Relazione Progetto WINSOME

Francesco Benocci 602495

Appello 19 luglio 2022

Contents

1 Introduzione

L'intero progetto è caricato su github al link https://github.com/Benocci/RetiWINSOME. Sono state utilizzate le seguenti librerie esterne entrambi presenti nella cartella src/lib:

- `jackson`: per la trasformazione dei file json di config in classi java;
- `gson`: per la conversione dei file json di backup in strutture dati e viceversa.

Il server è stato realizzato con *multiplexing dei canali mediante NIO e gestione delle richieste attraverso una threadpool.*

2 Esecuzione con file JAR

I file jar sono presenti all'interno della cartella jar. Rimanendo all'interno della cartella RetiWinsome è possibile eseguire il server lanciando il comando `java -jar java/server.jar src/configFile/configServer.json`, mentre per eseguire il client il comando è `java -jar jar/client.jar src/configFile/configClient.json`. In caso si voglia utilizzare dei file di config diversi personalizzati è possibile lanciare client e/o server senza specificare il file; in questo caso il programma provvederà a richiedere il path di un file json che comunque deve rispettare i campi presenti con il relativo nome in maniera obbligatoria. Nel caso di un file di config per il server i campi sono:

- server_address: indirizzo del server;
- server_port: porta del server;
- rmi_registration_name: nome pubblico dell'oggetto che contiene i metodi per la registrazione;
- rmi_registration_port: porta su cui viene creato il registry su cui viene registrato l'oggetto contenente i metodi per la registrazione;
- multicast_address: indirizzo per la connessione UDP multicast;
- multicast_port: porta per la connessione UDP multicast;
- rmi_callback_name: nome pubblico dell'oggetto che contiene i metodi per la callback;
- rmi_callback_port: porta su cui viene creato il registry su cui viene registrato l'oggetto contenente i metodi per la callback;
- rewards_timeout: tempo trascorso tra ogni calcolo dei rewards;
- author_percentual: percentuale di profitto che va all'autore di un post valutato.

Nel caso di un file di config per il client i campi sono:

- server_address: indirizzo del server;
- server_port: porta del server;
- server_registryRMI_name: nome su cui fare la lookup sul registry per il riferimento all'oggetto RMI per la registrazione;
- server_registryRMI_port: porta su cui è presente il registry RMI per la registrazione;
- multicast_address: indirizzo per la connessione UDP multicast;
- multicast_port: porta per la connessione UDP multicast;
- rmi_callback_name: nome su cui fare la lookup sul registry per il riferimento all'oggetto RMI per la callback;
- rmi_callback_port: porta su cui è presente il registry RMI per la callback;

3 Struttura del progetto

Il contenuto dei codici sorgenti del progetto è diviso in sei cartelle separate:

3.1 server

Contiene le classi che implementano il server nella sua interezza, tra le principali sono presenti le entità base del **socialnetwork** (User, Post, Vote, Comment, Wallet e Transaction). SocialNetwork che è la classe che utilizza queste entità per implementare le strutture dati del social, *ServerMainWINSOME* che come da nome è la classe main del server che gestisce le richieste demandandole alla classe *ServerRequestHandler*.

3.2 client

Contiene la classe main *ClientMainWINSOME* che contiene il flusso principale del client con il quale avviene la comunicazione con il server. La classe *RewardsNotification* implementa runnable e viene utilizzata dalla classe main con un thread per la gestione della ricezione dei reward e, infine, l'interfaccia *NotifyEventInterface* che è implementata da *NotifyEvent* e si occupano della ricezione delle notifiche attraverso la callback del cambiamento dei follower.

3.3 exception

Contiene tutte le eccezioni personalizzate che vengono utilizzate a livello di classe SocialNetwork facendo propagare l'eccezione fino a livello di ServerRequestHandler per poter settare opportunamente il campo di risposta da inviare al client:

- UserNotExistException;
- PostNotExistException;
- SameUserException;
- VoteNotValidException;
- AlreadyFollowerException;
- NoAuthorizationException.

3.4 configFile

Contiene i file json che vengono trasformati utilizzando la libreria esterna jackson in due ConfigServerWINSOME e ConfigClientWINSOME che contengono tutti i campi utili per l'esecuzione corretta di server e client.

3.5 manifest

Contiene i META_IF di server e client per la creazione dei file jar.

3.6 backupServerState

Contiene quattro file json che mantengono le informazioni di backup del server: usersBackup, followedBackup, followerBackup e postBackup.

