Relazione Progetto Laboratorio di Reti: Winsome, a reWardING SOcial MEdia

Università di Pisa - Dipartimento di Informatica
Chiara Maggi, 578517

A.A. 2021/22

Indice

1 Descrizione generale architettura

Il progetto realizzato rappresenta un social network con funzionalità base, dove ciascun utente registrato può seguire altri utenti ed essere a sua volta seguito. Questo meccanismo permette di presentare a un utente solo i contenuti pubblicati dagli utenti che segue. L'utente viene ricompensato dal servizio se pubblica contenuti (post) che riscuotono interesse da parte della comunità, e/o se contribuisce attivamente votando o commentando contenuti pubblicati da altri utenti. L'utente ha la possibilità di interagire con il social network tramite l'inserimento di comandi a runtime in un processo client, che sarà poi correttamente servito da uno dei thread del pool del server. Il server sfrutta la connessione TCP o RMI (a seconda della richiesta) per elaborare una risposta da mandare al client e quindi da mostrare all'utente.

Una prima scelta implementativa è quindi quella di realizzare il server con meccanismo Java I/O e Threadpool (in particolare CachedThreadpool, per la sua elasiticità) per ottimizzare la gestione dei vari client: in questo modo è possibile identificare univocamente lo stato di login di un determinato utente e gestire indipendentemente e contemporaneamente le richieste di diversi client.
Un'altra scelta implementativa riguarda la gestione degli accessi di un utente al social network: in particolare un utente che ha fatto il login su un determinato client non potrà collegarsi su un client differente fino a quando non effettuerà il logout dal primo.
Sono stati utilizzati dal client due sistemi di notifica: RMI callback permette di ottenere gli aggiornamenti sui followers dell'utente collegato, e servizio Multicast di ottenere le notifiche del periodico calcolo delle ricompense e conseguente aggiornamento del portafoglio.

2 Schema generale threads

Di seguito è illustrato lo schema generale dei threads attivati dalle 2 componenti del sistema, il Server e il Client.

2.1 ServerMain

L'implementazione del **ServerMain** è multithreading: all'avvio vengono infatti avviati vari thread che svolgono diversi compiti durante tutto il periodo di attività del Server. Dopo aver configurato correttamente i parametri (default o tramite file di configurazione) necessari per le varie connessioni e timeout, vengono ripristinate, tramite deserializzazione JSON dei file *backupPosts* e *backupUsers*, le informazioni (se presenti) dell'intero Social Network. Viene in seguito generato un threadpool contenente thread specializzati nello svolgere richieste provenienti dai client. Inoltre, il ServerMain si occupa della creazione di un Registry per permettere l'uso del servizio RMI da parte dei Client e della configurazione della connessione multicast, utilizzata per le notifiche realtive alle ricompense. Infine, il Server si mette in attesa di richieste di connessione tramite un `ServerSocket` (*listener*): una volta che una connessione viene accettata, viene creato il `Socket`(*clientSocket*) utilizzato dal worker per la comunicazione con il client.

2.1.1 Worker

Il compito del **thread worker** è quello di eseguire le richieste provenienti dal Client al quale è associato, fino a quando quest'ultimo non decide di disconnettersi. Il worker resta in attesa delle richieste dell'utente tramite una `readUTF`, che si sbloccherà all'arrivo di una stringa della quale verrà effettuato il parsing per ottenere l'effettiva richiesta e i relativi parametri. Dopo il parsing per mezzo delle funzioni offerte dalla classe SocialNetowrk (che verrà spiegata più avanti) viene eseguita la richiesta e viene elaborata una risposta che verrà inviata al Client tramite `writeUTF`. In alcuni casi, per evitare una stringa potenzialmente lunga, la risposta viene segmentata in più stringhe inviate singolarmente al Client.

2.1.2 Backup

Il compito del **backupThread** è quello di serializzare in JSON lo stato del social in 2 appositi file, *backupUsers* (per mantenere traccia degli utenti del social network) e *backupPosts* (per mantenere traccia di tutti i post creati). La libreria utilizzata è *Gson*, che fornisce la classe `JsonWriter` per la scrittura di stringhe JSON su file. Il salvataggio avviene periodicamente in base al timeout configurato all'avvio del Server e i metodi utilizzati

per la serializzazione sono `synchronized`, così da non avere inconsistenze tra il file JSON e lo stato del social network.

