Relazione WINSOME

Simone Passèra

1. Introduzione

Questo documento descrive in maniera generale il funzionamento e l'architettura complessiva di WINSOME, una rete sociale orientata ai contenuti e caratterizzata da un insieme minimo ma significativo di funzionalità. Per un approfondimento maggiore si consiglia la visione dei file sorgenti.

2. Architettura

Le due componenti principali del sistema sono:

Client. Gestiste l'interazione con l'utente tramite interfaccia a riga di comando, comunica con il server per eseguire le azioni richieste dall'utente, inoltre partecipa ad un gruppo di multicast da cui riceve la notifica di avvenuto calcolo delle ricompense e quindi l'aggiornamento dei portafogli ed infine dopo la fase di login da parte di un utente, riceve aggiornamenti relativi alla lista dei propri follower in tempo reale tramite il meccanismo di RMI callback.

Server. Gestiste la fase di registrazione di un utente mettendo a disposizione dei client un metodo *register* sullo stesso oggetto remoto con cui registrarsi alla callback,

calcola ad ogni intervallo di tempo regolare configurabile (e alla chiusura del server) le ricompense all'autore e a tutti i curatori per ogni post all'interno della rete sociale ed invia la notifica al gruppo di multicast, effettua il salvataggio di tutte le strutture dati persistenti ad un intervallo di tempo regolare configurabile (e alla chiusura del server) in un file json. Infine per gestire tutte le richieste da parte dei client, il server è strutturato come un CachedThreadPool modificato per avere almeno 10 thread nel pool sempre attivi, quando un utente si connette al server viene gestito per tutta la durata della connessione da un thread nel pool chiamato *UserManager.* La scelta di fare un thread per connessione rispetto ad altri meccanismi tipo il multiplexing dei canali con NIO, è sicuramente una maggior facilità di implementazione di questa scelta e nell'ottica di non avere un numero eccessivo di thread attivi in ogni istante.

Messaggi

Lo scambio di messaggi tra client e server avviene su un unica connessione TCP persistente fino alla chiusura del processo client (o server in caso di chiusura di quest'ultimo). La comunicazione avviene tramite uno stream di caratteri bufferizzato di input e output. Un messaggio è sostanzialmente una linea di testo formata da un numero arbitrario di caratteri terminata da fine linea.

Richiesta. La richiesta di un client è formata da uno o più messaggi a seconda del comando da eseguire. Il primo messaggio è sempre il nome dell'azione che deve essere eseguita dal server.

Comando interfaccia utente
> follow Simone

Richiesta generata dal client verso il server
```
outputStream.println(“follow”);
outputStream.println(“Simone”);
```

Risposta. Il server può avere due tipologie di risposte:

1. Oggetto serializzato con Gson, se più di uno invia prima il numero totale di oggetti che dovrà spedire
2. Codice di risposta in stile http (codice di stato, frase)
   codice di stato - intero a tre cifre che rappresenta il risultato dell’operazione richiesta
   frase - descrizione testuale del codice di stato

Codici di stato WINSOME

2xx - Successo
200 ok
201 richiesta accettata ma non ancora terminata

4xx - Errori Client
400 uno o più argomenti uguali a null
401 argomento vuoto
402 troppi argomenti
403 utente già esistente
404 utente non esiste
405 utente connesso
406 nessun utente connesso
407 password non corretta
408 username troppo lungo
409 utente non può essere seguito (nessun tag in comune)
410 utente già seguito
411 utente non seguito
412 lista dei tag contiene duplicati
413 titolo del post troppo lungo
414 contenuto del post troppo lungo
415 errore di conversione
416 post non esiste
417 post non è presente nel feed/blog
418 post già presente nel blog
419 utente non può essere seguito (se stessi)
420 valore del voto non accettato
421 post già votato
422 commento troppo lungo
423 utente non è autore del post

5xx - Errori Server
500 utente già registrato alla callback
501 conversione wincoin/bitcoin non riuscita
502 server in chiusura

Threads attivati

Client.

WalletUpdate - Thread che attende la notifica relativa al calcolo delle ricompense tramite il gruppo multicast, informa quindi il client impostando a true una variabile atomica e si rimette in ascolto per nuove notifiche.

