



UNIVERSITÀ DI PISA

Social Gossip

un servizio di chat multifunzionale

Laboratorio di Reti, Corsi A e B
Progetto di Fine Corso – A.A. 2017/18

STUDENTE : EMILIO PANTI
MAT: 531844

INDICE

- Introduzione	1
• Sistema operativo	
• Esecuzione	
• Piano della relazione	
- Cartella ‘UserFiles’	2
- Lato Client	3
• Suddivisione dei files	3
• Funzionamento client e connessioni	4
• Src	5
• Interfaccia grafica	6
- Lato Server	11
• Suddivisione dei files	11
• Src	11
• Funzionamento server	12
• Strutture dati	12
• Threads	14
- Connessioni	15
- Codice lato client	16
• <u>package client:</u>	
◦ ClientMain.java	17
• <u>package enumerations:</u>	
◦ Operations.java	20
• <u>package graphicInterfaces:</u>	
◦ ChatHandlerGUI.java	21
◦ ChatNicknameGUI.java	25
◦ ChatroomGUI.java	28
◦ LoginGUI.java	31
◦ OperativeGUI.java	35
◦ RegistrationGUI.java	42
◦ RequestGUI.java	48
◦ ResponseGUI.java	52
• <u>package notificationRMI:</u>	
◦ NotifyEventImpl.java	53
◦ NotifyEventInterface.java	55
◦ ServerInterface.java	56
• <u>package threads.chatroomsOp:</u>	
◦ AddToCRThread.java	57
◦ CloseCRThread.java	60
◦ CreateCRThread.java	62
◦ CRLListThread.java	65
• <u>package threads.friendsOp:</u>	
◦ FriendshipThread.java	68
◦ FriendsListThread.java	70

○	LookUpThread.java	72
○	StartChatThread.java	74
•	<u>package threads.listeners:</u>	
○	ListenerCR.java	76
○	ListenerFile.java	78
○	ListenerMsg.java	80
○	UnloaderFileThread.java	83
•	<u>package threads.senders:</u>	
○	FileFriendThread.java	85
○	MsgCRTThread.java	89
○	MsgFriendThread.java	92
•	<u>package threads.usersOp:</u>	
○	CloseThread.java	94
○	LoginThread.java	95
○	RegistrationThread.java	100
- Codice lato server		104
•	<u>package server:</u>	
○	ServerMain.java	104
•	<u>package dataStructures:</u>	
○	Chatroom.java	107
○	HashChatrooms.java	111
○	ListChatrooms.java	114
○	ListUsers.java	117
○	User.java	120
•	<u>package enumerations:</u>	
○	Operations.java *	20
•	<u>package notificationRMI:</u>	
○	NotifyEventInterface.java *	55
○	ServerInterface.java *	56
○	ServerImpl.java	128
•	<u>package threads:</u>	
○	HandlerMsgChatrooms.java	129
○	ThreadPool.java	131
○	Worker.java	132

NOTE:

1. Il simbolo (*) riportato in alcuni files del codice lato server significa che essi fanno parte anche del lato client. Per questo motivo la pagina indicata al loro fianco è la stessa dei corrispondenti files appartenenti al lato client.
2. Nelle pagine dove è riportato il codice ci sono due numerazioni:
 - una in basso al centro che indica il numero della pagina all'interno del file java,
 - una in basso a destra che indica il numero della pagina all'interno di questa relazione (l'indice fa riferimento a questa)

INTRODUZIONE

In questa relazione presenterò le scelte progettuali adottate nella realizzazione del progetto e per le quali è stata lasciata libera iniziativa allo studente.

Non verranno ri-discusse a fondo le implementazioni dettate dalle specifiche del problema.

All'interno della cartella '**SocialGossip**', sono contenuti (oltre a questa relazione) i seguenti elementi:

- il file eseguibile 'SocialGossipClient.jar',
- il file eseguibile 'SocialGossipServer.jar',
- la cartella 'UserFiles- la cartella 'SocialGossipClient- la cartella 'SocialGossipServer

- SISTEMA OPERATIVO:

Il progetto è stato sviluppato e testato completamente su **Windows 10**, non è garantito il corretto funzionamento in altri sistemi operativi.

- ESECUZIONE:

Eseguire da shell (trovandosi all'interno della cartella SocialGossip):

1. ***java -jar SocialGossipServer.jar***: per far partire il server,
2. ***java -jar SocialGossipClient.jar localhost***: per avviare un client.

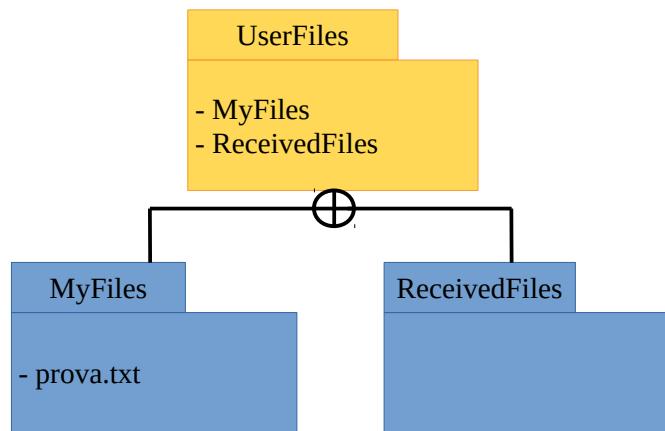
NB: l'avvio del client prevede di specificare l'host del server, ma il progetto è stato testato interamente sulla stessa macchina (sia per il server che per i vari clients). Non è garantito il corretto funzionamento se il progetto viene testato su più macchine o se viene passato al client un host diverso da localhost.

- PIANO DELLA RELAZIONE:

1. Spiegazione dell'utilizzo della cartella UserFiles.
2. Lato client del servizio SocialGossip; come funziona (brevemente) e l'interfaccia grafica offerta all'utente.
3. Lato server del servizio SocialGossip; discutendo le strutture dati utilizzate e i threads attivati per la gestione delle connessioni.
4. Breve panoramica sulle varie connessioni utilizzate all'interno del progetto.

NB: le classi e i metodi sono già stati commentati nel modo più esaustivo possibile all'interno del codice in allegato alla relazione.

Cartella ‘UserFiles’



La cartella ‘UserFiles’ è usata come supporto al client per alcune operazioni, in particolare le due cartelle al suo interno servono a:

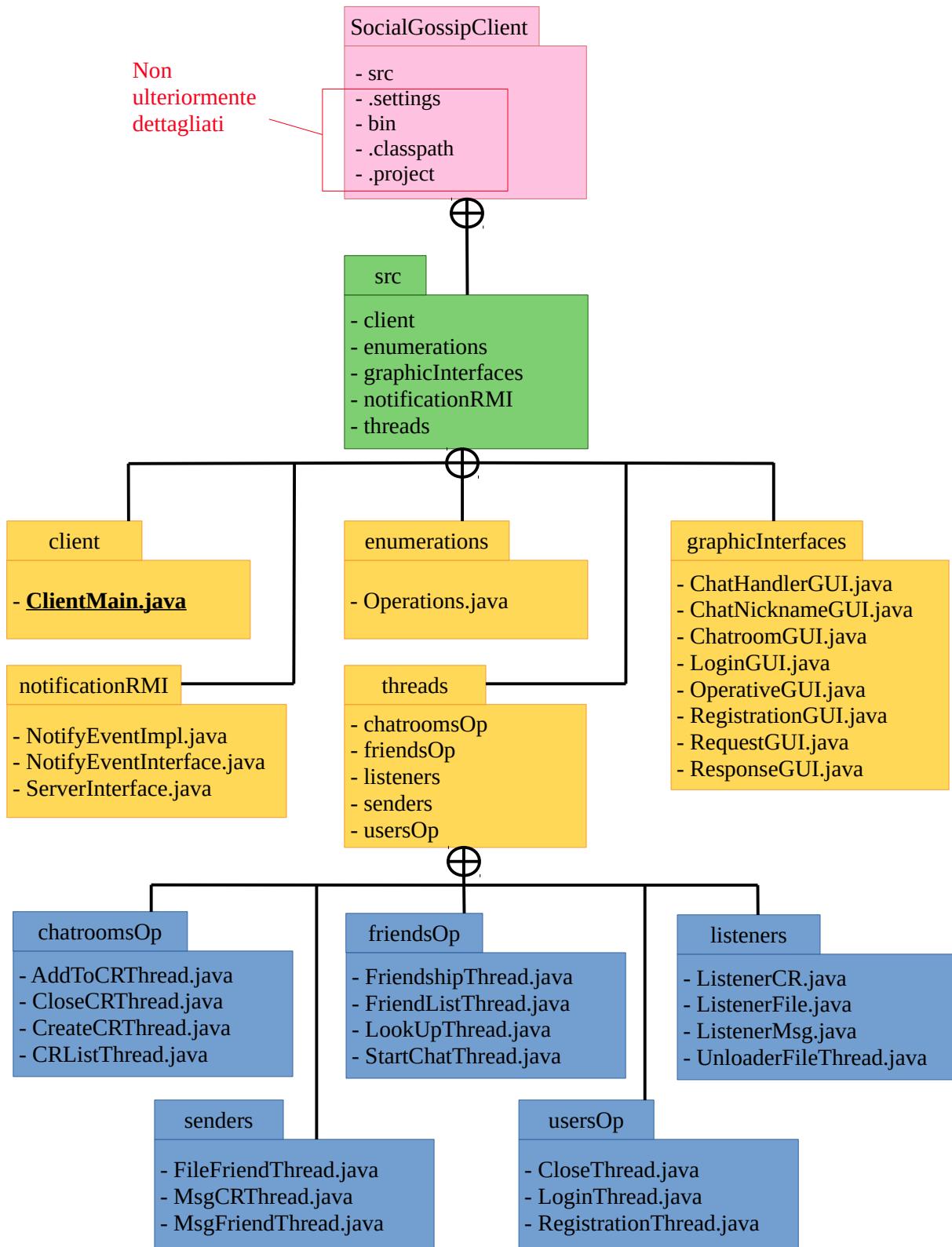
- ‘MyFiles’: il client cercherà qua dentro gli eventuali files che l’utente vorrà inviare ad altri iscritti.
- ‘ReceivedFiles’: è dove il client salva (sovrascrive nel caso in cui esistano di già) i files ricevuti da altri utenti.

NB: all’interno di ‘MyFiles’ è presente un file di prova testuale.

LATO CLIENT

SUDDIVISIONE DEI FILES

I file relativi allo sviluppo del lato client del servizio sono contenuti all'interno della cartella ‘SocialGossipClient’, suddivisa come segue:



FUNZIONAMENTO CLIENT E CONNESSIONI

Descriverò brevemente il funzionamento del client da un punto di vista molto astratto:

1. viene avviato il client ([ClientMain.java](#)), che apre:
 - la prima connessione TCP verso il server, utilizzata per richiedere operazioni e ricevere i relativi responsi,
 - il multicast socket per la ricezione dei messaggi provenienti dalle chatrooms (e per iscriversi ad esse),
 - il server socket channel per permettere agli altri utenti di connettersi ed inviare i files,
2. viene mostrata all'utente l'interfaccia di login (da cui è possibile passare a quella di registrazione),
3. se l'operazione di login (o di registrazione) va a buon fine:
 - viene effettuata l'iscrizione (utilizzando il multicast socket aperto dal ClientMain.java al punto 1) a tutti i gruppi multicast relativi alle chatrooms a cui l'utente era già iscritto (questo passaggio è effettuato solo nel caso in cui sia stato effettuato il login e non la registrazione),
 - viene creato un listener ([ListenerMsg.java](#)) che apre la seconda connessione TCP verso il server per ricevere i messaggi testuali provenienti dagli altri utenti,
 - viene creato un listener ([ListenerFile.java](#)) che si mette in attesa sul server socket channel (aperto dal ClientMain.java al punto 1) di nuove connessioni da parte degli utenti che vogliono inviare dei file a questo client. Per ogni nuova connessione viene attivato un thread ([UnloaderFileThread.java](#)) incaricato di scaricare il file inviatogli,
 - viene creato un listener ([ListenerCR.java](#)) che si mette in ascolto sul multicast socket (aperto dal ClientMain.java al punto 1) dei messaggi provenienti dalle chatrooms a cui l'utente è iscritto,
 - viene registrato lo stub per ricevere le notifiche tramite RMI (utilizzando le classi all'interno del package notificationRMI),
4. viene mostrata l'interfaccia operativa all'utente, il quale adesso è libero di effettuare tutte le operazioni che vuole (stringere amicizie, spedire messaggi, iscriversi alle chatrooms ecc.),
5. Se si verificano degli errori nella comunicazione con il server oppure se l'utente decide di chiudere l'interfaccia operativa (o ancora prima quella di login o di registrazione) viene terminato il client, chiudendo tutte le interfacce grafiche aperte e i thread attivi.

SRC

All'interno di src sono presenti tutti i file java per lo sviluppo del lato client, suddivisi nei seguenti packages:

- **client**: contiene ClientMain.java.
- **enumerations**: contiene il file Operations.java, enumerazione delle operazioni di richiesta e delle possibili risposte.
- **notificationRMI**: contiene i files per la registrazione al servizio delle notifiche offerto dal server.
- **graphicInterfaces**: contiene i files per la gestione dell'interfaccia grafica.
- **threads**: suddiviso in ulteriori packages:
 - chatroomsOp: contiene i files relativi ai threads che eseguono operazioni sulle chatrooms.
 - friendsOp: contiene i files relativi ai threads che eseguono operazioni sugli amici.
 - listeners: contiene i files relativi ai threads in ascolto di messaggi e files spediti da altri iscritti e dalle varie chatrooms.
 - senders: contiene i files relativi ai threads incaricati di inviare messaggi e files agli altri utenti ed alle chatrooms.
 - usersOp: contiene i files relativi ai threads che eseguono le operazioni di login, registrazione e chiusura connessione.

NOTA PER THREADS:

Le funzionalità dei vari threads usati dal client (ma anche di tutte le altre classi java) sono descritte dettagliatamente all'inizio di ogni file nel codice java (sotto-forma di commenti) allegato alla relazione.

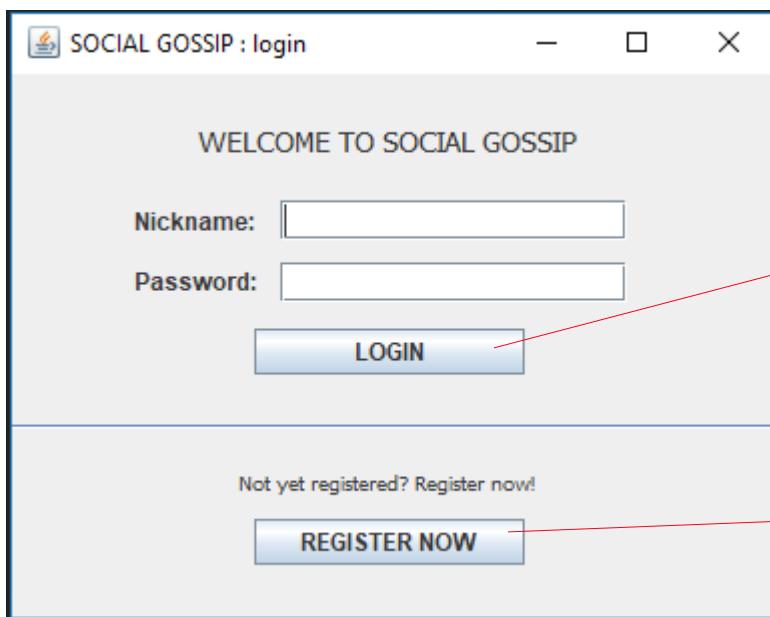
Tali informazioni non verranno pertanto riscritte qui.

INTERFACCIA GRAFICA

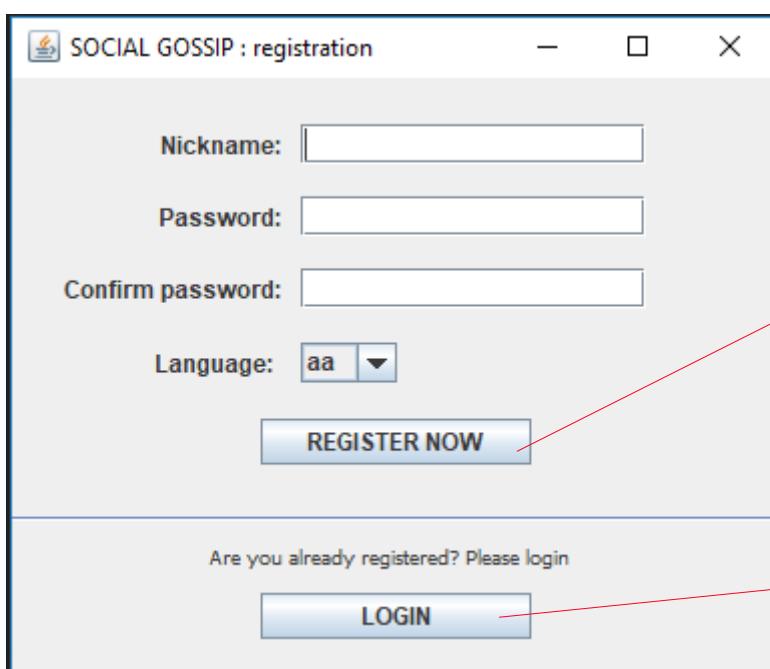
Presenterò le varie interfacce grafiche generate dalle classi del package graphicInterfaces.

NB: le interfacce grafiche sono state sviluppate utilizzando l' **API Swing**.

-LoginGUI.java



-RegistrationGUI.java

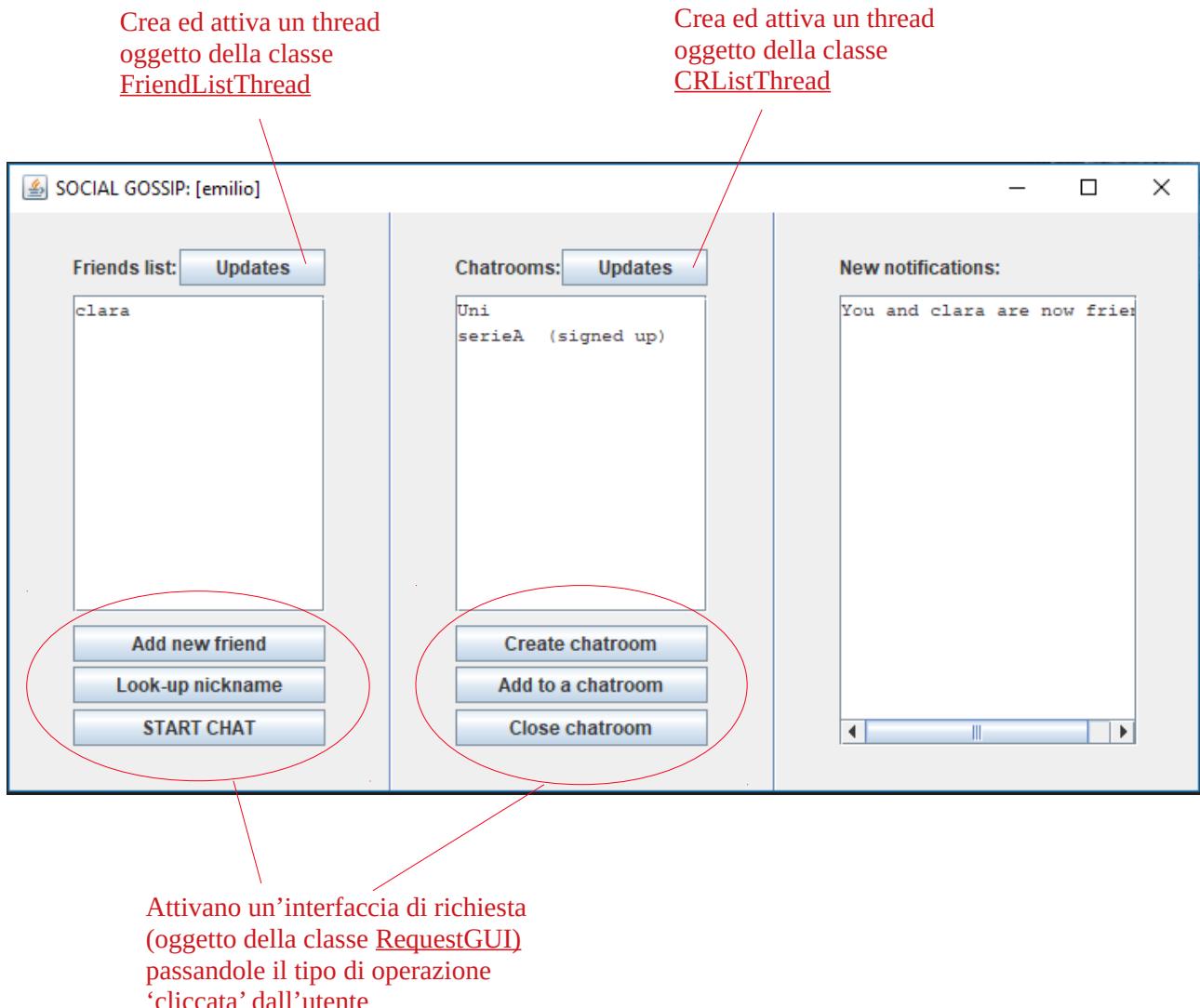


Nel caso in cui la richiesta di login (o di registrazione) vada a buon fine il thread oggetto della classe LoginThread (o RegistrationThread) prima di terminare crea due interfacce grafiche:

- un oggetto della classe OperativeGUI,
- un oggetto della classe ChatHandlerGUI

-OperativeGUI.java

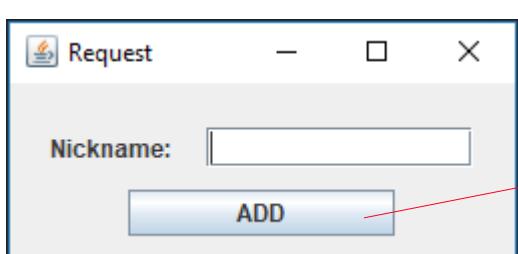
L'interfaccia offerta da questa classe mostra all'utente la sua lista amici, la lista delle chatrooms, le notifiche che lo riguardano e le possibili operazioni che può effettuare verso gli altri iscritti e verso le chatrooms.



