Lorenzo Bellomo 531423 | Nicolò Lucchesi 533515

RELAZIONE PROGETTO 'SOCIAL GOSSIP' LABORATORIO RETI

USAGE

L'esecuzione del server è senza parametri a linea di comando, in quanto non necessita di alcuna informazione aggiuntiva. L'esecuzione del client invece necessita di un argomento, che è l'indirizzo IP del server. In caso di numero errato o di parametro errato il client termina la propria esecuzione dopo aver fatto display dell'usage. Per eseguire un test in locale risulta necessario passare come parametro" localhost" o "127.0.0.1".

IL FORMATO DEI MESSAGGI

La prima scelta che è stata affrontata è quella riguardante il formato dei messaggi che il server ed il client si sarebbero dovuti scambiare, ed è ricaduta su una suddivisione in tre classi, ognuna delle quali avrebbe condiviso tre informazioni principali: il mittente, il destinatario ed il tipo di messaggio.

La prima sottoclasse che è stata individuata è quella dei messaggi di richiesta (RequestMessage), che si occupa di modellare tutte le richieste che il client può effettuare al server, dalla richiesta di login, registrazione, creazione di una chat room… all'invio di un messaggio ad un amico, con la differenza che quest'ultimo tipo di richiesta è strutturato in modo leggermente diverso, poiché in questo caso il messaggio di richiesta fa da 'involucro/busta' per il messaggio testuale interno. In modo analogo a quello che succede nella pila protocollare TCP/IP, il server, ricevuto un messaggio di richiesta di questo tipo, si occuperà di rimuovere la 'busta' su cui sarà indicato il mittente ed il destinatario (il server), per individuare nel messaggio di testo ivi contenuto il vero destinatario del messaggio, ossia un altro utente iscritto a Social Gossip (il 'target') e quindi procedere all'inoltro di questo messaggio contenuto nella busta al reale destinatario (individuato in maniera univoca dal suo nickname con cui si è registrato al servizio).

Proprio il messaggio testuale è il secondo tipo di messaggio che è stato modellato: TextMessage, è appunto l'astrazione di un messaggio che un utente invia ad un amico in una chat.

L'ultimo tipo di messaggio che abbiamo individuato è stato il ResponseMessage, che modella il messaggio di risposta del server ad una richiesta qualsiasi del client, con due sottocategorie principali, la risposta positiva o ACK, e la risposta negativa o NACK (che si differenzierà per il tipo di errore).

Un'unica eccezione alla regola si presenta nel caso di un particolare messaggio di richiesta, quella di OpenP2PConnection, che viene utilizzata per sincronizzare i due client che si vogliono scambiare un file. In seguito ad una richiesta di invio di file da client A verso client B, che viene fatta al server tramite una richiesta di tipo SendFileToFriend, il server invia al secondo client destinatario una richiesta di tipo OpenP2PConnection. A questo tipo di richiesta (l'unica con destinatario diverso dal server), il client risponde con un ACK comprensivo di proprio indirizzo IP e la porta su cui si è messo in ascolto. Il server dunque, ricevuto questo ACK, risponde al primo client inviando un responso comprensivo di questi campi aggiuntivi permettendo quindi lo scambio di file fra i due client.

La classe che si occupa della trasformazione dei messaggi in JSONObject e viceversa è MessageHandler.

Tale classe è fornita di tre metodi:

- Uno che si occupa di inviare una stringa su un determinato Socket;
- Uno che, data una stringa in formato JSON, ricostruisce il messaggio relativo e ne recupera i campi significativi;
- Uno che svolge il compito inverso, ovvero crea il JSONObject a partire da un messaggio.

In particolare, di un determinato messaggio verranno inviati soltanto i campi strettamente significativi (Per esempio in un messaggio di risposta ad una richiesta di tipo createChatroom, il campo address sarà significativo, mentre in una richiesta di friend list tale campo non sarà presente nel JSONObject).

