Social Gossip

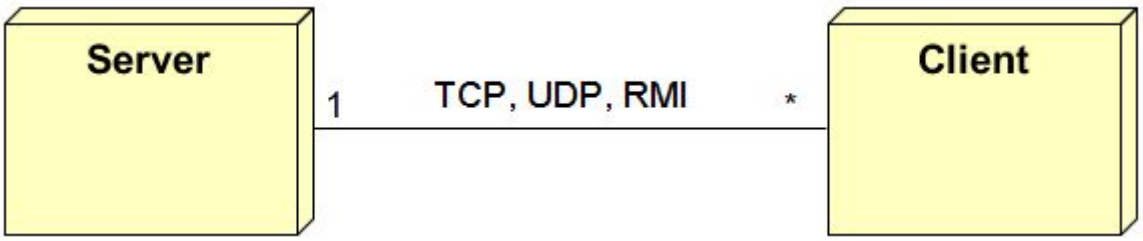
Motoc Stefan Daniel,
Tenucci Gabriele

1. Overview

Essendo Social Gossip un'applicazione basata su una struttura di tipo client-server, abbiamo deciso di suddividere il progetto in tre diversi package:

- Server: contenente le strutture dati e le classi del server
- Client: contenente le strutture dati e le classi del client
- Common: contenente le strutture dati comuni a client e server

Un'unica istanza del Server può gestire le richieste di più Client, con i quali comunica utilizzando connessioni TCP, UDP e RMI.



Lo scambio di messaggi tra singoli utenti avviene attraverso il protocollo TCP, mentre i messaggi all'interno dei gruppi vengono inviati tramite multicast UDP.
Per quanto riguarda RMI, esso viene utilizzato dai client per effettuare login/logout e ricevere notifiche riguardanti l'esito delle richieste effettuate al server e il cambio di stato degli utenti presenti nella propria lista amici.

Per facilitare la gestione dei messaggi ed elaborare le varie tipologie di richieste, abbiamo deciso di implementare una classe apposita, Message, così da agevolare le operazioni di conversione dal/al formato JSON, per implementare il quale ci siamo serviti della libreria esterna JSONSimple.

2. Classi

Le classi definite all'interno del progetto sono le seguenti. Per una visione più dettagliata delle classi e dei singoli metodi che esse contengono, viene fornita la documentazione Javadoc.

2.1. Client

ClientMain

- Avvia l'interfaccia grafica ed inizializza le strutture base del client.

ClientGui

- Gestisce l'interfaccia grafica del client.

SocialClientInterface

- Interfaccia per SocialClient.

SocialClient

- Crea ed inizializza tutte le strutture dati del client.
- Fornisce metodi di accesso e modifica dell'utente.
- Gestisce le operazioni richieste dall'utente tramite la GUI.
- Fornisce metodi RMI utilizzabili dal server.
- Avvia i thread necessari per l'invio e la ricezioni di messaggi e file.
- Gestisce la sincronizzazione tra i vari thread implementando metodi synchronized e lock sulle strutture dati, oltre che utilizzare strutture dati thread-safe (ConcurrentHashMap).

NIOListener

- Gestisce l'invio di file agli utenti che si collegano tramite NIO, utilizzando dei reader per la lettura dei file e dei writer per l'inoltro di essi sulle socket dei destinatari.

TCPListener

- Gestisce le connessioni TCP in arrivo dal server, oltre che quelle in arrivo dai client che vogliono spedire file.
    - In caso di ricezione file, invoca una funzione receiveFile() eseguita da un thread appositamente avviato, creando una connessione TCP con il client che ha richiesto l'invio del file.

UDPListener

- Gestisce i messaggi UDP ricevuti attraverso il gruppo multicast a cui è registrato il client.

2.2. Common

Translate

- Utilizza servizio offerto da API REST MyMemoryTranslated.net per effetuare la traduzione dei messaggi.

Message

- Classe utilizzata per lo scambio di messaggi.
- Contiene informazioni sul tipo del messaggio, il mittente, il destinatario, la lingua del mittente e la porta sulla quale avviene la comunicazione.
- Contiene anche un campo Data, utile per fornire dati aggiuntivi oltre a quelli previsti.
- Mette a disposizione metodi per la conversione dei campi da Message a JSON.

Settings

- Classe usata per leggere le costanti dal file di configurazione conf.txt.

TCPConnection

- Fornisce dei metodi per facilitare l'apertura e la chiusura delle connessioni tcp.

2.3. Server

Group

- Utilizzata dal server per mantenere informazioni su uno specifico gruppo.
- Fornisce metodi di accesso alle informazioni sul gruppo e di aggiunta membri.