4 Implementazione

4.1 Lato server

4.1.1 SocialNetwork

La classe main ServerMainWINSOME al suo lancio inizializza l'oggetto Social-Network che rappresenta dal punto di vista delle strutture dati il social, nello specifico sono tutte mappe del tipo *ConcurrentHashMap*:

- `users`: associa ad ogni nome utente l'entità utente;
- `followersMap`: associa ad ogni nome utente una lista (ConcurrentLinkedQueue) contenente i nomi utente delle persone che lo seguono;
- `followedMap`: associa ad ogni nome utente una lista (ConcurrentLinkedQueue) contenente i nomi utente delle persone che lui segue;
- `postMap`: associa ad ogni intero (id_post) il post con quell'id, l'id viene mantenuto attraverso un campo AtomicInteger incrementato ad ogni nuovo post per garantire l'unicità.

4.1.2 ServerMainWINSOME

Nella classe main avviene il ciclo su cui viene fatto il multiplexing dei canali NIO in modo non bloccante con il selector; distinguiamo quindi i casi di:

- **Accettazione di un nuovo canale**: viene quindi fatta l'`accept` sul canale e viene fatta la `register` con **OP_READ** e attachment null preparando quindi il canale in lettura;
- **Lettura su un canale**: nel caso di attachment null viene creato il buffer di lettura allocando la lunghezza del messaggio in arrivo che è sempre posta in testa ai messaggi. A questo punto può essere fatta la read che, essendo il tutto **non bloccante**, potrà fare **letture parziali**. Per ovviare a questo, il tutto è predisposto per continuare a fare le read finché non è stato letto l'intero messaggio di cui, come detto prima, sappiamo la lunghezza. Una volta fatto ciò la richiesta viene gestita da una threadpool che esegue il *Runnable ServerRequestHandler* di cui si discute in seguito;
- **Scrittura su un canale:** la scrittura viene fatta inviando l'attachment su cui in precedenza era stata posta la risposta. Anche in questo caso l'operazione viene fatta finché tutto il contenuto di response non è stato inviato e solo dopo verrà controllato il contenuto che, nel caso di una 'exit', chiuderà la connessione su quel canale. In caso contrario su di esso verrà fatta una register con **OP_READ** con attachment null preparandolo quindi in lettura.

4.1.3 ServerRequestHandler

E' la classe che si occupa della gestione delle richieste. Il messaggio viene analizzato e viene fatto uno switch sul primo campo che identifica il comando in modo da distinguere il comportamento del server di conseguenza. In generale il comportamento è quello di eseguire, se possibile, l'operazione e di settare un campo res che viene poi inviata al client come messaggio di risposta. Esso viene inserito come attachemnt nella register del canale con **OP_WRITE**. Si presenta in breve la spiegazione di come vengono gestite le richieste:

- login/logout: dopo aver controllato la password lo username viene inserito come valore in una struttura ConcurrentHashMap associato alla chiave channel che corrisponde all'indirizzo del client, mentre nel logout viene semplicemente rimosso;
- exit: l'invio del messaggio serve per fare il logout di un utente nel caso in cui non sia stato fatto in modo esplicito e, inoltre serve per comunicare al server la terminazione della comunicazione;
- show post/feed, list follwing/users, blog: tutte queste richieste non modificano lo stato del social, ma vanno a leggerne il contenuto inserendolo opportunamente nella stringa res, già formattato per la stampa sul client. L'invio della stringa res avviene come detto in precedenza;

- follow/unfollow: aggiunge/rimuove in maniera opportuna alle mappe tramite il metodo del social e attiva la notifica al client precedentemente registrato con la callback;
- post: viene fatto il parsing di titolo e contenuto tra doppi apici e viene creato un nuovo post con id univoco;
- delete: provoca la rimozione del post dal sistema. Si è deciso per questioni di semplicità di non recuperare l'id dei post eliminati che risulteranno, dopo un eliminazione, non riutilizzabili e quindi "vuoti", tranne nel caso in cui il post eliminato non sia l'ultimo post creato;
- rewin: provoca la creazione di un post con stesso titolo e contenuto di quello indicato con la differenza che sarà indicato il post di partenza e il suo autore. L'eliminazione del post di partenza inoltre provoca l'eliminazione di tutti i suoi rewin;
- rate: aggiunge alla lista dei voti del post indicato un oggetto voto con data, valore (+1 o -1) e autore del voto;
- comment: aggiunge alla lista dei commenti del post indicato un oggetto commento con data, contenuto e autore;
- wallet/wallet btc: restituisce il valore del wallet del user che lo richiede (eventualmente in Bitcoin);

In tutte le richieste che non richiedono la restituzione di informazioni se sono andate a buon fine il messaggio di risposta è res = "ok".

