Relazione progetto di Reti di calcolatori e laboratorio

Cioni Federico

580718

Gennaio 2022

Contenuti

1. Introduzione

Nel progetto è stato impiegato l'utilizzo di multiplexing dei canali tramite NIO per la ricezione dei pacchetti TCP, inoltre è stato affiancato ad esso l'utilizzo del multithreading per rendere il server ancora più scalabile. Le modalità con cui il multithreading è stato implementato saranno spiegate in seguito nella sezione dedicata al server.
La scelta di affiancare NIO e multithreading è stata intrapresa dopo aver valutato che questo porti ad un ottimo rapporto pacchetti ricevuti/pacchetti inviati, infatti nella mia implementazione la spedizione dei pacchetti di risposta è delegata ai thread stessi, che quindi, tramite operazioni non bloccanti, si occuperanno di inviare immediatamente al client la risposta una volta disponibile.
L'unica libreria esterna utilizzata è stata quella di gson, utilizzata per la memorizzazione su disco delle strutture dati per mantenere la persistenza del server anche allo spegnimento e inoltre è stato usato anche per la gestione dell'invio delle richieste e delle risposte e quindi dei pacchetti.

2. Server

Come già introdotto precedentemente, il server utilizza il multiplexing dei canali associato al multithreading. La ricezione di un pacchetto, l'elaborazione della richiesta e l'invio della risposta avvengono nel seguente modo:
il thread main si occupa di gestire le richieste in arrivo in modalità non bloccante, in maniera da ottimizzare la ricezione di queste ultime. Una volta ricevuto un pacchetto questo viene immediatamente passato ad un thread facente parte del *pool* (ThreadPoolExecutor), si occuperà poi questo thread mediante la classe Runnable RequestHandler, a spacchettare e interpretare la richiesta del client e, a seconda del comando ricevuto, verrà elaborata la risposta e sarà poi il thread stesso a spedire la risposta al client in maniera non bloccante, quindi anche il thread potrà usufruire della NIO per evitare attese prolungate nel caso di client lenti.
È stata scelta questa implementazione per rendere il più rapida possibile la ricezione della risposta da parte del client, per ottimizzare ancora di più le prestazioni e la scalabilità del server, dato che un client riceverà la risposta non appena questa sarà disponibile, invece di attendere ancora tempo per la ricezione della risposta se questa venisse delegata al thread principale. Per essere sicuri che questo meccanismo fosse veramente efficiente e non

bloccante sono state effettuate diverse prove con un pool di un solo thread e due tipi di client: uno con una sleep prima della read della risposta e uno senza; il primo client ha fatto una richiesta prima del secondo e comunque il secondo ha ricevuto prima la risposta. Pertanto è stato assicurato che le risorse vengono comunque utilizzate in maniera ottimale, non ricadendo quindi nella struttura di un thread per connessione.

2.1 Implementazione della gestione delle richieste TCP

Spiegato il meccanismo generale, l'implementazione del thread che gestisce la ricezione delle richieste è molto semplice: innanzi tutto viene aperta una socket, che sarà la socket principale del server, questa viene aperta in un canale in maniera da poter utilizzare la NIO; successivamente viene registrato un selettore per la ricezione delle nuove connessioni e si entra in un loop infinito, dove si accettano richieste di connessione e pacchetti da parte dei client.
Una volta ricevuta la richiesta di connessione, viene creato un SocketChannel, importante è la classe *Attachment* che serve per identificare il client connesso in quel momento. Questa classe presenta tre parametri: il primo è lo stato di login (*loginStatus*), se un client effettua il login questa variabile verrà settata a *true*, altrimenti rimarrà *false*; il secondo parametro è un buffer (*bb*) dove verranno ricevute le richieste; l'ultimo importante parametro è l'id con cui il client si è loggato (*loggedAs*), qualora lo avesse fatto. Questo attachment servirà al thread che elaborerà la richiesta per capire se il client ha i diritti di fare determinate azioni sul server.
Una volta che la connessione è stabilita, si possono ricevere i pacchetti dal client, questi una volta arrivati vengono immediatamente passati ad un thread che si occuperà di elaborare la risposta nel seguente modo:
una volta spacchettata la richiesta, che per convenzione sarà una lista di stringhe, in cui ogni stringa è un parametro del comando, viene, tramite uno switch, interpretato il comando e lanciata una funzione che svolgerà la richiesta. Ci sono dei casi in cui la richiesta non può essere esaudita già da subito, questi casi sono: il caso in cui un utente inserisca un comando non riconosciuto e il caso in cui l'utente non sia loggato; infatti, nonostante ci sia un controllo lato client per queste cose, la sicurezza è garantita ulteriormente anche lato server.
Per la risposta al client viene utilizzato un metodo uguale per tutti, *SendResponse* , questo metodo prende come parametro una stringa e un ID di risposta: la stringa sarà il messaggio di risposta che verrà spedito al client; l'id indica con un 1 se la richiesta è andata a buon fine e con uno 0 se la richiesta è fallita o è stata rifiutata. Questi due parametri fanno parte della classe