GroupDatabase

- Classe usata per gestire i gruppi creati in Social Gossip.
- Fornisce metodi per la creazione e la cancellazione dei gruppi da parte degli utenti.
- Permette il salvataggio e il caricamento del database dei gruppi su file in formato JSON.

KeepAlive

- Classe utilizzata per controllare periodicamente che gli utenti segnalati come online lo siano effettivamente.
- Come funzione ausiliaria, la classe prevede il salvataggio periodico del database degli utenti e di quello dei gruppi.

ServerMain

- Classe di avvio del Server, che si occupa di istanziare un nuovo manager.
- Utilizzata per inizializzare le connessioni RMI, TCP e UDP del server.
- Gestisce le richieste TCP e UDP in arrivo, avviando un thread per ognuna di esse.

SocialManagerInterface

- Interfaccia per la classe SocialManager, utilizzata dai client per utilizzare i metodi RMI messi a disposizione dal server (login, register, logout).

SocialManager

- Crea ed inizializza tutte le strutture dati del server.
- Fornisce metodi di accesso e modifica dei database utenti e gruppi.
- Fornisce l'implementazione dei metodi RMI, utilizzabili dai client per effettuare la registrazione, il login ed il logout.
- Avvia in un thread una istanza della classe KeepAlive, per la gestione degli utenti online ed il salvataggio dei database sul disco.
- Gestisce la sincronizzazione tra i vari thread implementando metodi synchronized e lock sulle strutture dati.

User

- Utilizzata dal server per mantenere informazioni su uno specifico utente.
- Fornisce metodi di accesso alle informazioni sull'utente.
- Gestisce la sincronizzazione tra i vari thread implementando metodi synchronized e lock sulle strutture dati.
- Contiene il riferimento allo stub dell'utente.

UserDatabase

- Classe utilizzata per gestire gli utenti registrati a SocialGossip.
- Carica la lista degli utenti dal database ad avvio server, e mette a disposizione metodi per il salvataggio della struttura dati utenti nel disco.
- fornisce metodi necessari per aggiungere utenti, prelevare utenti dal database, controllare i dati di login.

TCPHandler

- Classe istanziata dal ServerMain quando arriva una nuova richiesta TCP.
- Contiene metodo run che permette di gestire la richiesta in arrivo e comunicare col relativo client in base alla operazione richiesta.
- Gestisce il forwarding dei messaggi tra amici e le richieste di invio file.

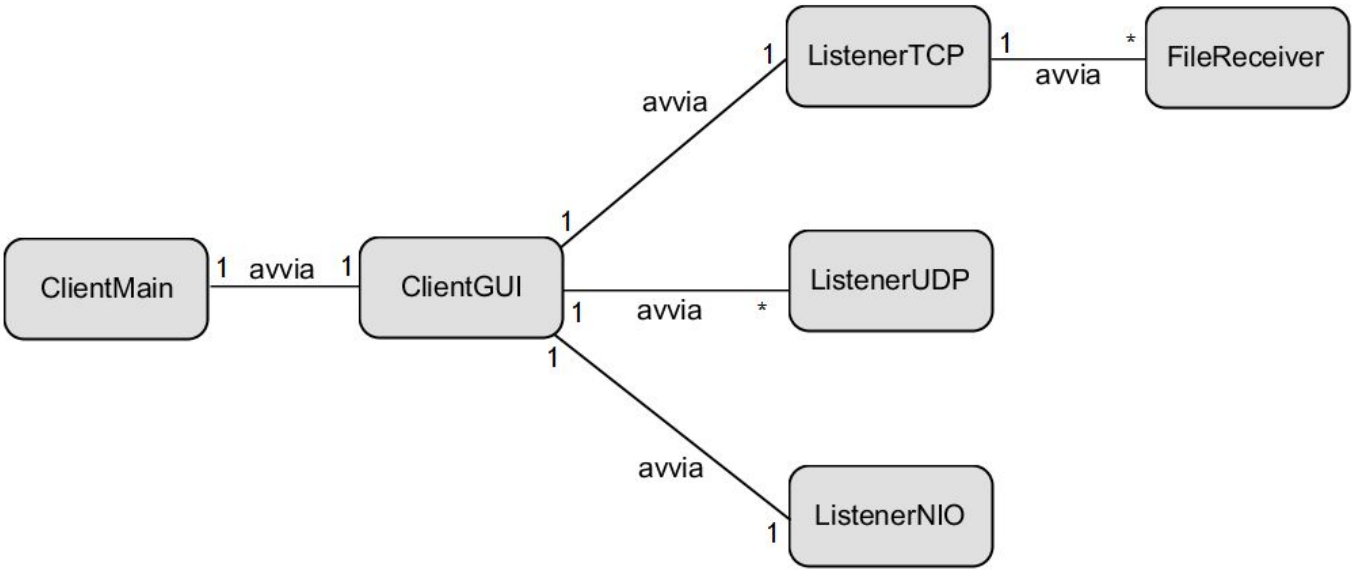
UDPHandler

- Gestisce le richieste UDP di invio messaggi multicast da parte degli utenti, e le richieste di cancellazione dei gruppi da parte dei loro amministratori.