4.1.4 RegistrationRMI

Classe che implementa l'apposita interfaccia RegistrationRMIInterface che estende Remote e che quindi permette ai suoi metodi, in questo caso al suo metodo register, di essere chiamato da un client tramite uno stub in remoto. Essa implementa la registrazione di un nuovo utente con inserimento della password e della lista dei tag creando una nuova entità User e inserendola nella mappa users.

4.1.5 BackupManager

Il sistema di backup è gestito interamente dalla classe Runnable BackupManager che fa uso della libreria esterna gson. Una volta avviato il thread, questa classe copia, ogni secondo, le quattro strutture portanti già presentate all'interno di SocialNetwork nei file json all'interno della cartella backupServerState. Questo come detto viene fatto con gson che trasforma la struttura passata in stringa e attraverso un WritableByteChannel viene scritta nel file indicato. Oltre al metodo run la classe offre anche un metodo loadBackup che viene chiamato allo start del server e permette di caricare le strutture dati trasformate dai file json indicati.

4.1.6 Callback

Come da specifica è richiesto che, ad ogni evento di follow o unfollow, venga notificato all'utente seguito (se esso è loggato) attraverso un sistema di callback. Segue lo schema di callback con RMI visto a lezione: lato server è presente l'interfaccia ServerCallbackInterface con relativa classe che la implementa, i metodi remoti permettono la registrazione al sistema di notifica identificato da una ConcurrentHashMap che associa all'utente registrato una NotifyEventInterface rappresentante uno stub dalla quale è possibile fare chiamate remote al metodo `notifyEvent` che aggiunge(o rimuove) alla lista locale del client il nuovo follower e notifica al client connesso l'operazione.

4.1.7 RewardsCalculation

Viene lanciata con un thread all'inizio dell'esecuzione del server; si tratta di un loop che viene ripetuto secondo un tempo indicato nel file di config (`rewards_period`) e che, ad ogni iterazione, prende tutti i post e controlla se hanno ricevuto voti o commenti dopo la data di ultimo controllo e, in questo caso, calcola il reward aggiungendo le transazioni ai wallet. Infine estrapola il calcolo del reward totale che viene inviato in multicast ai client in quel momento attivi.

4.2 Lato client

La classe principale è ClientMainWINSOME nella quale è contenuto il ciclo principale di lettura della CLI. Esso si traduce in una lettura da tastiera delle richieste che, dopo opportuni controlli, vengono inviate al server attraverso un **socketChannel** che è stato collegato al server all'avvio del client. Il quale leggerà in seguito la risposta del server da stampare a schermo. Le uniche richieste che esulano da questo comportamento sono la help, la register e la list followers: la prima si traduce in una semplice stampa di un messaggio fisso come da specifica, la seconda utilizza l'interfaccia *RegistrationRMIInterface* e la classe *RegistrationRMI* del livello server per una chiamata remota del metodo "register". Infine list followers utilizza l'ArrayList memorizzato in locale per stampare la lista dei seguaci senza dover fare richiesta al server; questo è possibile attraverso l'utilizzo di un sistema di callback identificato nel interfaccia *ServerCallbackInterface* e nella classe *ServerCallback* lato server e nel'interfaccia *NotifyEventInterface* e nella classe *NotifyEvent* lato client secondo lo schema RMI visto a lezione.

Lato client inoltre è presente un ulteriore thread su cui è in esecuzione la classe *Runnable RewardsNotification* che si occupa di ricevere i datagrammi UDP mandati in multicast dal server contenenti il calcolo dei rewards complessivi.

5 Schema generale dei thread

Lista dei thread attivati con le loro attività principali:

1. ServerMainWINSOME: thread principale del server.
2. backupThread: chiamato dal thread 1, esegue BackupManager.
3. rewardsThread: chiamato dal thread 1, esegue RewardsCalculation.
4. threadPool: chiamata dal thread 1 di tipo newCachedThreadPool, ad ogni richiesta esegue un ServerRequestHandler
5. ClientMainWINSOME: thread principale del client.
6. rewardsNotificationThread: chiamato dal thread 5, esegue RewardsNotification

6 Concorrenza

I problemi di concorrenza si presentano lato server poiché l'oggetto socialNetwork, e le sue strutture dati interne, vengono lette e modificate da vari thread della threadpool in contemporanea. Per ovviare a questo, ogni struttura dati all'interno del socialNetwork è di tipo **Concurrent**, come visto in precedenza, che permette di mantenerle sempre **consistenti**. Per la precisione l'utilizzo di queste strutture porta ad avere *Eventual Consistency* poiché i dati ottenuti dalle strutture sono sicuramente consistenti nel momento in cui vengono restituiti, ma essi potrebbero essere non perfettamente in linea con le ultime modifiche nel momento in cui il client riceve la risposta.