Response, che è la classe usata per le risposte, che quindi sostanzialmente è una coppia.
Secondo la mia implementazione quindi non vengono mai passate strutture dati o variabili come risposta, questo è stato fatto per avere una gestione più semplice e uguale per ogni richiesta.

2.2 Rmi e multicast

Come da richiesta del progetto la fase di registrazione e la notifica follower è gestita mediante rmi, vengono creati quindi i servizi *Register* e *Notifica_followers* ai quali è possibile accedere lato client per registrarsi e ricevere la notifica dei followers una volta fatto il login.
La classe che implementa la notifica followers contiene una struttura dati interna che viene aggiornata a mano a mano che vengono notificati i nuovi followers o chi smette di esserlo, questo si rifletterà in un aggiornamento lato client della struttura. Purtroppo, però, alla chiusura del client questa struttura viene cancellata, per questo ho ritenuto opportuno creare un comando aggiuntivo “followers” che permette di recuperare la lista dei followers dal server qualora si ritenesse opportuno farlo.
Il servizio di multicast è implementato tramite un thread apposito (t_reward) che è anche il thread che si occupa di calcolare le ricompense, sarà direttamente questo thread a notificare ai client tramite multicast la fine del calcolo delle ricompense per quel ciclo.

2.3 Strutture dati

La maggior parte delle operazioni che il server svolge sono svolte su strutture dati che devono essere per forza di cose sincronizzate, dato che si tratta di un server con più thread.
Innanzi tutto, ecco una lista delle strutture dati utilizzate:

- HashMap<String,User> Utenti
- HashMap<String,ArrayList<Post>> Posts
- ArrayList<Post> Allposts
- ArrayList<Activity> Activities
- HashMap<String,Wallet> Wallets
- ArrayBlockingQueue<Runnable> queue

La prima struttura dati *Utenti* è sostanzialmente un database contenente tutti gli user registrati al servizio. Dato che l’id utente è univoco è utilizzato come

chiave della map al quale corrisponde un utente. *User*, la classe che viene mappata con l'username, è una classe che presenta cinque parametri diversi attribuiti ad ogni utente della piattaforma:

- ID: id univoco dell'utente che serve da identificativo
- Password: questa stringa contiene la password codificata tramite lo SHA-256 dell'utente, la password quindi non viene memorizzata in chiaro
- Tags: questo array contiene i tags dell'utente
- Followers: questa lista contiene tutti i seguaci dell'utente
- Following: questa lista contiene tutti i seguiti dall'utente

Tramite questa struttura dati è possibile quindi effettuare alcune operazioni essenziali da parte del server, per esempio una volta ricevuta la richiesta di login, verrà verificata l'esistenza dell'utente in questione e verrà inoltre confrontata la password, ovviamente dopo averla codificata, con quella memorizzata associata allo user, se tutto sarà corretto si procederà al login e verrà aggiornato a *true* il valore *loginStatus* dell'attachment della chiave e anche lo username con cui si è loggati verrà aggiornato con l'id del login. Questa struttura serve, inoltre, per operazioni di ricerca di utenti e la visione dei suoi seguiti e seguaci, per esempio per le richieste di visione del Feed, verranno presi tutti i seguiti dello user che ha fatto questa richiesta dalla struttura *Utenti* e verranno spediti tutti i post associati a quegli utenti.

La struttura dati *Posts* come si intuisce serve per memorizzare tutti i post fatti da un determinato utente, la classe *Post* è così composta:

- Idpost: id univoco del post
- Title: titolo del post
- Content: contenuto del post
- Author: autore del post
- Reward_cicle: numero che serve per identificare il ciclo di reward in cui il post è stato creato (una spiegazione migliore verrà fornita in seguito nella sezione dedicata al calcolo delle ricompense)
- Likes: una lista contenente tutti gli utenti che hanno valutato il post
- N_likes: numero di like
- N_dislikes: numero di dislike
- Comments: una lista di commenti, Comment è una semplicissima classe, che non è altro che una coppia <titolo,commento>

la struttura associa ad ogni Idutente una sua lista di post, questa struttura dati viene utilizzata oltre che per operazioni di ricerca, anche per la richiesta di visione del Blog, dopo la quale verranno quindi restituiti tutti i post dell'utente.