- RequestGUI.java

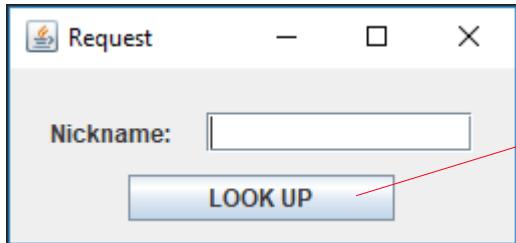
Questo tipo di interfaccia cambia ed attiva threads diversi in base a quale operazione ha richiesto l'utente dalla interfaccia operativa (OperativeGUI):

- button 'Add new friend':



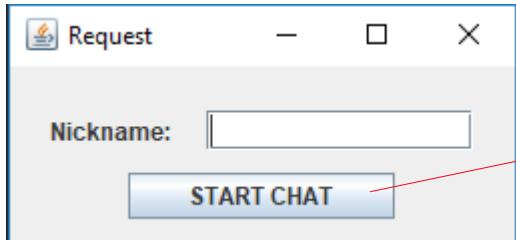
Crea ed attiva un thread
oggetto della classe
[FriendshipThread](#)

- button ‘Look-Up nickname’:



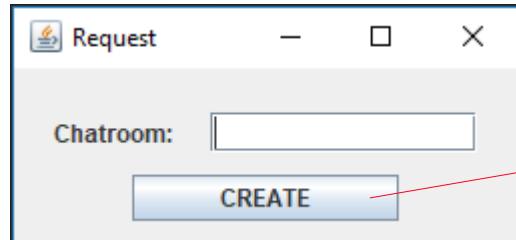
Crea ed attiva un thread
oggetto della classe
[LookUpThread](#)

- button ‘START CHAT’:



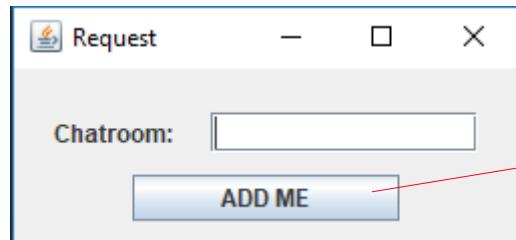
Crea ed attiva un thread
oggetto della classe
[StartChatThread](#)

- button ‘Create chatroom’:



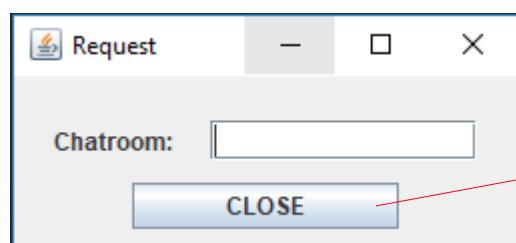
Crea ed attiva un thread
oggetto della classe
[CreateCRTThread](#)

- button ‘Add to a chatroom’:



Crea ed attiva un thread
oggetto della classe
[AddToCRTThread](#)

- button ‘Close chatroom’:



Crea ed attiva un thread
oggetto della classe
[CloseCRTThread](#)

- ChatHandlerGUI.java , ChatNickname.GUI e ChatroomGUI

La classe ChatHandlerGUI offre una interfaccia grafica per la gestione di tutte le chat aperte dall'utente, sia verso gli altri iscritti che verso le chatroom a cui è iscritto. Questa interfaccia è visibile all'utente se ha almeno una chat aperta.

Le chat verso gli altri utenti sono oggetti della classe ChatNicknameGUI.

Possono essere aperte a seguito di due eventi:

- l'utente apre una chat verso un amico tramite il pulsante 'START CHAT' della interfaccia operativa
- l'utente riceve un messaggio da un amico (verso cui non era ancora stata aperta nessuna chat)

Vengono chiuse nel caso in cui l'utente tenti di comunicare con un suo amico offline.

Le chat riguardati le chatroom a cui è iscritto l'utente sono oggetti della classe ChatroomGUI.

Possono essere aperte a seguito di diversi eventi:

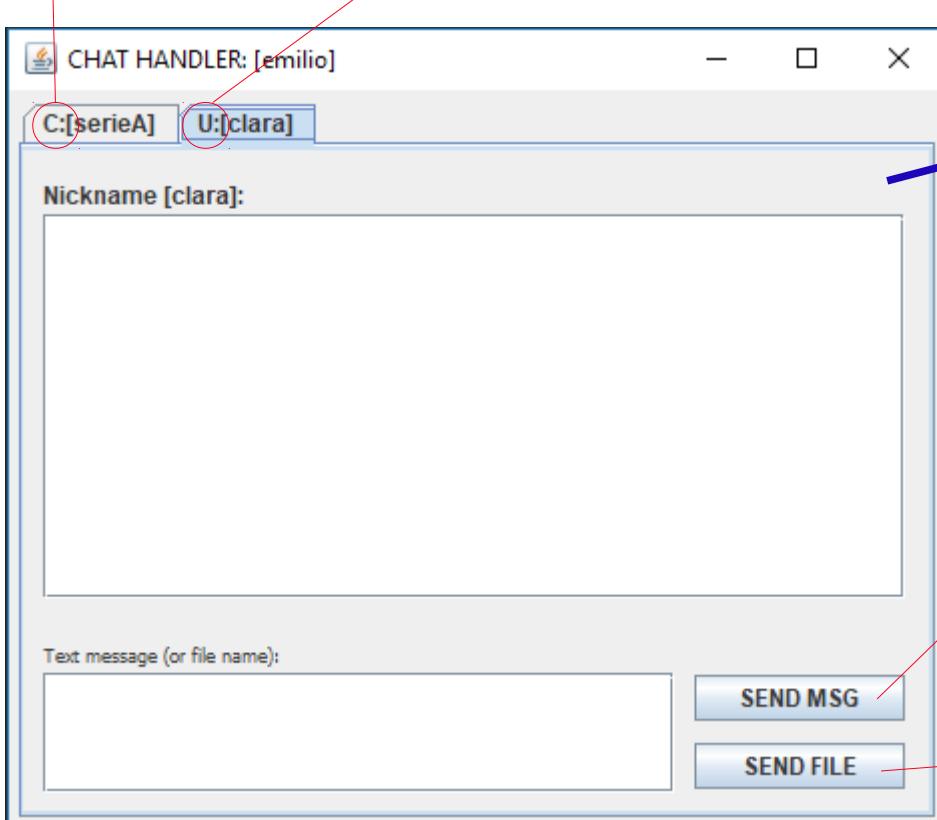
- l'utente crea una chatroom tramite il pulsante 'Create chatroom' della interfaccia operativa
- l'utente si unisce ad una chatroom tramite il pulsante 'Add to a chatroom' della interfaccia operativa
- quando l'utente effettua il login si aprono tutte le chat verso le chatrooms a cui si era iscritto in precedenti sessioni (e che sono ancora attive)

Vengono chiuse nel caso in cui il creatore di una chatroom decida di eliminarla.

La classe ChatHandlerGUI utilizza un oggetto JTabbedPane, che permette di gestire un gruppo di componenti (in questo caso gli oggetti creati dalle classi ChatNicknameGUI e ChatroomGUI).

'C' sta per chatroom,
rappresenta un oggetto
della classe ChatroomGUI

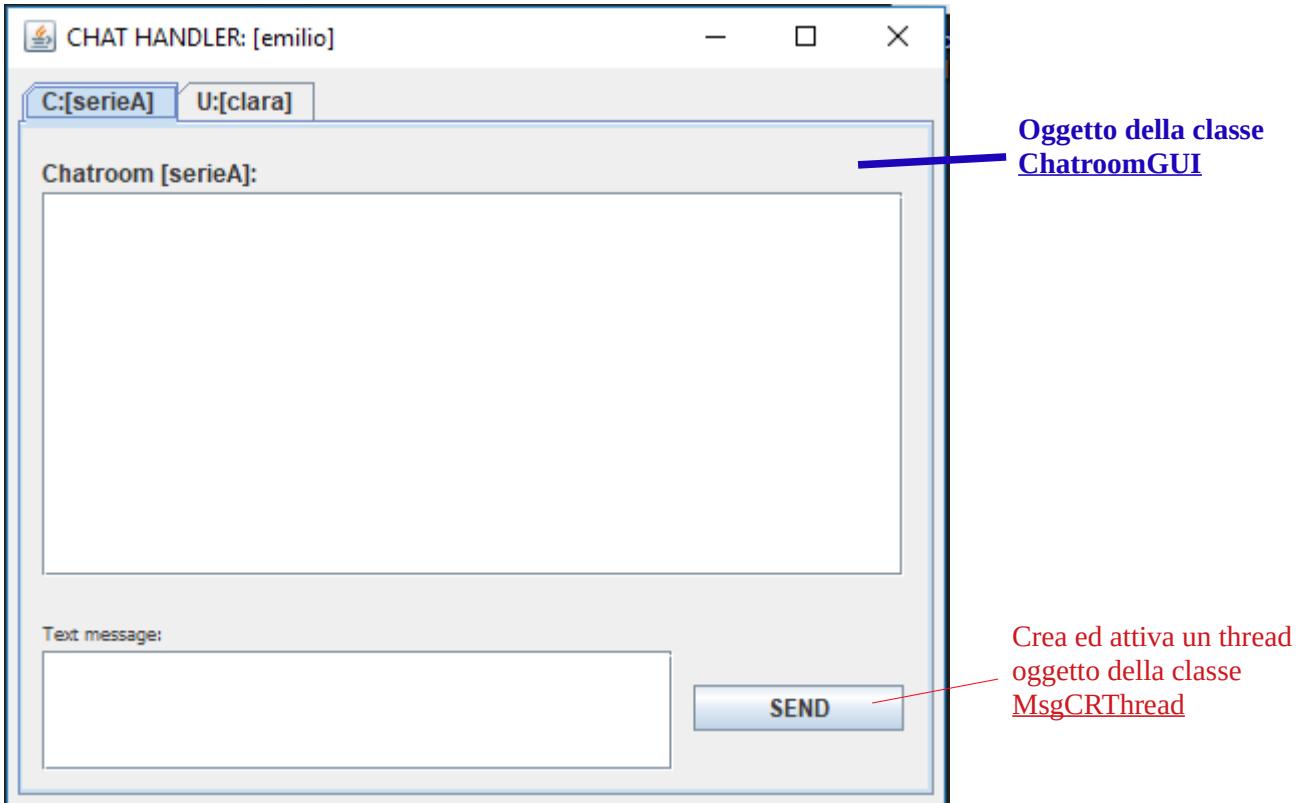
'U' sta per utente,
rappresenta un oggetto della
classe ChatNicknameGUI



Oggetto della classe
ChatNicknameGUI

Crea ed attiva un thread
oggetto della classe
MsgFriendThread

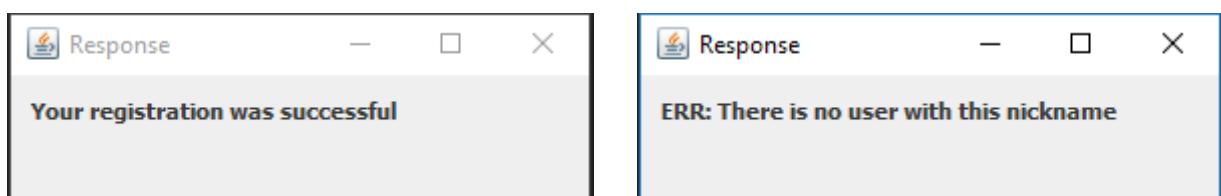
Crea ed attiva un thread
oggetto della classe
FileFriendThread



- ResponseGUI.java

Gli oggetti di questa classe sono usati per comunicare all’utente l’esito di una sua richiesta e per notificargli determinati eventi (come la chiusura di una chatroom).

Sono mostrati qui sotto due esempi:



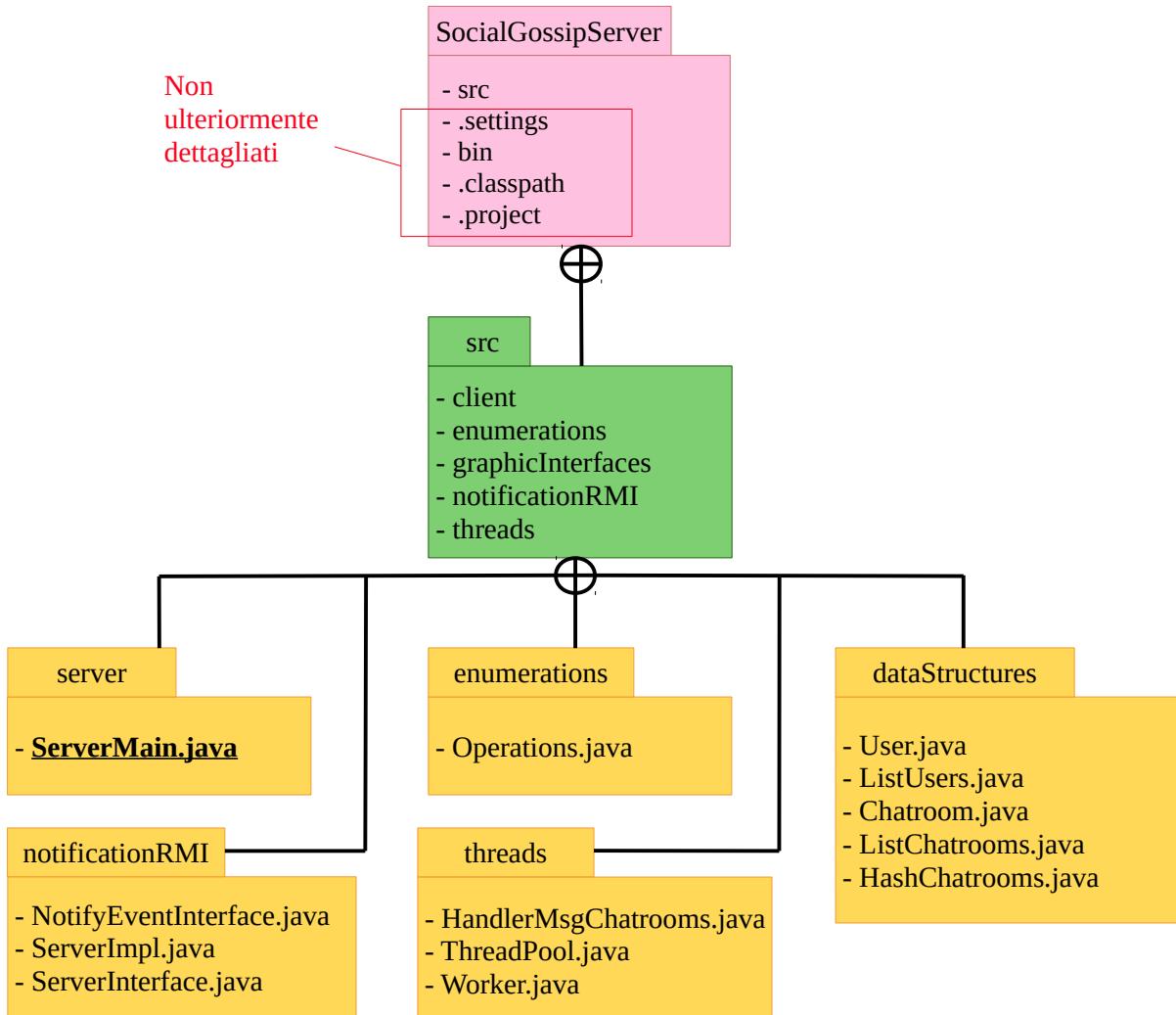
Nota sulla chiusura interfacce:

La chiusura delle interfacce grafiche offerte dalle classi LoginGUI, RegistrationGUI, OperativeGUI e ChatHandlerGUI da parte dell’utente sono interpretate come volontà di terminare il client. Pertanto tali eventi creano ed attivano un thread oggetto della classe [CloseThread](#) (incaricato di comunicare al server l’imminente chiusura e di invocare il metodo di clean-up della classe ClientMain.java).

LATO SERVER

SUDDIVISIONE DEI FILES

I files relativi allo sviluppo del lato server del servizio sono contenuti all'interno della cartella ‘SocialGossipServer’, suddivisa come segue:



SRC

All'interno di `src` sono presenti tutti i files java per lo sviluppo del lato server, suddivisi nei seguenti packages:

- **server**: contiene `ServerMain.java`.
- **enumerations**: contiene il file `Operations.java` che è una enumerazione delle operazioni di richiesta e delle possibili risposte.
- **notificationRMI**: contiene i files per offrire il servizio remoto di registrazione alle notifiche per i clients
- **dataStructures**: contiene i files delle strutture dati usate dal server (descritte in seguito)
- **threads**: i threads usati dal server per svolgere le sue funzioni (descritti successivamente).

FUNZIONAMENTO SERVER

Descriverò brevemente il funzionamento del server da un punto di vista molto astratto:

1. viene avviato il server ([ServerMain.java](#)) che:
 - crea una hash map per gli utenti registrati (ConcurrentHashMap <String,User>) e una per le chatrooms (oggetto della classe [HashChatrooms.java](#)).
 - crea un thread pool (oggetto della classe [ThreadPool.java](#)) per la gestione delle connessioni verso i vari clients.
 - fa partire un thread (oggetto della classe [HandlerMsgChatrooms.java](#)) che si occupa di smistare alle chatrooms di competenza i messaggi ricevuti dagli utenti e destinati ad esse.
 - offre un servizio remoto per permettere ai clients di registrarsi con lo scopo di ricevere le notifiche che li riguardano (utilizzando le classi all'interno del package notificationRMI).
2. Si mette in attesa ciclica di nuove connessioni, che vengono passate (e gestite) al thread pool creato precedentemente.
3. Il server termina tramite la chiusura del processo (es. Ctrl-C sulla shell in cui è in esecuzione).

STRUTTURE DATI

Breve descrizione delle strutture dati utilizzate dal server:

- **User:** rappresenta un utente iscritto al servizio SocialGossip. Le variabili (private) che caratterizzano il suo stato sono:
 - String nickname;
 - String psw;
 - String language;
 - boolean online; ([true se è online, false se offline](#))
 - ListUsers friends; ([lista degli amici, oggetto della classe \[ListUsers.java\]\(#\) descritta sotto](#))
 - ListChatrooms chatrooms; ([lista delle chatrooms a cui l'utente è iscritto, oggetto della classe \[ListChatrooms.java\]\(#\) descritta sotto](#))
 - NotifyEventInterface stub; ([stub per notificare gli eventi al client se l'utente è online](#))
 - Socket socketMsg;
 - DataOutputStream writerMsg; [(servono ad inviare i messaggi testuali all'utente)
 - DataInputStream readerMsg;
 - long portFile; ([porta in cui l'utente, se online, è in ascolto per ricevere files](#))

La maggioranza dei metodi di questa classe sono sincronizzati (o hanno parti di codice sincronizzate) dato che più threads possono accedere contemporaneamente a questa struttura dati.

Nota sulla variabile boolean online: dato che il client si connette con il server , apre la seconda connessione TCP (per ricevere i messaggi testuali) e si registra alle notifiche in tre momenti distinti, l’utente non risulterà online immediatamente dopo il login (o la registrazione).

Risulterà online quando tutte e tre le operazioni sopracitate saranno effettuate, cioè quando le variabili stub e socketMsg saranno entrambe **non null** (vengono poste a null quando l’utente va offline).

Questa scelta è stata presa per far sì che qualsiasi thread voglia eseguire una operazione su un utente online possa effettivamente farlo senza creare situazioni di inconsistenza (es: un thread del server vuole notificare un evento ad un utente online il cui client non ha ancora effettuato la registrazione dello stub).

- **Chatroom:** rappresenta un gruppo di chat per utenti. Le variabili (private) che caratterizzano il suo stato sono:

- String id;
- String creator;
- int usersConn; ([numero degli utenti connessi, se < 2 i messaggi non vengono spediti](#))
- ListUsers subscribes; ([lista degli iscritti, oggetto della classe ListUsers.java descritta sotto](#))
- InetSocketAddress address; ([multicast address associato alla chatroom](#))
- boolean active; ([variabile per stabilire se una chatroom è attiva o in ‘chiusura’](#))

La maggioranza dei metodi di questa classe sono sincronizzati (o hanno parti di codice sincronizzate) dato che più threads possono accedere contemporaneamente a questa struttura dati.

