Progetto RCL 17/18

Social Gossip, un servizio di chat multifunzione.

Codice e relazione a cura di Crea Giuseppe, matricola 501922

Introduzione alla struttura del progetto

Introduzione

Il progetto consiste in due moduli, Client e Server, che fanno entrambi affidamento sul modulo condiviso Communications, contente moduli di supporto alle funzionalità di entrambi i moduli.

La comunicazione tcp fra Client e Server avviene via la classe Message, che codifica ogni operazione in un JSON con due campi, “OP_CODE” (codice operazione) e “DATA”. Questi messaggi saranno inviati dal Client al Server attraverso una connessione di controllo che resterà aperta fino alla disconnessione.
I messaggi inviati su questa connessione di controllo verranno parsati da un thread ClientInstance nel Server. Ogni thread ClientInstance è istanziato dal server quando riceve una connessione sul suo ServerSocket.

Messaggi amici-amici

Quando un utente cerca di creare una chat 1-a-1 fra amici esso invia un messaggio con codice operazione “OP_MSG_FRD” e data il nome dell’utente col quale vuole comunicare al server.
Dopo aver controllato che i due siano effettivamente amici, il server avvia un thread MessageHandler che si occupa di controllare che i due utenti siano entrambi loggati allo stesso momento, ed in caso positivo invia un messaggio di handshake ai ServerSocket dei due client, aprendo 2 socket con loro. Questi socket saranno poi passati a due MessageRoutingThreads, uno usato per indirizzare i messaggi ricevuti da socket uno a socket due, un altro per fare l’opposto.
I due thread continueranno a girare finché uno dei client non si disconnette, o forzatamente o tramite una handshake finale, inviando un messaggio con operazione END_CHT e data il nome utente col quale si era in comunicazione. END_CHT sarà poi rigirato dal server al secondo utente, che parsandolo nella sua ChatInstance saprà di dover chiudere il thread correlato e bloccare la tab. In caso di chiusura forzata il metodo di logout dell’oggetto utente sul server si assicura di chiudere questi thread settando a false la variabile di controllo User.listOfConnections. Una safeguard contro il mancato invio di un END_CHT si ha nei client sotto forma di un blocco del thread e della tab associata via callback RMI al logout di un amico.

Gruppi e Messaggi Multicast

La creazione, aggiunta, rimozione e distruzione di gruppi multicast è richiesta interamente dai client tramite le corrispettive operazioni OP_CRT_GRP, OP_JON_GRP, OP_LEV_GRP ed OP_DEL_GRP. Inoltre, si avrà OP_GET_GRP per ottenere la lista di gruppi multicast attivi sul server. Il client si aspetta una risposta dal server per ognuna di queste operazioni.

Alla ricezione di un'operazione di OP_CRT_GRP il server creerà un oggetto wrapper chiamando il metodo createChatGroup sull'oggetto User correlato all'utente che lo ha richiesto, con id gruppo la stringa contenuta nel campo data del messaggio. Questo metodo controlla se un gruppo dallo stesso id esista già, e se no, lo crea, usando come porte di ingresso ed uscita le prime due porte udp libere e lanciando un nuovo thread ChatroomUDP e salvando un ThreadWrapper con tutte le info necessarie ad accedere a quella chatroom nella map statica creata in Server.Core. Dopodiché eseguirà il join su quel server, chiamando il metodo addUser() del ThreadWrapper in questione. Inoltre, aggiungerà il nome del gruppo ad una lista, transiente, di chatroom id al quale l'utente è iscritto.
Ad ogni disconnessione, l'utente si disiscrive da tutte le chatroom.
Il procedimento di aggiunta e rimozione da un gruppo è equivalente, basandosi sulla ConcurrentHashMap di ThreadWrapper.
Per decisione personale, l'unico utente autorizzato a cancellare un gruppo è il suo creatore. Il server può comunque forzare questa limitazione.
I gruppi NON vengono salvati e ripristinati fra sessioni.

Traduzione Messaggi

La traduzione messaggi avviene completamente dal lato Server, se e solo se i due utenti per i quali è stata richiesta una chat con traduzione sono registrati con lingue diverse.
In caso positivo, ed in caso la chat con traduzione sia stata richiesta, il server farà passare, nel MessageRoutingThread, i messaggi per la classe statica TranslationEngine.

Trasferimento file NIO

Il trasferimento file avviene in maniera quasi completamente trasparente al server. Come da specifica, è peer to peer fra i client stessi. Un utente che vuole inviare un suo file ad un altro utente contatta il server, richiedendo l'indirizzo di quell'utente e la porta sulla quale il ServerSocketChannel di quell'utente è in ascolto. A questo punto l'utente che sta cercando di inviare il file prova a connettersi a quella coppia indirizzo/porta, comunicando prima la dimensione del suo nome utente, del nome del file che vuole inviare e la dimensione del file stesso. Una volta ricevuti questi 3 dati via ByteBuffers di Integer e Long il ricevente può finalmente ricevere i dati veri e propri sul nome utente e nome file, chiedendo all'utente in ricezione via finestra di dialogo se e dove vuole che il file venga salvato.
L'utente in ricezione può accettare o bloccare il trasferimento file, mandando un'appropriata risposta al mittente via un altro ByteBuffer, passando in modalità di WRITE o chiudendo il channel e cancellando la chiave, rispettivamente alla scelta. Letta una risposta affermativa dal ricevente, il mittente finalmente trasferisce il file ed informa l'utente via finestra popup al completamento.

Database

Il database utenti è una semplice serializzazione della ConcurrentHashMap di String, User che il server usa internamente. Per assicurare l'aggiornamento continuo si elimina il precedente file database ogni volta che si riserializza.
Per estrarre il database ho la classe apposita Server.Deserializer, mentre per salvarlo in continuazione ho il Runnable Server.SavestateDeamon.