Allposts è una struttura dati di supporto che viene utilizzata per rendere più veloci operazioni di ricerca di post, dato che la ricerca di un post in una map è abbastanza dispendiosa. Questa struttura tiene un riferimento ai post creati e li memorizza in una lista, che è quindi più rapida da accedere qualora avessi bisogno di verificare l'esistenza di un post, per esempio per rewinnarlo, commentarlo o votarlo.

Activities è una struttura dati che serve per il calcolo delle ricompense. Lo scopo verrà spiegato meglio in seguito, ma sostanzialmente è una struttura che memorizza delle attività, ovvero like e commenti, quindi interazioni con i post, che verranno utilizzate dal meccanismo di calcolo delle ricompense per calcolare le reward e poi distribuirle ai curatori e all'autore del post.

Wallets è una struttura dati che associa ad ogni utente registrato un portafoglio che è una classe *Wallet;* la classe *Wallet* è così composta:

- Wincoin: ammontare di wincoin posseduti
- Transactions: lista delle ricompense ricevute con associato un timestamp
- Iduser: il proprietario del portafoglio

Questa struttura serve per tenere traccia dei movimenti e dell'ammontare di wincoin posseduti da ogni utente, oltre ovviamente ad avere dei riferimenti per accreditare nuove ricompense agli utenti che hanno interagito con i post.

Quque è la coda utilizzata dal pool di thread per la gestione delle richieste.

2.4 Gestione della concorrenza

La gestione della concorrenza è stata gestita in maniera tale da ridurre il meno possibile il degrado delle prestazioni del server, questo è stato possibile tramite l'utilizzo delle ReadWriteLock, questi tipi di lock vengono utilizzati per permettere l'accesso in lettura delle strutture a più threads contemporaneamente, ma permettono la scrittura a un solo thread, che inoltre non permette la lettura in quel momento. L'utilizzo delle lock è necessario perché altrimenti si rischia di fare letture inconsistenti quando qualcuno sta modificando una struttura dati e contemporaneamente si prova a leggerla. Per esporre al meglio questo meccanismo prendiamo un metodo come esempio:

```
private int follow(String tofollow) {
    Attachment att = (Attachment) key.attachment();
    if(tofollow.equals(att.loggedAs)) {
        sendResponse("Non puoi seguire te stesso",0);
        return 0;
    }

    //acquisisco la readlock degli utenti
    readlockUtenti.lock();
    if(!Utenti.containsKey(tofollow)) {
        sendResponse("Utente inesistente",0);
        readlockUtenti.unlock();
        return 0;
    }
    else {
        if(Utenti.get(tofollow).addFollower(att.loggedAs) == 0) {

            readlockUtenti.unlock();
            sendResponse("Segui già " + tofollow,0);
            return 0;
        }
        if(Utenti.get(att.loggedAs).addFollowed(tofollow) == 0) {

            readlockUtenti.unlock();
            sendResponse("Segui già " + tofollow,0);
            return 0;
        }
        readlockUtenti.unlock();
        try {
            server.update(att.loggedAs,tofollow);
        } catch (RemoteException e) {

            e.printStackTrace();
        }
        sendResponse("Hai iniziato a seguire " + tofollow,1);
        return 1;
    }
}

//-------------------------------------unfollow-----------------------
```

In questo esempio dove il metodo serve per seguire un utente, si nota come la readLock venga usata prima di fare una lookup dell'utente che si vuole seguire nella struttura Utenti, questo perché si potrebbe incorrere in una situazione dove utenti si stanno registrando mentre io sto accedendo alla mappa, rischiando di ottenere situazioni imprevedibili. Questo esempio serve anche per introdurre un altro meccanismo che è stato utilizzato per gestire la concorrenza, si può notare infatti che tramite il metodo addFollower e il metodo addFollowed, si va sostanzialmente ad aggiornare la struttura dati interna alla classe User, questo non provocherà nessun problema dato che il metodo è sincronizzato grazie all'utilizzo dei monitor (*synchronized)*, infatti ogni tentativo di aggiornare strutture interne alle classi, come i like, i followers, aggiunta di commenti ecc. viene fatta tramite metodi sincronizzati che quindi garantiscono la modifica atomica del campo. I valori di ritorno delle get, per esempio per ritornare la lista dei seguiti, ritornano una copia della struttura dati, non la struttura dati stessa, quindi non si incorre nel problema in cui, sebbene il metodo sia sincronizzato, quando vado a lavorare esternamente sulla struttura, essa possa essere modificata, infatti lavorerò su una copia esatta e quindi non incorrerò in questi problemi.