3. Threads

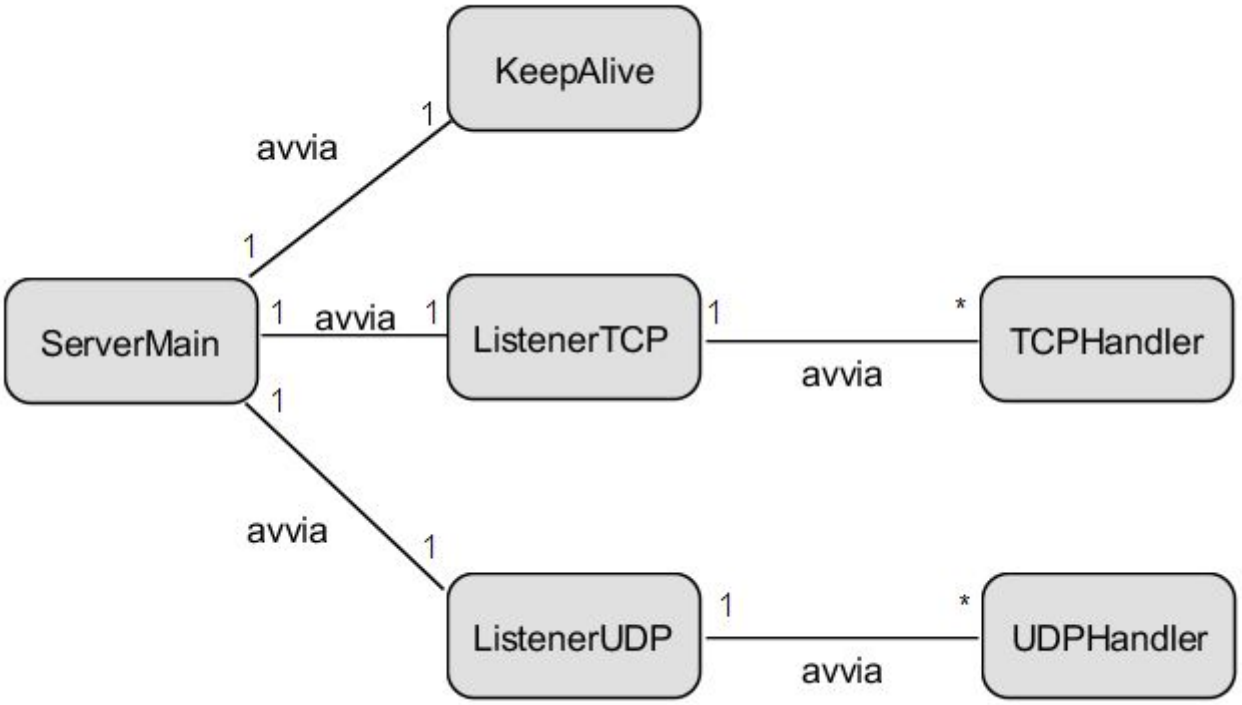
3.1. Client

- L'interfaccia grafica avvia vari thread, gestiti da Swing, ed esporta lo stub RMI, che si occupa di gestire le callback invocate dal server.
- TCPListener, inserito in executorTCP (singleThreadExecutor), è una istanza della classe che si occupa della ricezione e della gestione dei messaggi tramite TCP.
  - Avvia un thread ausiliario ogni volta che riceve una richiesta di invio file, in quanto il trasferimento potrebbe richiedere molto tempo, bloccando la gestione dei messaggi TCP e quindi lo scambio di messaggi fra utenti.
- UDPListener, inserite in ExecutorUDP (CachedThreadPool), sono istanze della classe che gestisce la ricezione e la gestione dei messaggi che arrivano tramite UDP Multicast da uno specifico gruppo. Ogni gruppo a cui l'utente è iscritto ha la sua istanza di UDPListener, alla quale viene assegnata una posta specifica, scelta casualmente all'avvio.
- NIOListener, inserito in ExecutorNIO (SingleThreadExecutor), è una istanza della classe NIOListener, che si occupa dell'invio dei file tramite TCP + NIO. L'utilizzo di NIO permette di avere un solo thread listener, che può quindi gestire più richieste di invio file in contemporanea.



