Relazione di Davide Marchi
Matricola 602476

Relazione del progetto di laboratorio di reti WINSOME: a reWardING SOcial Media. A.A. 2021/22

Introduzione

Il progetto consiste nella implementazione di WINSOME, un social media che si ispira a STEEMIT (https://steemit.com/), 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 (i curatori) che votano/commentano tali contenuti. La ricompensa viene data in Steem, una criptovaluta che può essere scambiata, tramite un Exchanger, con altre criptovalute come Bitcoin o con fiat currency. La gestione delle ricompense è effettuata mediante la blockchain Steem, su cui vengono registrati sia le ricompense che tutti i contenuti pubblicati dagli utenti. WINSOME, a differenza di STEEMIT, utilizza un'architettura client server in cui tutti i contenuti e le ricompense sono gestiti da un unico server piuttosto che da una blockchain, inoltre il meccanismo del calcolo delle ricompense è notevolmente semplificato rispetto a quello di STEEMIT. Ad esempio, in WINSOME non viene considerata la possibilità di distribuire parte della ricompensa in VEST, ovvero in quote di possesso della piattaforma.

Descrizione dell'architettura del sistema

L'implementazione di WINSOME ha richiesto la realizzazione di due entità principali: il client e il server. Queste devono comunicare per fornire i vari servizi, e tali comunicazioni avvengono tramite mezzi distinti: TCP, RMI e Multicast.

Lato client

Lato client l'utente non dovrà fare altro che scrivere i vari comandi che desidera eseguire e attendere la risposta del server. L'ordine di tali comandi ovviamente non è casuale, ma dei controlli sia lato client che lato server impediranno esecuzioni scorrette come la pubblicazione di un post prima di aver effettuato il login.
Il **ClientMain** dopo l'avvio provvederà a leggere la configurazione dal file **ClientConfiguration.json** e inizierà a ciclare fino a quando l'utente non inserirà l'apposito comando da tastiera (exit). Nel mentre si preoccuperà di leggere tutte le altre cose che l'utente scrive, impacchettarle nell'apposita struttura **Request**, e poi si impegnerà ad inoltrare tali richieste al **ServerMain** tramite una connessione TCP, la quale verrà aperta e chiusa per ogni ciclo di login e logout.
Dopo aver inoltrato la **Request** rimarrà in attesa della **Response** da parte del server, ed una volta ricevuta ne mostrerà il contenuto il contenuto all'utente.
Il client oltre che l'inoltro di richieste e la ricezione di risposte dovrà occuparsi di due cose: la gestione di una lista locale dei followers dell'utente correntemente loggato e della ricezione di messaggi multicast che il server invia periodicamente per notificare che sono stati pagati gli utenti per i propri contenuti.

Gestione della lista locale dei followers

La gestione della lista locale dei followers avviene sfruttando l'RMI, infatti il cliente terrà tale lista in un oggetto **NotifyEventImplementation** che sarà direttamente modificabile dal server qualora i followers cambino. Questo permette di non intasare la connessione TCP e consentire un aggiornamento della lista dei followers in "tempo reale" senza doverne chiedere l'aggiornamento esplicitamente.

Ricezione dei messaggi multicast

Ogni volta che un login va a buon fine il client provvederà a registrarsi nel gruppo multicast dove il server invia i messaggi di notifica per poterli ricevere. L ricezioni di tali messaggi tuttavia avviene in modalità bloccante, e per questo non è fatta direttamente dal **ClientMain** ma da un thread apposito che esegue la classe runnable **MulticastReceiver** che ha il compito unico di ricevere tali messaggi e stamparne il contenuto a schermo.
Le coordinate multicast alle quali connetterti sono ottenute tramite chiamate apposite all'oggetto remoto esposto dal client: **RmiServerImplementation**.

Lato server

Il compito fondamentale del server sarà quello di ricevere le **Request**, eseguirne le richieste e rispondere con delle apposite **Response**.
Le richieste, seppur non particolarmente onerose, non possono essere gestite da un unico thread, ciò infatti porterebbe ad un deterioramento delle prestazioni all'aumento del numero di client, quindi viene utilizzato un thread pool.
Inoltre per evitare lo schema richiesta risposta basato su chiamate bloccanti ho deciso di gestire le richieste ed i vari canali di comunicazioni sfruttando un selector e le letture e scritture non bloccanti.
Prese tali decisioni il **ServerMain** dopo essersi avviato e aver letto il proprio file di configurazione **ServerConfiguration.json** si trova dunque con il compito di dover accettare le connessioni, leggere le **Request** (scritte come stringhe json) sui vari canali, fare partire un apposito **RequestHandler** che se ne prenda cura all'interno del thread pool e poi inviare al client la risposta ottenuta dall'elaborazione della richiesta.
Questa continuità tra lettura e scrittura viene garantita usando l'oggetto **Attachment** delle chiavi, usate dal selector.

Request handler e classe Winsome

La vera esecuzione delle richieste avviene all'interno dei singoli thread, i quali modificano le risorse che caratterizzano il server: ovvero gli utenti, i post e gli wallets.
Tali risorse sono contenute all'interno di un oggetto apposito, ovvero **Winsome**.
Ovviamente sarebbero potute essere direttamente gestite all'interno del **ServerMain** stesso, tuttavia il separarle in un altro oggetto ha permesso di diminuire le interazioni (logiche quantomeno, poiché winsome è comunque istanziato all'interno del ServerMain) tra il **ServerMain** e il **RequestHandler**, permettendo una più facile gestione della concorrenza.