Interfaccia Grafica

L'interfaccia grafica è costruita col framework javafx facendo ampio uso di file fxml associati ad una classe controller. Chiamando metodi sulla classe controller si possono cambiare i componenti caricati dal file fxml. Javafx richiede che ogni modifica dell'interfaccia grafica sia eseguita dal thread principale dell'UI. Se è necessario modificare l'interfaccia da un thread che non sia il principale, occorre chiamare il metodo statico

di javafx Platform.runlater(), usando come argomento un oggetto runnable che eseguirà le modifiche richieste. Questo uso dei runnable è presente dappertutto nel lato Client del progetto.

How to Use

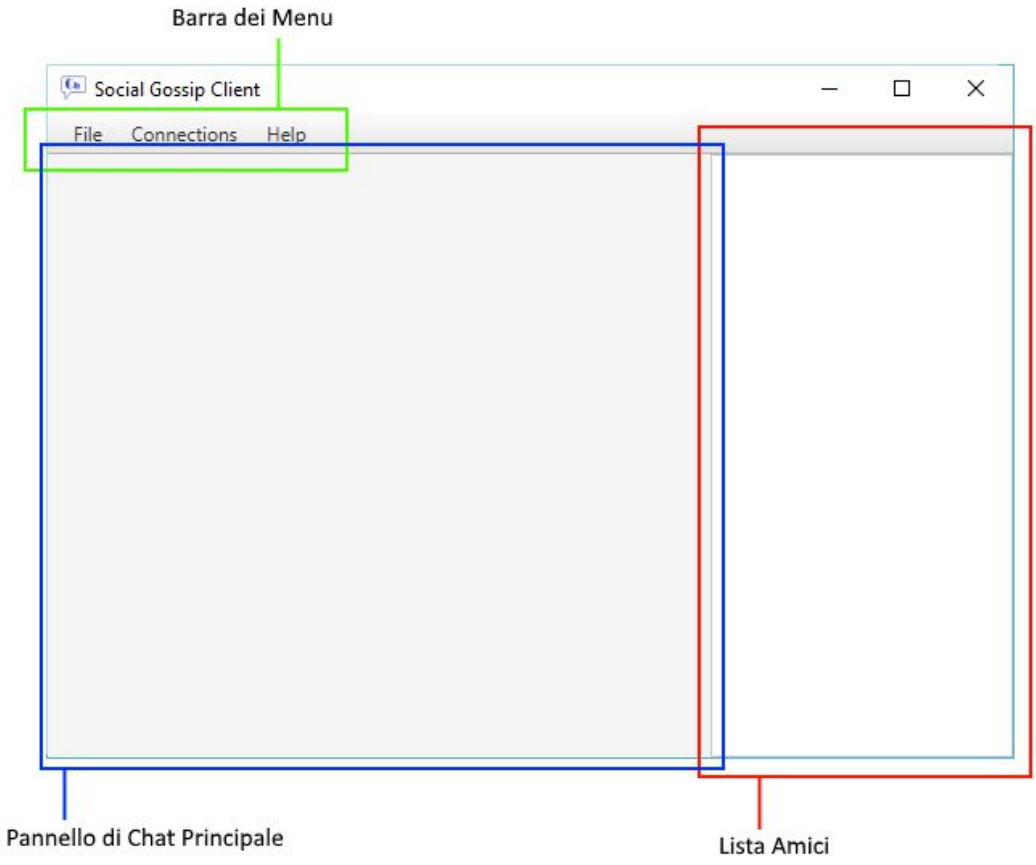
Server

Il server viene fornito come singolo file .jar che deve essere avviato da linea di comando usando **java 1.8** via comando:

```
java -jar [path]\Server.jar
```

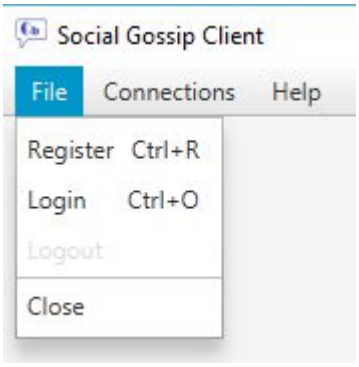
Al primo avvio il server creerà un file server.properties tramite il quale si potranno configurare parametri quali la porta sulla quale avviare il server in ascolto, la porta sulla quale avviare il registro RMI, e l'IP delle macchine server nella rete, importante se si vuole che il server sia acceduto da altre macchine. Un breve commento è sopra ogni opzione, a spiegazione della loro funzione.
Una volta avviato, scrivere "exit" e premere invio nella linea di comando sarà abbastanza per spegnere correttamente il server.

Client

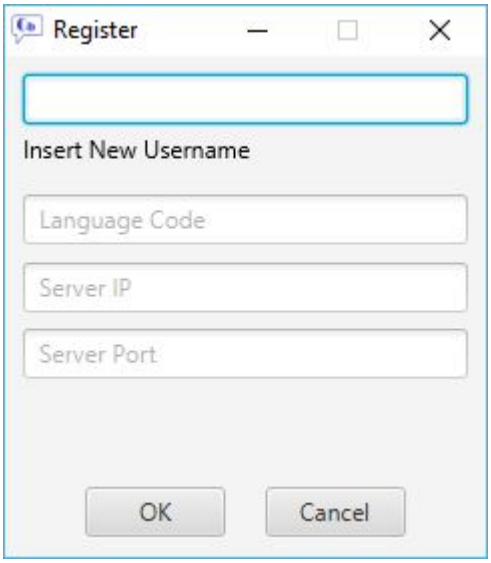


Il client viene fornito come file .jar avviabile per doppio click. Alla sua apertura ci si trova davanti un'interfaccia composta da due pannelli, uno grigio ed uno bianco. Il pannello grigio sarà il