2.1.3 Reward

Il compito del **rewardThread** è quello di calcolare periodicamente, in base al timeout impostato all'avvio del server, le ricompense da assegnare agli autori di post e ai vari curatori. Quando una nuova ricompensa è stata calcolata, viene inviata una notifica tramite `DatagramSocket` a tutti i client collegati. Per il calcolo della ricompensa viene utilizzata una formula specifica, dipendente da numerosi fattori, applicata ad ogni singolo post del social. La ricompensa risultante viene poi suddivisa tra autore e curatori del post, secondo una percentuale scelta all'avvio del Server, ai quali viene aggiornato il wallet con il nuovo totale e l'aggiunta della specifica transazione nella lista di tutte le transazioni.

2.1.4 ServerCloser

Il compito del **closerThread** è quello di rimanere in attesa che venga digitato il comando '`close`' o '`closeNow`' per terminare il server. In entrambi i casi vengono chiusi il `ServerSocket`, il `DatagramSocket` e viene interrotto il **rewardThread**. Digitando '`close`' viene effettuata una chiusura lenta del threadpool, che dà la possibilità ai worker di terminare di servire le richieste dei client già connessi entro un tempo settato di 5 minuti. Digitando invece '`closeNow`' viene chiuso immediatamente il threadpool con una `shuthdownNow`. Infine viene interrotto il **backupThread** e viene svolto l'ultimo backup del social.

2.2 ClientMain

Il **ClientMain** è il processo che si occupa di gestire le richieste degli utenti collegati. All'avvio viene fatto un parsing di un file per configurare i vari parametri (oppure viene usato il settaggio di default) da utilizzare per le connessioni e timeout, dopodichè vengono effettivamente configurate la connessione TCP e e quella multicast (i parametri vengono passati dal server all'apertura della connessione TCP). Viene quindi creato il thread ClientUDPThread per ricevere le notifiche dell'avvenuto calcolo della ricompensa svolto dal rewardThread del Server. Inoltre viene fatta la configurazione RMI per poter accedere ai servizi offerti dal registry del Server e inizializzato il servizio di notifiche relativo alla classe **NotifyClient** (di cui alcuni metodi sono invocati da remoto tramite RMI dal Server) che offre l'implementazione di metodi utili alla manipolazioni dell'oggetto LinkedList locale che tiene traccia dei followers dell'utente connesso. Dopo la creazione del thread, il client mostra una semplice interfaccia a linea di comando con cui interagire per fare richieste e svolgere azioni. La lista dei comandi possibili è la seguente:

- `register <username> <password> <tags>`: l'utente deve fornire un username, una password e massimo 5 tags per potersi registrare. Il server restituisce errore se l'username è già utilizzato da qualcuno o se la password non è compresa fra gli 8 e i 16 caratteri;
- `login <username> <password>`: permette all'utente di effettuare il login su un certo client. Il server restituisce errore se l'utente è già collegato su un altro client, se la password inserita non è corretta per quell'username o se l'username non esiste;
- `logout`: permette all'utente di effettuare il logout dal social network;