Nota sulla variabile boolean active: se settata a true la chatroom è ‘attiva’ ed è possibile effettuare qualsiasi operazione su di essa; se false è in ‘chiusura’, ovvero sta per essere eliminata. Ciò è per evitare che vengano effettuate operazioni sulla chatroom (di cui l’utente creatore ha richiesto la rimozione) nel lasso di tempo che intercorre tra la notifica della sua chiusura agli utenti iscritti e l’effettiva cancellazione di essa dalla hash map (oggetto della classe HashChatrooms.java, descritta sotto) che gestisce le chatrooms.

- **ListUsers:** contiene i metodi per creare/gestire liste di utenti ed operarci. La struttura ‘base’ su cui si poggia è un ArrayList <User>. Questa classe è usata all’interno delle strutture User (per rappresentare la lista amici di un utente) e Chatroom (per rappresentare la lista degli iscritti alla chatroom).
Tutti i metodi sono sincronizzati dato che questa struttura dati può essere usata da più threads contemporaneamente.

- **ListChatrooms:** contiene i metodi per creare/gestire liste di chatrooms ed operarci. La struttura ‘base’ su cui si poggia è un ArrayList <Chatroom>. Questa classe è usata all’interno della struttura User (per rappresentare la lista delle chatrooms a cui è iscritto un utente).

Tutti i metodi sono sincronizzati dato che questa struttura dati può essere usata da più threads contemporaneamente.

- **HashChatrooms:** Struttura dati per la gestione delle chatrooms, che utilizza due 'sotto-strutture dati':
 - una hash map (oggetto della classe `HashMap <String, Chatroom>`),
 - una lista di `InetAddress` (oggetto della classe `LinkedList <InetAddress>`).

Nella hash map vengono inserite e rimosse le chatrooms. I metodi che operano su di essa sono sincronizzati in modo manuale per evitare inconsistenze. E' previsto un numero massimo di chatrooms esistenti, deciso dalla variabile statica `MAX`.

La lista contiene `MAX InetAddress` di indirizzi multicast. Ogni volta che viene inserita una chatroom nella hash map le viene assegnato un indirizzo multicast, prelevato dalla lista. Se la lista è vuota e se la hash map ha raggiunto `MAX` elementi non è possibile inserire nuove chatrooms. Quando una chatroom viene chiusa e rimossa dalla hash map, viene reinserito nella lista degli `InetAddress` l'indirizzo che utilizzava, così da renderlo nuovamente disponibile.

NB: Per la gestione degli utenti non ho creato una nuova struttura dati come per le chatrooms ma ho utilizzato una `ConcurrentHashMap <String, User>`. Questo perché le uniche operazioni necessarie erano inserimento e ricerca dei vari utenti da eseguire in maniera sincronizzata.

THREADS

Il server utilizza un thread pool e due tipologie di thread per svolgere le sue funzioni:

- **HandlerMsgChatrooms:** il server fa partire questo thread prima di mettersi in attesa ciclica di connessioni. La funzione di questo thread è di smistare i messaggi destinati alle chatrooms. Per fare ciò apre un `DatagramSocket` in cui successivamente si mette in ascolto di messaggi provenienti dagli utenti (ma destinati alle chatrooms). Ogni pacchetto che riceve lo trasforma in formato json. Da quest'ultimo preleva l'inet address della chatroom di destinazione e spedisce il messaggio a tale indirizzo multicast.
- **ThreadPool:** thread pool creato ed utilizzato dal server per la gestione delle connessioni. Ogni thread del pool (oggetti della classe `Worker.java`) gestisce una connessione verso un client finché essa non viene chiusa. Il numero massimo di threads nel pool (e di conseguenza delle connessioni) è deciso dalla variabile statica `MAX_CONN` della classe `ServerMain.java`.
- **Worker:** thread che gestisce una connessione verso un client finché essa non viene terminata. Una volta avviato si mette in attesa ciclica di richieste da parte del client. In base alla operazione che deve eseguire sceglie quale metodo (privato) chiamare per creare la risposta da poi inviare al client.
Un thread di questa classe può terminare per uno dei seguenti motivi:
 - il client comunica che chiuderà la connessione,
 - viene rilevato qualche problema nella comunicazione con il client,
 - viene richiesta dal client l'apertura della seconda connessione TCP (dedicata alla ricezione dei messaggi). In questo caso il worker salva il `Socket`, il `DataOutputStream` e il `DataInputStream` aperti verso il client nella struttura dati relativa all'utente; dopodichè termina.

CONNESSIONI

Poichè le connessioni e i relativi protocolli utilizzati sono stati applicati in base alle specifiche dettate dal progetto. Questa che segue è una breve descrizione di quelle utilizzate:

- Una connessione **TCP** persistente tra il client ed il server per comunicare **richieste** (login, registrazione, ecc) e **risposte** tra loro. Viene aperta all'avvio del client e chiusa quando esso si disconnette.
- Una connessione **TCP** persistente sempre tra client e server per permettere a quest' ultimo di inoltrare i **messaggi testuali** destinati all'utente che sono spediti dai suoi amici. Questa connessione viene aperta quando l'operazione di login (o di registrazione) ha successo, e viene chiusa quando il client si disconnette.
- Una possibile connessione **TCP** non persistente tra due clients quando uno di essi vuole **inviare un file** all'altro. Tale connessione viene creata tramite l'aiuto del server, che comunica (su richiesta) al client mittente l'indirizzo e la porta dove è in ascolto il destinatario. Una volta inviato il file la connessione viene chiusa.
- Varie connessioni **UDP** per la gestione dei **messaggi** riguardanti le **chatrooms**. In particolare la “vita” di un messaggio destinato ad una chatroom è divisa nelle seguenti fasi:
 - il client invia il messaggio tramite pacchetto UDP al server (usando il thread della classe [MsgCRThread.java](#)),
 - il server riceve il messaggio (tramite il listener della classe [HandlerMsgChatrooms.java](#)) e lo spedisce all'indirizzo **multicast** relativo alla chatroom di destinazione,
 - i clients iscritti alla chatroom di riferimento ricevono il messaggio sul MultcastSocket (dove è in ascolto il thread della classe [ListenerThread.java](#)) e lo mostrano all'utente.
- Una connessione **RMI** tramite cui i clients possono iscriversi al servizio di **notifiche** offerto dal server.

NOTA SERVIZIO DI TRADUZIONE: il thread della classe Worker.java quando riceve una richiesta di invio messaggio testuale ad un utente controlla le lingue del mittente e del destinatario. Se non sono uguali apre una connessione TCP verso il **server REST MemoryTranslated** e tramite protocollo HTTP richiede la traduzione del messaggio. Dalla risposta ricevuta viene estratto il messaggio tradotto, che viene poi inviato al destinatario.

NOTA FORMATI JSON: tutte le connessioni sopra citate (ad eccezione del protocollo RMI) utilizzano formati json per rappresentare le informazioni di interesse.

CODICE LATO CLIENT

(Cartella SocialGossipClient/src)

ClientMain.java

```
1 package client;
2
3 import java.io.BufferedInputStream;
18
19
20 /**
21 * Classe ClientMain.
22 * Contiene il main che gestisce il client, prende come argomento
23 * l'host del server SocialGossip.
24 * Apre:
25 * - la connessione TCP verso il server,
26 * - il multicast socket per la ricezione di messaggi dalle chatroom,
27 * - il server socket channel per la ricezione di eventuali file inviati
28 * da altri client,
29 * - infine l'interfaccia grafica per il login all'applicazione SocialGossip.
30 * Offre anche un metodo per la chiusura di tutte le connessioni aperte.
31 * @author Emilio Panti mat:531844
32 */
33 public class ClientMain {
34
35     //porte per connessioni e hostname del server
36     public static int PORT_TCP = 6012;
37     public static int PORT_UDP = 6013;
38     public static int PORT_RMI = 6014;
39     public static String HOSTNAME = null;
40
41     //prima connssione TCP
42     //socket e streams utilizzati per fare richieste e ricevere i relativi responsi da/verso
43     //il server
44     public static Socket SOCKET = null;
45     public static DataOutputStream WRITER = null;
46     public static DataInputStream READER = null;
47
48     //seconda connessione TCP
49     //socket e stream utilizzati per ricevere i messaggi dagli altri utenti
50     public static Socket SOCKET_MSG = null;
51     public static DataInputStream READER_MSG = null;
52     public static DataOutputStream WRITER_MSG = null;
53
54     //Porta e multicast socket per unirmi alle chatrooms e riceverne i messaggi
55     public static int PORT_MS = 6000;
56     public static MulticastSocket MS = null;
57
58     //servizio remoto offerto dal server
59     public static ServerInterface SERVER_RMI;
60
61     //nickname dell'utente dopo che si è loggato
62     public static String NICKNAME = null;
63
64     //Porta e server socket channel per ricevere i file da altri utenti
65     public static long PORT_FILE = 0;
66     public static ServerSocketChannel SERVER_SOCKET_FILE = null;
67
68     //dove cercare i file da inviare
69     public static String PATH_MY_FILE =
70         "UserFiles" + File.separator + "MyFiles" + File.separator;
71     //dove salvare i file ricevuti
```

ClientMain.java

```
71  public static String PATH_FILE RECEIVED =
72      "UserFiles" + File.separator + "ReceivedFiles"+File.separator;
73
74
75  public static void main(String[] args) {
76      //l'host del server è passato come argomento
77      if (args.length != 1) {
78          System.out.println("Usage: java -jar SocialGossipClient.jar HostServer");
79          return;
80      }
81
82      //prendo l'host del server
83      HOSTNAME = args[0];
84
85      try {
86          //apro la prima connessione TCP verso il server
87          SOCKET = new Socket(HOSTNAME, PORT_TCP);
88          WRITER = new DataOutputStream(new BufferedOutputStream(SOCKET.getOutputStream()));
89          READER = new DataInputStream(new BufferedInputStream(SOCKET.getInputStream()));
90
91          //apro il multicast socket, utilizzato per le chatrooms
92          MS = new MulticastSocket(PORT_MS);
93
94          //prendo dal registry l'oggetto esportato dal server per
95          //la registrazione del client alle notifiche
96          Registry registry = LocateRegistry.getRegistry(PORT_RMI);
97          SERVER_RMI = (ServerInterface) registry.lookup("Server");
98
99          //apro il server socket channel per la ricezione dei file
100         SERVER_SOCKET_FILE = ServerSocketChannel.open();
101         ServerSocket ss = SERVER_SOCKET_FILE.socket();
102         InetSocketAddress address = new InetSocketAddress(0);
103         ss.bind(address);
104         PORT_FILE = ss.getLocalPort();
105
106         //faccio partire la schermata di login
107         LoginGUI loginGUI = new LoginGUI();
108         loginGUI.setVisible(true);
109     }
110     catch (Exception e) {
111         //per qualsiasi cosa non vada bene
112         e.printStackTrace();
113         //termino il client
114         cleanUp();
115     }
116 }
117
118 /**
119 * Metodo statico per chiudere tutte le connessioni verso il server
120 * e far terminare il client.
121 */
122
123 public static void cleanUp() {
124     //chiudo tutti gli streams ed i socket
125     try {
126         if (WRITER!=null) WRITER.close();
127         if (READER!=null) READER.close();
```

ClientMain.java

```
128         if (SOCKET!=null) SOCKET.close();
129         if (WRITER_MSG!=null) WRITER.close();
130         if (READER_MSG!=null) READER_MSG.close();
131         if (SOCKET_MSG!=null) SOCKET_MSG.close();
132         if (MS!=null) MS.close();
133         if (SERVER_SOCKET_FILE!=null) SERVER_SOCKET_FILE.close();
134     }
135     catch (Exception e) {}
136
137     //termino tutti gli eventuali threads
138     System.exit(0);
139 }
140
141 }
142
```

Operations.java

```
1 package enumerations;
2
3
4 /**
5 * Enumerazione Operations.
6 * Contiene l'enumerazione di tutte le operazioni richieste dal client
7 * ed i possibili responsi inviati dal server.
8 * @author Emilio Panti mat:531844
9 */
10 public enum Operations {
11
12     //POSSIBILI RESPONSI DAL SERVER
13     OP_OK,
14     OP_ERR,
15
16     //OPERAZIONI RIGUARDANTI L'UTENTE
17     REG,           //registrazione
18     LOG,           //login
19     CLOSE,          //chiusura connessione
20     CONN_MSG,        //connessione per ricevere messaggi dagli altri utenti
21
22     //OPERAZIONI VERSO GLI AMICI
23     LOOKUP,         //ricerca di un nickname
24     FRIENDSHIP,      //richiedere l'amicizia verso un utente
25     LISTFRIEND,     //richiedere la lista degli amici
26     STARTCHAT,       //aprire una chat con un amico
27     MSG_FRIEND,      //mandare un messaggio testuale ad un amico
28     FILE_FRIEND,     //mandare un file ad un amico
29
30     //OPERAZIONI VERSO LE CHATROOMS
31     CREATE_CHATROOM, //creare una chatroom
32     ADDME_CHATROOM,  //aggiungersi ad una chatroom
33     CHATLIST,         //ottenere la lista delle chatrooms
34     CLOSE_CHATROOM,   //chiudere una chatroom
35     MSG_CHATROOM      //mandare un messaggio testuale su una chatroom
36 }
37
```

ChatHandlerGUI.java

```
1 package graphicInterfaces;
2
3 import java.awt.event.WindowAdapter;
17
18
19 /**
20 * Classe ChatHandlerGUI.
21 * Offre una interfaccia grafica che permette all'utente di visionare e
22 * controllare tutte le chat che ha aperto verso altri utenti o verso
23 * le chatrooms a cui è iscritto.
24 * Le chat verso gli utenti sono oggetti della classe ChatNicknameGUI.
25 * Le chat verso le chatrooms sono oggetti della classe ChatroomGUI.
26 * Un oggetto di questa classe viene creato dopo una operazione di login o
27 * di registrazione effettuata dall'utente.
28 * Questa interfaccia grafica diventa visibile all'utente solamente quando
29 * c'è almeno una chat attiva.
30 * Se una chatroom a cui l'utente è iscritto viene chiusa dal suo creatore
31 * questa classe procede alla chiusura della chat relativa.
32 * Se l'utente invia un messaggio ad un utente che non è più online la chat
33 * verso tale utente viene chiusa in automatico.
34 * NB: la chiusura di questa interfaccia grafica da parte dell'utente
35 *      causa la terminazione del client.
36 * @author Emilio Panti mat:531844
37 */
38 public class ChatHandlerGUI extends JFrame {
39
40     private static final long serialVersionUID = 1L;
41     private JTabbedPane tabbedPane;
42
43     //numero delle chat aperte, se = 0 l'interfaccia diventa invisibile
44     private int numberChats = 0;
45
46     //variabile dove salvo questo gestore chats per passarla come parametro ai listener
47     private ChatHandlerGUI chatHandlerGUI;
48
49     //multicast socket del client, usato per disiscriversi dalle chatroom quando
50     //vengono chiuse
51     private MulticastSocket ms = null;
52
53
54     /**
55      * Costruttore classe ChatHandlerGUI.
56      */
57     public ChatHandlerGUI() {
58         chatHandlerGUI = this;
59         ms = ClientMain.MS;
60
61         setTitle("CHAT HANDLER: [" + ClientMain.NICKNAME + "]");
62         setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
63         //creo un window listener per personalizzare la gestione della chiusura
64         //dell'interfaccia
65         WindowListener exitListener = new WindowAdapter() {
66             @Override
67             public void windowClosing(WindowEvent e) {
68                 //faccio partire il thread che si occupa di chiudere la connessione e
69                 //terminare il client
70                 CloseThread closeThread = new CloseThread();
```

ChatHandlerGUI.java

```
70         Thread thread = new Thread(closeThread);
71         thread.start();
72     }
73 }
74 addWindowListener(exitListener);
75
76 setBounds(100, 100, 483, 407);
77 tabbedPane = new JTabbedPane(JTabbedPane.TOP);
78 tabbedPane.setBorder(new EmptyBorder(5, 5, 5, 5));
79 setContentPane(tabbedPane);
80 }
81
82 /**
83 * Aggiunge una chat verso l'utente che ha per nickname il parametro passato.
84 * @param String nickname: stringa contenente il nickname dell'utente.
85 * @param String msg: primo messaggio ricevuto (se è l'utente a mandare il primo messaggio
86 * è null)
87 */
88 synchronized public void addChatNickname (String nickname, String msg) {
89     int index = tabbedPane.indexOfTab("U:[" + nickname + "]");
90     //se non c'è già una chat aperta verso tale utente
91     if (index== -1) {
92         tabbedPane.addTab("U:[" + nickname + "]", new
93             ChatNicknameGUI(chatHandlerGUI, nickname, msg));
94         if(numberChats==0) this.setVisible(true);
95         numberChats++;
96     }
97
98 /**
99 * Aggiunge una chat verso la chatroom passata da parametro.
100 * @param String chatroom: stringa contenente il nome della chatroom.
101 * @param InetAddress address: indirizzo multicast della chatroom.
102 */
103 synchronized public void addChatroom (String chatroom, InetAddress address) {
104     int index = tabbedPane.indexOfTab("C:[" + chatroom + "]");
105     //se non c'è già una chat aperta verso tale chatroom
106     if (index== -1) {
107         tabbedPane.addTab("C:[" + chatroom + "]", new ChatroomGUI(chatroom, address));
108         if(numberChats==0) this.setVisible(true);
109         numberChats++;
110     }
111 }
112
113
114 /**
115 * Rimuove una chat verso l'utente che ha per nickname il parametro passato.
116 * @param String nickname: stringa contenente il nickname dell'utente.
117 */
118 synchronized public void removeChatNickname (String nickname) {
119     int index = tabbedPane.indexOfTab("U:[" + nickname + "]");
120     //se esiste una chat verso tale utente
121     if (index!= -1) {
122         tabbedPane.remove(index);
123         numberChats--;
124     }
```

ChatHandlerGUI.java

```
125         if(numberChats==0) this.setVisible(false);
126     }
127 }
128
129
130 /**
131 * Rimuove la chat relativa alla chatroom passata da parametro.
132 * @param String chatroom: stringa contenente il nome della chatroom.
133 * @param InetAddress address: address della chatroom.
134 */
135 synchronized public void removeChatroom (String chatroom,InetAddress address) {
136     int index = tabbedPane.indexOfTab("C:[ "+chatroom+"]");
137     //se esiste una chat verso tale chatroom
138     if (index!=-1) {
139         try {
140             ms.leaveGroup(address);
141         } catch (IOException e) {
142             e.printStackTrace();
143         }
144         tabbedPane.remove(index);
145         numberChats--;
146         if(numberChats==0) this.setVisible(false);
147     }
148 }
149
150
151 /**
152 * Posta il messaggio ricevuto da un utente nella relativa chat.
153 * Se la chat non è stata ancora aperta viene creata ed aggiunta a tabbedPane
154 * @param String nickname: stringa contenente il nickname dell'utente.
155 * @param String msg: messaggio da postare nella chat.
156 */
157 public void postMsgChatNickname (String nickname,String msg) {
158     ChatNicknameGUI chat = null;
159     synchronized (this) {
160         //controllo se esiste o meno una chat aperta verso tale utente
161         int index = tabbedPane.indexOfTab("U:[ "+nickname+"]");
162         if (index!=-1) chat = (ChatNicknameGUI) tabbedPane.getComponentAt(index);
163     }
164     //se ho già una chat verso tale utente mando il messaggio
165     if (chat!=null) chat.appendTextChat(msg);
166     //altrimenti creo una chat verso tale utente
167     else addChatNickname (nickname,msg);
168 }
169
170
171 /**
172 * Posta il messaggio ricevuto da una chatroom nella relativa chat.
173 * E' possibile che un messaggio ricevuto da una chatroom venga gestito
174 * dopo una sua eventuale chiusura, in questo caso scarto il messaggio.
175 * @param String chatroom: stringa contenente il nome della chatroom.
176 * @param String msg: messaggio da postare nella chat.
177 */
178 public void postMsgChatroom (String chatroom,String msg) {
179     ChatroomGUI chat = null;
180     synchronized (this) {
181         //controllo se esiste o meno una chat aperta della relativa chatroom
```

ChatHandlerGUI.java

```
182         int index = tabbedPane.indexOfTab("C:[" + chatroom + "]");
183         if (index != -1) chat = (ChatroomGUI) tabbedPane.getComponentAt(index);
184     }
185     //se ho una chat aperta relativa a tale chatroom
186     if (chat != null) chat.appendTextChat(msg);
187 }
188
189 }
190
```