Server.

RewardManager - Thread che calcola periodicamente la ricompensa globale per ogni post e accredita una percentuale configurabile di wincoin all'autore e a tutti i curatori equamente, successivamente invia la notifica al gruppo di multicast.

DataManager - Thread che effettua periodicamente il salvataggio di tutte le strutture dati persistenti in un file json. La serializzazione di ogni struttura dati avviene con Gson.

UserManager - Thread eseguito all'interno del thread pool, gestisce la connessione persistente con un client. Quando il client si connette, comunica indirizzo ip e porta di multicast in modo tale che il client può mettersi in ascolto sul gruppo di multicast. Questa scelta è pensata per rendere trasparente al client il meccanismo di notifica ed un possibile cambiamento dei riferimenti al gruppo non necessita che il client ne sia a conoscenza.

exitThread - Thread definito all'interno del server, effettua il parsing dello standard input in attesa della stringa *exit,* successivamente chiude il listen socket in modo da far iniziare la chiusura del server.

Strutture dati e concorrenza

Ho scelto di utilizzare strutture dati thread-safe offerte da java, insieme al meccanismo linguistico monitor, ovvero metodi e blocchi dichiarati synchronized. Sono consapevole del fatto che è possibile una gestione della concorrenza più specifica in modo tale ad esempio da permettere a due thread che fanno uso di sola lettura di non bloccarsi. Questo può essere fatto utilizzando lock esplicite ad esempio ReadWriteLock. Per quanto riguarda il server è stato sviluppato aggiungendo strutture dati thread-safe dichiarate globali quando ne avevo necessità, questo ha portato una maggior complessità di gestione della concorrenza su quelle operazioni che fanno uso di più strutture e devono essere atomiche rispetto ad altre. Ho scelto di usare il monitor perché è un meccanismo che permette di semplificare la sincronizzazione evitando di acquisire e rilasciare i lock ogni volta, però soffre di aspetti negativi citati prima. Per il tempo che avevo a disposizione ho scelto di mantenere questa implementazione per evitare di introdurre nuovi errori e inficiare sul corretto funzionamento del sistema.

Lista delle principali strutture dati:

Client.
Vector<String> **listFollowers**
Lista dei follower appartenenti all'utente attualmente connesso. Viene aggiornata dal server in tempo reale attraverso RMI callback.

Server.

ConcurrentHashMap<String, String> users
Mantiene la corrispondenza tra un utente e la sua password.

ConcurrentHashMap<String, ArrayList<String>> tags
Mantiene la corrispondenza tra un utente e la lista dei suoi tag, aggiunti al momento della registrazione, compresi tra 1 e 5.

ConcurrentHashMap<String, ArrayList<String>> followers
Mantiene la corrispondenza tra un utente e la lista dei propri followers. La modifica della lista viene notificato client se connesso.

ConcurrentHashMap<String, ArrayList<String>> followings
Mantiene la corrispondenza tra un utente e la lista degli utenti che segue all'interno della rete sociale.

ConcurrentHashMap<Integer, Post> posts
Mappa che contiene tutti i post all'interno della rete sociale, sono indicizzati dal proprio id univoco per ogni post.

ConcurrentHashMap<String, Vector<Post>> blogs
Mantiene la corrispondenza tra un utente e la lista dei post appartenenti al proprio blog. L'autore dei post possono essere anche diversi dall'utente se quest'ultimo ha fatto il rewin di qualche post.

ConcurrentHashMap<String, Wallet> wallets
Mantiene la corrispondenza tra un utente e il proprio portafoglio, contenente il valore totale dei wincoin posseduti e le transazioni effettuate.

3. Classi e Interfacce

Client.

ClientMain
Classe principale contenente il metodo main.

Hash
Classe contenente il metodo statico *encrypt*, utilizzato per cifrare la password dell'utente.

WalletUpdate
Vedi sopra.

NotifyFollowers
Oggetto remoto inviato dal client al server (RMI callback), utilizzato dal server per aggiornare la lista dei follower del client.

Server.

ServerMain
Classe principale contenente il metodo main.

RewardManager
Vedi sopra.

DataManager
Vedi sopra.