LE STRUTTURE DATI

Il secondo punto cruciale che è stato affrontato è quello della scelta delle strutture dati: partendo dal 'cuore' di Social Gossip, è stata presa la decisione di realizzare un grafo che fosse in grado di descrivere con semplicità gli utenti del servizio e le relazioni di amicizia fra essi. A tale scopo dunque è stata creata un'interfaccia con generics ed una classe che implementasse tale interfaccia e le fornisse un minimo di flessibilità in più per la gestione delle operazioni di gestione di una social network. In particolare, il grafo cosi implementato contiene un insieme di nodi e un insieme di archi: il primo è stato rappresentato da una HashMap concorrente (per le modifiche alla struttura che i vari thread del server avrebbero apportato durante l'esecuzione) che associa ad ogni username un oggetto utente, mentre l'insieme degli archi, che corrispondono alle relazioni di amicizia fra utenti, è stato rappresentato come una HashMap che mette in relazione una chiave, l'username di un utente, con una ulteriore struttura HashMap, che rappresenterà i suoi amici associando ad un username la sua struttura utente. Entrambe le Map utilizzate sono Concurrent, per il motivo prima citato, e la chiave di entrambe le strutture è l'username poiché questo sarà un identificatore univoco per ogni utente, in quanto username uguali non possono essere assegnati a client diversi.

Un'altra struttura dati che si è rivelata necessaria è quella degli utenti online: per realizzarla è stato preferito l'utilizzo di un vettore (concorrente) contenente istanze di una nuova classe, OnlineUser, la quale modella la rappresentazione di un utente online, necessaria per individuare con facilità il destinatario di messaggi fra gli utenti online (eliminando la necessità di dover consultare il grafo ogni volta) o di notifiche RMI

Proprio per quest'ultimo motivo si è rilevato necessario conservare lo stub RMI di ogni utente online, per essere in grado di eseguire callback con facilità, legando all'interno di un OnlineUser il suo stub al suo username e ai suoi Socket (che verranno descritti più avanti).

Un client che desideri entrare nella social network con un'operazione di registrazione o di login, deve inviare, insieme con tale richiesta, uno stub RMI per le notifiche di cambio stato di un utente. Il server, ad ogni connessione e disconnessione di un amico di tale utente, avrà il compito di mandare una notifica tramite tale stub inviato e mantenuto nel vettore di utenti online. Ulteriore compito dello stub è la notifica di una nuova amicizia. Quando un client invia una richiesta di amicizia ad un utente e quest'ultimo è online, il server invierà all'utente che è stato aggiunto agli amici una notifica RMI di tipo NewOnlineFriend(nickname). Sarà compito del client verificare che tale utente

non era precedentemente amico proprio, e quindi aggiungerlo alla propria struttura dati atta a mantenere tale informazione.

Per la rappresentazione delle chatroom e degli utenti appartenenti è stata utilizzata una lista di ChatRoom, una classe contente un idchat (l'identificativo univoco di una Chat Room), una lista di utenti appartenenti ad essa ed un indirizzo multicast associato alla Chat Room; questo indirizzo viene assegnato alla creazione della chat room, incrementando una variabile statica contenente il primo indirizzo multicast che assegniamo, 225.0.0.0.

Anche la gestione dei messaggi destinati ad un gruppo avviene in modo simile a quella di una chat con un amico, poiché il mittente invia la richiesta di invio messaggio al server, il quale si occuperà dell'invio in multicast del messaggio testuale, ottenendo l'indirizzo per l'invio dalla struttura sopra citata.

Lato client si trovano invece due strutture dati più semplici, un vettore che conterrà gli username degli amici online, e un altro che conterrà invece quelli offline; queste strutture sono necessarie al client per individuare i propri amici, tenere conto di chi fra questi è online e offline, aggiornando le due liste ogni qual volta il server ci segnala il cambiamento dello stato online/offline di un amico e per riconoscere facilmente quando siamo stati aggiunti da un altro utente, con il meccanismo spiegato nella breve parentesi sul funzionamento delle callback RMI.

IL SERVER

Per quanto riguarda il Server, è stata privilegiata la scelta di utilizzare un thread listener (Acceptor) ed una pool di thread 'worker', che servono una richiesta alla volta ed i quali condividono una coda di WorkerTask.

Il funzionamento più nel dettaglio del ciclo di vita del server è il seguente:

il thread Acceptor si occupa di stare in ascolto su un ServerSocket, su una porta nota ad ogni client, accettando ogni nuova connessione e creando così una nuova connessione per ogni client che effettua la connessione (sfruttando il funzionamento dell'accept).