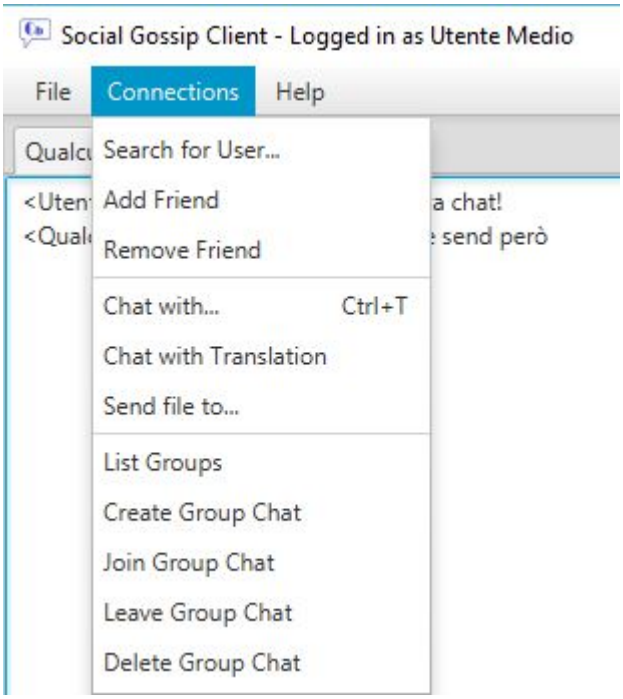
pannello di chat principale, nel quale tutte le conversazioni avranno luogo. Il pannello bianco sarà, all'atto del login, popolato con la lista amici di questo utente.



Prima che l'utente abbia eseguito il login o la registrazione tutte le opzioni della barra dei menu, tranne File -> Register, File -> Login, File -> Close ed Help -> About, saranno bloccate, a lui inusabili.



Register e Login offrono due finestre comparativamente simile, la cui unica differenza è la presenza o assenza del campo di input per la lingua utente. A destra possiamo vedere la schermata di registrazione utente. Lasciando i campi bianchi il client cercherà di connettersi a localhost sulla porta (scelta a caso da me come porta di default fra quelle publiche effimere) 62453. Il primo campo, dedicato al nome utente, deve essere sempre compilato e se lasciato vuoto bloccherà il client dal mandare la richiesta di registrazione al server. Il campo Language Code deve seguire lo standard ISO 639-1, il cui corretto utilizzo verrà controllato direttamente dal client. Nel caso il codice inserito NON rispecchi lo standard richiesto l'utente sarà registrato con linguaggio quello del sistema sul quale il client è in esecuzione, e così anche in caso il campo sia lasciato bianco.

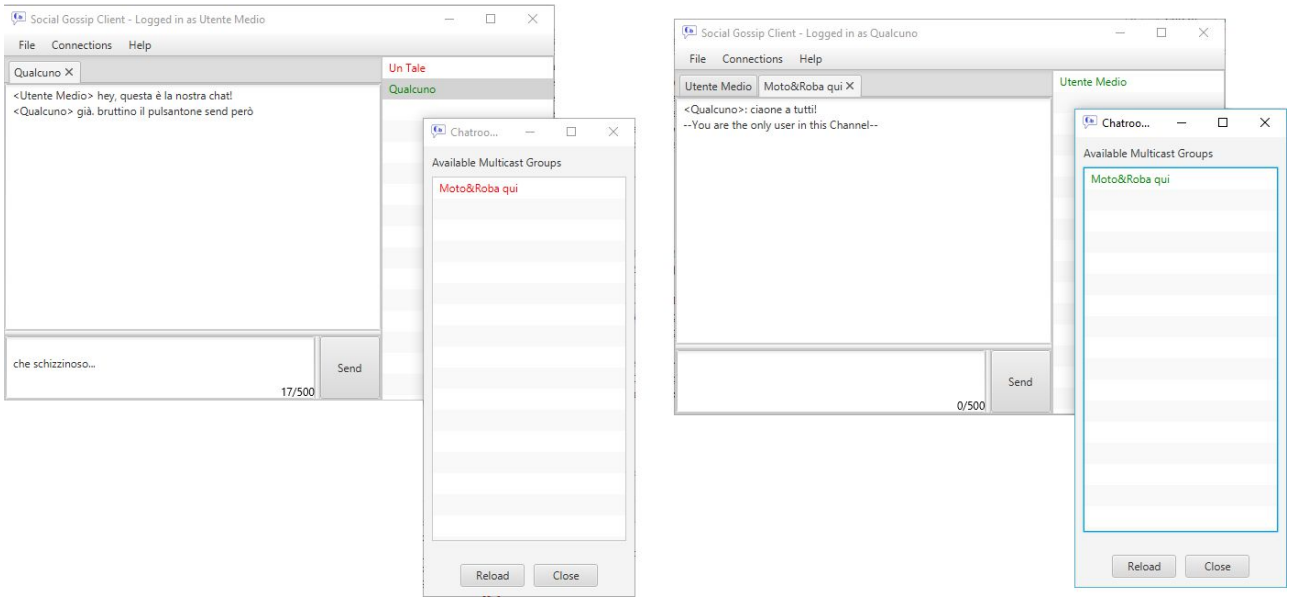


Dopo aver effettuato il login o la registrazione le relative opzioni saranno disabilitate e la lista amici sarà scaricata dal server. Allo stesso tempo le voci del menu Connections saranno abilitate, permettendo ora di richiedere servizi al server.
I sistemi di ricerca utenti ed aggiunta amici mostrano una finestra di dialogo ognuno, con uno spazio per inserire il nome dell'utente a noi interessato. La finestra di ricerca utente, inoltre, ci permette di aggiungere direttamente un utente agli amici se egli esiste. In un certo senso è la versione più potente della semplice finestra di aggiunta amici.
La finestra di rimozione amici funziona in modo analogo a quella di aggiunta, chiedendo all'utente il nome dell'amico da rimuovere.