3.2. Server

- Il thread Main del server si occupa di esportare lo stub RMI, che è necessario per garantire servizi RMI ai client che si collegano.
- Thread TCP, avviato dal Main in un SingleThreadExecutor, è attivo per gestire le richieste TCP in entrata, provenienti dai client connessi al server.
  - Per ogni connessione TCP in entrata viene avviato un thread di tipo TCPHandler, inserito in un CachedThreadPool, che la gestisce.
- Thread UDP, avviato dal Main in un SingleThreadExecutor, è attivo per gestire le richieste UDP in entrata, provenienti dai client connessi al server.
  - Per ogni messaggio UDP in entrata viene avviato un thread di tipo UDPHandler, inserito in un CachedThreadPool, che lo gestisce.
- Thread KeepAlive, gestito da un SingleThreadExecutor, è necessario per controllare che gli utenti settati come online lo siano effettivamente (e scollegarli in caso negativo), richiamando una callback RMI e scoprendo quindi se ci sono client che non sono più raggiungibili.
  Il controllo viene effettuato periodicamente (ogni 30 secondi).
  Il thread svolge anche funzioni periodiche di salvataggio database utenti e gruppi su disco, in appositi file JSON all'interno della cartella contenente i file JAR di client e server.
  Il salvataggio avviene ogni 3 iterazioni della funzione di keepalive.



4. **Gestione concorrenza**

   Le strutture dati del client sono memorizzate in una istanza di SocialClient, che le tiene in memoria utilizzando delle ConcurrentHashMap.
   La sincronizzazione avviene grazie alle strutture thread-safe utilizzate, ad alcuni metodi synchronized implementati nella SocialClient, ed al fatto che la GUI rimane bloccata durante l'esecuzione di alcune operazioni, impedendo quindi di poter eseguire più operazioni in contemporanea.
   Inoltre, abbiamo ritenuto ragionevole il fatto di lasciare un solo thread client per la gestione delle connessioni TCP. Questo implica che la struttura dati che memorizza i messaggi fra utenti non viene mai modificata da più thread in contemporanea, garantendo quindi che i dati all'interno della struttura siano corretti.

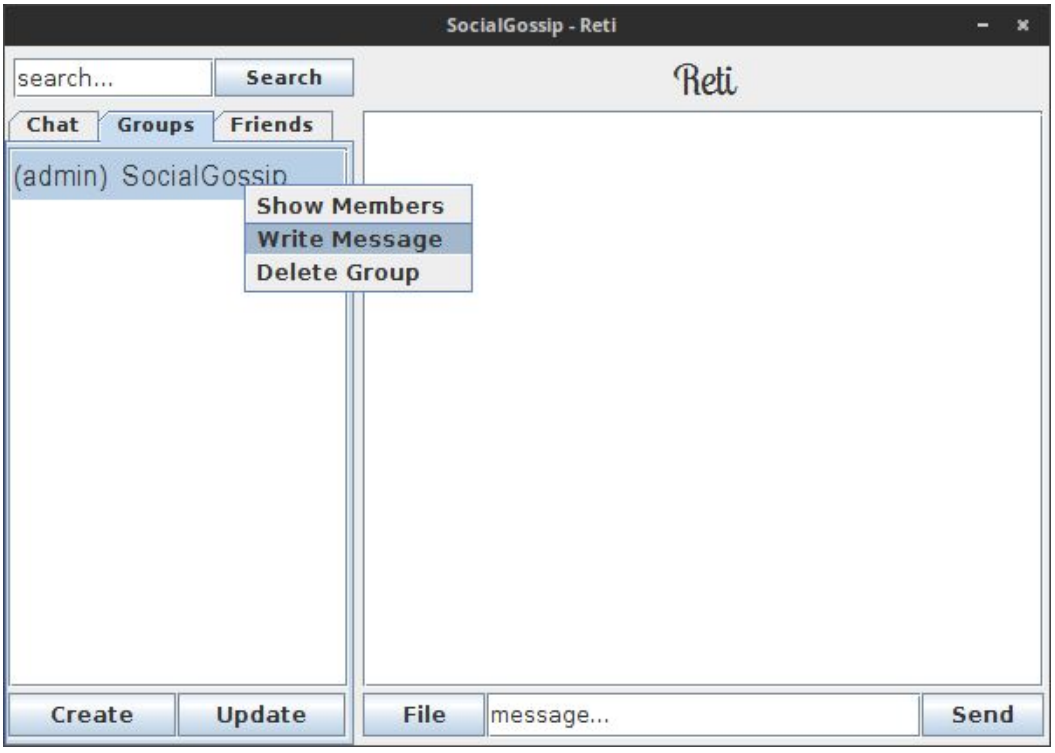
   Per il server è stato necessario utilizzare delle Lock, per garantire che alcune operazioni critiche sul database e sui singoli utenti non venissero effettuate in contemporanea da più thread.
   Il server implementa il database utenti tramite la classe UserDatabase, che prevede, oltre ad alcuni metodi synchronized, una Lock, utilizzata per ottenere i permessi di modifica del database.
   Il database gruppi viene gestito in modo simile, utilizzando la classe GroupDatabase.
   I singoli gruppi e gli utenti sono gestiti dalle loro apposite classi, anche esse provviste di Lock e metodi synchronized, per impedire la modifica in contemporanea da parte di più thread.

