Relazione Progetto SOL 2020-2021

Sofia Pisani
Matricola: 646301

06/09/2022

Contents

1 Istruzioni per l'uso

1.1 Compilazione

Il progetto non include eseguibili precompilati. Prima di utilizzarlo bisogna utilizzare uno dei seguenti comandi make:

make all : Genera gli eseguibili server.out e client.out nella cartella out.

make server : Genera l'eseguibile server.out nella cartella out.

make client : Genera l'eseguibile client.out nella cartella out.

make clean : Ripulisce la cartella out e la cartella obj dai moduli oggetto compilati.

make test[1,2,3] : Genera gli eseguibili necessari ed esegue automaticamente il test richiesto.

1.2 Client

1.2.1 Avvio

Per lanciare il client, dopo averlo compilato, eseguire il file client.out, dandogli delle opzioni valide.

1.2.2 Comandi

Di seguito una lista dei comandi disponibili e i loro effetti:

-h : Stampa una lista dei comandi disponibili e i loro effetti.

-f filename : Connette il client al socket di nome filename.

-w dirname[,n=0] : Scrive sul server fino a n file dalla cartella dirname, visitando ricorsivamente le subdirectory. Se n=0 o non è specificato scrive tutti i file trovati.

-W file1[,file2] : Scrive sul server tutti i file specificati.

-D dirname : Imposta la cartella in cui il client salvera i file espulsi dal server a seguito di un capacity miss. Se non impostata i file espulsi dal server verrano ignorati.

-r file1[,file2] : Legge dal server tutti i file specificati.

-R [N=0] : Legge n file qualsiasi dal server. Se n=0 o non è specificato scrive tutti i file trovati.

-d dirname : Imposta la cartella in cui il client salvera i file letti. Se non impostata i file letti verranno ignorati.

-t time : Imposta il tempo in millisecondi che intercorrerà tra una richiesta al server e la prossima (0 di default).

-l file1[,file2] : Ottiene una lock sui file specificati.

-u file1[,file2] : Rilascia la lock sui file specificati.

-c file1[,file2] : Rimuove tutti i file specificati dal server.

-p : Abilita le stampe sullo stdout.

1.3 Server

1.3.1 Avvio

Per lanciare il server, dopo averlo compilato, eseguire il file server.out, passandogli opzionalmente da linea di comando un path al file di configurazione da usare. Se non specificato cerchera un file config.txt nella directory corrente, e se non trovato usera delle impostazioni di default.

1.3.2 File di Configurazione

Un file di configurazione valido è una serie di coppie CHIAVE=VALORE, ognuna su una linea diversa. Il file di configurazione può anche includere commenti, cioè linee che iniziano con // verranno ignorate. Le chiavi disponibili, insieme a una breve descrizione dei loro effetti e ai loro valori di default, possono essere trovati nel file config.txt generato durante la compilazione del server.

2 Architettura del Server

Il client ha un architettura estremamente semplice, limitandosi a parsare le opzioni da linea di comando e lanciando le richieste necessarie al server tramite l'API. Per questo non è particolarmente interessante, e ci limiteremo a parlare dell' architettura del server.
Il server ha infatti una struttura ben più complessa, dovendo gestire i file memorizzati al proprio interno, e connessioni simultanee da più client.

2.1 Layout dei Thread

Il server funziona su multipli thread:

- Il main si occupa dello startup del server, facendo il parsing del file di config, inizializzando appropriatamente le strutture dati necessarie, e inizializzando la ThreadPool. Una volta fatto questo si occuperà unicamente di accettare connessioni, e passarle alla ThreadPool perchè le gestisca.
  Si occupa inoltre della terminazione, essendo l'unico thread che non blocchi i segnali SIGINT, SIGQUIT e SIGHUP.
- Il manager della ThreadPool è un thread che si occupa della gestione di tale struttura. Si occupa principalmente di generare e uccidere thread a seconda della necessità, oltre a svolgere un ruolo nella terminazione della ThreadPool. I thread generati al suo interno si occupano poi indipendentemente di consumare la queue di lavoro interna alla ThreadPool.
- I worker thread si occupano di consumare le task submittate alla ThreadPool. Il server genererà una task per ogni connessione, e quindi i worker thread andranno a occuparsi della gestione delle connessioni, accettando richieste, processandole (andando quindi anche a modificare il FileSystem), e restituendo risposte appropriate.