La finestra di Chat with… permette d'iniziare una chat con un utente nella nostra lista amici, in modo esattamente analogo al semplice click del mouse sul nome di quell'utente nella lista amici a destra dell'applicazione. La finestra di Chat with Translation si occupa di richiedere al server quel particolare servizio, ma funziona esattamente come la finestra precedente altrimenti.
La finestra di invio file ci chiede di individuare, via file chooser, il file che vogliamo inviare dal nostro client, e di inserire il nome dell'utente (nostro amico) al quale vogliamo inviare tale file. L'utente in questione riceverà una notifica col nome del file e il nostro nome utente, chiedendo se e dove vuole salvare il file

prima che il trasferimento vero e proprio inizi. Alla fine del trasferimento entrambi gli utenti saranno notificati.

L'ultimo insieme di elementi del menu riguardano i gruppi multicast. List Groups aprirà una nuova finestra, nella quale una lista dei vari gruppi multicast attualmente attivi sul server sarà mostrata. I gruppi ai quali l'utente è iscritto (quelli per i quali ha una chat tab attiva in questo momento) saranno colorati in verde, quelli a cui non è iscritto in rosso. Inoltre, cliccando su un gruppo rosso manderà una richiesta di join per quel gruppo, ed una relativa tab verrà aperta sul suo client.



Sopra, un esempio di chat di gruppo, chiamata "Moto&Roba qui", alla quale l'utente Qualcuno è iscritto ma l'utente Utente Medio no.

Create, Join, Leave e Delete Group Chat si comportano tutti allo stesso modo, richiedendo l'utente inserisca il nome della chat sulla quale vuole eseguire l'appropriata operazione.
Importante notare che l'operazione di Delete può essere eseguita solo dall'utente che originariamente ha creato quella chat room. Inoltre, chiudere la tab di una chatroom equivale ad eseguire il Leave da quella chat room, proprio come chiudere una tab di chat fra amici interrompe la chat.

Quando l'utente esegue il logout o chiude il client in qualsivoglia modo (crash esclusi, quelli sono gestiti dal server via un heartbeat timer) tutte le connessioni fra amici aperte verranno chiuse e tutte le iscrizioni ai gruppi di chat verranno cancellate.

Dopo un logout, un secondo utente può usare l'applicativo senza avere nessun effetto sull'utente precedente.

Panoramica Moduli

Client Module

Il modulo client si divide in altri 4 sotto-moduli e contiene al suo interno 6 classi.
Le classi qui presenti sono quelle che, a livello logico durante la prima fase di progettazione, risultano indispensabili al funzionamento del client stesso. Ne segue una breve descrizione.
Core: Classe contenente funzioni statiche di connessione e trasmissione dati fra client e server.
ChatInstance: Ogni chat fra amici diversa sarà un' istanza di questo Runnable, che ha il compito di parsare e mostrare i messaggi all'utente.

FriendchatsListener: Runnable singolo la cui istanza nasce a login e muore a logout, mettendosi in ascolto su di un socket predeterminato per possibili nuove chat o friend/unfriend requests in arrivo da utenti singoli.
FirstMessageListener: Classe di supporto a FriendchatsListener, si occupa di parsare il contenuto del messaggio di handshake fra utenti (inviato a noi dal server a richiesta di un altro utente).
Heartbeat: Un servizio di heartbeat per notificare il server della nostra continua presenza. Se vi è una IOException su quel thread sappiamo che il server è andato offline, e forziamo il client alla chiusura.
User: Classe statica che conterrà il nostro oggetto utente una volta effettuata la registrazione o il login con un server. Quasi tutte le altre classi del client faranno riferimento a questo oggetto per svolgere i propri compiti.

Client.NIO Module

Il primo sottomodulo, in ordine alfabetico, è quello del sistema di trasferimento file NIO. Vi appartengono solo 2 classi, FileReceiverServer che, come il nome suggerisce, si occupa di eseguire un processo continuo d'ascolto su di un apposito ServerSocketChannel, istanziato su di una porta scelta casualmente al boot dell'applicazione e comunicata al momento del login al server.
La classe si occupa di accettare nuove richieste NIO, leggere il nome del file e del mittente, e chiedere all'utente se vuole accettare il trasferimento. Ottenuto input dall'utente comunica, sempre via NIO, la risposta al mittente, che procede allora a trasferire il file oppure interrompere l'operazione. Questa classe inoltre chiede, per ogni trasferimento, dove l'utente voglia salvare i file.

Sender invece si occupa solamente di mandare un file già selezionato dall'utente ad un indirizzo e porta che gli sono già stati comunicati via SocketChannel e FileChannel, attendendo una risposta dall'interlocutore. Il file, l'indirizzo e la porta in questione gli sono stati comunicati dal suo chiamante, l'elemento UI Client.UI.NIOui.SenderController che si occupa di prendere il nome utente del destinatario via input, far scegliere all'utente un file dal suo sistema operativo ed inviare al Server un messaggio di richiesta trasferimento file, con codificato il nome dell'utente che si vuole raggiungere. Il Server si accerterà che i due utenti abbiano una relazione d'amicizia ed invierà una risposta al SenderController. Se la risposta è positiva, nei dati conterrà l'indirizzo dell'utente da raggiungere e la porta del FileReceiverServer di quell'utente, che poi saranno passate al processo Sender.

Client.RMI

Contenente l'interfaccia RMI UserCallback e la sua implementazione, UserCallbackImplementation. Interfaccia e classe contengono solo i metodi hasLoggedIn e hasLoggedOut per avvisare il client di chat che un utente dal nome dato è loggato in o out dal server. Questi metodi eseguono il Runnable Client.UI.ModifyFriendlistStatus, il cui unico compito è la modifica della lista amici.

Client.UDP

Dopo aver ricevuto le porte sulle quali un dato gruppo multicast è in esecuzione questo runnable si occupa di ricevere messaggi multicast UDP su di esse e pusharli nella tab appropriata. Il thread viene interrotto quando l'utente che esegue il logout o quando il richiede di essere rimosso dal gruppo multicast. Chiudere la tab conta come richiedere la propria rimozione dal gruppo. Al logout/chiusura client il client stesso richiede la rimozione dal gruppo. L'invio dei messaggi è invece eseguito dalla tab di chat stessa.