5. **Interfaccia grafica**

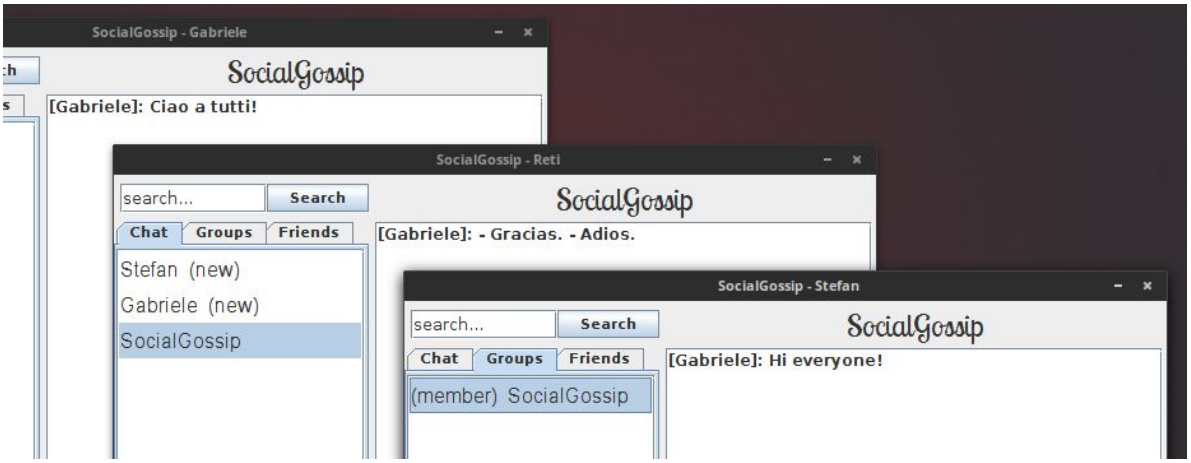
   L'utente interagisce con il client tramite l'interfaccia grafica implementata in ClientGui.
   All'avvio, l'utente può scegliere di registrarsi, oppure di collegarsi al server con un account già registrato.
   Se username e password sono corretti per il login, o se la registrazione è andata a buon fine, allora l'utente risulterà collegato, e verrà mostrata l'interfaccia principale del Social.
   In caso di errore, invece, verrà mostrato un messaggio popup, e l'utente non risulterà collegato al server.

   Una volta entrato nell'interfaccia principale, l'utente potrà scorrere le tab per vedere la lista gruppi, gli amici collegati, e le chat attive.
   La lista gruppi mostra tutti i gruppi presenti sul server. Il client può aggiornare la lista tramite il tasto update, oppure può creare un nuovo gruppo con la funzione create.

Selezionando un gruppo col tasto destro, invece, è possibile visualizzare la lista membri, entrare nel gruppo, oppure eliminarlo (ciò è permesso solo se si è amministratori del gruppo).
Una volta entrati in un gruppo è possibile mandare messaggi selezionandolo dalla lista gruppi, cliccando col tasto destro e selezionando “write message”. A questo punto, il gruppo verrà messo nella tab delle chat attive, e sarà possibile mandare messaggi a tutti i suoi membri attualmente online.