2.5 Calcolo delle ricompense

Il calcolo delle ricompense è un'operazione delegata ad un thread specifico, che, come detto in precedenza, si occupa anche di notificare i client tramite multicast.
Il calcolo delle ricompense fa uso di una struttura dati *Activities* che sostanzialmente è una lista di attività, le quali sono definite da una classe *Activity* che ha i seguenti parametri:

- Type: il tipo di attività, 0 se è un like, 1 se è un commento
- Idpost: id del post verso il quale è stata fatta l'interazione
- Rewardcicle: ciclo di reward in cui il post è stato creato
- Author: autore del post con cui si è interagito
- Madeby: chi ha compiuto l'interazione
- Numcomment: numero di commenti totali sul post fatti da chi ha fatto il commento

Il calcolo avviene secondo questa modalità: viene acquisita la lock sulla lista delle attività e viene creata una copia di quest'ultima, successivamente vengono tolte dalla lista (*Activities)* tutte le attività che verranno utilizzate per il calcolo della reward. Tramite vari passaggi vengono divisi in gruppi i vari post su cui verrà effettuato il calcolo della ricompensa e vengono ulteriormente suddivisi like e commenti, inoltre per i commenti viene calcolato precedentemente un valore relativo al peso del commento: tramite l'utilizzo del campo numcomment, infatti, vengono calcolati i commenti totali della persona sul post e quindi viene fatta l'operazione che ne calcola il peso effettivo. Infine viene tutto passato alla funzione che calcolerà, tramite l'utilizzo della funzione dataci dal testo del progetto, la reward effettiva che verrà suddivisa in percentuali pari ai valori contenuti in *authorReward e curatorReward* , e distribuita, secondo la percentuale, all'autore e , una volta presa la quota riservata ai curatori e suddivisa in parti uguali, a tutti gli utenti che hanno interagito. È importante sottolineare come il server salverà su disco anche la lista delle attività, così facendo anche se il server viene spento ma sono presenti interazioni da valutare, alla riaccensione queste verranno recuperate e usate nel prossimo calcolo delle ricompense.

2.6 Persistenza del server

Il server fa uso della libreria gson per memorizzare su disco tutte le strutture dati che immagazzinano informazioni sugli utenti e tutto ciò a loro collegato. Per cercare di mantenere il più possibile i dati aggiornati su disco, ogni lasso di tempo (configurabile modificando il parametro checkpont_time del file di

configurazione) viene fatto un salvataggio delle strutture dati su disco, di questo si occupa un thread apposito (t_save) lanciato all'inizio dell'avvio del server, che fa uso di una funzione *writeFile*, definita nella classe *Utils,* che va a serializzare e memorizzare le strutture su disco, secondo i percorsi file passati anch'essi nel file di configurazione. In aggiunta al metodo precedente, il server può essere terminato digitando il comando "end" da tastiera, che farà sì che un ulteriore thread lanciato all'avvio (t_end) inizierà il graduale spegnimento di quest'ultimo, attendendo che i thread finiscano le richieste che stanno esaudendo o hanno in coda, ma non accettandone altre; una volta fatto questo, il thread provvederà a salvare tutti i dati su disco.

All'avvio tutti i dati memorizzati su disco saranno recuperati, se è la prima volta che il server si avvia non succede sostanzialmente niente, mentre se qualche file è corrotto o inesistente (per esempio non esistono post dato che nessuno li ha pubblicati) il server inizializza normalmente la struttura dati come se fosse la prima volta che il server si avvia.

2.7 Gestione errori di connessione

È stato cercato il più possibile di gestire eventuali errori derivati da client che si disconnettono inaspettatamente o prima di aver fatto il logout, tramite la chiusura del canale e l'eliminazione della chiave qualora venisse lanciata un'eccezione relativa alla socket. Per il servizio di rmi della notifica dei follower, è presente un meccanismo dove viene fatta la try della notifica, qualora il client si fosse disconnesso senza fare l'unregister delle callbacks.