2.2 Stato del server

Il server deve tenere traccia di:

Informazioni relative ai file memorizzati Vengono conservate nella struttura FileSystem, e gestite tramite le funzioni che usano questa struttura.

Informazioni relative alle singole connessioni Ogni thread gestisce una connessione singola, e ne ricorda lo stato in una struttura ConnState. Informazioni rilevanti sono ad esempio i file aperti al momento e il file lockato se presente.

Informazioni relative ai thread in esecuzione Queste vengono gestite autonomamente dalla ThreadPool.

2.2.1 FileSystem

Il server inizia dal ServerMain, che fa poi partire gli altri thread nella fase di setup.

2.2.2 ThreadPool

Il ServerMain si occupa principalmente dello startup del server, all completamento del quale entra in un loop in attesa di un comando di ”exit” da linea di comando.

- Durante lo startup inizializza il file di configurazione. Da questo prende il path per il serverData, e, se questo si trova sul disco, lo inizializza caricandolo da disco, alternativamente inizializza un serverData pulito.
- Va poi ad assegnare delle task periodiche ad uno scheduler, in particolare il salvataggio dei dati del server e il calcolo delle ricompense. Lo scheduler si assicurerà di gestire i thread lanciandoli ad intervalli appropriati.
- Inizializza quindi il socket TCP e il servizio di registrazione RMI, che gestirà indipendentemente i thread necessari alla registrazione di nuovi utenti.
- Infine, prima di entrare in un loop in cui ascolta comandi dell'utente, inizializza un thread TCPHandler con le informazioni sulla connessione TCP, e lo lancia.
- Nel caso riceva un comando di ”exit” si occuperà poi della chiusura pulita del server, notificando tutti i thread iniziati di terminare appena possibile e facendo un ultimo salvataggio dei dati in serverData.

2.2.3 ConnState

Il TCPHandler si occupa di aspettare nuove connessioni e creare dei thread ConnectionHandler per gestirle. I thread vengono gestiti da una threadpool, questo permette flessibilità nel numero di connessioni attive e va a minimizzare il tempo di accettazione di una nuova connessione.

2.2.4 ConnectionHandler

Ogni thread ConnectionHandler gestisce quindi una singola connessione. Questi tengono traccia delle informazioni relative alla singola connessione TCP, e quindi anche del corrente stato di autorizzazione della connessione. Il ConnectionHandler si occupa di rispondere a Request del client, aggiornando inoltre serverData in modo appropriato. Lavora in un ciclo del tipo:

1. Aspetta una Request dal client.
2. Avendo ricevuto una Request la gestisce all interno di HandleRequest, effettuando eventuali query/alterazioni a serverData e allo stato della connessione, e generando una Response adeguata.
3. Restituisce al client la Response generata al passo precedente.

Il grosso della logica necessario per aggiornare correttamente lo stato del server è implementato in serverData, e ConnectionHandler si occupa principalmente di interpretare le richieste e passarne i parametri ai metodi di serverData adeguati. Questo approccio ci permette di effettuare una forte separazione delle responsabilità. Ci semplifica inoltre notevolmente il compito di tenere correttamente aggiornate le complesse strutture dati di supporto interne a ServerData.

2.2.5 SaveHandler

Il SaveHandler viene eseguito periodicamente dallo scheduler, con tempo tra un esecuzione e un altra preso dal file di configurazione. All'interno del SaveHandler si fa solo una chiamata al metodo save di serverData, che contiene al suo interno la logica necessaria al salvataggio dei propri dati. Un salvataggio periodico dei dati permette agli amministratori del server di scegliere a piacere un tempo massimo tra un backup e il prossimo, ottenendo quindi il tradeoff performance/affidabilità ottimale per il proprio caso.

