Simple Social

Progetto di Laboratorio di Reti

Nicolò Tonci

Agosto 2016

Indice

1 Introduzione

Lo scopo di questa relazione è descrivere le metodologie e scelte implementative nella realizzazione di una semplice rete sociale formata da due componenti *Social Client* e *Social Server* che costituiscono un'architettura client-server.
Il sistema mette a disposizione le seguenti funzionalità: registrazione utente, login utente, invio di richieste di amicizia, possibilità di confermare o rifiutare richieste di amicizia ricevute, visualizzare la propria lista degli amici, ricercare utenti iscritti, pubblicare contenuti (testo), esprimere interesse per i contenuti pubblicati da un utente (equivale al concetto di seguire una persona), visualizzazione dei contenuti, logout utente e una funzionalità interna al sistema per tenere traccia degli utenti attualmente online denominata *Keep-alive*.
L'attenzione è stata rivolta principalmente all'architettura di rete e alle strutture dati della rete sociale e, sebbene sia stato deciso di realizzare un' interfaccia grafica per la componente *Social Client*, essa risulta piuttosto minimale poichè non considerata di fondamentale importanza.
Per una descrizione completa del progetto si rimanda ai documenti forniti dai docenti.

2 Architettura di rete

Come in ogni applicazione client-server il punto fondamentale è la comunicazione tra questi due e tutto ciò che ne concerne.
Per la relizzazione delle features richieste dal docente sono state utilizzate connessioni TCP, datagrammi UDP (Unicast e Multicast) e il sistemi di invocazione remota di metodi Java (RMI).

2.1 Main Server

Il "Main Server" rappresenta il principale mezzo di comunicazione tra le componenti e viene usato per la maggior parte delle funzionalità, in particolare per: registrare un nuovo utente, effettuare il login utente, comunicare al server la volontà di inviare una nuova richiesta di amicizia, confermare o rifiutare richieste di amicizia pervenute, richiedere la lista degli amici, ricercare utenti iscritti, inviare al server un nuovo contenuto (testo) ed effettuare il logout.
Tale server usa connessioni TCP, è in ascolto sulla porta 1500 ed è capace di gestire più clients contemporaneamente tramite un *pool* di thread.
Ad ogni connessione il cliente invia un oggetto di tipo *Request* contenente tutti i parametri della richiesta (token per l'identificazione dell'utente che effettua la richiesta, il comando che caratterizza la richiesta e una stringa con i parametri), il server risponde alla richiesta ed infine il client chiude la

connessione. La struttura dell'oggetto *Request* è visibile in figura 1.

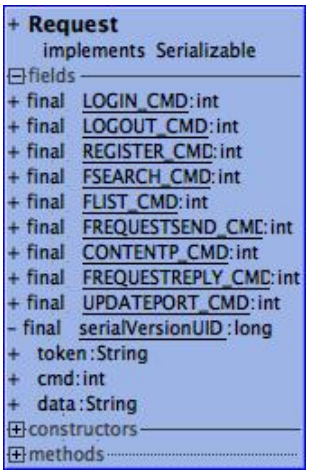


Figura 1: Diagramma UML della classe Request

Il protocollo completo è descritto nella tabella seguente che riassume per ogni richiesta la struttura dei parametri e il risultato atteso.

Tabella 1: Messaggi del protocollo per le comunicazioni sul Main Server

Nome Operazione	**CMD**	**Token richiesto**	**(String) Data**	**(Object) Risposta**	**Descrizione**
LOGIN	1	No	{username}|||||{*password*}	(String) {token} (successo) 'error' (altrimenti)	Login utente, restituisce il token per l'identificazione nella sessione
LOGOUT	2	Si		(String) 'ok'	Effettua il logout
REGISTER	3	No	{email}|||||{*username*}|||||{*password*}	(String) {token} (successo) 'InUse' (username in uso) 'error' (altrimenti)	Registrazione nuovo utente, restituisce il token per l'identificazione nella sessione.
FSEARCH	4	Si	{filter}	(List<String >)	Restituisce una lista di utenti che contengono {filter} nel nome utente.
FLIST	5	Si		(List<String >)	Restituisce la lista degli amici
FREQESTSEND	6	Si	{username}	(String) 'ok' (se inoltrata) 'error' (altrimenti)	Invia una richiesta di amicizia, il server prova a consegnarla, se consegnata restituisce 'ok', 'error' altrimenti.
CONTENTP	7	Si	{testo}	(String) 'ok'	Invia un nuovo contenuto al server per la distribuzione agli utenti interessati.
FREQUESTREPLY	8	Si	{username}|||||{*'yes'/'no'*}	(String) 'ok' (successo) 'error' (altrimenti)	Risponde ad una richiesta di amicizia.
UPDATEPORT	255	Si	{portClient}	(String) 'ok'	Aggiorna la porta su cui è in ascolto il client per l'inoltro di richieste di amicizia.