ChatNicknameGUI.java

```
1 package graphicInterfaces;
2
3 import javax.swing.JPanel;
18
19
20 /**
21 * Classe ChatNicknameGUI.
22 * Offre una interfaccia grafica della chat verso un utente.
23 * Gli oggetti di questa classe vengono mostrati all'utente tramite
24 * l'interfaccia del gestore delle chat (ChatHandlerGUI).
25 * Permette di inviare sia messaggi testuali che file.
26 * @author Emilio Panti mat:531844
27 */
28 public class ChatNicknameGUI extends JPanel {
29
30     private static final long serialVersionUID = 1L;
31
32     //caratteri max per un messaggio
33     protected static int MAX_CAR = 150;
34
35     private JTextArea textChat;
36
37     //variabile dove salvo questa chat interface per passarla come parametro ai listener
38     private ChatNicknameGUI chatNicknameGUI;
39
40     //interfaccia gestore delle chats
41     @SuppressWarnings("unused")
42     private ChatHandlerGUI chatHandlerGUI;
43
44
45     /**
46      * Costruttore classe ChatNicknameGUI.
47      * @param ChatHandlerGUI chatHandlerGUI: interfaccia che raccoglie tutte le chat aperte
48      * dell'utente.
49      * @param String nickname: nickname dell'utente con cui è aperta la chat.
50      * @param String msg: primo messaggio ricevuto (se è l'utente a mandare il primo messaggio
51      * è null).
52      */
53     public ChatNicknameGUI(ChatHandlerGUI chatHandlerGUI, String nickname, String msg) {
54         this.chatHandlerGUI = chatHandlerGUI;
55         this.chatNicknameGUI = this;
56
57         setLayout(null);
58
59         //LABEL NICKNAME
60         JLabel lblNickname = new JLabel("Nickname ["+nickname+"]:");
61         lblNickname.setBounds(10, 11, 430, 22);
62         add(lblNickname);
63
64         //TEXT AREA CHAT
65         textChat = new JTextArea();
66         textChat.setFont(new Font("Monospaced", Font.PLAIN, 11));
67         textChat.setEditable(false);
68         textChat.setLineWrap(true);
69         JScrollPane scrolltextChat = new JScrollPane(textChat);
70         scrolltextChat.setBounds(10, 32, 430, 192);
71         add(scrolltextChat);
```

ChatNicknameGUI.java

```
70     //appendo il messaggio passato da parametro
71     if (msg!=null) appendTextChat(msg);
72
73     //LABEL INFO
74     JLabel lblInfo = new JLabel("Text message (or file name):");
75     lblInfo.setFont(new Font("Tahoma", Font.PLAIN, 9));
76     lblInfo.setBounds(10, 245, 158, 14);
77     add(lblInfo);
78
79     //TEXT AREA MSG
80     JTextArea textMsg = new JTextArea();
81     textMsg.setFont(new Font("Monospaced", Font.PLAIN, 11));
82     textMsg.setLineWrap(true);
83     JScrollPane scrolltextMsg = new JScrollPane(textMsg);
84     scrolltextMsg.setBounds(10, 261, 315, 60);
85     add(scrolltextMsg);
86
87     //BUTTON MSG
88     JButton btnMsg = new JButton("SEND MSG");
89     btnMsg.setBounds(335, 262, 105, 23);
90     add(btnMsg);
91
92     //BUTTON FILE
93     JButton btnFile = new JButton("SEND FILE");
94     btnFile.setBounds(335, 296, 105, 23);
95     add(btnFile);
96
97     JLabel lblCheckMsg = new JLabel("");
98     lblCheckMsg.setFont(new Font("Tahoma", Font.PLAIN, 9));
99     lblCheckMsg.setForeground(Color.RED);
100    lblCheckMsg.setHorizontalAlignment(SwingConstants.RIGHT);
101    lblCheckMsg.setBounds(185, 245, 140, 14);
102    add(lblCheckMsg);
103
104
105    //-----ALL LISTENER-----
106
107    //LISTENER FIELD NICKNAME
108    textMsg.addMouseListener(new MouseAdapter() {
109        //quando 'ento' nell'area di textMsg con il mouse
110        @Override
111        public void mouseEntered(MouseEvent e) {
112            lblCheckMsg.setText("(max "+MAX_CAR+" characters)");
113        }
114        //quando 'esco' dall'area di textMsg con il mouse
115        @Override
116        public void mouseExited(MouseEvent e) {
117            //controllo che il messaggio scritto non sia più lungo di MAX_CAR caratteri
118            if (textMsg.getText().length() <= MAX_CAR) lblCheckMsg.setText("");
119            else lblCheckMsg.setText("Message too long!");
120        }
121    });
122
123    //LISTENER SEND MSG
124    btnMsg.addMouseListener(new MouseAdapter() {
125        @Override
126        public void mouseClicked(MouseEvent arg0) {
```

ChatNicknameGUI.java

```
127     String msg = textMsg.getText();
128     //controllo che il messaggio scritto non sia più lungo di MAX_CAR
129     //caratteri e che non sia vuoto
130     if (msg.length()<=MAX_CAR && msg.length()>0) {
131         //avvio il thread che si occupa di inviare il messaggio all'amico
132         MsgFriendThread msgFriendThread =
133             new MsgFriendThread(chatNicknameGUI,chatHandlerGUI, nickname,
134             msg);
135         Thread thread = new Thread(msgFriendThread);
136         thread.start();
137         //ripulisco la textMsg
138         textMsg.setText("");
139     }
140 }
141 });
142
143 //LISTENER SEND FILE
144 btnFile.addMouseListener(new MouseAdapter() {
145     @Override
146     public void mouseClicked(MouseEvent e) {
147         String file = textMsg.getText();
148         //controllo che il file name non sia più lungo di MAX_CAR
149         //caratteri e che non sia vuoto
150         if (file.length()<=MAX_CAR && file.length()>0) {
151             //avvio il thread che si occupa di inviare il file all'amico
152             FileFriendThread fileFriendThread =
153                 new FileFriendThread(chatNicknameGUI,chatHandlerGUI,nickname,
154                 file);
155             Thread thread = new Thread(fileFriendThread);
156             thread.start();
157             //ripulisco la textMsg
158             textMsg.setText("");
159         }
160     }
161 });
162 }
163
164 /**
165 * Metodo per appendere una stringa all'area di testo della chat.
166 * @param String txt: stringa da appendere.
167 */
168 public void appendTextChat (String txt) {
169     //stringa end of line
170     String eol = System.getProperty("line.separator");
171     txt = txt + eol;
172
173     synchronized(textChat) {
174         textChat.append(txt);
175     }
176 }
177
178 }
```

ChatroomGUI.java

```
1 package graphicInterfaces;
2
3 import java.awt.Font;
17
18
19 /**
20 * Classe ChatroomGUI.
21 * Offre una interfaccia grafica della chat riguardante una chatroom
22 * a cui l'utente è iscritto.
23 * Gli oggetti di questa classe vengono mostrati all'utente tramite
24 * l'interfaccia del gestore delle chat (ChatHandlerGUI).
25 * Permette di inviare messaggi testuali alla chatroom.
26 * @author Emilio Panti mat:531844
27 */
28 public class ChatroomGUI extends JPanel {
29
30     private static final long serialVersionUID = 1L;
31     private JTextArea textChat;
32
33     //caratteri max per un messaggio
34     protected static int MAX_CAR = 150;
35
36
37     /**
38      * Costruttore classe ChatroomGUI.
39      * @param String chatroom: nome della chatroom.
40      * @param InetAddress address: inet address del gruppo multicast della chatroom.
41      */
42     public ChatroomGUI(String chatroom, InetAddress address) {
43         setLayout(null);
44
45         //LABEL CHATROOM
46         JLabel lblChatroom = new JLabel("Chatroom ["+chatroom+"]:");
47         lblChatroom.setBounds(10, 11, 430, 22);
48         add(lblChatroom);
49
50         //TEXT AREA CHAT
51         textChat = new JTextArea();
52         textChat.setFont(new Font("Monospaced", Font.PLAIN, 11));
53         textChat.setEditable(false);
54         textChat.setLineWrap(true);
55         JScrollPane scrolltextChat = new JScrollPane(textChat);
56         scrolltextChat.setBounds(10, 32, 430, 192);
57         add(scrolltextChat);
58
59         //LABEL INFO
60         JLabel lblInfo = new JLabel("Text message:");
61         lblInfo.setFont(new Font("Tahoma", Font.PLAIN, 9));
62         lblInfo.setBounds(10, 245, 105, 14);
63         add(lblInfo);
64
65         //TEXT AREA MSG
66         JTextArea textMsg = new JTextArea();
67         textMsg.setFont(new Font("Monospaced", Font.PLAIN, 11));
68         textMsg.setLineWrap(true);
69         JScrollPane scrolltextMsg = new JScrollPane(textMsg);
70         scrolltextMsg.setBounds(10, 261, 315, 60);
```

ChatroomGUI.java

```
71         add(scrolltextMsg);
72
73     //BUTTON SEND
74     JButton btnSend = new JButton("SEND");
75     btnSend.setBounds(335, 278, 105, 23);
76     add(btnSend);
77
78     JLabel lblCheckMsg = new JLabel("");
79     lblCheckMsg.setFont(new Font("Sylfaen", Font.PLAIN, 9));
80     lblCheckMsg.setHorizontalTextPosition(SwingConstants.RIGHT);
81     lblCheckMsg.setForeground(Color.RED);
82     lblCheckMsg.setBounds(162, 245, 163, 14);
83     add(lblCheckMsg);
84
85     //-----ALL LISTENER-----
86
87     //LISTENER FIELD NICKNAME
88     textMsg.addMouseListener(new MouseAdapter() {
89         //quando 'ento' nell'area di textMsg con il mouse
90         @Override
91         public void mouseEntered(MouseEvent e) {
92             lblCheckMsg.setText("(max "+MAX_CAR+" characters)");
93         }
94         //quando 'esco' dall'area di textMsg con il mouse
95         @Override
96         public void mouseExited(MouseEvent e) {
97             //controllo che il messaggio scritto non sia più lungo di MAX_CAR caratteri
98             if (textMsg.getText().length() <= MAX_CAR) lblCheckMsg.setText("");
99             else lblCheckMsg.setText("Message too long!");
100        }
101    });
102
103    //LISTENER SEND
104    btnSend.addMouseListener(new MouseAdapter() {
105        @Override
106        public void mouseClicked(MouseEvent arg0) {
107            String msg = textMsg.getText();
108            //controllo che il messaggio scritto non sia più lungo di MAX_CAR
109            //caratteri e che non sia vuoto
110            if (msg.length()<=MAX_CAR && msg.length()>0) {
111                //avvio il thread che si occupa di inviare il messaggio
112                //alla chatroom
113                MsgCRThread msgCRThread =
114                    new MsgCRThread(chatroom, address, msg);
115                Thread thread = new Thread(msgCRThread);
116                thread.start();
117
118                //ripulisco la textMsg
119                textMsg.setText("");
120            }
121        }
122    });
123 }
124
125 /**
126 * Metodo per appendere una stringa all'area di testo della chat.
127 * @param String txt: stringa da appendere.
```

ChatroomGUI.java

```
128 */  
129 public void appendTextChat(String txt) {  
130     //string a end of line  
131     String eol = System.getProperty("line.separator");  
132     txt = txt + eol;  
133  
134     synchronized(textChat) {  
135         textChat.append(txt);  
136     }  
137 }  
138  
139 }  
140
```

LoginGUI.java

```
1 package graphicInterfaces;
2
3
4 import javax.swing.JFrame;
5
6
7 /**
8  * Classe LoginGUI.
9  * Interfaccia grafica per il login dell'utente al servizio Social Gossip.
10 * Quando l'utente clicca sul bottone per il login viene
11 * creato e fatto partire il thread (oggetto della classe LoginThread)
12 * che si occuperà di effettuare la richiesta al server.
13 * Offre anche la possibilità di accedere all'interfaccia grafica
14 * per la registrazione a SocialGossip.
15 * NB: la chiusura di questa interfaccia grafica da parte dell'utente
16 * causa la terminazione del client.
17 * @author Emilio Panti mat:531844
18 */
19 public class LoginGUI extends JFrame {
20
21     private static final long serialVersionUID = 1L;
22
23     //lunghezza massima per i campi nickname e password
24     static int MAX_LENGTH = 16;
25
26     JPanel contentPane;
27     JTextField fieldNickname;
28     JPasswordField fieldPassword;
29     JButton btnLogin;
30
31     //variabile dove salvo l'interfaccia di login creata per
32     //passarla ai metodi invocati dai vari listener.
33     private LoginGUI loginGUI;
34
35     /**
36      * Costruttore classe LoginGUI.
37      */
38     public LoginGUI() {
39         loginGUI = this;
40
41         setTitle("SOCIAL GOSSIP : login");
42         setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
43         //creo un window listener per personalizzare la gestione della chiusura
44         //dell'interfaccia
45         WindowListener exitListener = new WindowAdapter() {
46             @Override
47             public void windowClosing(WindowEvent e) {
48                 //faccio partire il thread che si occupa di chiudere la connessione e il
49                 //client
50                 CloseThread closeThread = new CloseThread();
51                 Thread thread = new Thread(closeThread);
52                 thread.start();
53             }
54         };
55         addWindowListener(exitListener);
56     }
57 }
```

LoginGUI.java

```
74     setBounds(100, 100, 397, 310);
75     contentPane = new JPanel();
76     contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
77     setContentPane(contentPane);
78     contentPane.setLayout(null);
79
80     //LABEL WELCOME
81     JLabel lblWelcome = new JLabel("WELCOME TO SOCIAL GOSSIP");
82     lblWelcome.setHorizontalAlignment(SwingConstants.CENTER);
83     lblWelcome.setFont(new Font("Tahoma", Font.PLAIN, 14));
84     lblWelcome.setBounds(87, 26, 200, 14);
85     contentPane.add(lblWelcome);
86
87     //LABEL NICKNAME
88     JLabel lblNickname = new JLabel("Nickname:");
89     lblNickname.setBounds(61, 61, 63, 24);
90     contentPane.add(lblNickname);
91
92     //FIELD NICKNAME
93     JTextField fieldNickname = new JTextField();
94     fieldNickname.setBounds(134, 63, 173, 20);
95     contentPane.add(fieldNickname);
96     fieldNickname.setColumns(10);
97
98     //LABEL PASSWORD
99     JLabel lblPassword = new JLabel("Password:");
100    lblPassword.setBounds(61, 96, 63, 14);
101    contentPane.add(lblPassword);
102
103    //FIELD PASSWORD
104    JPasswordField fieldPassword = new JPasswordField();
105    fieldPassword.setBounds(134, 94, 173, 20);
106    contentPane.add(fieldPassword);
107
108    //BUTTON LOGIN
109    JButton btnLogin = new JButton("LOGIN");
110    btnLogin.setBounds(121, 127, 135, 23);
111    contentPane.add(btnLogin);
112
113    //LINE SEPARATOR
114    JSeparator separator = new JSeparator();
115    separator.setBounds(-11, 175, 404, 2);
116    contentPane.add(separator);
117
118    //INFO LABEL ("Not yet registered? Register now")
119    JLabel lblInfo = new JLabel("Not yet registered? Register now!");
120    lblInfo.setFont(new Font("Tahoma", Font.PLAIN, 10));
121    lblInfo.setBounds(113, 197, 155, 14);
122    contentPane.add(lblInfo);
123
124    //BUTTON REGISTER
125    JButton btnRegister = new JButton("REGISTER NOW");
126    btnRegister.setBounds(121, 222, 135, 23);
127    contentPane.add(btnRegister);
128
129
130
```

LoginGUI.java

```
131     //-----ALL LISTENER-----
132
133
134     //LISTENER BUTTON LOGIN
135     btnLogin.addMouseListener(new MouseAdapter() {
136         @Override
137         public void mouseClicked(MouseEvent arg0) {
138
139             //prendo il nickname e la psw
140             String nickname = fieldNickname.getText();
141             String psw = new String(fieldPassword.getPassword());
142
143             //controllo che il nickname e la password siano della lunghezza giusta
144             if (checkLenght(nickname)&&checkLenght(psw)) {
145
146                 //disabilito il bottone
147                 btnLogin.setEnabled(false);
148
149                 //chiama l'operazione di login
150                 LoginThread loginThread = new LoginThread(loginGUI, nickname, psw);
151                 Thread thread = new Thread(loginThread);
152                 thread.start();
153
154             }
155             //comunico l'errore all'utente senza fare richieste inutili al server
156             else {
157                 ResponseGUI responseGUI = new ResponseGUI("ERR: Nickname or password
158                 incorrect");
159                 responseGUI.setVisible(true);
160             }
161         }
162     });
163
164
165     //LISTENER BUTTON REGISTER NOW
166     btnRegister.addMouseListener(new MouseAdapter() {
167         @Override
168         public void mouseClicked(MouseEvent e) {
169             //apro l'interfaccia per la registrazione
170             RegistrationGUI frame = new RegistrationGUI();
171             frame.setVisible(true);
172
173             //chiudo la login interface
174             dispose();
175         }
176     });
177
178 }
179
180 /**
181 * Controlla la lunghezza della stringa passata.
182 * @param String str: stringa
183 * @return boolean: true se la stringa è più corta di MAX_LENGTH,
184 * false altrimenti
185 */
186 */
```

LoginGUI.java

```
187     private boolean checkLength(String str){  
188         return (str.length()<=MAX_LENGTH && str.length()>0);  
189     }  
190  
191  
192     /**  
193      * Attiva il bottone per il login.  
194      */  
195     public void enabledBtnLogin() {  
196         btnLogin.setEnabled(true);  
197     }  
198 }  
199
```

OperativeGUI.java

```
1 package graphicInterfaces;
2
3 import javax.swing.JFrame;
25
26
27 /**
28 * Classe OperativeGUI.
29 * Offre l'interfaccia grafica operativa del servizio SocialGossip.
30 * Permette all'utente di effettuare le operazioni di:
31 * - aggiornamento lista amici,
32 * - look up di un nickname,
33 * - amicizia
34 * - apertura chat verso un amico,
35 * - aggiornamento lista chatrooms,
36 * - creazione chatroom,
37 * - iscrizione ad una chatroom,
38 * - chiusura di una chatroom.
39 * Mostra all'utente:
40 * - la lista amici,
41 * - la lista delle chatrooms (indicando a quali di esse è iscritto),
42 * - le notifiche che lo riguardano.
43 * Un oggetto di questa classe viene creato dopo una operazione di login o
44 * di registrazione effettuata dall'utente.
45 * NB: la chiusura di questa interfaccia grafica da parte dell'utente
46 *      causa la terminazione del client.
47 * @author Emilio Panti mat:531844
48 */
49 public class OperativeGUI extends JFrame {
50
51     private static final long serialVersionUID = 1L;
52     private JPanel contentPane;
53     private JTextArea textFriendsList;
54     private JTextArea textChatrooms;
55     private JTextArea textNotifications;
56     private JButton btnUpdatesFriends;
57     private JButton btnUpdatesChatrooms;
58
59     //Interfaccia che gestisce le chat
60     @SuppressWarnings("unused")
61     private ChatHandlerGUI chatHandlerGUI;
62
63     //variabile dove salvo questa interaccia grafica per poi passarla
64     //come parametro anche all'interno dei vari listener
65     private OperativeGUI operativeGUI;
66
67     //nickname dell'utente
68     private String nickname;
69
70     /**
71      * Costruttore classe OperativeGUI.
72      * @param: ChatHandlerGUI chatHandlerGUI: gestore delle chat aperte.
73      */
74     public OperativeGUI(ChatHandlerGUI chatHandlerGUI) {
75         this.operativeGUI = this;
76         this.chatHandlerGUI = chatHandlerGUI;
77         nickname = ClientMain.NICKNAME;
78     }
```

OperativeGUI.java

```
79      setTitle("SOCIAL GOSSIP: ["+nickname+"]");
80      setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
81      //creo un window listener per personalizzare la gestione della chiusura
82      //dell'interfaccia
83      WindowListener exitListener = new WindowAdapter() {
84          @Override
85          public void windowClosing(WindowEvent e) {
86              //faccio partire il thread che si occupa di chiudere la connessione e il
87              //client
88              CloseThread closeThread = new CloseThread();
89              Thread thread = new Thread(closeThread);
90              thread.start();
91          }
92      };
93      addWindowListener(exitListener);
94
95      setBounds(100, 100, 765, 404);
96      contentPane = new JPanel();
97      contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
98      setContentPane(contentPane);
99      contentPane.setLayout(null);
100
101     //LABEL FRIENDS LIST
102     JLabel lblFriendsList = new JLabel("Friends list:");
103     lblFriendsList.setBounds(40, 27, 70, 14);
104     contentPane.add(lblFriendsList);
105
106     //BUTTON UPDATES LIST FRIENDS
107     JButton btnUpdatesFriends = new JButton("Updates");
108     btnUpdatesFriends.setBounds(107, 23, 92, 23);
109     contentPane.add(btnUpdatesFriends);
110
111     //TEXT AREA FRIENDS LIST
112     JTextArea textFriendsList = new JTextArea();
113     textFriendsList.setEditable(false);
114     JScrollPane scrollFriendsList = new JScrollPane(textFriendsList);
115     scrollFriendsList.setFont(new Font("Monospaced", Font.PLAIN, 11));
116     scrollFriendsList.setBounds(40, 52, 159, 200);
117     contentPane.add(scrollFriendsList);
118
119     //BUTTON ADD FRIEND
120     JButton btnAddNewFriend = new JButton("Add new friend");
121     btnAddNewFriend.setBounds(40, 260, 159, 23);
122     contentPane.add(btnAddNewFriend);
123
124     //BUTTON LOOK-UP NICKNAME
125     JButton btnLookUpNickname = new JButton("Look-up nickname");
126     btnLookUpNickname.setBounds(40, 286, 159, 23);
127     contentPane.add(btnLookUpNickname);
128
129     //BUTTON START CHAT
130     JButton btnStartChat = new JButton("START CHAT");
131     btnStartChat.setBounds(40, 313, 159, 23);
132     contentPane.add(btnStartChat);
133
134     //SEPARATOR
135     JSeparator separator_1 = new JSeparator();
```