UserManager
Vedi sopra.

WinsomeRMI
Oggetto remoto esportato dal server.

ListInteractions
Classe contenente le interazioni (informazioni utili per il calcolo delle ricompense) per ogni post all'interno della rete sociale.

Interaction
Classe che contiene le informazioni relative ad un post necessarie per il calcolo delle ricompense, come commenti, voti ed età del post. Le informazioni sono valide dall'ultimo calcolo delle ricompense.

Classi utilizzate da client e server.
WinsomeRMIServices
Interfaccia dell'oggetto remoto esportato dal server, fornisce il servizio di registrazione di un utente e la gestione della callback.

CodeReturn
Classe contenente codice e messaggio di risposta, restituito dall'oggetto remoto WinsomeRMI.

NotifyFollowersInterface
Interfaccia dell'oggetto remoto inviato dal client al server (RMI callback), utilizzato dal server per aggiornare la lista dei follower del client.

Wallet
Classe che modella il portafoglio di un utente, contiene il valore in wincoin e le varie transazioni.

Post
Classe che modella un post all'interno della rete sociale. Contiene informazioni come autore, titolo, contenuto ecc. Alcuni campi sono annotati con @Expose, meccanismo

fornito da gson per escludere la serializzazione di alcune variabili, viene utilizzato per eliminare informazioni non necessarie al client quando ne richiede la visualizzazione.

4. File di configurazione

Ci sono due file di configurazione, uno per il client (client.conf) e uno per il server (server.conf) all'interno della directory *conf*. I file sono strutturati nel seguente modo:

1) Righe vuote o che iniziano con # verranno ignorate
2) <stringa di configurazione> = <valore>
   Stringa e valore non devono avere spazi all'interno

5. Compilazione

Il progetto può essere compilato con l'ultima versione di java disponibile (17.0.2). All'interno della classe *WalletUpdate* ho utilizzato le funzioni viste a lezione joinGroup() e leaveGroup() anche se deprecate dalla versione 14. Unica libreria esterna usata è Gson contenuta nella directory *lib*.

Spostarsi nella directory del progetto ed eseguire :

Client.

```
javac -cp ./WinsomeShared:./WinsomeClient:./lib/gson-2.8.9.jar WinsomeShared/*.java WinsomeClient/*.java -d WinsomeExec
```

Server.

```
javac -cp ./WinsomeShared:./WinsomeServer:./lib/gson-2.8.9.jar WinsomeShared/*.java WinsomeServer/*.java -d WinsomeExec
```

6. Esecuzione

Spostarsi nella directory del progetto ed eseguire :

Client.

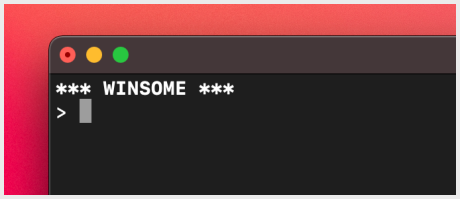
Comando per eseguire il client dopo aver già compilato

```
java -cp ./WinsomeExec:./lib/gson-2.8.9.jar ClientMain ./conf/client.conf
```

Esecuzione automatica del client

```
./execClient                              (script bash - compila ed esegue)
Oppure
java -jar ./jar/client.jar ./conf/client.conf    (file jar)
```

Interfaccia



Digitare il comando help per la lista completa dei comandi e la loro sintassi

Server.

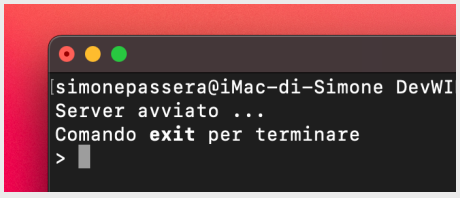
Comando per eseguire il server dopo aver già compilato

```
java -cp ./WinsomeExec:./lib/gson-2.8.9.jar ServerMain ./conf/server.conf
```

Esecuzione automatica del server

```
./execServer                              (script bash - compila ed esegue)
Oppure
java -jar ./jar/server.jar ./conf/server.conf    (file jar)
```

Interfaccia



Digitare il comando exit per chiudere correttamente il server