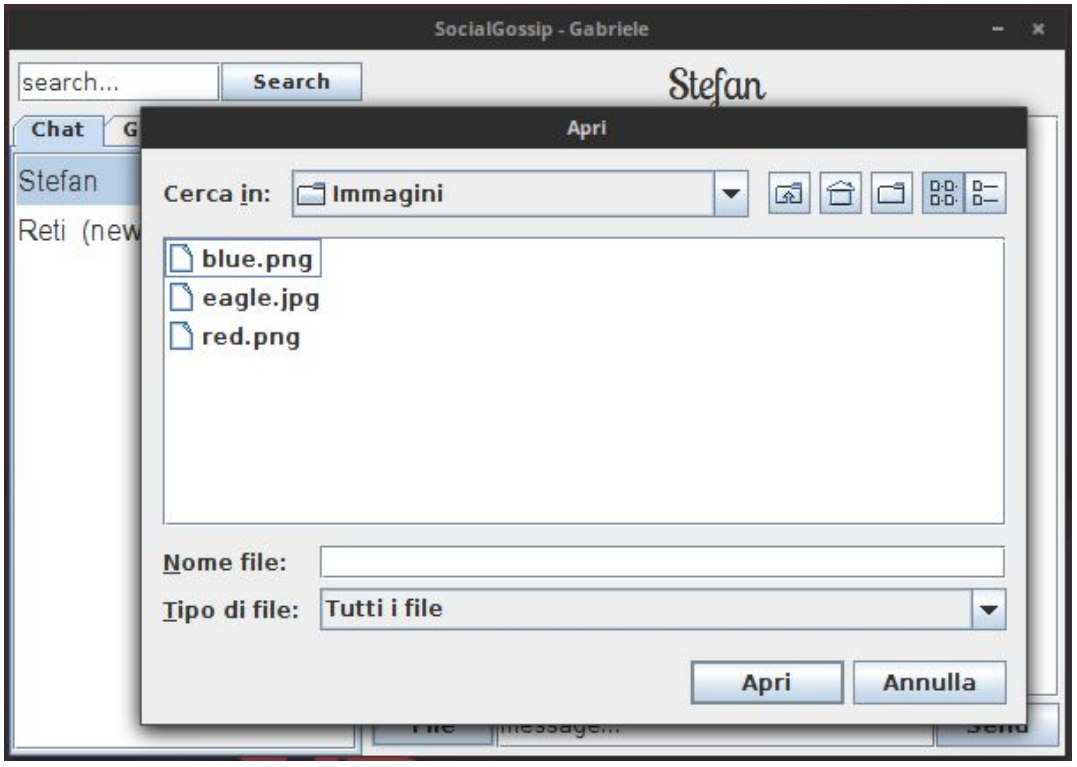


La lista amici, invece, mostra gli amici dell’utente (online e non).
Per avviare una chat con un amico basta selezionarlo dalla lista. A quel punto verrà inserito nella lista delle chat attive, e sarà possibile inviare messaggi.
Le notifiche di messaggi non letti vengono inviate nella lista delle chat attive e saranno visibile tramite l’apposito tag “(new)”.



Nell’interfaccia principale è anche possibile cercare amici o gruppi, utilizzando la funzione di ricerca. La ricerca fornisce una lista che comprende tutti gli utenti e i gruppi che contengono nel nome una sottostringa della parola cercata. Selezionandone uno dal menù a tendina e cliccando ok si aggiunge l’utente come amico, oppure si entra a far parte del gruppo.

Nella chat è presente uno spazio dove vengono visualizzati i messaggi ricevuti in una specifica chat, ed è possibile inviare messaggi tramite il tasto Send.
E' inoltre possibile inviare file ad un altro utente cliccando sul tasto File. Si aprirà una interfaccia di selezione file, e cliccando "select" verrà inoltrata al server la richiesta di invio file.
Al termine dell'invio, sia mittente che destinatario riceveranno una notifica di successo.



5.1. Avvio

Per avviare il server è necessario spostarsi nella cartella contenente i file JAR ed il file di configurazione *conf.txt* da terminale, ed eseguire il comando:

java -jar server.jar

Per avviare il client, invece, eseguire il comando:

java -jar client.jar

6. Ulteriori scelte implementative

Abbiamo deciso di implementare una classe apposita per i messaggi. Questo garantisce:

- modularità tra i vari metodi utilizzati
- maggiore pulizia del codice

La classe Message include metodi di conversione da Message a JSON, così da poter inviare e ricevere stringhe JSON opportunamente parsate.
Message ha un campo tipo, che descrive il tipo di operazione richiesta dall'utente.
Tra gli altri, sono anche definiti tipi utilizzati per segnalare l'esito delle operazioni effettuate dal server (ACK, NACK).