OperativeGUI.java

```
134 separator_1.setOrientation(SwingConstants.VERTICAL);
135 separator_1.setBounds(239, 0, 2, 394);
136 contentPane.add(separator_1);
137
138 //LABEL CHATROOMS
139 JLabel lblChatrooms = new JLabel("Chatrooms:");
140 lblChatrooms.setBounds(281, 27, 70, 14);
141 contentPane.add(lblChatrooms);
142
143 //BUTTON UPDATES CHATROOMS
144 JButton btnUpdatesChatrooms = new JButton("Updates");
145 btnUpdatesChatrooms.setBounds(348, 23, 92, 23);
146 contentPane.add(btnUpdatesChatrooms);
147
148 //TEXT AREA CHATROOMS
149 JTextArea textChatrooms = new JTextArea();
150 textChatrooms.setEditable(false);
151 JScrollPane scrollChatrooms = new JScrollPane(textChatrooms);
152 textChatrooms.setFont(new Font("Monospaced", Font.PLAIN, 11));
153 scrollChatrooms.setBounds(281, 52, 159, 200);
154 contentPane.add(scrollChatrooms);
155
156 //BUTTON CREATE CHATROOM
157 JButton btnCreateChatroom = new JButton("Create chatroom");
158 btnCreateChatroom.setBounds(281, 260, 159, 23);
159 contentPane.add(btnCreateChatroom);
160
161 //BUTTON ADD TO CHATROOM
162 JButton btnAddToChatroom = new JButton("Add to a chatroom");
163 btnAddToChatroom.setBounds(281, 286, 159, 23);
164 contentPane.add(btnAddToChatroom);
165
166 //BUTTON CLOSE CHATROOM
167 JButton btnCloseChatroom = new JButton("Close chatroom");
168 btnCloseChatroom.setBounds(281, 313, 159, 23);
169 contentPane.add(btnCloseChatroom);
170
171 //SEPARATOR
172 JSeparator separator = new JSeparator();
173 separator.setOrientation(SwingConstants.VERTICAL);
174 separator.setBounds(481, 0, 2, 394);
175 contentPane.add(separator);
176
177 //LABEL NOTIFICATIONS
178 JLabel lblNewNotifications = new JLabel("New notifications:");
179 lblNewNotifications.setBounds(523, 27, 108, 14);
180 contentPane.add(lblNewNotifications);
181
182 //TEXT AREA NOTIFICATIONS
183 JTextArea textNotifications = new JTextArea();
184 textNotifications.setEditable(false);
185 JScrollPane scrollNotification = new JScrollPane(textNotifications);
186 textNotifications.setFont(new Font("Monospaced", Font.PLAIN, 11));
187 scrollNotification.setBounds(523, 52, 188, 284);
188 contentPane.add(scrollNotification);
189
190
```

OperativeGUI.java

```
191     //-----ALL LISTENER-----
192
193
194     //LISTENER UPDATES FRIENDS LIST
195     btnUpdatesFriends.addMouseListener(new MouseAdapter() {
196         @Override
197         public void mouseClicked(MouseEvent arg0) {
198             //disabilito il bottone
199             btnUpdatesFriends.setEnabled(false);
200
201             //attivo il thread che si occupa di richiedere la lista amici
202             FriendsListThread friendsListThread =
203                 new FriendsListThread(operativeGUI);
204             Thread thread = new Thread(friendsListThread);
205             thread.start();
206         }
207     });
208
209
210     //LISTENER ADD FRIEND
211     btnAddNewFriend.addMouseListener(new MouseAdapter() {
212         @Override
213         public void mouseClicked(MouseEvent e) {
214             //apro l'interfaccia per la richiesta e passo al costruttore la relativa
215             operazione
216             RequestGUI frame = new
217             RequestGUI(Operations.FRIENDSHIP,operativeGUI,chatHandlerGUI);
218             frame.setVisible(true);
219         }
220     });
221
222     //LISTENER LOOK-UP NICKNAME
223     btnLookUpNickname.addMouseListener(new MouseAdapter() {
224         @Override
225         public void mouseClicked(MouseEvent arg0) {
226             //apro l'interfaccia per la richiesta e passo al costruttore la relativa
227             operazione
228             RequestGUI frame = new
229             RequestGUI(Operations.LOOKUP,operativeGUI,chatHandlerGUI);
230             frame.setVisible(true);
231         }
232     });
233
234     //LISTENER START CHAT
235     btnStartChat.addMouseListener(new MouseAdapter() {
236         @Override
237         public void mouseClicked(MouseEvent e) {
238             //apro l'interfaccia per la richiesta e passo al costruttore la relativa
239             operazione
240             RequestGUI frame = new
241             RequestGUI(Operations.STARTCHAT,operativeGUI,chatHandlerGUI);
242             frame.setVisible(true);
243         }
244     });
245
```

OperativeGUI.java

```
242
243     //LISTENER UPDATES CHATROOMS LIST
244     btnUpdatesChatrooms.addMouseListener(new MouseAdapter() {
245         @Override
246         public void mouseClicked(MouseEvent arg0) {
247             //disabilito il bottone
248             btnUpdatesChatrooms.setEnabled(false);
249
250             //attivo il thread che si occupa di chiedere la lista amici
251             CRLListThread cRLListThread =
252                 new CRLListThread(operativeGUI);
253             Thread thread = new Thread(cRLListThread);
254             thread.start();
255         }
256     });
257
258
259     //LISTENER CREATE CHATROOM
260     btnCreateChatroom.addMouseListener(new MouseAdapter() {
261         @Override
262         public void mouseClicked(MouseEvent e) {
263             //apro l'interfaccia per la richiesta e passo al costruttore la relativa
264             operazione
265             RequestGUI frame = new
266             RequestGUI(Operations.CREATE_CHATROOM,operativeGUI,chatHandlerGUI);
267             frame.setVisible(true);
268         }
269     });
270
271     //LISTENER ADD TO CHATROOM
272     btnAddToChatroom.addMouseListener(new MouseAdapter() {
273         @Override
274         public void mouseClicked(MouseEvent arg0) {
275             //apro l'interfaccia per la richiesta e passo al costruttore la relativa
276             operazione
277             RequestGUI frame = new
278             RequestGUI(Operations.ADDME_CHATROOM,operativeGUI,chatHandlerGUI);
279             frame.setVisible(true);
280         }
281     });
282
283     //LISTENER CLOSE CHATROOM
284     btnCloseChatroom.addMouseListener(new MouseAdapter() {
285         @Override
286         public void mouseClicked(MouseEvent e) {
287             //apro l'interfaccia per la richiesta e passo al costruttore la relativa
288             operazione
289             RequestGUI frame = new
290             RequestGUI(Operations.CLOSE_CHATROOM,operativeGUI,chatHandlerGUI);
291             frame.setVisible(true);
292         }
293     });
294 }
```

OperativeGUI.java

```
293
294 /**
295  * Metodo per scrivere nell'area di testo relativa alle amicizie.
296  * @param String txt: stringa contenente il testo da scrivere.
297 */
298 public void setTextFriends (String txt) {
299     synchronized(textFriendsList) {
300         textFriendsList.setText(txt);
301     }
302 }
303
304
305 /**
306  * Metodo per scrivere nell'area di testo relativa alle chatrooms.
307  * @param String txt: stringa contenente il testo da scrivere.
308 */
309 public void setTextChatrooms (String txt) {
310     synchronized(textChatrooms) {
311         textChatrooms.setText(txt);
312     }
313 }
314
315
316 /**
317  * Metodo per appendere una stringa all'area di testo relativa alle amicizie.
318  * @param String txt: stringa da appendere.
319 */
320 public void appendTextFriends (String txt) {
321     //stringa end of line
322     String eol = System.getProperty("line.separator");
323     txt = txt + eol;
324
325     synchronized(textFriendsList) {
326         textFriendsList.append(txt);
327     }
328 }
329
330
331 /**
332  * Metodo per appendere una stringa all'area di testo relativa alle chatrooms.
333  * @param String txt: stringa da appendere.
334 */
335 public void appendTextChatrooms (String txt) {
336     //stringa end of line
337     String eol = System.getProperty("line.separator");
338     txt = txt + eol;
339
340     synchronized(textChatrooms) {
341         textChatrooms.append(txt);
342     }
343 }
344
345
346 /**
347  * Metodo per appendere una stringa all'area di testo relativa alle notifiche.
348  * @param String txt: stringa da appendere.
349 */
```

OperativeGUI.java

```
350 public void appendTextNotifications (String txt) {  
351     //stringa end of line  
352     String eol = System.getProperty("line.separator");  
353     txt = txt + eol;  
354  
355     synchronized(textNotifications) {  
356         textNotifications.append(txt);  
357     }  
358 }  
359  
360  
361 /**  
362 * Attiva il bottone per la richiesta della lista amici.  
363 */  
364 public void enabledBtnUpdatesFriends() {  
365     btnUpdatesFriends.setEnabled(true);  
366 }  
367  
368  
369 /**  
370 * Attiva il bottone per la richiesta della lista chatrooms.  
371 */  
372 public void enabledBtnUpdatesChatrooms() {  
373     btnUpdatesChatrooms.setEnabled(true);  
374 }  
375 }  
376
```

RegistrationGUI.java

```
1 package graphicInterfaces;
2
3 import java.awt.Font;
26
27
28 /**
29 * Classe RegistrationGUI.
30 * Interfaccia grafica per la registrazione dell'utente al servizio
31 * Social Gossip.
32 * Quando l'utente clicca sul bottone per la registrazione viene
33 * creato e fatto partire il thread (oggetto della classe RegistrationThread)
34 * che si occuperà di effettuare la richiesta al server.
35 * Offre anche la possibilità di accedere all'interfaccia grafica di login.
36 * NB: la chiusura di questa interfaccia grafica da parte dell'utente
37 * causa la terminazione del client.
38 * @author Emilio Panti mat:531844
39 */
40 public class RegistrationGUI extends JFrame {
41
42     //lunghezza massima per i campi nickname e password
43     static int MAX_LENGTH = 16;
44
45     private static final long serialVersionUID = 1L;
46     private JPanel contentPane;
47     private JTextField fieldNickname;
48     private JPasswordField fieldPassword;
49     private JPasswordField fieldPassword2;
50     private JButton btnRegister;
51     private JComboBox<String> comboBox;
52
53     /**
54      * variabile dove salvo l'interfaccia di registrazione creata per
55      * passarla ai metodi invocati dai vari listener.
56      */
57     private RegistrationGUI registrationGUI;
58
59
60     /**
61      * Costruttore classe RegistrationGUI.
62      */
63     public RegistrationGUI() {
64         registrationGUI = this;
65
66         setTitle("SOCIAL GOSSIP : registration");
67         setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
68         //creo un window listener per personalizzare la gestione della chiusura
69         //dell'interfaccia
70         WindowListener exitListener = new WindowAdapter() {
71             @Override
72             public void windowClosing(WindowEvent e) {
73                 //faccio partire il thread che si occupa di chiudere la connessione e il
74                 //client
75                 CloseThread closeThread = new CloseThread();
76                 Thread thread = new Thread(closeThread);
77                 thread.start();
78             }
79         };
80     }
81 }
```

RegistrationGUI.java

```
78     addWindowListener(exitListener);
79
80     setBounds(100, 100, 397, 336);
81     contentPane = new JPanel();
82     contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
83     setContentPane(contentPane);
84     contentPane.setLayout(null);
85
86     //LABEL NICKNAME
87     JLabel lblNickname = new JLabel("Nickname:");
88     lblNickname.setHorizontalAlignment(SwingConstants.RIGHT);
89     lblNickname.setBounds(34, 21, 100, 24);
90     contentPane.add(lblNickname);
91
92     //FIELD NICKNAME
93     fieldNickname = new JTextField();
94     fieldNickname.setBounds(144, 23, 172, 20);
95     contentPane.add(fieldNickname);
96     fieldNickname.setColumns(10);
97
98     //LABEL CHECK NICKNAME
99     JLabel lblCheckNickname = new JLabel("");
100    lblCheckNickname.setForeground(Color.RED);
101    lblCheckNickname.setFont(new Font("Tahoma", Font.PLAIN, 9));
102    lblCheckNickname.setBounds(134, 43, 172, 14);
103    contentPane.add(lblCheckNickname);
104
105    //LABEL PASSWORD
106    JLabel lblPassword = new JLabel("Password:");
107    lblPassword.setHorizontalAlignment(SwingConstants.RIGHT);
108    lblPassword.setBounds(20, 62, 114, 14);
109    contentPane.add(lblPassword);
110
111    //FIELD PASSWORD
112    fieldPassword = new JPasswordField();
113    fieldPassword.setBounds(144, 59, 172, 20);
114    contentPane.add(fieldPassword);
115
116    //LABEL CHECK PASSWORD
117    JLabel lblCheckPassword = new JLabel("");
118    lblCheckPassword.setForeground(Color.RED);
119    lblCheckPassword.setFont(new Font("Tahoma", Font.PLAIN, 9));
120    lblCheckPassword.setBounds(134, 79, 172, 14);
121    contentPane.add(lblCheckPassword);
122
123    //LABEL PASSWORD2
124    JLabel lblPassword2 = new JLabel("Confirm password:");
125    lblPassword2.setHorizontalAlignment(SwingConstants.RIGHT);
126    lblPassword2.setBounds(10, 98, 124, 14);
127    contentPane.add(lblPassword2);
128
129    //FIELD PASSWORD2
130    fieldPassword2 = new JPasswordField();
131    fieldPassword2.setBounds(144, 95, 172, 20);
132    contentPane.add(fieldPassword2);
133
134    //LABEL CHECK PASSWORD2
```

RegistrationGUI.java

```
135     JLabel lblCheckPassword2 = new JLabel("");
136     lblCheckPassword2.setForeground(Color.RED);
137     lblCheckPassword2.setFont(new Font("Tahoma", Font.PLAIN, 9));
138     lblCheckPassword2.setBounds(134, 114, 172, 14);
139     contentPane.add(lblCheckPassword2);
140
141     //LABEL LANGUAGE
142     JLabel lblLanguage = new JLabel("Language:");
143     lblLanguage.setBounds(71, 135, 63, 14);
144     contentPane.add(lblLanguage);
145
146     //JCOMBOBOX LANGUAGES
147     comboBox = new JComboBox<String>();
148     //prendo un array di String delle lingue in formato ISO e le inserisco nella combo box
149     String[] languages = java.util.Locale.getISOLanguages();
150     comboBox.setModel(new DefaultComboBoxModel<String>(languages));
151     comboBox.setBounds(144, 132, 48, 20);
152     contentPane.add(comboBox);
153
154     //BUTTON REGISTER
155     btnRegister = new JButton("REGISTER NOW");
156     btnRegister.setBounds(124, 170, 135, 23);
157     contentPane.add(btnRegister);
158
159     //LINE SEPARATOR
160     JSeparator separator = new JSeparator();
161     separator.setBounds(-11, 219, 404, 2);
162     contentPane.add(separator);
163
164     //INFO LABEL ("Are you already registered? Please login")
165     JLabel lblInfo = new JLabel("Are you already registered? Please login");
166     lblInfo.setFont(new Font("Tahoma", Font.PLAIN, 10));
167     lblInfo.setBounds(98, 232, 192, 14);
168     contentPane.add(lblInfo);
169
170     //BUTTON LOGIN
171     JButton btnLogin = new JButton("LOGIN");
172     btnLogin.setBounds(124, 257, 135, 23);
173     contentPane.add(btnLogin);
174
175
176
177     //-----ALL LISTENER-----
178
179
180     //LISTENER FIELD NICKNAME
181     fieldNickname.addMouseListener(new MouseAdapter() {
182         //quando 'entro' nell'area di fieldNickname con il mouse
183         @Override
184         public void mouseEntered(MouseEvent e) {
185             lblCheckNickname.setText("(max "+MAX_LENGTH+" characters)");
186         }
187         //quando 'esco' dall'area di fieldNickname con il mouse
188         @Override
189         public void mouseExited(MouseEvent e) {
190             //controllo che il nickname inserito dall'utente sia della lunghezza giusta
191             String nickname = fieldNickname.getText();
```

RegistrationGUI.java

```
192         if (nickname.length()<=MAX_LENGTH) lblCheckNickname.setText("");
193         else lblCheckNickname.setText("Nickname too long!");
194     }
195 });
196
197
198 //LISTENER FIELD PASSWORD
199 fieldPassword.addMouseListener(new MouseAdapter() {
200     //quando 'entro' nell'area di fieldPassword con il mouse
201     @Override
202     public void mouseEntered(MouseEvent e) {
203         lblCheckPassword.setText("(max "+MAX_LENGTH+" characters)");
204     }
205     //quando 'esco' dall'area di fieldPassword con il mouse
206     @Override
207     public void mouseExited(MouseEvent e) {
208         //controllo che la password inserita dall'utente sia della lunghezza giusta
209         String psw = new String(fieldPassword.getPassword());
210         if (psw.length()<=MAX_LENGTH) lblCheckPassword.setText("");
211         else lblCheckPassword.setText("Password too long!");
212     }
213 });
214
215
216 //LISTENER FIELD PASSWORD2
217 fieldPassword2.addMouseListener(new MouseAdapter() {
218     //quando 'entro' nell'area di fieldPassword2 con il mouse
219     @Override
220     public void mouseEntered(MouseEvent e) {
221         lblCheckPassword2.setText("(max "+MAX_LENGTH+" characters)");
222     }
223     //quando 'esco' dall'area di fieldPassword2 con il mouse
224     @Override
225     public void mouseExited(MouseEvent e) {
226         //controllo che le due password siano uguali
227         if
228             (checkEqualsPassword(fieldPassword.getPassword(),fieldPassword2.getPassword()))
229             lblCheckPassword2.setText("");
230         else lblCheckPassword2.setText("Passwords must match!");
231     }
232 });
233
234 //LISTENER BUTTON REGISTER NOW
235 btnRegister.addMouseListener(new MouseAdapter() {
236     @Override
237     public void mouseClicked(MouseEvent e) {
238         //prendo il nickname, le passwords e la lingua
239         String nickname = fieldNickname.getText();
240         String psw = new String(fieldPassword.getPassword());
241         String language = (String) comboBox.getSelectedItem();
242
243         //chiamo l'operazione di registrazione se tutti i campi sono corretti
244         if(checkLengthNickname(nickname) &&
245            checkLengthPassword(fieldPassword.getPassword())
246            &&
247            checkEqualsPassword(fieldPassword.getPassword(),fieldPassword2.getPassword()))
```

RegistrationGUI.java

```
245         ) {
246             //disabilito il bottone
247             btnRegister.setEnabled(false);
248
249             //chiama l'operazione di registrazione
250             RegistrationThread registrationThread =
251                 new RegistrationThread(registrationGUI, nickname, psw, language);
252             Thread thread = new Thread(registrationThread);
253             thread.start();
254         }
255         //altrimenti comunico l'errore all'utente senza fare richieste inutili al
256         //server
257     }  
258     else {
259         ResponseGUI responseGUI = new ResponseGUI("ERR: Nickname or password
260         incorrect");
261         responseGUI.setVisible(true);
262     }
263 }
264
265 //LISTENER BUTTON LOGIN
266 btnLogin.addMouseListener(new MouseAdapter() {
267     @Override
268     public void mouseClicked(MouseEvent arg0) {
269         //apro l'interfaccia per il login
270         LoginGUI frame = new LoginGUI();
271         frame.setVisible(true);
272
273         //chiudo la registration interface
274         dispose();
275     }
276 });
277
278
279
280
281 /**
282 * Controlla la lunghezza del nickname.
283 * @param String nickname: stringa contenente il nickname
284 * @return boolean: true se il nickname è più corto di MAX_LENGTH,
285 *                     false altrimenti
286 */
287 private boolean checkLengthNickname (String nickname){
288     return (nickname.length()<=MAX_LENGTH && nickname.length()>0);
289 }
290
291 /**
292 * Controlla la lunghezza di una password.
293 * @param char[] password: rappresenta la password
294 * @return boolean: true se la password è più corta di MAX_LENGTH,
295 *                     false altrimenti
296 */
297 private boolean checkLengthPassword (char[] password){
298     String psw = new String(password);
```