2.2.6 RewardHandler

Il RewardHandler viene eseguito periodicamente dallo scheduler. All'interno del RewardHandler si vanno ad effettuare tutti i calcoli relativi al ciclo corrente di reward, effettuando query a serverData e aggiornando poi questa con i nuovi valori dei Wallet degli utenti. Infine, prima di terminare, effettua un write in multicast a tutti i client in ascolto, che riceveranno quindi una notifica non appena viene completato un ciclo di reward.

3 Classi Principali

Le classi sono divise in 3 package:

Client Di poco interesse, le sue classi implementano solo la poca logica necessaria al funzionamento del thinclient.

Server Contiene le classi relative ai thread listati sopra, con un eccezione interessante; ServerData.

Common Contiene classi rappresentanti i costrutti principali del modello di WinSome, quali User, Post, Comment etc.

Analizziamo le strutture dati del modello di WinSome con un approccio top-down, iniziando quindi da ServerData.

3.1 ServerData

La classe ServerData gestisce tutti i dati relativi ad un istanza di WinSome. Questi sono fondamentalmente una collezione di User, una collezione di Post, e una collezione di coppie username:passwordHash, più delle strutture di supporto. User, hash delle password e Post vengono conservati in delle HashMap con chiavi i rispettivi identificatori univoci (Username e Id), in modo da permettere un accesso rapido ai dati necessari. Le hash delle password vengono conservate al di fuori dell'User a cui sono relative per questioni di sicurezza, permettendo più granularità su da dove queste siano visibili.
Abbiamo poi una serie di strutture di supporto, di cui spieghiamo brevemente l'utilità.

UsersByTags rappresenta un mapping (tag $\rightarrow$ lista usernames). Permette di risalire efficientemente da una determinata tag alla lista degli User con quella tag. Questo velocizza l'operazione di list users. Viene aggiornata ogni volta che si registra un nuovo utente.

ActivePosts rappresenta una collezione di id dei post che hanno ricevuto attività dall'ultimo ciclo di calcolo dei reward. Permette di calcolare i reward solo per i post che lo necessitano, velocizzando l'operazione. Un post ci viene aggiunto quando riceve commenti o voti positivi, e viene rimosso alla rimozione del post. Viene svuotata alla fine di ogni ciclo di reward.

Rewinners rappresenta un mapping (id $\rightarrow$ lista usernames). Permette di risalire efficientemente dall'id di un Post alla lista degli User che ne hanno effettuato il rewin. Questo è necessario in quanto alla rimozione di un post, bisogna eliminare i riferimenti a questo dal blog di ognuno degli utenti che ne hanno effettuato il rewin. Viene aggiornata ogni volta che un utente rewinna un post, e ogni volta che un post viene eliminato.

FollowNotifiers rappresenta un mapping (username $\rightarrow$ FollowerClient). È qui che si tiene traccia dei meccanismi di callback RMI dei singoli client. Questa viene aggiornata tramite richieste di subscribe/unsubscribe del client via RMI; inoltre, quando un FollowerClient non ci risulta raggiungibile viene rimosso, in modo da non dover tenere traccia dei client di utenti che assumiamo essere inattivi.

Ogni server ha la propria istanza di ServerData, che viene inizializzata dal ServerMain e poi condivisa tra ogni suo thread. Per gestire la concorrenza viene utilizzato un ReadWriteLock, che permette ai thread di condividere il ServerData in lettura, necessitandone controllo esclusivo solo in casi di scrittura. Aggiornamenti ai dati del server vengono generalmente effettuati tramite metodi propri di ServerData, all interno dei quali ci assicuriamo di aggiornare in maniera appropriata le strutture dati di supporto.

3.2 Common

Le classi del package Common più interessanti sono User, Post, e Request/Response.

