aver ricevuto risposta dal server, la connessione verrà chiusa e il processo terminerà.

ISTRUZIONI COMPILAZIONE

Compilazione ed esecuzione server

Per compilare il server è necessario eseguire il seguente comando:

- javac -cp lib/jackson-annotations-2.9.7.jar:lib/jackson-core-2.9.7.jar:lib/jackson-databind-2.9.7.jar src/WinsomeServer/*.java src/Risorse/*.java src/WinsomeClient/*.java -d out/production

Comando per eseguire il server:

- java -cp ./out/production:lib/jackson-annotations-2.9.7.jar:lib/jackson-core-2.9.7.jar:lib/jackson-databind-2.9.7.jar WinsomeServer.WinsomeServerMain

Nella file del progetto è anche presente un script ./compila_eseguiServer.sh e anche il file jar associato, presente nella cartella out/artifacts, comando per eseguirlo:

- java -jar out/artifacts/ServerWinsome.jar

Compilazione ed esecuzione client

Per avere l'interfaccia da linea di comando è necessario compilare il WinsomeClient con il seguente comando:

- javac -cp ./lib/jackson-annotations-2.9.7.jar:./lib/jackson-core-2.9.7.jar:./lib/jackson-databind-2.9.7.jar src/WinsomeClient/*.java src/WinsomeServer/*.java src/Risorse/*.java -d out/production

E successivamente eseguire WinsomeClient con il seguente comando:

- java -cp ./out/production:./lib/jackson-annotations-2.9.7.jar:./lib/jackson-core-2.9.7.jar:./lib/jackson-databind-2.9.7.jar WinsomeClient.WinsomeClientMain

Anche per il client è disponibile lo script ./compila_eseguiClient.sh e il file jar associato, presente nella cartella out/artifacts comando per eseguirlo:

- java -jar out/artifacts/Client.jar

Ps: Io ho utilizzato questi comandi per il terminale del computer MacBook Air