- `list users`: permette di visualizzare tutti gli utenti (username e tags) del social network che hanno almeno un tag in comune con l'utente che digita il comando;
- `list followers`: operazione lato Client che permette di visualizzare tutti i followers dell'utente connesso. Quando registra un nuovo follow o unfollow, il Server aggiorna tramite callback la lista mantenuta localmente dal Client.
- `list following`: restituisce la lista degli utenti (username e tags) seguiti dall'utente connesso al Client;
- `follow <username>`: permette all'utente connesso di iniziare a seguire un utente. Il server restituisce errore se l'utente segue già quell'username, se l'username non esiste o se l'username è il nome dell'utente stesso.
- `unfollow <username>`: permette all'utente connesso di smettere di seguire un utente. Il server restituisce errore se l'utente non segue già quell'username, se l'username non esiste o se l'username è il nome dell'utente stesso.
- `blog`: operazione per recuperare la lista dei post di cui l'utente è autore. Viene restituita una lista dei post presenti nel blog dell'utente. Per ogni post viene fornito id del post, autore e titolo.
- `post <title> <content>`: permette all'utente di creare un post. il Server restituisce errore se il titolo è più lungo di 20 caratteri o se il contenuto è più lungo di 500 caratteri.
- `show feed`: permette all'utente di visualizzare il proprio feed contenente tutti i post degli utenti che segue e quelli ricondivisi dagli stessi;
- `show post <id>`: permette di visualizzare tutte le informazioni (titolo, contenuto, numero di voti positivi, numero di voti negativi e commenti) relative al post con l'id digitato. Restituisce errore se l'id non esiste;
- `delete <idPost>`: operazione per cancellare un post. La richiesta viene accettata ed eseguita solo se l'utente è l'autore del post. Il Server cancella il post con tutto il suo contenuto associato (commenti e voti). Il Server restituisce errore se il post non esiste;
- `rewin <idPost>`: permette all'utente di ricondividere un post di un altro utente. Il Server restituisce errore se il post non esiste, se l'utente è l'autore del post, o se il post non è contenuto nel feed dell'utente;
- `rate <idPost> <vote>`: permette ad un utente di votare un post con un voto negativo (-1) o postivio (1). Il Server restituisce errore se il post non esiste, se l'utente è l'autore del post, se il post non è nel feed dell'autore, se l'utente ha già votato quel post o se il voto è diverso da 1 o -1;
- `comment <idPost> <comment>`: comando per commentare un post. Il commento deve essere di lunghezza inferiore ai 200 caratteri. Il Server restituisce errore se il post con l'id digitato non esiste, se l'utente che vuole commentare è l'autore del post o se il post non è contenuto nel suo feed.
- `wallet`: permette di visualizzare lo stato del portafoglio dell'utente. Vengono visualizzati il totale di winscoin e la lista di tutte le transazioni relative alle ricompense ottenute;

- `wallet btc`: permette di visualizzare il totale del wallet in bitcoin;
- `help`: permette di visualizzare tutti i comandi che è possibile digitare;
- `quit`: utilizzato per terminare il processo Client;

Oltre ai vari controlli lato Server, il Client si occupa di controllore se la notazione del comando digitato è corretta, altrimenti solleva subito un errore senza inoltrare la richiesta al Server.

2.2.1 ClientUDPThread

Il compito del **notifyRewardThread** è quello di rimanere in attesa di notifiche da parte del server e, se l'utente ha fatto il login, di stampare a schermo la scritta di avvenuto aggiornamento del portafoglio. Questa notifica è uguale per tutti, a prescindere dal fatto che l'utente che la riceve abbia ottenuto ricompense o meno.

3 Classi e strutture dati

Di seguito sono riportate le classi utilizzate nell'intero sistema con particolare attenzione alle strutture dati di principale importanza e alle tipologie di sincronizzazioni utilizzate.

3.1 SocialNetwork

La classe **SocialNetwork** è il vero e proprio fulcro di tutto il sistema. Questa classe si occupa di gestire tutte le informazioni degli utenti registrati / connessi e dei post creati. I thread worker per soddisfare le richieste dei client invocano esclusivamente metodi forniti dalla classe SocialNetwork. Le strutture dati principali sono due `ConcurrentHashMap`, utilizzate per gestire l'insieme degli utenti e l'insieme dei Post. Grazie all'utilizzo della concurrent collection è possibile svolgere operazioni di base (add, get e remove) sulla mappa, con un'ottima efficenza e senza trattare esplicimentamente la sincronizzazione. Quando viene prelevato un elemento da una di queste strutture, la gestione della sincronizzazione per lavorare su di esso è rimandata alle classi di seguito, che offrono un sistema di `ReentrantLock` e/o metodi `synchronized`.