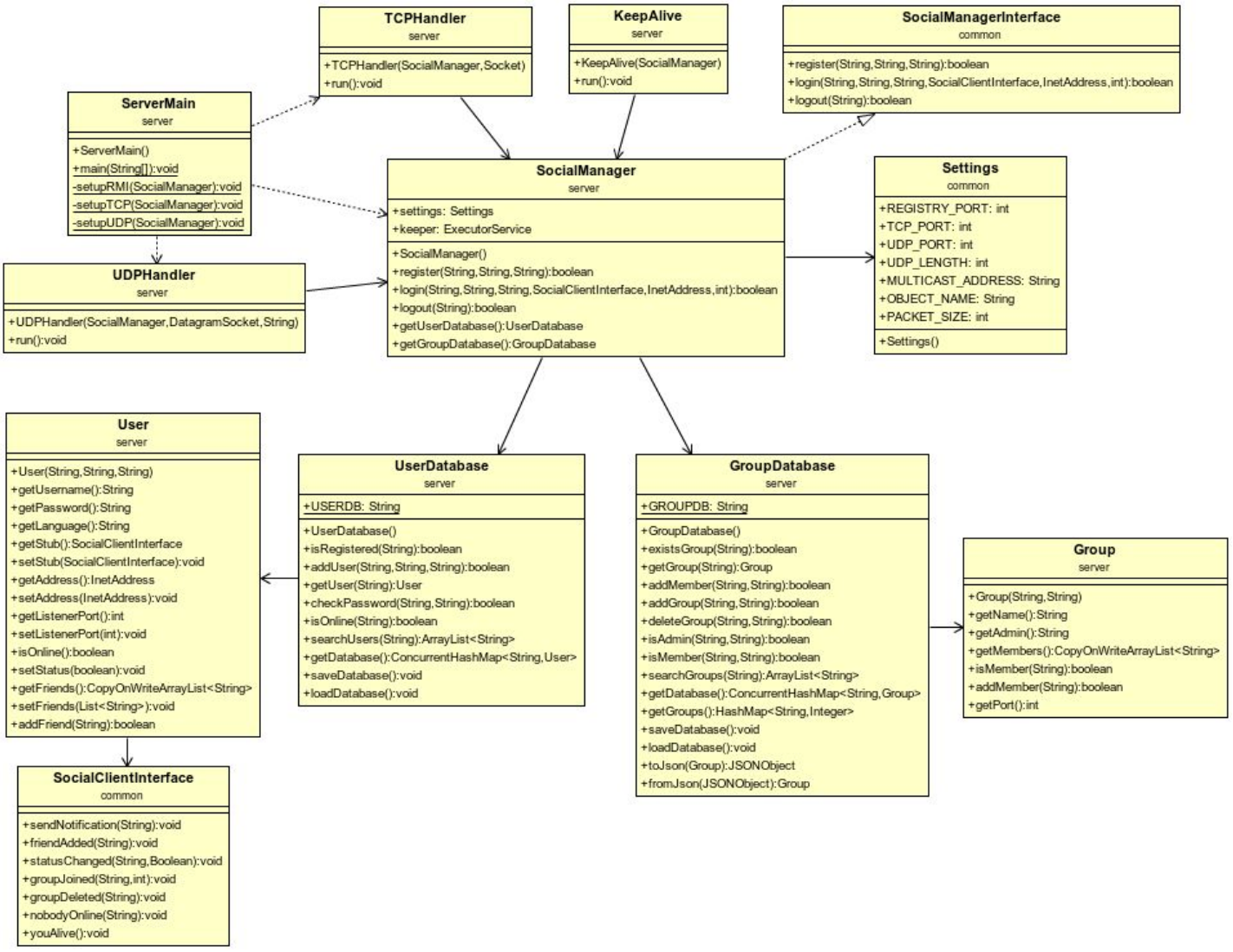
Gli utenti ed i gruppi vengono memorizzati in un file JSON che viene aggiornato periodicamente da un thread apposito nel server, e viene caricato all'avvio di esso.

Per evitare che le password venissero salvate in chiaro nel database, abbiamo deciso di implementare una funzione di hashing (SHA256).

