Winsome: a reWardINg SOcial MEdia

Università di Pisa - Facoltà di Informatica
Jacopo Raffi 598092

5 Gennaio 2022

Indice

1 Introduzione

Il progetto riguarda la realizzazione di $WINSOME$ un *social media network* dove gli utenti possono seguire altri utenti, pubblicare *post* e valutare i contenuti pubblicati dai propri $followers$.
Inoltre c'è la possibilità di ricevere ricompense in *wincoin*, sulla base delle valutazioni positive recenti, i quali possono essere convertiti in *bitcoin*. Il lavoro consiste nell'implementazione sia del *Server* sia del *Client*.

2 Server

2.1 Descrizione generale

L'implementazione del *server* è $Multi\ Threading$, vengono generati diversi *thread* specializzati in differenti compiti. La gestione dei *client* avviene tramite $cachedThreadPool$, si genera un $Thread\ Worker$ che esegue tutte le richieste di un *client*.
Le connessioni avvengono tramite TCP, UDP e RMI.

2.2 Classi lato Server

2.2.1 ServerWinsomeSocial

La classe $ServerWinsomeSocial$ rappresenta il *social*. I "Thread Worker" invocano i metodi e utilizzano le strutture dati di questa classe.
Questa classe utilizza due $ConcurrentHashMap$, una per gli utenti e l'altra per i *posts*.
La sincronizzazione tra i vari *thread* avviene tramite metodi *synchronized* e $ReentrantLock$ delle classi $ServerPost$

e *ServerUser*. In caso ci sia necessità di una mutua esclusione su più *lock*, i *thread* acquisiscono le *lock* secondo un preciso ordine così da evitare possibili *deadlock*.
La struttura dati utilizzate da questa classe, sia per i *posts* che per gli utenti, è la *ConcurrentHashMap*. Un motivo di questa scelta è quello di ottenere sincronizzazione nelle operazioni di *add*, *remove* e *get*; nel caso della *get*, dopo aver ottenuto l'oggetto, si gestisce la sincronizzazione tramite *ReentranLock*. L'altro motivo che ha portato all'uso di questa struttura dati è l'efficenza delle operazioni che avvengono in tempo O(1).

2.2.2 ServerUser

La classe *ServerUser* rappresenta l'utente all'interno del *Server*. Dentro questa classe vengono utilizzate diverse strutture dati per contenere *followers*, *followed*, *blog* e *feed*. Per i *followers* e i *followed* sono stati utilizzati dei *LinkedHashSet*, è stata scelta questa struttura dati per evitare possibili copie dello stesso *username* all'interno di queste collezioni. Il motivo per cui è stata scelta un *LinkedHashSet* e non un *TreeSet* è dovuta alla maggiore efficenza nelle operazioni di *add*, *remove* e *get*.
Nel caso di *blog* e *feed* viene utilizzata una *HashMap*, scelta per l'efficenza delle operazioni elementari.
All'interno del *ServerUser* sono presenti tre *ReentrantLock*, salvate in un singolo *array* per avere codice più pulito. Ad ogni *Lock* è associata una particolare struttura dati:

1. Indice 0: *Lock* per il *feed*
2. Indice 1: *Lock* per i *followers*
3. Indice 2: *Lock* per il *blog*

Ogni volta che vengono invocati i metodi *lock* e *unlock* bisogna specificare anche l'indice. Per concludere la scelta di una *HashMap* invece di una *ConcurrentHashMap* è motivata dal minore spazio che occupa la prima struttura dati.

2.2.3 ServerPost

Il tipo di dato *ServerPost* rappresenta i *posts* all'interno di *Winsome*. Dentro questa classe venngono utilizzate due strutture dati per contenere i *likes/dislikes* e i commenti relativi ad un *post*.
Per i *likes*, rappresentati dalla classe *FeedBack* viene utilizzata una *LinkedList*, questo perchè interessa solo avere tempo O(1) nell'operazione di *add* e non interessano le operazioni di *get* e *remove*.
Per i commenti viene utilizzata una *HashMap* dove le chiavi sono gli *username* degli utenti che hanno commentato mentre il valore associato è la lista di commenti eseguiti da quella chiave.
Questo perchè si cerca di ridurre i tempi di calcolo delle ricompense relativa ad un *post*.

2.2.4 ServerUtilities

Dentro questa sezione verranno descritte tre classi utilizzate all'interno di *ServerPost* e *ServerUser*.

- ServerWallet: Questa classe viene utilizzata all'interno del *ServerUser* ed al suo interno viene salvato il totale di *wincoin* dell'utente e le varie transazioni.
- ServerFeedback: Questa classe rappresenta i *likes* e i *dislikes* del *social*. L'utilità di questa classe è quello di tenere traccia degli autori, del tempo in cui è stato aggiunto un *feedback* e infine se questo *feedback* rappresenti un *like* o un *dislike*.
- ServerComment: Questo tipo di dato è un'estensione della classe sopra citata, in aggiunta ha soltanto il contenuto del commento(una stringa di massimo 500 caratteri).