Client.UI

Questo è forse il modulo più corposo dell'intero progetto.
La classe CoreUI si occupa di istanziare l'oggetto User principale e le porte che verranno usate nei ServerSocket e ServerSocketChannel dei processi FriendchatsListener e NIO.FileReceiverServer. Inoltre, si

occupa di caricare il file clientGUI.fxml, file principale dell'interfaccia utente, e salvare un riferimento al suo controller, un oggetto della classe statica Controller. Questo riferimento sarà usato per ogni azione sull'UI.
La classe Controller mappa ogni possibile interazione con l'interfaccia utente principale, occupandosi di creare nuove finestre, avviare thread e aggiornare finestre attive. Qui sono anche istanziate le variabili di controllo per i thread di chat fra utenti.
ColoredText è un wrapper per la combo colore/testo degli oggetti coi quali costruirò la lista amici, mostrata nel lato destro dell'interfaccia principale come ListView con metodo di display personalizzato, rosso per amici offline, verde per amici online. Sarà anche usata per la lista dei gruppi multicast, con lo stesso principio di colorazione per i gruppi ai quali siamo o meno connessi.
FriendListUpdate e ModifyFriendlistStatus sono entrambi Runnable per aggiornare la lista amici da thread diversi dal thread principale dell'UI via il metodo Platform.runlater() di javafx.
Il modulo chatPane contiene le classi delle tab di chat ed il file fxml da caricare quando si aggiunge una nuova chat tab.
La funzione di ChatTabController merita di essere approfondita. Una tab di chat può essere creata in 2 modi diversi, a seconda del tipo. Le tab di chat TCP fra amici sono avviate da ChatInstance, sia che siano state richieste da me, che da un altro utente, mentre quelle UDP possono essere chiamate solo da una finestra chiamata dal controller stesso, poiché l'utente deve attivamente richiedere di parteciparvi. Il controller allora avrà una variabile privata per stabilire la modalità nella quale ogni tab sta girando, sia essa tcp o udp, ed i suoi metodi di invio e display messaggi, insieme ai metodi di chiusura tab, saranno diversi per le due. Si hanno nel modulo UI.chatPane altri runnable necessari ad operare sulle tab da thread non UI, come quello di creazione, quello di aggiornamento per scrivere un messaggio su una tab, ed infine quello di lock, per bloccare l'area di scrittura ed il pulsante di invio in una connessione chiusa dal peer/server.
PopupWindows sono una collezione di file fxml e relativi classi controller usate quasi esclusivamente per piccole, veloci finestre popup necessarie a raccogliere informazioni dall'utente.
I controller delle finestre di registrazione e login, rispettivamente RegisterController e LoginController meritano un momento d'approfondimento. Essi controllano i vari campi di testo messi da loro a disposizione e comunicano col server per effettuare la registrazione o il login utente tramite i metodi presenti in Client.Core, comunicando all'utente eventuali errori. Inoltre, riempiono i campi della variabile statica CoreUI.myUser e aggiornano l'interfaccia grafica principale.
Un'altra finestra leggermente diversa dalle altre è la multicastGroupListWindow, che mostra i gruppi multicast attivi sul server, differenziando in verde quelli ai quali l'utente è iscritto ed in rosso quelli ai quali non lo è. Inoltre i Runnable Alerts, BigErrorAlert e Warning servono come supporto per mostrare velocemente messaggi di avviso all'utente da thread non-UI.
Infine una piccola menzione alle classi del modulo Client.UI.NIOui, SenderController di cui abbiamo già parlato e ReceiveConfirmation, che si attiverà solo quando username di un mittente e filename del file che esso ci vuole inviare sono stati ottenuti dall'istanza di Client.NIO.FileReceiverServer, chiedendo all'utente se vuole accettare quel file e dove vorrà salvarlo.

Questo conclude la panoramica sulle classi ed i moduli del Client.

Communication Module

Il modulo Communication contiene classi che, durante la fase di progettazione iniziale del progetto, avevo ritenuto potessero tornare utili sia al Client che al Server, o comunque classi che possono essere usate in maniera intercambiabile senza bisogno di modifiche da entrambi i moduli.
Delle 3 classi qui presenti in realtà solamente la classe Message viene condivisa, mentre le classi GetProperties e IsoUtil sono usate esclusivamente da Server e Client, rispettivamente.
La classe IsoUtil è la più piccola delle 3, fornendo solo un supporto per controllare che un linguaggio inserito alla registrazione dall'utente rispetti le norme ISO 639-1.
La classe GetProperties si occupa di aprire il file server.properties e leggerne le proprietà richieste.

Infine la classe Message, decisamente la più complessa del pacchetto, costituisce il cuore dei messaggi tcp che saranno scambiati fra Client e Server. La logica di questa classe gira attorno al serializzare messaggi in 2 campi, operation e data, uno dedicato a scambiare codici operazione e l'altro a fornire dati riguardanti tali operazioni al ricevente. Questi campi sono salvati come stringhe e serializzati in un unico oggetto JSON come da specifiche, per poi essere inviati tramite un BufferedWriter sul socket designato. Al termine della scrittura di un dato JSON si aggiunge una newLine, permettendo così al metodo receive di questa classe, chiamato parallelamente dall'interlocutore, di sapere dove fermarsi. Il metodo receive parserà il JSON e ne metterà i valori nei campi operation e data dell'oggetto Message sul quale è stato chiamato. Message presenta 2 costruttori, uno vuoto per oggetti che si intende usare in ricezione, mentre uno per oggetti che si vuole usare in spedizione che permette di settare già al momento della creazione i campi operation e data, nonché di creare l'oggetto JSON associato. Inoltre, si ha il metodo setFields, in caso si voglia riutilizzare lo stesso oggetto Message per più comunicazioni tcp.