Ad ogni richiesta il server provvede a verificare se il token è valido e se l'utente è risultato online nei precedenti 10 secondi, andando a consultare l'apposita lista che introdurremo nel prossimo paragrafo. Quando una richiesta va a buon fine il server aggiorna la lista degli utenti online corrente (anche questa verrà introdotta nel prossimo paragrafo).
I thread coinvolti sono: il *main thread* per il *ServerSocket* e un pool, come prima accennato, per la gestione dei singoli socket dei client (classe RequestHandler).

2.2 Keep-Alive

Il meccanismo di Keep-Alive serve per tenere traccia degli utenti online nella rete sociale. Come da richiesta, un utente che non comunica con il server per più di 10 secondi deve ricevere un nuovo token e quindi rieseguire il login. Per questo motivo il server mantiene due liste: una (in sola lettura) degli utenti che nello slot (di 10 secondi) sono risultati online ed hanno quindi comunicato con il server ed un'altra in modifica per raccogliere gli utenti che in questo slot di tempo hanno inviato qualcosa al server. Ogni 10 secondi il server copia la lista dello slot appena terminato in quello precedente e svuota quella dello slot corrente, invia poi un datagramma UDP in un gruppo multicast a cui sono iscritti i client, ogni client che riceve il datagramma di keep-alive dal server risponde con un altro datagramma UDP, questa volta unicast, al server contentente il proprio token.
Ogni token ricevuto va inserito (se non già presente) nella lista dello slot corrente di utenti online.
Le liste degli utenti online sono implementate con dei Vector in quanto *thread-safe*.
I thread per implementare questo meccanismo nel server sono due: uno che si sveglia ogni 10 secondi e invia il messaggio multicast (classe KeepAlive) ed uno che ascolta le risposte di KeepAlive alla porta 1500 UDP (classe KeepAliveReceiver). Nel client invece vi è un solo thread che è in ascolto sul gruppo multicast (classe KeepAlive).

2.3 Distribuzione di contenuti

Ad ogni contenuto pubblicato tramite il comando CONTENTP sul Main Server, la componente server invia tale contenuto a tutti gli utenti amici dell'autore che hanno manifestato interesse per i suoi contenuti.
L'espressione dell'interesse e la distribuzione dei contenuti (tramite callback) avviene tramite RMI.
In particolare ogni client dopo il login registra la propria callback, definita nell'interfaccia *ContentClientInterface*, per ricevere in modo asincrono i contenuti, chiamando il metodo *RegisterCallback* messo a disposizione dal server e definito nell'interfaccia *ContentRemoteInterface* che comprende anche il metodo *Follow* per esprimere l'interesse verso un amico.
Tutte le callback sono salvate in un Hashtable (*thread-safe*) associate al token del client che l'ha registrata. In fase di registrazione della callback il server invia tutti i contenuti che ancora non sono stati visualizzati al client. Successivamente per ogni contenuto pubblicato, il server consulta la lista degli utenti interessati a quell'utente e per ognuno di esse chiama la sua callback per inviargli il post.

Il server Registry necessario al funzionamento del sistema RMI è avviato direttamente via codice all'interno della componente Server.

2.4 Inoltro delle richieste di amicizia

Quando il server riceve da parte di un utente x la volontà di richiedere l'amicizia ad un altro utente u, esso prova ad aprire una connessione TCP con u sulla porta trasmessa da u dopo il login attraverso il comando UPDATEPORT sul Main Server, per comunicare che un utente x gli ha inviato una richiesta di amicizia. Infatti ogni client avvia un thread (classe FRequestServer) in attesa di richieste di amicizia inoltrategli dal server.
Se il server riesce ad inoltrare la richiesta di amcizia ad u, la salva (per verificare poi i riscontri) e risponde poi ad x che tale richiesta è andata a buon fine.
L'utente u può in seguito accettare o negare richieste di amicizia inviando un riscontro per ognuna di esse tramite il comando FREQESTREPLY al Main Server, che procederà a verificare se c'è una richiesta di amicizia pendente tra l'utente x e u ed aggiornare la rete sociale.

3 Interfaccia al database

È stato deciso di usare un database sqlite per lo storage delle informazioni in quanto i dati sono gestiti dal motore ed è facile poter fare query (anche relazionali tra più tabelle) e poterle rileggere tramite SQL. Inoltre è risultato più facile gestire la concorrenza in primis per la presenza di transazioni e del lock del sistema operativo in scrittura sul file. Tuttavia per non incorrere in eccezioni causate dal lock del database è stato implementato un metodo per effettuare tutte le scritture sul database syncronized.
Per le letture invece non vi è alcun problema in caso di concorrenza e questo ha contribuito ad una maggior efficienza in fase di lettura.
Il database è stato creato seguendo i buoni principi di progettazione del corso di database e si consiglia la visione della in primis della struttura per comprendere le query e del contenuto in fase di test per verificare il buon esito delle funzionalità che vi interagiscono.

4 Test

I principali test sono stati dinamici tramite il debugger dell'ide. Sono stati svolti anche test su LAN locale tra più macchine OSX e Windows, fatta eccezione per le funzionalità che usano Multicast.
I pacchetti .jar generati necessitano di essere eseguiti sulla stessa macchina.