L'architettura delle connessioni è (come da specifiche), basata sul fatto che il client fa richieste al server e riceve messaggi su due connessioni TCP differenti. Tali connessioni devono essere perennemente collegate (cioè legate al proprio client) e mai svincolate, in quanto dipendono strettamente dal client cui riferiscono. Su di tali connessioni viaggiano:

1. Connessione di Controllo: potrebbe essere definita connessione sincrona. Non è sempre attiva in quanto vive solo nel lasso di tempo compreso fra una richiesta del client e la relativa risposta del server. Su di questo canale viaggiano solo questo tipo di messaggi.
2. Connessione dei Messaggi: potrebbe invece essere definita connessione asincrona. Su di essa viaggiano tutti i messaggi che il server invia per conto di altri client, oltre che tutte le richieste di apertura connessione P2P per lo scambio di file e le relative risposte del client verso il server.

Dalle due connessioni prendono dunque nome i due tipi di Socket: il primo, quello di controllo, è quello che ha origine dal thread acceptor, e viene creato nel modo 'classico' appena descritto, mentre la seconda connessione viene aperta dal server in un modo differente (ma simile all'apertura della connessione dati di FTP). Ottenuto il Socket di controllo da un client appena connesso,

l'Acceptor si fa carico dell'inserimento nella coda condivisa di un WorkerTask di tipo ACCEPT, questo significa che questo oggetto conterrà al suo interno solo il Socket di controllo, poiché quello dei messaggi non è ancora stato creato; nel mentre, un thread Worker generico svolgerà il suo ciclo di vita, che consiste nell'estrarre un WorkerTask dalla coda condivisa e servire la richiesta. Estratto un WorkerTask di tipo ACCEPT, sarà il Worker stesso che procederà con l'apertura del Socket cosiddetto 'dei messaggi', aprendo verso lo stesso client una nuova connessione. Il server reperisce la porta su cui aprire la connessione dal Socket di controllo, in quanto il client stesso, dopo aver aperto con successo la connessione di controllo con il Server, invierà su di essa la porta su cui accetterà la connessione dei messaggi. Ricapitolando, l'apertura attiva della connessione di controllo è dunque svolta dal client ed è su una porta nota, mentre l'apertura attiva della connessione dei messaggi è svolta dal server ed avviene su una porta inviata dal client sulla connessione di controllo.

A questo punto il thread Worker crea un nuovo WorkerTask di tipo 'Serve', ossia contente entrambi i Socket, e lo inserisce nella coda condivisa, tornando all'inizio del suo ciclo. La creazione di questo WorkerTask garantisce che la correlazione tra i due socket venga mantenuta. Se il worker estrae un nuovo WorkerTask di tipo 'Serve' al ciclo successivo, può procedere all'esecuzione del suo compito principale, avendo a disposizione entrambi Socket: legge la richiesta di un client generico, la riconosce effettuando il parsing del messaggio in formato JSON ricevuto (come da specifica), effettua l'operazione richiesta (se valida), invia la risposta al client da cui ha ricevuto la richiesta con un ResponseMessage, indicando dunque l'esito dell'operazione ed eventualmente allegando dati al messaggio e poi reinserisce il WorkerTask nella coda condivisa.

IL CLIENT

Il client utilizza anch'esso un thread Listener, detto ClientListener, insieme con i thread supplementari generati dalla JVM (in particolare dal supporto RMI e da quello della graphic user interface). A differenza del server, il client prende appunto l'input dell'utente attraverso l'utilizzo di un'interfaccia grafica, ed invia richieste sotto forma di RequestMessage, create in modo analogo al server grazie all'utilizzo di tutte le classi condivise con il server stesso riguardo la gestione dei messaggi.

Le operazioni specificate nella documentazione sono disponibili attraverso l'uso di un'interfaccia grafica che è stata realizzata con lo scopo di essere intuitiva, ed ogni 'trigger' di un'azione richiama una funzione in ClientOps, classe che implementa tutte le operazioni disponibili; è in effettiva 'la controparte' lato client delle operazioni presenti in RequestHandler lato server, ossia l'implementazione delle interfacce fornite nelle specifiche. Ogni operazione (tranne quella dei file, discussa dopo) porta alla creazione di un messaggio di richiesta spedito al server, alla conseguente attesa attiva della risposta, alla sua decodifica (JSON->Message in ricezione, viceversa all'invio) e all'aggiornamento della GUI in base all'esito dell'operazione contenuto nella risposta. Mentre per le richieste effettuate in prima persona c'è appunto un'attesa attiva, è stata prediletta una politica differente per quanto riguarda i messaggi in arrivo, che possono arrivare in ogni momento.