2.3 Thread Main

Il *Thread Main*(lato *Server*), oltre a creare la *threadPool* e a generare i vari *thread* specializzati, si occupa della configurazione del *Server* tramite lettura di un *file* di configurazione in formato *txt*. Successivamente viene eseguito il ripristino dello stato del *Server* prima della precedente chiusura.
Il ripristino avviene tramite la deserializzazione *JSON* dei *files usersStatus* e *postStatus*. La deserializzazione avviene tramite la classe *JsonReader* della libreria *gson*.
Il primo *file* contiene le informazioni riguardanti gli utenti registrati al *social*, il secondo invece contiene informazioni riguardanti i *posts* creati dagli utenti.
Il *Thread Main* inoltre si occupa della creazione di un *registry* mediante *RMI*; tramite questo *registry* gli utenti possono registrarsi al *social* e recuperare la loro lista dei *followers*.
Infine questo *thread* si occupa di accettare le connessioni tramite *WelcomeSocket*, una volta accettata la connessione viene creato il *clientSocket* utilizzato da un *Thread Worker* per la comunicazione col *client*.

2.4 Thread Reward

Il *Thread Reward* si occupa del calcolo della ricompensa, in *wincoin*, di ogni *post*.
Scansiona uno ad uno i *posts* del *social*, calcola il numero di *likes* e di commenti recenti, poi tramite un'apposita formula viene calcolato il guadagno totale derivante dal singolo *post*, infine questo guadagno viene suddiviso tra autore e curatori; la percentuale dell'autore, default 80%, può essere stabilita nel $file$ di configurazione.
Una volta calcolata la ricompensa invia una notifica unica a tutti gli utenti registrati, che hanno fatto il *login*, tramite *DatagramSocket*.

2.5 Thread Backup

Il compito del *Thread Backup* è quello di serializzare in $JSON$ lo stato del *social* in appositi $files$. Per la scrittura di stringhe $JSON$ su $file$ viene utilizzata la classe $JsonWriter$, fornita dalla libreria *gson*.
All'interno dei *blog* e $feed$ degli utenti vengono salvati solamente gli *id* dei *posts*.
I *posts* e gli utenti vengono scritti manualmente uno ad uno così da evitare possibili stringhe di grandi dimensioni in memoria principale.

2.6 Thread Worker

Il compito dei *worker* è quello di eseguire le richieste provenienti dai *client*. Ad ogni *worker* è associato un solo *client*, il *thread* termina il suo lavoro quando il *client* si disconnette. Il *thread* rimane in attesa della richiesta dell'utente, tramite $readUTF$, effettua il *parsing* della stringa per ottenere il tipo di richiesta e i possibili parametri.
Dopo il *parsing* viene eseguita la richiesta e successivamente il *worker* invia la risposta al *client* tramite una $writeUTF$. Ci sono alcuni casi in cui, per evitare di inviare una stringa potenzialmente lunga, la risposta viene segmentata in più stringhe inviate una ad una al *client*.

2.7 Thread di terminazione

Il *Thread di terminazione* viene avviato dal *Thread Main* prima di accettare connessioni TCP. Il *thread* rimane in attesa che venga scritta da tastiera la stringa *quit* o la stringa *quitNow*.
Una volta mandato il comando *quit* o *quitNow* vengono chiusi il *welcomeSocket* e il *DatagramSocket*, vengono interrotti tutti i *thread* specializzati, viene eseguito uno spegnimento della *threadPool* e per concludere viene eseguito un *backup* finale.
Con il comando *quit* viene terminata la *threadPool* e si attendono 10 minuti per permettere ai worker di terminare le richieste dei *client* già connessi. IL comando *quitNow* invece non attende la terminazione dei *Thread Worker* ma esegue uno *shutdownNow* della *threadPool* in modo da chiudere velocemente il *Server*.

3 Client

3.1 Client Main

Il processo *Client* comincia con l'esecuzione della classe *ClientThinMain*. All'inizio vengono impostate le politiche di sicurezza del sistema con quanto contenuto nel file *MyGrantAllPolicy.policy* e selezionando come *securitymanager* un nuovo oggetto di tipo *SecurityManager*. Questa classe successivamente carica la classe *ClientClass* tramite la *loadClass* di *RMIClassLoader*.

3.2 ClientClass

Questa classe rappresenta il cuore vero e proprio del *Client*. All'inizio viene fatto un *parsing* del file di configurazione, dopo aver assegnato i valori alle porte e agli indirizzi si crea la connessione col *Server*, tramite TCP.
Una volta connessi il *Server* comunica i parametri necessari per la connessione *MultiCast*, una volta ricevuti i parametri viene creato un *MulticastSocket* che verrà passato come parametro al *ClientUDPThread*.
Dopo la creazione del *thread* è possibile eseguire i comandi da linea di comando. La lista dei comandi è la seguente:

- register: il comando è *register username password tag*1 *tag*2 *tag*3 *tag*4 *tag*5. L'utente deve fornire *username*, *password* e una lista di *tag*(massimo 5 *tag*). Il *server* risponde con un codice che può indicare l'avvenuta registrazione, oppure, se lo username è già presente, o se la *password* è vuota, restituisce un messaggio d'errore. Lo username dell'utente deve essere univoco.