Server Module

Non dotato di interfaccia grafica, il server stesso si presenta molto più semplice da comprendere che il client.
Core: è la classe principale del server, quella che sarà avviata dal jar. Si occupa di leggere le proprietà, istanziare il database utenti, il registro RMI, il thread di accettazione di nuove connessioni e di ascoltare input utente per un comando di `exit`.
ServerAcceptThread: Attende nuove connessioni sul ServerSocket tcp passatogli dal Core, ed avvia un Runnable ClientInstance per ogni nuova connessione.
ClientInstance: Attende che l'utente esegua la registrazione o il login. In caso di errore in questa prima fase tuttavia la clientinstance si chiuderà, poiché il client richiede una nuova connessione col server, troncando il socket precedente. Una volta eseguito il login questo Runnable rimarrà in un loop di ascolto e risposta col client per messaggi di controllo, associati a richieste di operazioni da parte del client stesso, ognuna correlata da un diverso OP_CODE. I messaggi di heartbeat saranno inviati assieme a questi altri messaggi di controllo, e non saranno corrisposti da un ack di alcun tipo. La loro unica funzione sarà decrementare un timer che, superati i 4 secondi, assumerebbe l'utente come disconnesso e procederebbe a chiudere il thread. Una volta ricevuta una richiesta di logout questo thread si occuperà di pulire le connessioni attive che l'utente a lui associato ha lasciato in sospeso, cosa che dovrebbe succedere solamente in caso di crash.
MessageHandler e MessageRoutingThread: Sono due classi che estendono Thread, usate per la comunicazione fra amici. Dato che da specifica la comunicazione fra amici deve avvenire su un socket diverso da quella fra client e server, ma allo stesso tempo tutti i messaggi diretti ad un amico devono passare per il server, il thread MessageHandler si occuperà di creare 2 diversi MessageRoutingThread, uno per inviare tutto ciò che utente A scrive ad utente B, uno per inviare tutto ciò che utente B scrive ad utente A, richiedendo di connettersi al serversocket inizializzato da FriendchatsListener per i due utenti. Gli utenti a loro volta manderanno e riceveranno messaggi su questo nuovo socket aperto col server. Tutto questo risulta molto comodo quando voglio aggiungere il servizio REST di traduzione, facendo semplicemente passare i messaggi gestiti dai MessageRoutingThread(s) per la mia classe statica TranslationEngine.
ChatConnectionWrapper: Classe che uso in congiunzione ad una ConcurrentHashMap nei miei oggetti User per controllare il ciclo while delle chat amici. Settando il valore del boolean all'interno a false posso chiudere un MessageRoutingThread il cui ciclo while fa riferimento a quel boolean, passando dall'oggetto User al quale esso è associato.

Server.UDP

Similmente a ChatConnectionWrapper, qui ho la classe ThreadWrapper che incapsulerà tutte le variabili ed i metodi necessari a controllare l'esecuzione dei thread ChatroomUDP, tramite un public static ConcurrentHashMap che associa un id (String) ad ogni nuovo ThreadWrapper direttamente nella classe

Server.Core.
Il Runnable ChatroomUDP stesso ha un funzionamento semplice, facendo il multicast dei datagrammi utente ricevuti da chiunque sia connesso al DatagramSocket a lui associato, aggiungendo un avviso se questi è l'unico utente online in quella chatroom. Quando grazie al wrapper chiamerò il metodo close del DatagramSocket a lui associato il thread terminerà. Non uso una variabile di controllo associata al ciclo while interno del thread poiché la receive è bloccante.

Server.RMI

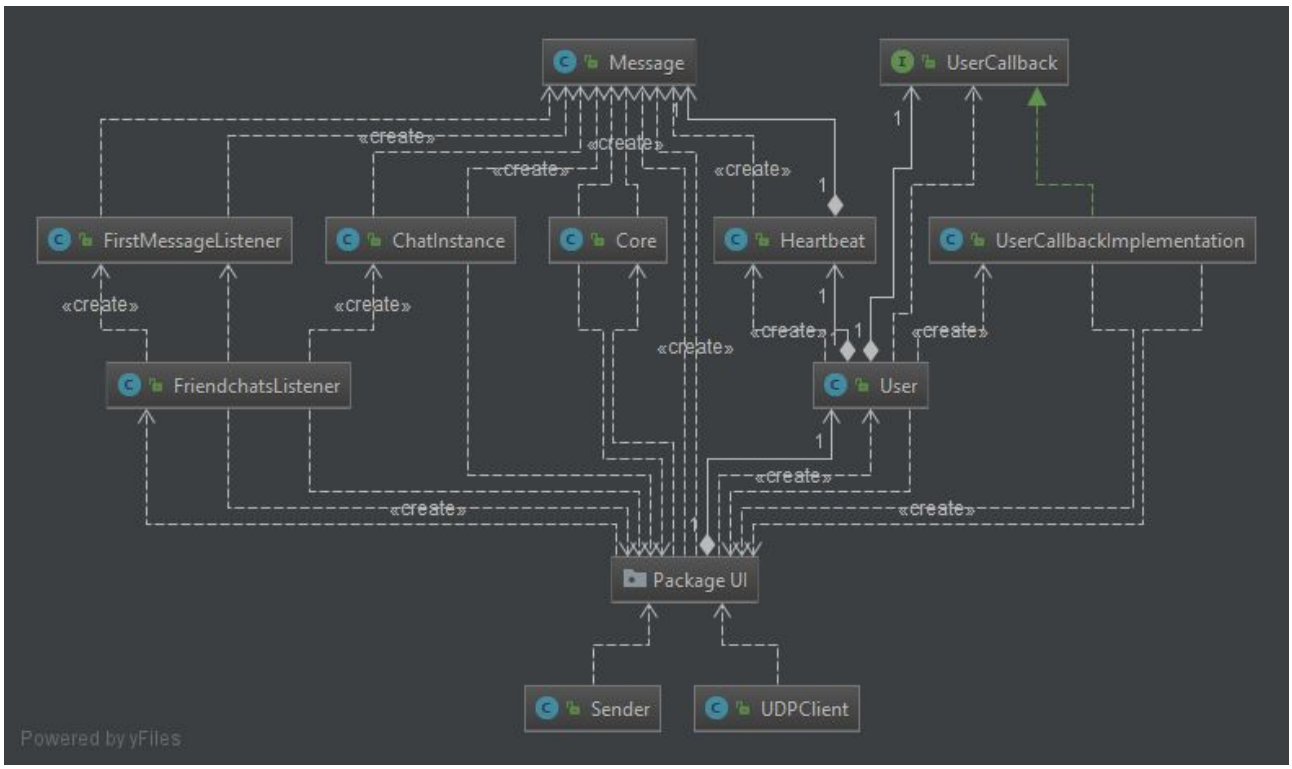
L'implementazione dell'interfaccia di RMI è piuttosto semplice, attivandosi solo a login/logout di utenti, o quando un utente invoca il metodo getFriendOnlineStatus dal suo oggetto User, per aggiornare lo status di un particolare amico nella sua lista.