Per inviare un messaggio ad un amico con il supporto grafico è necessario cliccare sull'amico con cui si vuole dialogare sulla destra, cui seguirà un'apparizione di uno slot in basso nell'area delle chat attive (come facebook). Cliccando adesso su questa chat si aprirà il box specifico per la messaggistica e sarà possibile procedere all'invio dei messaggi.

Una menzione riguardo al ciclo di vita del thread ClientListener è necessaria in quanto il suo compito non si limita all'ascolto attivo sulla connessione dei messaggi, ma consta anche del controllo delle chatroom e della gestione delle richieste dei file. Nel suo ciclo perenne ClientListener ascolta per un lasso di tempo determinato sulla connessione dei messaggi. In caso di ricezione di messaggi di testo aggiorna la GUI, altrimenti passa al controllo del proprio MulticastSocket, che è registrato su tutti i gruppi relativi alle chatroom a cui è registrato. Dopo aver atteso datagrammi su questo Socket, passa al controllo di connessioni pendenti sul Server Socket dei file. Se il client riceve una connessione per lo scambio di file, la serve immediatamente accettando tale connessione e gestendo nella sua interezza lo scambio di file. Il clientListener è incaricato di tale scambio, e può di conseguenza ritardare l'avvio di messaggi pendenti. Tale scelta è stata fatta considerando che lo scambio di file sarà meno frequente dello scambio di messaggi e che un numero esagerato di thread sarebbe stato un carico eccessivo sulla JVM in particolare in caso di test in locale.

Sia Client che Server fanno uso di una serie di eccezioni (oltre a quelle di base di Java) che sono state definite per individuare dei tipi di errore propri di Social Gossip (come ad esempio NameAlreadyInUse) e su cui sia client che server possono trarre più informazioni dal risultato di un'operazione, ed agire di conseguenza in modo più appropriato, nonché fornire un'astrazione più corretta del problema.

RECAP DELLE CLASSI

Vengono adesso elencate tutte le classi (escluse le eccezioni) ed in breve ne vengono spiegate le funzioni:

SERVER

- Acceptor: Codice del thread che accetta le connessioni sulla porta nota e fa dispatch di task per i thread worker del pool
- ChatRoom: Modella la chatroom e gestisce l'assegnazione degli indirizzi
- Graph & SocialGraph: Interfaccia e classe che implementano il grafo sociale
- OnlineUser: Modella un utente online come caratterizzato da un username e legato al proprio RMI Notifier, oltre che ai propri socket di controllo e dei messaggi.
- RequestHandler: Gestisce, data una richiesta dal client, la sua esecuzione.
- SocialGossipServer: Contiene il metodo main, che inizializza alcune strutture e fa lo spawn dei thread del pool e dell'acceptor
- Translator: gestisce la traduzione dei messaggi REST
- User: modella un utente
- Worker: Contiene il codice dei thread del pool: leggono le richieste dal client, chiamano la funzione relativa di RequestHandler e rispondono con l'esito.
- WorkerTask: Contiene una coppia di Socket se la connessione dei messaggi è stata aperta, altrimenti solo il Socket di controllo.

CLIENT

- AccessWindow: Contiene il codice della GUI al momento dell'accesso
- ChatOps & ChatRoomOps & SocialOps: Interfacce che specificano le operazioni che fornisce il server e le loro funzionalità (nessuna classe le implementa, solo dimostrative)
- ClientListener: Contiene il codice del thread listener del client

- RMINotifier: Implementa Notifier (vedi sotto), classe remota per la gestione delle notifiche RMI
- SG_Home: Contiene il codice della GUI della sessione vera e propria
- SocialGossipClient: Contiene il Main del client, inizializza alcune strutture dati, fa lo spawn del thread listener e fa partire la AccessWindow.

CONDIVISE (fra Server e Client)

- Message & ResponseMessage & TextMessage & RequestMessage: Modellano i messaggi.
- MessageHandler: Classe che è incaricata di trasformare i messaggi in stringhe in formato JSON e viceversa.
- Notifier: Interfaccia per gli stub RMI dei client
- RequestCode & ResponseCode: Contengono i codici numerici delle richieste e delle risposte

Lorenzo Bellomo 531423, Nicolò Lucchesi 533515