RegistrationGUI.java

```
300         return (psw.length()<=MAX_LENGTH && psw.length()>0);
301     }
302
303
304     /**
305      * Controlla che due password siano uguali.
306      * @param char[] password1: rappresenta la prima password
307      * @param char[] password2: rappresenta la seconda password
308      * @return boolean: true se le due passwords sono uguali,
309                      false altrimenti
310      */
311     private boolean checkEqualsPassword (char[] password1, char[] password2){
312         String psw1 = new String(password1);
313         String psw2 = new String(password2);
314         return (psw1.equals(psw2));
315     }
316
317
318     /**
319      * Attiva il bottone per la registrazione.
320      */
321     public void enabledBtnRegister() {
322         btnRegister.setEnabled(true);
323     }
324 }
```

RequestGUI.java

```
1 package graphicInterfaces;
2
3 import javax.swing.JFrame;
22
23
24 /**
25 * Classe RequestGUI.
26 * Interfaccia grafica per effettuare una richiesta al server
27 * di SocialGossip.
28 * @author Emilio Panti mat:531844
29 */
30 public class RequestGUI extends JFrame {
31
32     //lunghezza massima per i campi nickname e password
33     static int MAX_LENGTH = 16;
34
35     private static final long serialVersionUID = 1L;
36     JPanel contentPane;
37     JTextField textField;
38
39
40 /**
41 * Costruttore classe RequestGUI.
42 * @param Operations op: operazione richiesta dall'utente.
43 * @param OperativeGUI operativeGUI: interfaccia operativa del servizio.
44 * @param ChatHandlerGUI chatHandlerGUI: interfaccia gestore delle chat.
45 */
46 public RequestGUI(Operations op, OperativeGUI operativeGUI,
47                   ChatHandlerGUI chatHandlerGUI) {
48
49     setTitle("Request");
50     setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
51     setBounds(100, 100, 268, 126);
52     contentPane = new JPanel();
53     contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
54     setContentPane(contentPane);
55     contentPane.setLayout(null);
56
57     //LABEL
58     JLabel lbl = new JLabel("Nickname:");
59     lbl.setBounds(20, 25, 85, 14);
60     contentPane.add(lbl);
61
62     //FIELD
63     textField = new JTextField();
64     textField.setBounds(98, 22, 133, 20);
65     contentPane.add(textField);
66     textField.setColumns(10);
67
68     //BUTTON
69     JButton btnNewButton = new JButton("New button");
70     btnNewButton.setBounds(59, 53, 133, 23);
71     contentPane.add(btnNewButton);
72
73     //in base alla operazione richiesta viene generata un interfaccia grafica diversa
74     switch(op) {
75
```

RequestGUI.java

```
76     case FRIENDSHIP:
77         btnNewButton.setText("ADD");
78         break;
79
80     case LOOKUP:
81         btnNewButton.setText("LOOK UP");
82         break;
83
84     case STARTCHAT:
85         btnNewButton.setText("START CHAT");
86         break;
87
88     case CREATE_CHATROOM:
89         lbl.setText("Chatroom:");
90         btnNewButton.setText("CREATE");
91         break;
92
93     case ADDME_CHATROOM:
94         lbl.setText("Chatroom:");
95         btnNewButton.setText("ADD ME");
96         break;
97
98     case CLOSE_CHATROOM:
99         lbl.setText("Chatroom:");
100        btnNewButton.setText("CLOSE");
101        break;
102
103    default:
104        dispose();
105        break;
106    }
107
108 //-----ALL LISTENER-----
109
110 //LISTENER BUTTON
111 btnNewButton.addMouseListener(new MouseAdapter() {
112     @Override
113     public void mouseClicked(MouseEvent arg0) {
114         //prendo cosa ha scritto l'utente
115         String txt = textField.getText();
116
117         //se txt non è vuoto
118         if(txt.length()>0) {
119
120             //se txt è più lungo di MAX_LENGTH
121             if(txt.length()>MAX_LENGTH) {
122                 //comunico all'utente l'errore
123                 if(op==Operations.FRIENDSHIP || op==Operations.LOOKUP ||
124                     op==Operations.STARTCHAT) {
125                     ResponseGUI responseGUI = new ResponseGUI("The nickname is too
126                     long!");
127                     responseGUI.setVisible(true);
128                 }
129                 else {
130                     ResponseGUI responseGUI = new ResponseGUI("The chatroom is too
131                     long!");
132                     responseGUI.setVisible(true);
133                 }
134             }
135         }
136     }
137 }
```

RequestGUI.java

```
131     }
132 }
133 //se l'utente prova ad aggiungere, cercare, chattare con sè stesso
134 else if(checkTxt(op,txt)==false) {
135     ResponseGUI responseGUI = new ResponseGUI("You can't insert your
136     nickname!");
137     responseGUI.setVisible(true);
138 }
139 //se tutto è corretto
140 else {
141     //in base alla operazione richiesta attivo il thread relativo
142     switch(op) {
143
144         case FRIENDSHIP:
145             //faccio partire il thread che esegue la richiesta di amicizia
146             FriendshipThread friendshipThread = new FriendshipThread(txt);
147             Thread thread = new Thread(friendshipThread);
148             thread.start();
149             break;
150
151         case LOOKUP:
152             //faccio partire il thread che esegue la richiesta di
153             //ricerca utente
154             LookUpThread lookUpThread = new LookUpThread(txt);
155             Thread thread2 = new Thread(lookUpThread);
156             thread2.start();
157             break;
158
159         case STARTCHAT:
160             //faccio partire il thread che esegue la richiesta di
161             //aprire una nuova chat verso un utente
162             StartChatThread startChatThread =
163                 new StartChatThread(chatHandlerGUI,txt);
164             Thread thread3 = new Thread(startChatThread);
165             thread3.start();
166             break;
167
168         case CREATE_CHATROOM:
169             //faccio partire il thread che esegue la richiesta
170             //di creare una nuova chatroom
171             CreateCRTThread createCRTThread =
172                 new CreateCRTThread(operativeGUI,chatHandlerGUI,txt);
173             Thread thread4 = new Thread(createCRTThread);
174             thread4.start();
175             break;
176
177         case ADDME_CHATROOM:
178             //faccio partire il thread che esegue la richiesta di
179             //aggiunta ad una chatroom
180             AddToCRTThread addToCRTThread =
181                 new AddToCRTThread(chatHandlerGUI,txt);
182             Thread thread5 = new Thread(addToCRTThread);
183             thread5.start();
184             break;
185
186         case CLOSE_CHATROOM:
187             //faccio partire il thread che esegue la richiesta di chiusura
```

RequestGUI.java

```
187                     //chatroom
188                     CloseCRTThread closeCRTThread = new CloseCRTThread(txt);
189                     Thread thread6 = new Thread(closeCRTThread);
190                     thread6.start();
191                     break;
192
193             default:
194                 break;
195         }
196     }
197     dispose();
198 }
199 }
200 });
201 }
202
203 /**
204 * Metodo per controllare che il testo passato da parametro non sia
205 * uguale al nickname dell'utente, per evitare che tenti di aggiungere,
206 * chattare o cercare sè stesso.
207 * @param Operations op: operazione richiesta dall'utente.
208 * @param String txt: testo inserito dall'utente nella richiesta.
209 * @return boolean: true, se non ha inserito il suo stesso nickname nella richiesta,
210 *                     false, altrimenti.
211 */
212 private boolean checkTxt(Operations op, String txt) {
213     //se è una operazione verso un altro utente
214     if(op==Operations.FRIENDSHIP || op==Operations.LOOKUP ||
215         op==Operations.STARTCHAT) {
216         if(ClientMain.NICKNAME.equals(txt)) return false;
217         else return true;
218     }
219     //se è una operazione verso una chatroom
220     else return true;
221 }
222 }
223 }
224 }
```

ResponseGUI.java

```
1 package graphicInterfaces;
2
3 import javax.swing.JFrame;
8
9
10 /**
11  * Classe ResponseGUI.
12  * Interfaccia grafica per mostrare all'utente l'esito di una
13  * sua richiesta al server.
14  * @author Emilio Panti mat:531844
15 */
16 public class ResponseGUI extends JFrame {
17
18     private static final long serialVersionUID = 1L;
19     private JPanel contentPane;
20
21
22     /**
23      * Costruttore classe ResponseGUI.
24      * @param string: messaggio di risposta
25      */
26     public ResponseGUI(String string) {
27         setTitle("Response");
28         setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
29         setBounds(100, 100, 304, 100);
30         contentPane = new JPanel();
31         contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
32         setContentPane(contentPane);
33         contentPane.setLayout(null);
34
35         //TEXT AREA
36         JTextArea textArea = new JTextArea(string);
37         textArea.setFont(new Font("Tahoma", Font.BOLD, 11));
38         textArea.setBounds(10, 11, 268, 67);
39         contentPane.add(textArea);
40         textArea.setLineWrap(true);
41         textArea.setWrapStyleWord(true);
42         textArea.setOpaque(false);
43         textArea.setEditable(false);
44     }
45 }
46
```

NotifyEventImpl.java

```
1 package notificationRMI;
2
3 import java.net.InetAddress;
10
11
12 /**
13 * Classe NotifyEventImpl.
14 * Contiene le implementazioni dei metodi dichiarati nell'interfaccia
15 * NotifyEventInterface.
16 * @author Emilio Panti mat:531844
17 */
18 public class NotifyEventImpl extends RemoteObject implements NotifyEventInterface {
19
20     private static final long serialVersionUID = 1L;
21
22     //interfacce grafiche dove verrano segnalati gli eventi
23     private ChatHandlerGUI chatHandlerGUI;
24     private OperativeGUI operativeGUI;
25
26
27     /**
28     * Costruttore classe NotifyEventImpl.
29     * crea un nuovo callback client.
30     * @param ChatHandlerGUI chatHandlerGUI: interfaccia che gestisce
31     *         le chat aperte.
32     * @param OperativeGUI operativeGUI: interfaccia operativa.
33     * @throws RemoteException
34     */
35     public NotifyEventImpl(ChatHandlerGUI chatHandlerGUI,
36                           OperativeGUI operativeGUI) throws RemoteException {
37         this.chatHandlerGUI = chatHandlerGUI;
38         this.operativeGUI = operativeGUI;
39     }
40
41
42     /**
43     * Metodo che può essere richiamato dal server per notificare al client
44     * che un suo amico è adesso online.
45     * @param String Nickname: nickname dell'utente che è online
46     * @throws RemoteException
47     */
48     public void notifyFriendOnline(String nickname) throws RemoteException {
49         //scrivo l'evento nell'area di testo delle notifiche
50         operativeGUI.appendTextNotifications(nickname + " is now online");
51     }
52
53
54     /**
55     * Metodo che può essere richiamato dal server per notificare al client
56     * che un suo amico è adesso offline.
57     * @param String Nickname: nickname dell'utente che è andato offline
58     * @throws RemoteException
59     */
60     public void notifyFriendOffline(String nickname) throws RemoteException {
61         //scrivo l'evento nell'area di testo delle notifiche
62         operativeGUI.appendTextNotifications(nickname + " is now offline");
63     }
```

NotifyEventImpl.java

```
64
65
66 /**
67 * Metodo che può essere richiamato dal server per notificare al client
68 * una nuova amicizia.
69 * @param String Nickname: nickname dell'utente diventato amico.
70 * @throws RemoteException
71 */
72 public void notifyNewFriend(String nickname) throws RemoteException {
73     //scrivo l'evento nell'area di testo delle notifiche
74     operativeGUI.appendTextNotifications("You and "+ nickname + " are now friends");
75
76     //aggiungo il nuovo amico nell'area di testo riservata alla lista amici
77     operativeGUI.appendTextFriends(nickname);
78 }
79
80
81 /**
82 * Metodo che può essere richiamato dal server per notificare al client
83 * la chiusura di una chatroom da parte del suo creatore.
84 * @param String chatroom: nome della chatroom che è stata chiusa
85 * @param InetSocketAddress address: address della chatroom.
86 * @throws RemoteException
87 */
88 public void notifyCloseChatroom(String chatroom, InetSocketAddress address)
89     throws RemoteException {
90     //scrivo l'evento nell'area di testo delle notifiche
91     operativeGUI.appendTextNotifications("The chatroom '"+ chatroom + "' has been
closed");
92
93     //cancello la chat aperta verso la chatroom chiusa
94     chatHandlerGUI.removeChatroom(chatroom, address);
95
96 /**
97 * apro anche una interfaccia grafica per comunicare che la chat è stata rimossa
98 * dalla interfaccia che gestisce le chat aperte, questo perchè ritengo opportuno
99 * richiamare l'attenzione dell'utente in modo più rilevante rispetto alle notifiche
100 * di cambiamento status di un amico e di nuova amicizia.
101 */
102 ResponseGUI responseGUI = new ResponseGUI("The chatroom '" + chatroom + "' has been
closed");
103 responseGUI.setVisible(true);
104 }
105 }
106 }
```

NotifyEventInterface.java

```
1 package notificationRMI;
2
3 import java.net.InetAddress;
4
5
6
7
8 /**
9  * Interfaccia NotifyEventInterface.
10 * Contiene le dichiarazioni dei metodi utilizzati dal server per notificare
11 * al client remoto alcuni eventi, fra cui:
12 * - cambiamento di status di un amico (online/offline),
13 * - nuova amicizia,
14 * - chiusura di una chatroom.
15 * @author Emilio Panti mat:531844
16 */
17 public interface NotifyEventInterface extends Remote {
18
19     /**
20      * Metodo che può essere richiamato dal server per notificare al client
21      * che un suo amico è adesso online.
22      * @param String Nickname: nickname dell'utente che è online
23      * @throws RemoteException
24     */
25     public void notifyFriendOnline(String nickname) throws RemoteException;
26
27
28     /**
29      * Metodo che può essere richiamato dal server per notificare al client
30      * che un suo amico è adesso offline.
31      * @param String Nickname: nickname dell'utente che è andato offline
32      * @throws RemoteException
33     */
34     public void notifyFriendOffline(String nickname) throws RemoteException;
35
36
37     /**
38      * Metodo che può essere richiamato dal server per notificare al client
39      * una nuova amicizia .
40      * @param String Nickname: nickname dell'utente che ha stretto l'amicizia
41      * @throws RemoteException
42     */
43     public void notifyNewFriend(String nickname) throws RemoteException;
44
45
46     /**
47      * Metodo che può essere richiamato dal server per notificare al client
48      * la chiusura di una chatroom da parte del suo creatore.
49      * @param String chatroom: nome della chatroom che è stata chiusa
50      * @param InetAddress address: address della chatroom.
51      * @throws RemoteException
52     */
53     public void notifyCloseChatroom(String chatroom,InetAddress address)
54             throws RemoteException;
55 }
```

ServerInterface.java

```
1 package notificationRMI;
2
3 import java.rmi.Remote;
4
5
6 /**
7  * Interfaccia ServerInterface.
8  * Contiene le dichiarazioni dei metodi offerti in remoto dal server
9  * per permettere ai clients di registrarsi alla callback e ricevere
10 * così le notifiche che li riguardano.
11 * @author Emilio Panti mat:531844
12 */
13 public interface ServerInterface extends Remote {
14
15     /**
16      * Metodo che può essere richiamato dal client per registrarsi alla callback
17      * e poter ricevere le notifiche che lo riguardano.
18      * @param NotifyEventInterface clientInterface: stub del client utilizzato
19      *                      dal server per richiamare i metodi che notificano l'eventi.
20      * @param String nickname: nickname dell'utente che ha richiesto la registrazione
21      *                      alla callback
22      * @throws RemoteException
23     */
24     public void registerForCallback (NotifyEventInterface clientInterface,
25                                     String nickname) throws RemoteException;
26 }
27
```

AddToCRThread.java

```
1 package threads.chatroomsOp;
2
3 import java.io.DataInputStream;
15
16
17 /**
18 * Classe AddToCRThread.
19 * Thread che richiede al server (e ne gestisce la risposta) di
20 * iscrivere l'utente ad una specifica chatroom.
21 * Se l'operazione va a buon fine viene aperta una chat verso
22 * tale chatroom.
23 * NB: la lista chatrooms presente nell'interfaccia operativa non
24 * viene aggiornata (è lasciato all'utente il compito di
25 * aggiornarla quando lo ritiene più opportuno).
26 * @author Emilio Panti mat:531844
27 */
28 public class AddToCRThread implements Runnable {
29
30     //interfaccia che gestisce le chat
31     private ChatHandlerGUI chatHandlerGUI;
32
33     //nome della chatroom a cui si vuole unire l'utente
34     private String id;
35
36
37     /**
38      * Costruttore classe AddToCRThread.
39      * @param ChatHandlerGUI chatHandlerGUI: interfaccia che gestisce le chat.
40      * @param String id: nome della chatroom a cui si vuole unire l'utente.
41      */
42     public AddToCRThread(ChatHandlerGUI chatHandlerGUI, String id) {
43         this.chatHandlerGUI = chatHandlerGUI;
44         this.id = id;
45     }
46
47
48     /**
49      * Task che esegue l'operazione di iscrizione ad una chatroom.
50      */
51     @SuppressWarnings("unchecked")
52     public void run() {
53         //prendo gli streams per comunicare con il server
54         DataOutputStream writer = ClientMain.WRITER;
55         DataInputStream reader = ClientMain.READER;
56
57         //creo la richiesta di iscrizione ad una chatroom
58         JSONObject request = new JSONObject();
59         request.put("OP", "ADDME_CHATROOM");
60         request.put("ID", id);
61
62         //variabili per gestire la risposta del server
63         String response = null;
64         JSONObject responseJSON = null;
65
66         try {
67             //mando la richiesta
68             writer.writeUTF(request.toJSONString());
```

AddToCRThread.java

```
69         writer.flush();
70
71         //ricevo la risposta
72         response = reader.readUTF();
73
74         //trasformo in formato JSON la risposta ricevuta dal server
75         responseJSON = (JSONObject) new JSONParser().parse(response);
76
77     } catch (Exception e) {
78         //se c'è qualche problema di comunicazione con il server
79         //o se non è possibile parsare il messaggio ricevuto
80         e.printStackTrace();
81         //termino il client
82         ClientMain.cleanUp();
83     }
84
85
86     //controllo l'esito della risposta
87     Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
88     if (esito==Operations.OP_OK) {
89         //prendo l'indirizzo della chatroom
90         String address = (String) responseJSON.get ("ADDRESS");
91
92         //prendo il multicast socket
93         MulticastSocket ms = ClientMain.MS;
94
95         //boolean per sapere se la registrazione al gruppo è andata bene
96         boolean check = true;
97
98         //mi registro al gruppo multicast della chatroom
99         InetAddress ia = null;
100        try {
101            ia = InetAddress.getByName(address);
102            ms.joinGroup (ia);
103        } catch (Exception e) {
104            //se fallisce la ricerca dell'indirizzo ip dell'host
105            //o se fallisce l'iscrizione al gruppo multicast
106            e.printStackTrace();
107            check = false;
108        }
109
110        //se la registrazione al gruppo è andata bene
111        if(check) {
112            //apro una chat per la chatroom
113            chatHandlerGUI.addChatroom(id, ia);
114        }
115        //se c'è stato qualche errore
116        else {
117            //apro l'interfaccia grafica per comunicare l'errore
118            ResponseGUI responseGUI =
119                new ResponseGUI("ERR: An error occurred");
120            responseGUI.setVisible(true);
121        }
122    }
123    else {
124        //prendo il messaggio di errore trasmesso dal server
125        String msgErr = (String) responseJSON.get ("MSG");
```

AddToCRThread.java

```
126
127      //apro l'interfaccia grafica per comunicare l'errore
128      ResponseGUI responseGUI = new ResponseGUI(msgErr);
129      responseGUI.setVisible(true);
130  }
131
132
133 }
134
```

CloseCRThread.java

```
1 package threads.chatroomsOp;
2
3 import java.io.DataInputStream;
12
13
14 /**
15 * Classe CloseCRThread.
16 * Thread che richiede al server (e ne gestisce la risposta) di
17 * chiudere una specifica chatroom.
18 * Se l'operazione va a buon fine viene chiusa la chat che era
19 * aperta verso tale chatroom.
20 * NB: la lista chatrooms presente nell'interfaccia operativa non
21 * viene aggiornata (è lasciato all'utente il compito di
22 * aggiornarla quando lo ritiene più opportuno).
23 * @author Emilio Panti mat:531844
24 */
25 public class CloseCRThread implements Runnable {
26
27     //nome della chatroom che si vuole chiudere.
28     private String id;
29
30
31     /**
32      * Costruttore classe CloseCRThread.
33      * @param String id: nome della chatroom che l'utente vuole chiudere.
34      */
35     public CloseCRThread(String id) {
36         this.id = id;
37     }
38
39
40     /**
41      * Task che esegue l'operazione di chiusura chatroom.
42      */
43     @SuppressWarnings("unchecked")
44     public void run() {
45         //prendo gli streams per comunicare con il server
46         DataOutputStream writer = ClientMain.WRITER;
47         DataInputStream reader = ClientMain.READER;
48
49         //creo la richiesta di cancellazione di una chatroom
50         JSONObject request = new JSONObject();
51         request.put("OP", "CLOSE_CHATROOM");
52         request.put("ID", id);
53
54         //variabili per gestire la risposta del server
55         String response = null;
56         JSONObject responseJSON = null;
57
58         try {
59             //mando la richiesta
60             writer.writeUTF(request.toJSONString());
61             writer.flush();
62
63             //ricevo la risposta
64             response = reader.readUTF();
65         }
```