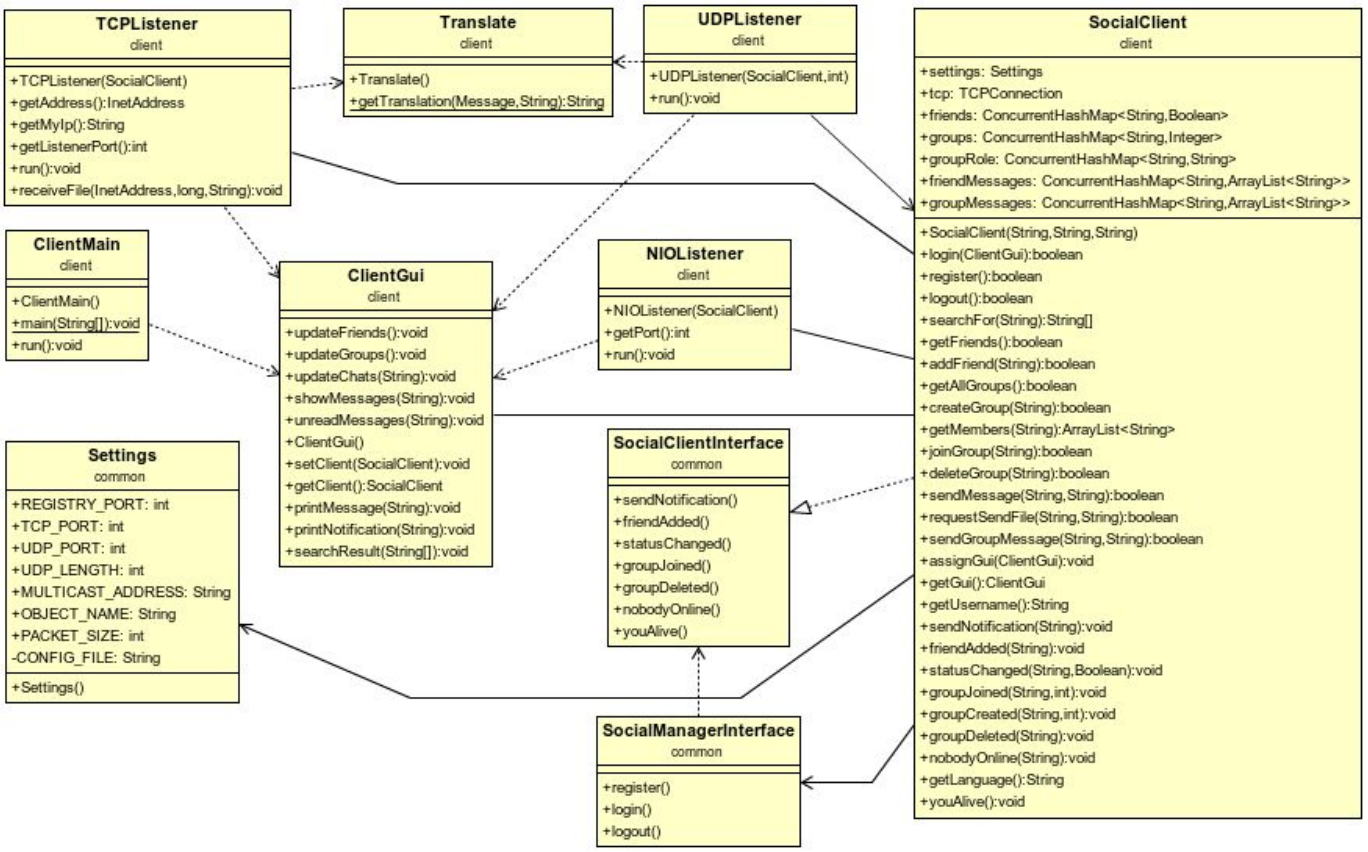
Al login viene effettuato l'hashing della password. Il risultato viene confrontato con l'hash della password salvata nel database alla registrazione e, se corrisponde, l'utente verrà loggato.

7. Appendice

7.1. Server



7.2. Client


TCPListener
client
+TCPListener(SocialClient)
+getAddress():InetAddress
+getMyIp():String
+getListenerPort():int
+run():void
+receiveFile(InetAddress,long,String):void
Translate
client
+Translate()
+getTranslation(Message,String):String
UDPListener
client
+UDPListener(SocialClient,int)
+run():void
SocialClient
client
+settings: Settings
+tcp: TCPConnection
+friends: ConcurrentHashMap<String,Boolean>
+groups: ConcurrentHashMap<String,Integer>
+groupRole: ConcurrentHashMap<String,String>
+friendMessages: ConcurrentHashMap<String,ArrayList<String>>
+groupMessages: ConcurrentHashMap<String,ArrayList<String>>
+SocialClient(String,String,String)
+login(ClientGui):boolean
+register():boolean
+logout():boolean
+searchFor(String):String[]
+getFriends():boolean
+addFriend(String):boolean
+getAllGroups():boolean
+createGroup(String):boolean
+getMembers(String):ArrayList<String>
+joinGroup(String):boolean
+deleteGroup(String):boolean
+sendMessage(String,String):boolean
+requestSendFile(String,String):boolean
+sendGroupMessage(String,String):boolean
+assignGui(ClientGui):void
+getGui():ClientGui
+getUsername():String
+sendNotification(String):void
+friendAdded(String):void
+statusChanged(String,Boolean):void
+groupJoined(String,int):void
+groupCreated(String,int):void
+groupDeleted(String):void
+nobodyOnline(String):void
+getLanguage():String
+youAlive():void
ClientMain
client
+ClientMain()
+main(String[]):void
+run():void
ClientGui
client
+updateFriends():void
+updateGroups():void
+updateChats(String):void
+showMessages(String):void
+unreadMessages(String):void
+ClientGui()
+setClient(SocialClient):void
+getClient():SocialClient
+printMessage(String):void
+printNotification(String):void
+searchResult(String[]):void
NIOListener
client
+NIOListener(SocialClient)
+getPort():int
+run():void
Settings
common
+REGISTRY_PORT: int
+TCP_PORT: int
+UDP_PORT: int
+UDP_LENGTH: int
+MULTICAST_ADDRESS: String
+OBJECT_NAME: String
+PACKET_SIZE: int
-CONFIG_FILE: String
+Settings()
SocialClientInterface
common
+sendNotification()
+friendAdded()
+statusChanged()
+groupJoined()
+groupDeleted()
+nobodyOnline()
+youAlive()
SocialManagerInterface
common
+register()
+login()
+logout()