5 Avvio

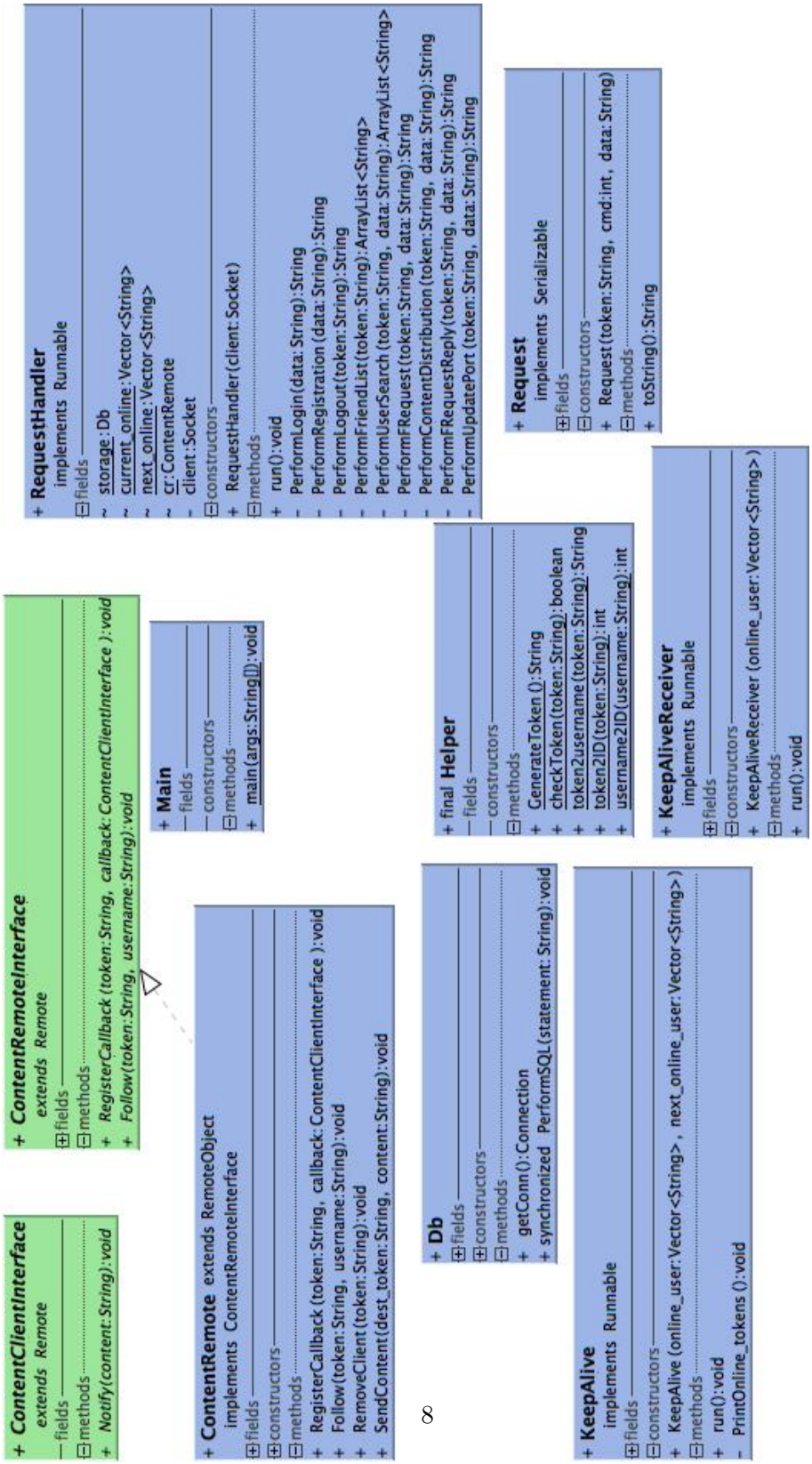
Passi per avviare il sistema (sullo stesso calcolatore) tramite Artifacts JAR:

1. Verificare che nella cartella SocialServer vi sia il database *database.sqlite3*
2. Avviare il server tramite il pacchetto *SocialServer.jar* (da console per vederne l'interazione)
3. Avviare i client tramite il pacchetto *SocialClient.jar*

6 Conclusioni

Il progetto è stato sviluppato grazie agli IDE IntelliJ Idea e DataGrip di JetBrains grazie al quale è stato possibile creare l'interfaccia grafica ed il database in maniera efficiente e veloce, gran parte del tempo impiegato è stato dedicato all'organizzazione del codice e allo sviluppo del protocollo del Main Server.
Anche lo sviluppo dell'interfaccia grafica è stato motivo di piacere e approfondimento. Questa è stata per me una esperienza bella e di eccellente valore formativo, in quanto la rete è destinata ad essere sempre più parte della vita quotidiana di ognuno e la sua conoscenza è necessaria per un futuro nel campo dell'informatica.
Per ulteriori delucidazioni sulle scelte progettuali e implementative si rimanda al codice e ai relativi commenti.

Server classes diagram



Client classes diagram