3.2.1 User

La classe User rappresenta le informazioni relative a un singolo utente di WinSome. Contiene quindi informazioni quali l'username dell'utente e la sua lista di tag, oltre a un set contente gli id dei post presenti nel proprio blog. Conserviamo poi qui un set degli utenti che questo User segue o dai quali viene seguito. Teniamo due set separati follower/following e li aggiorniamo a ogni nuovo follow/unfollow in modo da rendere più efficienti le operazioni di list followers o list following. Gli User vengono creati all interno di ServerData alla registrazione di un nuovo utente, e possono poi essere ottenuti da questa da ogni thread del server. Per controllare la concorrenza si usa anche qui un unico ReadWriteLock.

3.2.2 Post

La classe Post rappresenta le informazioni relative a un singolo post su WinSome. Contiene informazioni quali il proprio autore, titolo e contenuto. Contiene inoltre informazioni più dinamiche quali una lista dei commenti effettuati sul post, e tiene traccia dei voti sul post e dei relativi autori. Presenta poi delle strutture dati di supporto utili al calcolo dei reward. nComments rappresenta un mapping (username $\rightarrow$ numero di commenti). Insieme a votes, però, non contiene tutte le informazioni necessarie a calcolare i reward del post, in quanto bisogna saper distinguere le interazioni con il post già conteggiate dalle altre. A questo fine il Post contiene anche un PostCheckpoint, che contiene una copia di nComments e di votes relativa all ultimo calcolo dei reward su di questo post. Con il PostCheckpoint si possono quindi trovare le interazioni non ancora conteggiate calcolando la differenza tra questo e lo stato corrente. Come gli User, i Post sono accessibili da tutti i thread del Server, e, per gestire la concorrenza si usa un unico ReadWriteLock.

3.2.3 Request/Response

Per le comunicazioni su TCP il sistema WinSome utilizza un sistema del tipo Richiesta/Risposta. Questo sta a significare che il Client farà richieste al server e aspetterà una risposta da questo. Le richieste dal Client al server prendono la forma di una classe Request serializzata. Questa è caratterizzata da un tipo enum e una lista di parametri opzionali. In risposta il Server manderà una Response serializzata, che è caratterizzata da un codice (i quali valori cercano di seguire quelli dei codici HTTP/1.1), un messaggio, ed eventuali contenuti richiesti. Questo modello di comunicazione tramite Request/Response serializzate è utile per semplificare il parsing delle richieste lato server, e delle risposte lato client, che consideriamo al momento più importante di un implementazione più complessa, seppur moderatamente più efficiente.

4 Migliorie Possibili

Per quanto il sistema WinSome nella forma corrente sia perfettamente funzionale, e capace di gestire un traffico moderatamente alto, sono possibili notevoli migliorie nella sua UI e performance. Un interfaccia grafica darebbe chiaramente un grande miglioramento della user experience. Dal punto di vista della performance, poi, identifichiamo alcuni bottleneck notevoli che, in caso di traffico troppo elevato potrebbero diventare dei problemi. Il primo e più notevole è il singolo ReadWriteLock su ServerData. In quanto ogni connessione condivide deve condividere l'accesso a questo, se qualunque di queste ottiene una lock di write andrà a bloccare ogni altra connessione nella durata della sua operazione. Ciò si potrebbe mitigare con una strategia di lock più granulare, avendo quindi lock separate per ogni struttura dati, al costo di inserire una certa complessità nel codice. La soluzione migliore sarebbe in realtà banalmente di usare un sistema di database esistente come struttura dati di supporto, e ciò ci darebbe certamente un efficienza maggiore rispetto a una soluzione creata da noi.
Inoltre, l'implementazione delle comunicazioni TCP tramite threadpool va a creare un thread separato per ogni connessione che viene gestita. Questo è inefficiente se confrontato ad esempio a una soluzione che usi NIO e il multiplexing di canale, che andrebbe invece a usare un numero fisso di thread per gestire le connessioni in arrivo, con costi di context-switching notevolmente minori. Queste due migliorie permetterebbero probabilmente al server di gestire traffici molto più elevati in futuro.

Detto questo, non crediamo necessarie al momento eccessive ottimizzazioni, che ci paiono premature rispetto al numero di utenti attivi (0).