- login: il comando è *login username password*. Se l'utente ha già effettuato la login o la password è errata, viene ricevuto un messaggio di errore.
- logout: il comando è *logout*. Effettua il logout dell'utente dal servizio.
- quit: il comando è *quit*. Termina il processo *Client*.
- follow: il comando è $follow\ iduser$. L'utente chiede di seguire l'utente che ha per *username idUser*.
- unfollow: il comando è $unfollow\ iduse$r. L'utente smette di seguire l'utente che ha per *username idUser*.
- createpost: il comando è *createpost titolo contenuto*. Viene creato un *post* all'interno del *social*. In caso il contenuto o il titolo contengono spazi è necessario metterli tra " ".
- showpost: il comando è *showpost idpost*. Viene mostrato il *post* identificato da *idpost*.
- rewinpost: il comando è *rewinpost idpost*. Operazione per effettuare il *rewin* di un *post*, ovvero per pubblicare nel proprio *blog* un post presente nel proprio $feed$.
- deletepost: il comando è *deletepost idpost*. Viene eliminato dal *social* il *post* identificato da *idpost*, solo se l'utente è l'autore.
- addcomment: il comando è *addcomment idpost contenuto*. Operazione per aggiungere un commento al *post* identificato da *idpost*. Se l'utente ha il *post* nel proprio $feed$, il commento viene accettato.
- ratepost: il comando è *ratepost idpost voto*. Operazione per aggiungere *like* o *dislike* al *post* identificato da *idpost*. Se l'utente ha il *post* nel proprio $feed$, il voto viene accettato.
- listusers: il comando è listusers. Il *serve* restituisce la lista di utenti che hanno almeno un *tag* in comune con l'utente.
- listfollowing: il comando è $listfollowing$. Viene restituita la lista degli *username* di cui l'utente è $follower$.
- listfollowers: il comando è $listfollowers$. Operazione lato *Client*, restituisce la lista dei $followers$. Quando, lato *Server*, viene eseguita una $follow$ o una $unfollow$ il *Server* aggiorna la lista dei $followers$, lato *Client*,con una *callback* alla quale l'utente si registra dopo aver effettuato il *login*.
- viewblog: il comando è *viewblog*. Operazione per recuperare la lista dei post di cui l'utente è autore. Per ogni *post* viene fornito *id* del *post*, autore e titolo.
- showfeed: il comando è $showfeed$. Operazione per recuperare la lista dei post nel proprio $feed$. Per ogni *post* viene fornito *id* del *post*, autore e titolo.
- getwallet: il comando è *getwallet*. Operazione per recuperare il valore del proprio portafoglio e la lista delle transazioni eseguite dal *Server*.
- getwalletinbitcoin: il comando è *getwalletinbitcoin*. Operazione per recuperare il valore del proprio portafoglio convertito in *Bitcoin*. Il *server*, per calcolare la conversione, utilizza il servizio di generazione di valori random decimali fornito da $RANDOM.ORG$ per ottenere un tasso di cambio casuale.

3.3 ClientUDPThread

Questo *thread* viene generato dalla classe *ClientClass* per rimanere in attesa di notifiche da parte del *Server*. Se l'utente ha fatto il *login* viene stampata a schermo la scritta "¡NOTIFICA: RICOMPENSA AGGIORNATA¿". Questa notifica è uguale per tutti i *client* all'interno del *social* a prescindere che abbiano ottenuto ricompense o meno.

4 Manuale d'Uso

- Guida su come compilare:
  1. Andare dentro la cartella src
  2. eseguire il comando javac -cp .;..\lib\gson-2.8.9.jar *.java -d ..\Out

     IMPORTANTE: NON cambiare destinazione cartella perchè potrebbero esserci problemi riguardanti il path relativo dei file di configurazione, di backup e di policy.

- Guida su come eseguire il Client(classe main ClientThinMain):
  1. prima eseguire il server(altrimenti non si connette e il programma termina immediatamente)
  2. andare dentro la cartella Out
  3. eseguire il comando java ClientThinMain - path del file di configurazione(opzionale) -
- Guida su come eseguire il Server(classe main ServerMain):
  1. andare dentro la cartella out
  2. eseguire il comando java -cp .;..\lib\gson-2.8.9.jar ServerMain - path del file di configurazione(opzionale) -
- Guida su come creare il server.jar:
  1. andare dentro la cartella Out
  2. eseguire il comando jar cmf ManifestServer.mf server.jar Server*.class IllegalRegisterException.class ClientNotifyInterface.class
- Guida su come creare il client.jar:
  1. andare dentro la cartella Out
  2. eseguire il comando jar cmf ManifestClient.mf client.jar Client*.class IllegalRegisterException.class ServerRegistryInterface.class

---

IMPORTANTE: Se si decide di configurare il *server*(o il *client*) tramite *file* di configurazione, assicurarsi PRIMA di eseguire i comandi che i parametri corrispondano altrimenti i due processi non riescono a comunicare tra loro.