User

La classe Server.User costituisce il cuore del database del mio server. Gli oggetti di questa classe saranno periodicamente serializzati dal SavestateDeamon e saranno ripristinati fra reboot/crash del server.
Qui il nome, la lingua, lo status online, la lista amici, l'indirizzo, le porte del FriendchatsListener e FileReceiverServer e tutte le connessioni aperte sono salvate, alcune in modo transient per non essere poi ripristinate dalle varie serializzazioni (dopotutto l'indirizzo e la porta possono cambiare ad ogni relog).
Oltre a fornire metodi per modificare lo status stesso di quest'oggetto, l'oggetto User offre metodi per operare sui gruppi UDP, e per rimuovere eventuali callback RMI lasciate da un crash del client a lui associato.

Concorrenza

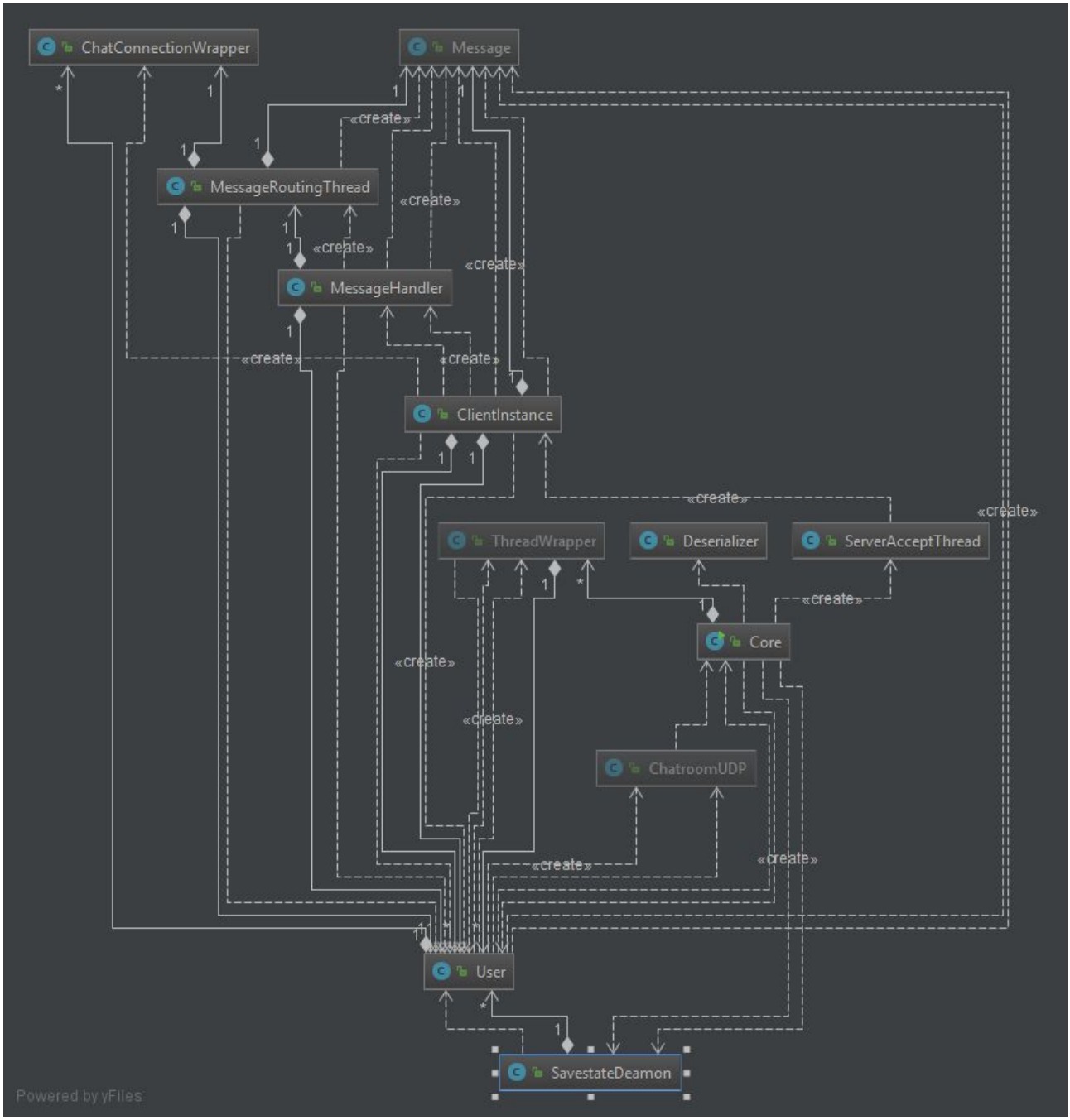
Client



La sincronizzazione fra thread è mantenuta principalmente dall'uso di strutture dati Concurrent.
Il client utilizza un thread principale per l'interfaccia utente, sempre attivo, sul quale le varie chiamate di