CloseCRThread.java

```
66 //trasformo in formato JSON la risposta ricevuta dal server
67 responseJSON = (JSONObject) new JSONParser().parse(response);
68
69 } catch (Exception e) {
70     //se c'è qualche problema di comunicazione con il server
71     //o se non è possibile parsare il messaggio ricevuto
72     e.printStackTrace();
73     //termino il client
74     ClientMain.cleanUp();
75 }
76
77
78 //controllo l'esito della risposta
79 Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
80 if (esito!=Operations.OP_OK) {
81     //prendo il messaggio di errore trasmesso dal server
82     String msgErr = (String) responseJSON.get ("MSG");
83
84     //apro l'interfaccia grafica per comunicare l'errore
85     ResponseGUI responseGUI = new ResponseGUI(msgErr);
86     responseGUI.setVisible(true);
87 }
88
89
90 }
91
```

CreateCRThread.java

```
1 package threads.chatroomsOp;
2
3 import java.io.DataInputStream;
16
17
18 /**
19 * Classe CreateCRThread.
20 * Thread che richiede al server (e ne gestisce la risposta) di
21 * creare una nuova chatroom.
22 * Se l'operazione va a buon fine la chatroom appena creata viene
23 * aggiunta alla lista delle chatrooms nell'interfaccia operativa.
24 * Se l'operazione va a buon fine viene aperta una chat
25 * verso la nuova chatroom.
26 * @author Emilio Panti mat:531844
27 */
28 public class CreateCRThread implements Runnable {
29
30     //interfacce grafiche aperte
31     private OperativeGUI operativeGUI;
32     private ChatHandlerGUI chatHandlerGUI;
33
34     //nome della chatroom che si vuole aprire
35     private String id;
36
37
38     /**
39      * Costruttore classe CreateCRThread.
40      * @param OperativeGUI operativeGUI: interfaccia operativa.
41      * @param ChatHandlerGUI chatHandlerGUI: interfaccia che gestisce le chat.
42      * @param String id: nome della chatroom che vuole creare l'utente.
43      */
44     public CreateCRThread(OperativeGUI operativeGUI, ChatHandlerGUI chatHandlerGUI, String id) {
45         this.operativeGUI = operativeGUI;
46         this.chatHandlerGUI = chatHandlerGUI;
47         this.id = id;
48     }
49
50
51     /**
52      * Task che esegue l'operazione di richiesta apertura chatroom.
53      */
54     @SuppressWarnings("unchecked")
55     public void run() {
56         //prendo gli streams per comunicare con il server
57         DataOutputStream writer = ClientMain.WRITER;
58         DataInputStream reader = ClientMain.READER;
59
60         //creo la richiesta di creazione chatroom
61         JSONObject request = new JSONObject();
62         request.put("OP", "CREATE_CHATROOM");
63         request.put("ID", id);
64
65         //variabili per gestire la risposta del server
66         String response = null;
67         JSONObject responseJSON = null;
68
69         try {
```

CreateCRThread.java

```
70      //mando la richiesta
71      writer.writeUTF(request.toJSONString());
72      writer.flush();
73
74      //ricevo la risposta
75      response = reader.readUTF();
76
77      //trasformo in formato JSON la risposta ricevuta dal server
78      responseJSON = (JSONObject) new JSONParser().parse(response);
79
80  } catch (Exception e) {
81      //se c'è qualche problema di comunicazione con il server
82      //o se non è possibile parsare il messaggio ricevuto
83      e.printStackTrace();
84      //termino il client
85      ClientMain.cleanUp();
86  }
87
88
89      //controllo l'esito della risposta
90      Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
91      if (esito==Operations.OP_OK) {
92          //prendo l'indirizzo della chatroom
93          String address = (String) responseJSON.get ("ADDRESS");
94
95          //prendo il multicast socket
96          MulticastSocket ms = ClientMain.MS;
97
98          //boolean per sapere se la registrazione al gruppo è andata bene
99          boolean check = true;
100
101         //mi registro al gruppo multicast della chatroom
102         InetAddress ia = null;
103         try {
104             ia = InetAddress.getByName(address);
105             ms.joinGroup (ia);
106         } catch (Exception e) {
107             //se fallisce la ricerca dell'indirizzo ip dell'host
108             //o se fallisce l'iscrizione al gruppo multicast
109             e.printStackTrace();
110             check = false;
111         }
112
113         //se la registrazione al gruppo è andata bene
114         if(check) {
115             //apro una chat per la chatroom
116             chatHandlerGUI.addChatroom(id, ia);
117
118             //aggiorno la lista chatroom nella operative interface
119             String txt = id + " (signed up)";
120             operativeGUI.appendTextChatrooms(txt);
121         }
122         //se c'è stato qualche errore
123     else {
124         //apro l'interfaccia grafica per comunicare l'errore
125         ResponseGUI responseGUI =
126             new ResponseGUI("ERR: An error occurred");
```

CreateCRThread.java

```
127         responseGUI.setVisible(true);
128     }
129 }
130 else {
131     //prendo il messaggio di errore trasmesso dal server
132     String msgErr = (String) responseJSON.get("MSG");
133
134     //apro l'interfaccia grafica per comunicare l'errore
135     ResponseGUI responseGUI = new ResponseGUI(msgErr);
136     responseGUI.setVisible(true);
137 }
138 }
139 }
140 }
```

CRLListThread.java

```
1 package threads.chatroomsOp;
2
3 import java.io.DataInputStream;
15
16
17 /**
18 * Classe CRLListThread.
19 * Thread che richiede al server (e ne gestisce la risposta) la
20 * lista di tutte le chatrooms aperte, specificando a quali di
21 * esse l'utente sia già iscritto.
22 * Se l'operazione va a buon fine viene aggiornata la lista
23 * delle chatrooms presente nell'interfaccia operativa.
24 * @author Emilio Panti mat:531844
25 */
26 public class CRLListThread implements Runnable {
27
28     //interfaccia operativa della chat
29     private OperativeGUI operativeGUI;
30
31
32     /**
33      * Costruttore classe CRLListThread.
34      * @param OperativeGUI operativeGUI: interfaccia operativa.
35      */
36     public CRLListThread(OperativeGUI operativeGUI) {
37         this.operativeGUI = operativeGUI;
38     }
39
40
41     /**
42      * Task che esegue l'operazione di richiesta lista chatroom.
43      */
44     @SuppressWarnings("unchecked")
45     public void run() {
46         //prendo gli streams per comunicare con il server
47         DataOutputStream writer = ClientMain.WRITER;
48         DataInputStream reader = ClientMain.READER;
49
50         //creo la richiesta per la lista chatrooms in formato JSON
51         JSONObject request = new JSONObject ();
52         request.put("OP", "CHATLIST");
53
54         //variabili per gestire la risposta del server
55         String response = null;
56         JSONObject responseJSON = null;
57
58         try {
59             //mando la richiesta
60             writer.writeUTF(request.toJSONString());
61             writer.flush();
62
63             //ricevo la risposta
64             response = reader.readUTF();
65
66             //trasformo in formato JSON la risposta ricevuta dal server
67             responseJSON = (JSONObject) new JSONParser().parse(response);
68         }
```

CRLListThread.java

```
69 } catch (Exception e) {
70     //se c'è qualche problema di comunicazione con il server
71     //o se non è possibile parsare il messaggio ricevuto
72     e.printStackTrace();
73     //termino il client
74     ClientMain.cleanUp();
75 }
76
77
78     //controllo l'esito della risposta
79     Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
80     if (esito==Operations.OP_OK) {
81         //prendo la lista chatrooms in formato JSONArray
82         JSONArray chatrooms = (JSONArray) responseJSON.get ("CHATROOMS");
83
84         if(chatrooms != null) {
85             //Stringa dove inserisco il nome delle chatrooms
86             //(specificando a quali l'utente è già iscritto)
87             String listChatrooms = "";
88
89             //stringa end of line
90             String eol = System.getProperty("line.separator");
91
92             //creo un iteratore per scorrere la lista delle chatrooms
93             Iterator<JSONObject> iterator = chatrooms.iterator();
94
95             while (iterator.hasNext()) {
96                 //prendo un oggetto json della lista
97                 JSONObject chatroomJSON = iterator.next();
98
99                 //prendo il nome della chatroom
100                String chatroom = (String) chatroomJSON.get("ID");
101                //boolean che mi dice se l'utente è iscritto ad essa o meno
102                Boolean signed = (Boolean) chatroomJSON.get("SIGNED");
103
104                //se l'utente è iscritto
105                if(signed) {
106                    //aggiorno la stringa della lista chatrooms
107                    listChatrooms = listChatrooms + chatroom + " (signed up)" + eol;
108                }
109                //se l'utente non è iscritto
110                else {
111                    //aggiorno la stringa della lista chatrooms
112                    listChatrooms = listChatrooms + chatroom + eol;
113                }
114            }
115
116            //appendo la lista delle chatrooms nell'area di text dell'interfaccia
117            operativeGUI.setTextChatrooms(listChatrooms);
118        }
119        //se non esistono chatrooms attive
120        else {
121            operativeGUI.setTextChatrooms("");
122        }
123    }
124 }
```

CRLListThread.java

```
125      //prendo il messaggio di errore trasmesso dal server
126      String msgErr = (String) responseJSON.get ("MSG");
127
128      //apro l'interfaccia grafica per comunicare l'errore
129      ResponseGUI responseGUI = new ResponseGUI(msgErr);
130      responseGUI.setVisible(true);
131  }
132
133      //riattivo il bottone per la richiesta della lista chatrooms
134      operativeGUI.enabledBtnUpdatesChatrooms();
135  }
136
137 }
138
```

FriendshipThread.java

```
1 package threads.friendsOp;
2
3 import java.io.DataInputStream;
12
13
14 /**
15 * Classe FriendshipThread.
16 * Thread che richiede al server (e ne gestisce la risposta) di
17 * instaurare una nuova amicizia tra l'utente ed un'altra persona
18 * che utilizza SocialGossip.
19 * Se l'operazione va a buon fine il nickname del nuovo amico viene
20 * aggiunto alla lista amici nell'interfaccia operativa.
21 * @author Emilio Panti mat:531844
22 */
23 public class FriendshipThread implements Runnable {
24
25     //nickname con cui l'utente vuole stringere amicizia
26     private String nickname;
27
28
29     /**
30      * Costruttore classe FriendshipThread.
31      * @param String nickname: nickname con cui l'utente vuole stringere amicizia.
32      */
33     public FriendshipThread(String nickname) {
34         this.nickname = nickname;
35     }
36
37
38     /**
39      * Task che esegue l'operazione di richiesta amicizia.
40      */
41     @SuppressWarnings("unchecked")
42     public void run() {
43         //prendo gli streams per comunicare con il server
44         DataOutputStream writer = ClientMain.WRITER;
45         DataInputStream reader = ClientMain.READER;
46
47         //creo la richiesta di amicizia
48         JSONObject request = new JSONObject();
49         request.put("OP", "FRIENDSHIP");
50         request.put("ID", nickname);
51
52         //variabili per gestire la risposta del server
53         String response = null;
54         JSONObject responseJSON = null;
55
56         try {
57             //mando la richiesta
58             writer.writeUTF(request.toJSONString());
59             writer.flush();
56
51
52             //ricevo la risposta
53             response = reader.readUTF();
54
55             //trasformo in formato JSON la risposta ricevuta dal server
56             responseJSON = (JSONObject) new JSONParser().parse(response);
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
778
779
779
780
781
782
783
784
785
786
787
788
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
878
879
879
880
881
882
883
884
885
886
887
888
888
889
889
890
891
892
893
894
895
896
897
897
898
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
918
919
919
920
921
922
923
924
925
926
927
928
928
929
929
930
931
932
933
934
935
936
937
938
938
939
939
940
941
942
943
944
945
946
947
947
948
948
949
949
950
951
952
953
954
955
956
957
957
958
958
959
959
960
961
962
963
964
965
966
967
968
968
969
969
970
971
972
973
974
975
976
977
977
978
978
979
979
980
981
982
983
984
985
986
987
987
988
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1047
1048
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1077
1078
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1087
1088
1088
1089
1089
1090
1091
1092
1093
1094
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1127
1128
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1137
1138
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1151
1152
1153
1154
1155
1156
1157
1157
1158
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1171
1172
1173
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1181
1182
1183
1184
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1191
1192
1193
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1207
1208
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1251
1252
1253
1254
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1271
1272
1273
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1281
1282
1283
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1291
1292
1293
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1301
1302
1303
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1311
1312
1313
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1321
1322
1323
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1331
1332
1333
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1341
1342
1343
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1351
1352
1353
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1361
1362
1363
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1371
1372
1373
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1381
1382
1383
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1391
1392
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1401
1402
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1411
1412
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1421
1422
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1431
1432
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1441
1442
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1451
1452
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1461
1462
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1471
1472
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1481
1482
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1491
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1501
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1511
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1521
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1531
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1541
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1551
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1561
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1571
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1581
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1591
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1601
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1611
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1621
1622
1623
1623
1624
1624
1625
1625
1626
1626
1627
1627
1628
1628
1629
1629
1630
1631
1632
1633
1633
1634
1634
1635
1635
1636
1636
1637
1637
1638
1638
1639
1639
1640
1641
1642
1643
1643
1644
1644
1645
1645
1646
1646
1647
1647
1648
1648
1649
1649
1650
1651
1652
1653
1653
1654
1654
1655
1655
1656
1656
1657
1657
1658
1658
1659
1659
1660
1661
1662
1663
1663
1664
1664
1665
1665
1666
1666
1667
1667
1668
1668
1669
1669
1670
1671
1672
1673
1673
1674
1674
1675
1675
1676
1676
1677
1677
1678
1678
1679
1679
1680
1681
1682
1683
1683
1684
1684
1685
1685
1686
1686
1687
1687
1688
1688
1689
1689
1690
1691
1692
1693
1693
1694
1694
1695
1695
1696
1696
1697
1697
1698
1698
1699
1699
1700
1701
1702
1703
1703
1704
1704
1705
1705
1706
1706
1707
1707
1708
1708
1709
1709
1710
1711
1712
1713
1713
1714
1714
1715
1715
1716
1716
1717
1717
1718
1718
1719
1719
1720
1721
1722
1723
1723
1724
1724
1725
1725
1726
1726
1727
1727
1728
1728
1729
1729
1730
1731
1732
1733
1733
1734
1734
1735
1735
1736
1736
1737
1737
1738
1738
1739
1739
1740
1741
1742
1743
1743
1744
1744
1745
1745
1746
1746
1747
1747
1748
1748
1749
1749
1750
1751
1752
1753
1753
1754
1754
1755
1755
1756
1756
1757
1757
1758
1758
1759
1759
1760
1761
1762
1763
1763
1764
1764
1765
1765
1766
1766
1767
1767
1768
1768
1769
1769
1770
1771
1772
1773
1773
1774
1774
1775
1775
1776
1776
1777
1777
1778
1778
1779
1779
1780
1781
1782
1783
1783
1784
1784
1785
1785
1786
1786
1787
1787
1788
1788
1789
1789
1790
1791
1792
1793
1793
1794
1794
1795
1795
1796
1796
1797
1797
1798
1798
1799
1799
1800
1801
1802
1803
1803
1804
1804
1805
1805
1806
1806
1807
1807
1808
1808
1809
1809
1810
1811
1812
1813
1813
1814
1814
1815
1815
1816
1816
1817
1817
1818
1818
1819
1819
1820
1821
1822
1823
1823
1824
1824
1825
1825
1826
1826
1827
1827
1828
1828
1829
1829
1830
1831
1832
1833
1833
1834
1834
1835
1835
1836
1836
1837
1837
1838
1838
1839
1839
1840
1841
1842
1843
1843
1844
1844
1845
1845
1846
1846
1847
1847
1848
1848
1849
1849
1850
1851
1852
1853
1853
1854
1854
1855
1855
1856
1856
1857
1857
1858
1858
1859
1859
1860
1861
1862
1863
1863
1864
1864
1865
1865
1866
1866
1867
1867
1868
1
```

FriendshipThread.java

```
66
67     } catch (Exception e) {
68         //se c'è qualche problema di comunicazione con il server
69         //o se non è possibile parsare il messaggio ricevuto
70         e.printStackTrace();
71         //termino il client
72         ClientMain.cleanUp();
73     }
74
75
76     //controllo l'esito della risposta
77     Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
78     if (esito!=Operations.OP_OK) {
79         //prendo il messaggio di errore trasmesso dal server
80         String msgErr = (String) responseJSON.get ("MSG");
81
82         //apro l'interfaccia grafica per comunicare l'errore
83         ResponseGUI responseGUI = new ResponseGUI(msgErr);
84         responseGUI.setVisible(true);
85     }
86 }
87 }
88 }
```

FriendsListThread.java

```
1 package threads.friendsOp;
2
3 import java.io.DataInputStream;
13
14
15 /**
16 * Classe FriendsListThread.
17 * Thread che richiede al server (e ne gestisce la risposta) la
18 * lista degli amici dell'utente.
19 * Se l'operazione va a buon fine viene aggiornata la lista degli
20 * amici presente nell'interfaccia operativa.
21 * NB: la richiesta di aggiornamento della lista amici da parte
22 * dell'utente è futile. Questo perchè essa viene aggiornata
23 * ogni volta che l'utente stringe una nuova amicizia
24 * (sia in modo attivo che passivo).
25 * @author Emilio Panti mat:531844
26 */
27 public class FriendsListThread implements Runnable {
28
29     //interfaccia operativa della chat
30     private OperativeGUI operativeGUI;
31
32
33     /**
34     * Costruttore classe FriendsListThread.
35     * @param OperativeGUI operativeGUI: interfaccia operativa.
36     */
37     public FriendsListThread(OperativeGUI operativeGUI) {
38         this.operativeGUI = operativeGUI;
39     }
40
41
42     /**
43     * Task che esegue l'operazione di richiesta lista amici.
44     */
45     @SuppressWarnings("unchecked")
46     public void run() {
47         //prendo gli streams per comunicare con il server
48         DataOutputStream writer = ClientMain.WRITER;
49         DataInputStream reader = ClientMain.READER;
50
51         //creo la richiesta per la lista amici in formato JSON
52         JSONObject request = new JSONObject ();
53         request.put("OP", "LISTFRIEND");
54
55         //variabili per gestire la risposta del server
56         String response = null;
57         JSONObject responseJSON = null;
58
59         try {
60             //mando la richiesta
61             writer.writeUTF(request.toJSONString());
62             writer.flush();
63
64             //ricevo la risposta
65             response = reader.readUTF();
66         }
```

FriendsListThread.java

```
67      //trasformo in formato JSON la risposta ricevuta dal server
68      responseJSON = (JSONObject) new JSONParser().parse(response);
69
70  } catch (Exception e) {
71      //se c'è qualche problema di comunicazione con il server
72      //o se non è possibile parsare il messaggio ricevuto
73      e.printStackTrace();
74      //termino il client
75      ClientMain.cleanUp();
76  }
77
78
79 //controllo l'esito della risposta
80 Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
81 if (esito==Operations.OP_OK) {
82     //prendo la lista amici e la appendo nell'area di text dell'interfaccia operativa
83     String friends = (String) responseJSON.get ("FRIENDS");
84     if (friends!=null) operativeGUI.setTextFriends(friends);
85 }
86 else {
87     //prendo il messaggio di errore trasmesso dal server
88     String msgErr = (String) responseJSON.get ("MSG");
89
90     //apro l'interfaccia grafica per comunicare l'errore
91     ResponseGUI responseGUI = new ResponseGUI(msgErr);
92     responseGUI.setVisible(true);
93 }
94
95 //riattivo il bottone per la richiesta della lista amici
96 operativeGUI.enabledBtnUpdatesFriends();
97 }
98
99 }
100 }
```

LookUpThread.java

```
1 package threads.friendsOp;
2
3 import java.io.DataInputStream;
12
13
14 /**
15 * Classe LookUpThread.
16 * Thread che richiede al server (e ne gestisce la risposta) se
17 * esista o meno un utente con il nickname specificato.
18 * @author Emilio Panti mat:531844
19 */
20 public class LookUpThread implements Runnable {
21
22     //nickname dell'utente da cercare
23     private String nickname;
24
25
26     /**
27      * Costruttore classe LookUpThread.
28      * @param String nickname: nickname dell'utente da cercare.
29      */
30     public LookUpThread(String nickname) {
31         this.nickname = nickname;
32     }
33
34
35     /**
36      * Task che esegue l'operazione di ricerca di un utente.
37      */
38     @SuppressWarnings("unchecked")
39     public void run() {
40         //prendo gli streams per comunicare con il server
41         DataOutputStream writer = ClientMain.WRITER;
42         DataInputStream reader = ClientMain.READER;
43
44         //creo la richiesta di look up
45         JSONObject request = new JSONObject();
46         request.put("OP", "LOOKUP");
47         request.put("ID", nickname);
48
49         //variabili per gestire la risposta del server
50         String response = null;
51         JSONObject responseJSON = null;
52
53         try {
54             //mando la richiesta
55             writer.writeUTF(request.toJSONString());
56             writer.flush();
57
58             //ricevo la risposta
59             response = reader.readUTF();
60
61             //trasformo in formato JSON la risposta ricevuta dal server
62             responseJSON = (JSONObject) new JSONParser().parse(response);
63
64         } catch (Exception e) {
65             //se c'è qualche problema di comunicazione con il server
```