Notifiche dei followers, aggiornamenti degli wallets e backup

Il server gestisce altri thread oltre a quelli nel thread pool. Nello specifico un thread nato dall'esecuzione di **ServerBackup** si occuperà ciclicamente di salvare tutte le risorse del server in appositi file che permettano di recuperarne lo stato una volta chiuso e riavviato.
Approccio simile è stato scelto per i pagamenti, i quali verranno gestiti dal **MulticastSender**, che ciclicamente farà in modo che gli utenti ricevano le proprie ricompense e che a questi sia notificato mediante comunicazione multicast.
Il client farà chiamate al proprio oggetto esportato (**RmiServerImplementation**) per fare in modo che questo chiamo i metodi remoti dell'oggetto del client per aggiornarne i followers contenuti.

Schema dei threads attivati e concorrenza

Lo schema dei lato client è il seguente:

1. Avvio di **ClientMain**
    a. Avvio del thread **MulticastReceiver**

Lo schema dei lato server threads è il seguente:

1. Avvio di **ServerMain**
    a. Avvio del thread **ServerBackup**
    b. Avvio del thread **MulticastSender**
2. Avvio di un thread **RequestHandler** appena ricevuta una richiesta

Gestione della concorrenza

Tutte le risorse condivise sono all'interno di **Winsome**.
Esse riceverebbero accessi concorrenti dai vari **RequestHandler** che ne comprometterebbero l'integrità, dunque ho deciso di gestirne gli accessi mediante blocchi synchronized.
Rendere ogni metodo synchronized avrebbe risolto ugualmente il problema, tuttavia dato che alcuni metodi modificano solo alcune delle risorse sarebbe stato inefficiente bloccarle tutte ogni singola richiesta
Dovendo talvolta accedere a più risorse alcuni blocchi ***synchronized*** sono uno dentro l'altro (***nested***). Ciò di base può creare problemi e deadlock, tuttavia non dovendo mai bloccare l'accesso a più di due strutture è stato facile decidere a priori un ordine con cui questi blocchi dovessero essere nestati.
Facendo le synchronized seguendo sempre un ordine preciso infatti possiamo togliere il rischio di creare deadlock.
Ordine delle sincronizzazioni usato nelle funzioni:

- Users -> loggedUsers
- Posts -> Utenti
- Users -> Wallets
- Posts -> Wallets

Ordine complessivo: Posts -> Users -> Wallets -> loggedUsers

Classi utilizzate

- ClientMain.java: Classe main che gestisce l'interazione con l'utente ed invia le richieste al server
- ClientConfiguration.java: Classe per la gestione dei parametri di configurazione del client
- MulticastReceiver.java: Classe runnable che il client userà per ricevere nel notifiche di aggiornamento degli wallets
- NotifyEventImplementation.java: Classe RMI del client
- NotifyEventInterface.java: Interfaccia RMI del client
- Attachment.java: Classe per gestire la comunicazone non bloccante fornita dal server
- ServerMain.java: Classe main che riceve le richieste e risponde con risposte
- ServerConfiguration.java: Classe contente i parametri di configurazione del server
- Request.jav: Classe per contenere in un unico oggetto convertibile in json tutte le informazioni riguardanti una richiesta del client
- Response.java: Classe per contenere le informazione delle risposte di winsome che il ServerMain inoltrerà al ClientMain
- RmiServerImplementation.java: Classe RMI del server
- RmiServerInterface.java: Interfaccia RMI del server
- RequestHandler.java: Classe runnable che contatterà winsome per una sola richiesta
- MulticastSender.java: Classe runnable che il server userà per inviare notiche periodiche al client e per gestire le ricompense
- ServerBackup.java: Classe runnable che il server userà per gestire i backup
- Winsome.java: Classe contente le informazioni sensisbili del social network gestendo la concorrenza
- User.java: Classe per contenere le informaizoni significative di un utente come followers e seguiti
- Wallet.java: Classe per contenere le informaizoni significative di uno wallet come la quantità di wincoin presenti sul conto
- Post.java: Classe per contenere le informaizoni significative di un post come valutazioni e commenti pagati e non

Istruzioni per l'esecuzione del progetto

Tutti i file .java sono presenti nella solita cartella e assieme a loro ci sono i .json delle configurazioni.
Sempre nella stessa directory saranno generati i .json per il backup.
Nella cartella **jars** ci saranno i .jar necessari per poter utilizzare la libreria jackson, mentre nella cartella **Client Server jars** ci saranno i .jar eseguibili di client e server richiesti da consegna.

Esecuzione con javac e java

L'esecuzione prevede di aprire la cartella ed eseguire due comandi per il server e due per il client (i primi creeranno i .class e i secondi avvieranno l'esecuzione).

Per il server

javac -cp .;jars\jackson-annotations-2.9.7.jar;jars\jackson-core-2.9.7.jar;jars\jackson-databind-2.9.7.jar ServerMain.java
java -cp .;jars\jackson-annotations-2.9.7.jar;jars\jackson-core-2.9.7.jar;jars\jackson-databind-2.9.7.jar ServerMain

Per il client

javac -cp .;jars\jackson-annotations-2.9.7.jar;jars\jackson-core-2.9.7.jar;jars\jackson-databind-2.9.7.jar ClientMain.java
java -cp .;jars\jackson-annotations-2.9.7.jar;jars\jackson-core-2.9.7.jar;jars\jackson-databind-2.9.7.jar ClientMain

A questo punto basterà utilizzare i comandi richiesti dalla specifica per le interazioni del client con il server.
L'unica aggiunta è un comando lato client "exit" per fare un logout e terminarne l'esecuzione.

Esecuzione dei .jar eseguibili

Dopo essere entrati con il terminale nella directory **Client Server jars** l'esecuzione dei .jar avviene attraverso il comando java -jar.
Quindi per un'esecuzione di un client e di un server:

java -jar server.jar
java -jar client.jar