Platform.runlater(<Runnable>) gireranno.
Effettuato il login o la registrazione tre nuovi thread verranno avviati, uno istanza della classe Heartbeat, uno della classe FriendchatsListener ed un ultimo istanza della classe FileReceiverServer.
FriendchatsListener potrà istanziare 2 runnable diversi, FriendListUpdate e FirstMessageListener, ma solo quest'ultimo darà luogo ad un nuovo thread, mentre il primo sarà eseguito sul thread UI.
ChatInstance eseguirà ogni Runnable da lui istanziato sul thread UI stesso, senza creare ulteriori thread. Accederà in modo statico alla CopyOnWriteArrayList allActiveChats presente in CoreUI.controller.
FileReceiverServer istanzia una FutureTask per ottenere il consenso dall'utente a ricevere il file, ma anche questa sarà eseguita sul thread UI.
Heartbeat non istanzierà nessun nuovo runnable, né farà riferimento a variabili condivise.
Tutti i thread sopra elencati, con l'eccezione del thread UI, vengono stoppati al logout, sia esso via l'opzione di menu o via la chiusura del client. Le varie istanze di ChatInstance vengono fermate dal loro stesso processo logico, quando il metodo onClose della tab UI corrispondente viene invocato nel processo di logout tramite clearChatPane().
Il thread UI è capace di avviare due altri tipi di thread, uno istanza di NIO.Sender, un altro istanza di UDP.UDPClient, entrambi su richiesta dell'utente.
I thread del primo tipo istanziano solo runnable che andranno ad essere eseguiti sul thread UI, e terminano automaticamente sia che il trasferimento sia andato a buon fine, sia che esso sia stato rifiutato, sia che la connessione si sia interrotta.
I thread istanza di UDPClient faranno riferimento alla variabile static openGroupChats per controllare il proprio flusso logico ed il runnable UpdateTab per aggiornare la tab di chat a loro corrispondente. A sua volta il runnable UpdateTab (eseguito sul thread principale dell'UI) invocherà i metodi statici writeToChatTab, writeToUdpChatTab e lockChatTabWrites di UI.Controller per raggiungere i controller grafici delle chat tab aperte e poter finalmente scrivere a schermo. Il Runnable UpdateTab è usato in modo analogo in thread ChatInstance. UDPClient termina quando riceve il messaggio di goodbye appropriato, generato dalla funzione onClose della sua stessa tab (o dal comando di leave/delete per chat tabs nel menu del client stesso).

Server



La classe principale del pacchetto Server, Core, istanzia 2 thread all'avvio, rispettivamente istanze di SavestateDeamon e ServerAcceptThread, ed un ThreadPool per le istanze di ClientInstance che saranno inizializzate da ServerAcceptThread. Inoltre Server.Core possiederà alcune strutture dati static, che userò ai fini della sincronizzazione fra thread, soprattutto fra UDP.ChatroomUDP e le istanze di ClientInstance. Tramite due diverse ConcurrentHashMap, una composta da chiavi String e valori Boolean, una da chiavi String e valori ThreadWrapper, controllerò l'esecuzione logica e lo shutdown corretto dei vari thread ChatroomUDP inizializzati dalle ClientInstance, avendone un riferimento in scoping globale.
Inoltre Core istanzierà la variabile myDatabase, una ConcurrentHashMap di chiave String e value User che sarà poi passata al ServerAcceptThread ed alle ClientInstance per funzionare come database unico di tutti gli utenti. Ogni modifica apportata agli oggetti utente si applica a questo database, ed i metodi della classe User nel pacchetto Server lo rispecchiano, usando strutture dati abilitate alla concorrenza e le keyword synchronized quando necessario, come nel caso del get e set dello stato online di un dato utente.

Passando ai thread generati da Core, SavestateDeamon fa uso della variabile myDatabase di cui sopra, ma in sola lettura. Non istanzia altri thread e termina quando comandato da Core.
ServerAcceptThread istanzierà oggetti della classe ClientInstance, aggiungendoli al thread pool creato in precedenza da Core. Termina quando il ServerSocket sul quale è in ascolto viene chiuso dal thread di Core.
ClientInstance istanzierà due tipi di thread, istanze di MessageHandler, alle quali associerà un oggetto di controllo ChatConnectionWrapper nella ConcurrentHashMap listOfConnections degli oggetti User relativi ai due utenti che stanno iniziando una connessione fra loro, ed istanze di ChatroomUDP tramite un metodo dell'oggetto User che sta trattando. User (sempre sul thread della ClientInstance) farà poi riferimento alla static ConcurrentHashMap chatroomsUDPWrapper in Core per le informazioni aggiuntive necessarie a trattare il thread ChatroomUDP e il suo wrapper di controllo.
ClientInstance termina alla disconnessione utente (per richiesta o per timeout) o al chiamarsi del System.exit in Core.
Le istanze di MessageHandler avviano due diverse istanze di MessageRoutingThread l'una, e termineranno al chiudersi di queste ultime dopo aver propagato il messaggio di fine chat da un client all'altro, usando la listOfConnections interna agli oggetti utente di cui sopra per controllare lo stato della chat.
NB: listOfConnections non è settata né in MessageRoutingThread né in MessageHandler, ma bensì nel ClientInstance associato all'utente che richiede la chiusura della connessione, quest'implementazione è stata scelta per mantenere separati i flussi logici di messaggi chat e messaggi controllo.
I MessageRoutingThread terminano quando gli utenti chiedono la chiusura della chat al thread ClientInstance loro associato, cambiando il valore di verità del chatroomsUDPWrapper associato a quella chat nella hashmap listOfConnections di ogni utente. La richiesta di chiusura chat da parte di un utente setta il wrapper di entrambi i membri di quella connessione a falso.
MessageRoutingThread inoltre termina se uno dei due utenti a se associato va offline.
ChatroomUDP fa riferimento al chatroomsUDPcontrolArray inizializzato in Core per controllare la sua esecuzione, e non crea nuovi thread.