2.8 Esecuzione del server

Per avviare l'esecuzione del server basta lanciare la classe main; è importante che il file di configurazione sia nella stessa cartella dove è situato il file del server. Quando si vorrà far terminare il server basta digitare "end" nella console e attendere la terminazione, questo farà si che il server avvii la chiusura corretta del server, chiudendo tutti i thread e salvando su disco le strutture dati.

3. Client

Il client è sostanzialmente un thin client, tutte le operazioni sono delegate al server, per cui al client è richiesto solamente di impacchettare la richiesta in maniera corretta e spedirla sottoforma di json al server, che si occuperà di esaudire la richiesta o meno.
Il client funziona in questo modo: al lancio è possibile fare richiesta di registrazione tramite il servizio rmi oppure fare il login; qualsiasi richiesta, a parte quella di "quit" o "register", se non si è loggati fallirà.
Una volta fatto il login una variabile globale *logged* viene settata a 1 e viene anche settato lo username con il quale siamo loggati e col quale ci è permesso fare richieste.
Il client una volta fatto il login con successo, può iniziare a fare le richieste, tutte le richieste possibili sono consultabili con il comando "help".
Il client può sostanzialmente eseguire una richiesta per volta, in quanto per rendere equa la sottomissione di richieste da parte di più client, il canale dal lato del client è impostato in modalità bloccante, si deve quindi attendere una risposta prima di riprendere a inviare ulteriori richieste.
La ricezione del pacchetto di risposta avviene in due fasi, il primo pacchetto ricevuto indica la dimensione del buffer da allocare per ricevere il secondo pacchetto, che sarà la risposta effettiva alla richiesta.
La risposta come già detto è una coppia: la risposta effettiva, che sarà la risposta alla richiesta in caso di successo oppure la motivazione del fallimento della richiesta, e l'esito della richiesta (1 successo, 0 fallimento).

2.1 Multicast

Per il servizio di multicast lato client la modalità per reperire i riferimenti per collegarvisi è stata quella di indicarli nel file di configurazione, nel quale vengono messi praticamente tutti i riferimenti per collegarsi al server, è stato ritenuto opportuno di metterci anche quest'ultimo. L'alternativa era di far inviare al server un pacchetto all'avvenuto login contenente i riferimenti per collegarvisi, ma ho ritenuto questa strada sostanzialmente dispendiosa per il server, che può semplicemente mantenere online un riferimento globale con tutti i parametri di configurazione per i client.
Il multicast viene effettuato nel seguente modo: all'avvenuto login, il client si registra al servizio di multicast; viene successivamente avviato un thread che si occuperà di attendere e stampare a schermo le notifiche delle ricompense. Una volta effettuato il logout, il gruppo viene abbandonato e il multicast chiuso,

questo causerà il lancio di un'eccezione nel thread che gestisce il multicast, che terminerà la propria esecuzione.

2.2 Esecuzione del client

Una tipica esecuzione del client prevede il lancio nella stessa cartella dove è presente il file di configurazione, successivamente se si è registrati si effettua il login, altrimenti ci si registra e poi si effettua il login.

Si può dunque iniziare a fare richieste al server. Per la lista delle richieste possibili e la loro dicitura, digitare "help".

Gli unici comandi aggiuntivi che ho deciso di inserire sono: "followers", per recuperare dal server la lista aggiornata dei propri followers dato che potrei essermi disconnesso e potrei non avere più un riferimento ai miei follower; "quit", che permette di spegnere il client in maniera corretta, ovvero effettuando logout prima di uscire.

4.Modalità compilazione ed esecuzione

Per la compilazione del progetto basta collocarsi nella cartella 'src' e utilizzare il comando:

```
javac -cp ".:./gson-2.8.9.jar" *.java
```

oppure

```
javac -cp ".;./gson-2.8.9.jar" *.java
```

per l'esecuzione invece bisogna lanciare il server e il client da due terminali diversi sempre all'interno della cartella 'src' con i comandi:

```
java -cp ".:./gson-2.8.9.jar" WinsomeServerMain
```

oppure

```
java -cp ".;./gson-2.8.9.jar" WinsomeServerMain
```

per il server, per il client invece:

java -cp ".:./gson-2.8.9.jar" WinsomeClientMain

oppure

java -cp ".:./gson-2.8.9.jar" WinsomeClientMain