3.2 User

La classe **User** è utilizzata per definire ogni utente e tenere traccia di tutte le informazioni relative ad esso. Vengono utilizzate diverse strutture dati per i *followers*, i *followed*, il *blog* e il *feed*. Per quanto riguarda followers e followed la scelta è stata quella di usare delle semplici `LinkedList` di stringhe per tenere traccia di tutti gli username e svolgere operazioni di aggiunta e rimozione in maniera efficente. Invece per quanto riguarda il blog e il feed le strutture dati sono delle `ConcurrentHashMap` che offrono una completa gestione della sincronizzazione per l'aggiunta e rimozioni di post da esse. L'unica lock esplicita è una `ReentrantLock` utilizzata per la gestione dei followers: nel momento in cui una qualsiasi componente del sistema (ad esempio la classe SocialNetowrk) vuole accedere la lista dei followers di un determianto utente, dovrà prima acquisire la followersLock (`followersLock()`), per poi rilasciarla a fine operazione (`followersUnlock()`). In generale la classe User fornisce vari metodi per apportare modifiche allo stato dell'utente: tutti questi metodi sono usati dal social network per completare determinate richieste provenienti dal Client. Possiede 2 costruttori, uno usato quando un utente si registra per la prima volta, l'altro per quando si ripristinano gli utenti del social network dal file JSON.

3.3 Post

La classe **Post** è utilizzata per definire ogni post creato all'interno del social network. Le principali strutture dati utilizzate sono due `LinkedList`: una contenente i voti relativi al post e l'altra i commenti (classe **Vote** e **Comment** più avanti). Per gestire la sincronizzazione di quest'ultime vi sono due `ReentrantLock` (votesLock e commentsLock), che dovranno essere acquisite (`votesLock()`, `commentsLock()`) prima di accedere o apportare una qualsiasi modifica alle risorse, e rilasciate alla fine (`votesUnlock()`, `commentsUnlock()`). Anche questa classe possiede numerosi metodi di utilità utilizzati dal social network per apportare modifiche ai vari post esistenti nel sistema.

3.4 Comment

La classe **Comment** è una semplice classe di utilità che serve per tenere traccia delle informazioni relative a un commento: *autore*, *contenuto* e *data creazione*. Queste informazioni oltre a fare parte del singolo post sono utilizzate anche per il calcolo della ricompensa.

3.5 Vote

La classe **Vote** è una semplice classe di utilità per tenere traccia delle informazioni relative ad un voto ovvero *autore*, *tipologia di voto* e *data della creazione*. Queste informazioni, oltre a fare parte del singolo post, sono utilizzate anche per il calcolo della ricompensa.

3.6 Wallet

La classe **Wallet** è una semplice classe di utilità usata all'interno della classe User. Tiene traccia del totale di wincoin dell'utente e dell'insieme di transazioni che modificano il portafoglio ogni volta che viene applicata la ricompensa a quell'utente. Tutti i metodi della classe sono definiti `synchronized`: in questo modo gestiscono la sincronizzazione dell'arrivo di più ricompense contemporaneamente, che modificano la lista delle transazioni e il totale.

3.7 Hash

La classe **Hash** è utilizzata per fare l'hashing della password di ogni utente ed evitare che nei file JSON sia visibile in chiaro.

4 Manuale d'Uso

Compilazione:

1. Aprire il terminale sulla cartella src
2. Digitare il seguente comando e premere invio:
   `javac -cp .;..\lib\gson-2.8.2.jar *.java`

Eseguire il ServerMain:

1. Aprire il terminale sulla cartella src
2. Digitare uno dei due comandi e premere invio:
   - Avvio con valori di default:
     `java -cp .;..\lib\gson-2.8.2.jar ServerMain.java`

- Avvio con valori prelevati dal file di configurazione:
  `java -cp .;..\lib\gson-2.8.2.jar ServerMain.java ..\config\ConfigServer.txt`

Eseguire il ClientMain:

1. Aprire il terminale sulla cartella src
2. Digitare uno dei due comandi e premere invio (è necessario che sia già stato avviato il server per poter mettere in funzione il client):
   - Avvio con valori di default:
     `java ClientMain.java`
   - Avvio con valori prelevati dal file di configurazione:
     `java ClientMain.java ..\config\ConfigClient.txt`

Eseguire con file Server.jar:

1. Aprire il terminale sulla cartella jar
2. Digitare il seguente comando e premere invio:
   - Avvio con valori di default:
     `java -jar Server.jar`
   - Avvio con valori prelevati dal file di configurazione:
     `java -jar Server.jar ..\config\ConfigServer.txt`

Eseguire con file Client.jar:

1. Aprire il terminale sulla cartella jar
2. Digitare il seguente comando e premere invio:
   - Avvio con valori di default:
     `java -jar Client.jar`
   - Avvio con valori prelevati dal file di configurazione:
     `java -jar Client.jar ..\config\ConfigClient.txt`