LookUpThread.java

```
66      //o se non è possibile parsare il messaggio ricevuto
67      e.printStackTrace();
68      //termino il client
69      ClientMain.cleanUp();
70  }
71
72
73      //controllo l'esito della risposta
74      Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
75      if (esito==Operations.OP_OK) {
76          //prendo il responso della ricerca
77          String msg = (String) responseJSON.get ("MSG");
78
79          //apro l'interfaccia grafica per comunicare l'errore
80          ResponseGUI responseGUI = new ResponseGUI(msg);
81          responseGUI.setVisible(true);
82      }
83      else {
84          //prendo il messaggio di errore trasmesso dal server
85          String msgErr = (String) responseJSON.get ("MSG");
86
87          //apro l'interfaccia grafica per comunicare l'errore
88          ResponseGUI responseGUI = new ResponseGUI(msgErr);
89          responseGUI.setVisible(true);
90      }
91  }
92
93 }
94 }
```

StartChatThread.java

```
1 package threads.friendsOp;
2
3 import java.io.DataInputStream;
4
5 /**
6  * Classe StartChatThread.
7  * Thread che richiede al server (e ne gestisce la risposta)
8  * il 'permesso' di aprire una chat verso l'utente specificato.
9  * @author Emilio Panti mat:531844
10 */
11 public class StartChatThread implements Runnable {
12
13     //interfaccia che gestisce le chat
14     private ChatHandlerGUI chatHandlerGUI;
15
16     //nickname dell'utente con cui il client vuole aprire una chat
17     private String nickname;
18
19
20     /**
21      * Costruttore classe StartChatThread.
22      * @param ChatHandlerGUI chatHandlerGUI: interfaccia gestore delle chat.
23      * @param String nickname: nickname dell'utente con cui il client vuole
24      *                         aprire una chat.
25      */
26     public StartChatThread(ChatHandlerGUI chatHandlerGUI, String nickname) {
27         this.chatHandlerGUI = chatHandlerGUI;
28         this.nickname = nickname;
29     }
30
31
32     /**
33      * Task che esegue l'operazione di apertura chat.
34      */
35     @SuppressWarnings("unchecked")
36     public void run() {
37         //prendo gli streams per comunicare con il server
38         DataOutputStream writer = ClientMain.WRITER;
39         DataInputStream reader = ClientMain.READER;
40
41         /**
42          * Task che esegue l'operazione di apertura chat.
43          */
44         try {
45             //creo la richiesta di apertura chat
46             JSONObject request = new JSONObject();
47             request.put("OP", "STARTCHAT");
48             request.put("ID", nickname);
49
50             //variabili per gestire la risposta del server
51             String response = null;
52             JSONObject responseJSON = null;
53
54             try {
55                 //mando la richiesta
56                 writer.writeUTF(request.toJSONString());
57                 writer.flush();
58
59                 //ricevo la risposta
60                 response = reader.readUTF();
61             } catch (IOException e) {
62                 e.printStackTrace();
63             }
64         } catch (JSONException e) {
65             e.printStackTrace();
66         }
67     }
68 }
```

StartChatThread.java

```
67
68     //trasformo in formato JSON la risposta ricevuta dal server
69     responseJSON = (JSONObject) new JSONParser().parse(response);
70
71 } catch (Exception e) {
72     //se c'è qualche problema di comunicazione con il server
73     //o se non è possibile parsare il messaggio ricevuto
74     e.printStackTrace();
75     //termino il client
76     ClientMain.cleanUp();
77 }
78
79
80     //controllo l'esito della risposta
81     Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
82     if (esito==Operations.OP_OK) {
83         //apro una chat verso l'utente
84         chatHandlerGUI.addChatNickname(nickname, null);
85     }
86     else {
87         //prendo il messaggio di errore trasmesso dal server
88         String msgErr = (String) responseJSON.get ("MSG");
89
90         //apro l'interfaccia grafica per comunicare l'errore
91         ResponseGUI responseGUI = new ResponseGUI(msgErr);
92         responseGUI.setVisible(true);
93     }
94 }
95 }
```

ListenerCR.java

```
1 package threads.listeners;
2
3 import java.net.DatagramPacket;
4
5
6 /**
7  * Classe ListenerCR.
8  * Thread in ascolto (sul multicast socket aperto dal ClientMain)
9  * di nuovi messaggi provenienti dalle varie chatroom a cui
10 * l'utente è iscritto.
11 * Posta i messaggi ricevuti nelle relative chat aperte
12 * nell'interfaccia della classe ChatHandlerGUI.
13 * @author Emilio Panti mat:531844
14 */
15 public class ListenerCR implements Runnable {
16
17     //interfaccia che gestisce le chat
18     private ChatHandlerGUI chatHandlerGUI;
19
20
21     /**
22      * Costruttore classe ListenerChatrooms.
23      * @param ChatHandlerGUI chatHandlerGUI: interfaccia che gestisce le chat.
24      */
25     public ListenerCR(ChatHandlerGUI chatHandlerGUI) {
26         this.chatHandlerGUI = chatHandlerGUI;
27     }
28
29
30     /**
31      * Task che ascolta i messaggi in arrivo dalle varie chatrooms
32      */
33     public void run() {
34         //multicast socket per ricevere i messaggi dalle chatrooms
35         MulticastSocket ms = ClientMain.MS;
36
37         /**
38          * Task che ascolta i messaggi in arrivo dalle varie chatrooms
39          */
40         byte [ ] buffer = new byte[8000];
41
42         //variabile per gestire i messaggi che ricevo
43         String msgReiceved = null;
44         JSONObject msgJSON = null;
45
46         //variabili dove salvo il messaggio ricevuto e la chatroom di destinazione
47         String chatroom = null;
48         String msg = null;
49
50         while(true){
51             try {
52                 //creo il datagram packet
53                 DatagramPacket dp = new DatagramPacket(buffer,buffer.length);
54
55                 //aspetto un messaggio
56                 ms.receive(dp);
57                 msgReiceved = new String(dp.getData());
58
59             }
60         }
61
62     }
63
64 }
```

ListenerCR.java

```
65 //prendo solo la parte della stringa che mi interessa
66 int lastIndex = msgReiceved.lastIndexOf('}');
67 String msgCorrect = msgReiceved.substring(0, lastIndex+1);
68
69 //trasformo in formato JSON il messaggio corretto
70 msgJSON = (JSONObject) new JSONParser().parse(msgCorrect);
71
72 //prendo il messaggio e la chat destinataria
73 chatroom = (String) msgJSON.get ("TO");
74 msg = (String) msgJSON.get ("MSG");
75
76 //passo il messaggio all'interfaccia che gestisce le chat
77 chatHandlerGUI.postMessage(chatroom,msg);
78 }
79 catch (Exception e) {
80 //nel caso si presentasse una eccezione nella ricezione o nel parsing
81 //o altro non faccio nulla,così da far continuare il lavoro del listener
82 }
83 }
84 }
85 }
86 }
```

ListenerFile.java

```
1 package threads.listeners;
2
3 import java.io.IOException;
4
5
6
7
8
9
10
11 /**
12  * Classe ListenerFile.
13  * Thread in attesa (sul ServerSocketChannel aperto dal ClientMain)
14  * di nuove connessioni da parte di utenti che vogliono inviare
15  * dei file a questo client.
16  * Per ogni nuova connessione viene creato un thread (della classe
17  * UnloaderFileThread) che si occupa di scaricare il file inviatogli
18  * dal client dell'altro utente.
19  * @author Emilio Panti mat:531844
20 */
21 public class ListenerFile implements Runnable {
22
23     //interfaccia che gestisce le chats
24     private ChatHandlerGUI chatHandlerGUI;
25
26
27     /**
28      * Costruttore classe ListenerFile.
29      * @param ChatHandlerGUI chatHandlerGUI: interfaccia che gestisce le chat.
30      */
31     public ListenerFile(ChatHandlerGUI chatHandlerGUI) {
32         this.chatHandlerGUI = chatHandlerGUI;
33     }
34
35
36     /**
37      * Task che è in ascolto sul socket server channel
38      * per ricevere eventuali file da altri utenti
39      */
40     public void run() {
41         //prendo il server socket channel e lo metto in modalità bloccante
42         ServerSocketChannel serverSocket = ClientMain.SERVER_SOCKET_FILE;
43         try {
44             serverSocket.configureBlocking(true);
45         } catch (IOException e1) {
46             // TODO Auto-generated catch block
47             e1.printStackTrace();
48         }
49
50         //mi metto in attesa di ricevere connessioni
51         while (true) {
52
53             SocketChannel socketChannel = null;
54             try {
55                 //ricevo una nuova connessione
56                 socketChannel = serverSocket.accept();
57                 socketChannel.configureBlocking(true);
58             } catch (IOException e) {
59                 //se la connessione verso un altro client va male
60                 socketChannel = null;
61             }
62         }
63     }
64 }
```

ListenerFile.java

```
63 if(socketChannel!=null) {  
64     //passo la connessione al thread che si occupa di scaricare il file  
65     UnloaderFileThread unloaderFileThread =  
66         new UnloaderFileThread(chatHandlerGUI, socketChannel);  
67     Thread thread = new Thread(unloaderFileThread);  
68     thread.start();  
69 }  
70 }  
71 }  
72 }  
73  
74 }  
75 }
```

ListenerMsg.java

```
1 package threads.listeners;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6
7
8 /**
9  * Classe ListenerMsg.
10 * Thread che apre una seconda connessione TCP verso il
11 * server di SocialGossip e in cui si mette in ascolto.
12 * Questa nuova connessione è dedicata alla ricezione di
13 * messaggi testuali inviati dagli altri utenti.
14 * Posta i messaggi ricevuti nelle relative chat aperte
15 * nell'interfaccia della classe ChatHandlerGUI.
16 * NB: al momento della richiesta di apertura della seconda
17 * connessione viene comunicato al server anche la porta
18 * in cui il client è in ascolto per la ricezione di file.
19 * @author Emilio Panti mat:531844
20 */
21 public class ListenerMsg implements Runnable {
22
23     //interfaccia che gestisce le chats
24     private ChatHandlerGUI chatHandlerGUI;
25
26     //nickname dell'utente
27     private String nickname;
28
29
30     /**
31      * Costruttore classe ListenerMsg.
32      * @param ChatHandlerGUI chatHandlerGUI: interfaccia che gestisce le chat.
33      * @param String nickname: nickname dell'utente, serve per aprire la
34      * seconda connessione verso il server.
35      */
36     public ListenerMsg(ChatHandlerGUI chatHandlerGUI, String nickname) {
37         this.chatHandlerGUI = chatHandlerGUI;
38         this.nickname = nickname;
39     }
40
41
42     /**
43      * Task che ascolta i messaggi in arrivo da altri utenti
44      */
45     @SuppressWarnings("unchecked")
46     public void run() {
47
48         //socket e streams per la seconda connessione TCP
49         Socket socket = null;
50         DataOutputStream writer = null;
51         DataInputStream reader = null;
52
53         //variabili per gestire la risposta del server
54         String response = null;
55         JSONObject responseJSON = null;
56
57         //creo una seconda connessione verso il server
58         try {
59             try {
60                 writer = new DataOutputStream(socket.getOutputStream());
61                 writer.writeUTF(chatHandlerGUI.getPort());
62
63                 reader = new DataInputStream(socket.getInputStream());
64                 response = reader.readUTF();
65
66                 responseJSON = new JSONObject(response);
67
68                 if (responseJSON.has("file")) {
69                     String file = responseJSON.getString("file");
70
71                     File fileObject = new File(file);
72                     if (!fileObject.exists()) {
73                         fileObject.createNewFile();
74
75                         try {
76                             writer.writeUTF("File created successfully!");
77                         } catch (IOException e) {
78                             e.printStackTrace();
79                         }
80
81                         responseJSON = new JSONObject("File created successfully!");
82                     }
83
84                     if (fileObject.exists()) {
85                         byte[] bytes = Files.readAllBytes(fileObject.toPath());
86                         String fileContent = new String(bytes);
87
88                         responseJSON = new JSONObject("File content: " + fileContent);
89                     }
90
91                     response = responseJSON.toString();
92
93                     chatHandlerGUI.receiveFile(file, response);
94
95                 } else {
96                     responseJSON = new JSONObject("File not found or invalid file path.");
97                     response = responseJSON.toString();
98
99                     chatHandlerGUI.receiveFile(file, response);
100                }
101            } catch (IOException e) {
102                e.printStackTrace();
103            }
104        } catch (Exception e) {
105            e.printStackTrace();
106        }
107    }
108}
```

ListenerMsg.java

```
70      //apro il socket e gli streams di input ed output
71      socket = new Socket(ClientMain.HOSTNAME, ClientMain.PORT_TCP);
72      writer = new DataOutputStream(new BufferedOutputStream(socket.getOutputStream()));
73      reader = new DataInputStream(new BufferedInputStream(socket.getInputStream()));
74
75      //creo la richiesta per la connessione dedicata all'ascolto dei messaggi
76      //e specifico anche la porta dove il client è in ascolto di files
77      JSONObject request = new JSONObject();
78      request.put("OP", "CONN_MSG");
79      request.put("ID", nickname);
80      request.put("PORT_FILE", ClientMain.PORT_FILE);
81
82      //mando la richiesta
83      writer.writeUTF(request.toJSONString());
84      writer.flush();
85
86      //ricevo la risposta
87      response = reader.readUTF();
88
89      //trasformo in formato JSON la risposta ricevuta dal server
90      responseJSON = (JSONObject) new JSONParser().parse(response);
91
92      //controllo l'esito della risposta
93      Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
94      //se non è ok chiudo il Client
95      if (esito!=Operations.OP_OK) ClientMain.cleanUp();
96
97      //Salvo il nuovo socket creato e lo strem in input nella classe ClientMain
98      ClientMain.SOCKET_MSG = socket;
99      ClientMain.READER_MSG= reader;
100     ClientMain.WRITER_MSG= writer;
101
102    } catch (Exception e) {
103        //se vi è una qualsiasi eccezione nel blocco sopra devo chiudere
104        //il client. Questo perchè l'apertura della seconda connessione TCP
105        //verso il server è fondamentale
106        e.printStackTrace();
107        //termino il client
108        ClientMain.cleanUp();
109    }
110
111
112    //inizio l'ascolto ciclico di messaggi
113    while(true) {
114
115        try {
116            //ricevo un messaggio
117            response = reader.readUTF();
118
119            //trasformo in formato JSON ciò che ho ricevuto
120            responseJSON = (JSONObject) new JSONParser().parse(response);
121        } catch (IOException e) {
122            //termino il client nel caso ci sia un problema di comunicazione
123            //con il server
124            ClientMain.cleanUp();
125        } catch (Exception e) {
126            //nel caso si presentasse una eccezione nel parsing del messaggio
```

ListenerMsg.java

```
127         //o altro non faccio nulla,così da far continuare il lavoro del listener
128         e.printStackTrace();
129     }
130
131     //controllo cosa ho ricevuto
132     Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
133     //se è un messaggio (dovrebbe ricevere solo messaggi)
134     if (esito==Operations.MSG_FRIEND) {
135         //prendo il mittente ed il messaggio
136         String sender = (String) responseJSON.get ("FROM");
137         String msg = (String) responseJSON.get ("MSG");
138
139         //passo il messaggio all'interfaccia che gestisce le chat
140         chatHandlerGUI.postMessage(sender,msg);
141     }
142 }
143
144
145 }
146
```

UnloaderFileThread.java

```
1 package threads.listeners;
2
3 import java.nio.ByteBuffer;
4
5
6 /**
7  * Classe UnloaderFileThread.
8  * Thread a cui viene passato (al momento della sua
9  * creazione) un socket channel, in cui verrà inviato
10 * un file.
11 * Compito di questo thread è scaricare tale file e
12 * salvarlo nella cartella specificata nel ClientMain.
13 * @author Emilio Panti mat:531844
14 */
15
16 public class UnloaderFileThread implements Runnable {
17
18     //interfaccia che gestisce le chats
19     private ChatHandlerGUI chatHandlerGUI;
20
21     //socket channel in cui verrà spedito il file
22     private SocketChannel socketChannel;
23
24     /**
25      * Costruttore classe UnloaderFileThread.
26      * @param ChatHandlerGUI chatHandlerGUI: interfaccia che gestisce le chat.
27      * @param SocketChannel socketChannel: socket channel in cui verrà spedito
28      *                                     il file.
29      */
30
31     public UnloaderFileThread(ChatHandlerGUI chatHandlerGUI,
32                               SocketChannel socketChannel) {
33         this.chatHandlerGUI = chatHandlerGUI;
34         this.socketChannel = socketChannel;
35     }
36
37
38     /**
39      * Task che riceve un file dal socket channel e lo
40      * salva nella apposita cartella.
41      */
42
43     public void run() {
44
45         //prendo il path della cartella dove salvare il file
46         String path = ClientMain.PATH_FILE RECEIVED;
47
48         ByteBuffer buffer = ByteBuffer.allocate(400);
49
50         try {
51             //leggo l'oggetto json contenente le info sul file
52             socketChannel.read(buffer);
53             buffer.flip();
54
55             //creo un byte array della lunghezza del numero di bytes presenti in 'input'
56             byte[] bytes = new byte[buffer.remaining()];
57             buffer.get(bytes);
58
59             //trasformo in stringa il bytes array
60             String info = new String(bytes);
61
62         } catch (IOException e) {
63             e.printStackTrace();
64         }
65     }
66
67 }
```

UnloaderFileThread.java

```
68     //trasformo in formato JSON la stringa letta e prendo i campi d'interesse
69     JSONObject infoJSON = (JSONObject) new JSONParser().parse(info);
70     String sender = (String) infoJSON.get("SENDER");
71     String fileName = (String) infoJSON.get("FILE");
72     long size = (long) infoJSON.get("SIZE");
73
74     //mando al sender un segnale che ho ricevuto le info
75     buffer.clear();
76     buffer.put("ok".getBytes());
77     buffer.flip();
78     while(buffer.hasRemaining()) socketChannel.write(buffer);
79
80     //creo il file e apro il file channel (se esiste di già lo sovrascrivo)
81     FileChannel fileChannel = FileChannel.open(Paths.get(path+fileName),
82         StandardOpenOption.WRITE,StandardOpenOption.CREATE,
83         StandardOpenOption.TRUNCATE_EXISTING);
84
85     //collego direttamente il file channel al socket channel per il trasferimento
86     fileChannel.transferFrom(socketChannel, 0, size);
87
88     //faccio sapere all'utente che ha ricevuto un file dal sender
89     String txt = "["+sender+"]: has sent you the file \\""+fileName+"\\"";
90     chatHandlerGUI.postMessage(sender, txt);
91
92 } catch (Exception e) {
93     //se l'utente che sta inviando il file si disconnette
94     //e vengono alzate delle eccezioni a seguito di ciò
95     //non faccio niente.
96     e.printStackTrace();
97 }
98
99
100 }
101
```

FileFriendThread.java

```
1 package threads.senders;
2
3 import java.io.DataInputStream;
22
23
24 /**
25 * Classe FileFriendThread.
26 * Thread che si occupa di inviare un file ad un altro utente.
27 * Per prima cosa controlla che il file esista all'interno della
28 * apposita cartella specificata nella classe ClientMain.
29 * Se esiste, il server invia l'address dell'utente destinatario
30 * e la porta in cui è in ascolto per eventuali file.
31 * Dopo aver raccolto le informazioni necessarie invia il file
32 * tramite un socket channel.
33 * Se l'operazione va a buon fine aggiorna la chat verso l'utente
34 * destinatario, altrimenti la chat viene chiusa.
35 * @author Emilio Panti mat:531844
36 */
37 public class FileFriendThread implements Runnable {
38
39     //interfaccia della chat verso l'utente
40     private ChatNicknameGUI chatNicknameGUI;
41
42     //interfaccia che gestisce le chat aperte
43     private ChatHandlerGUI chatHandlerGUI;
44
45     //nickname dell'amico a cui si vuole inviare il file
46     private String receiver;
47
48     //nome del file da inviare
49     private String fileName;
50
51
52     /**
53      * Costruttore classe FileFriendThread.
54      * @param ChatNicknameGUI chatNicknameGUI: chat aperta verso l'utente destinatario.
55      * @param ChatHandlerGUI chatHandlerGUI: interfaccia che gestisce le chat.
56      * @param String receiver: nickname dell'utente destinatario.
57      * @param String fileName: nome del file da inviare.
58      */
59     public FileFriendThread(ChatNicknameGUI chatNicknameGUI, ChatHandlerGUI chatHandlerGUI,
60                           String receiver, String fileName) {
61         this.chatNicknameGUI = chatNicknameGUI;
62         this.chatHandlerGUI = chatHandlerGUI;
63         this.receiver = receiver;
64         this.fileName = fileName;
65     }
66
67
68     /**
69      * Task che esegue l'operazione di invio file ad un utente.
70      */
71     @SuppressWarnings("unchecked")
72     public void run() {
73         //path della cartella in cui cerco il file da inviare
74         String path = ClientMain.PATH_MY_FILE;
75     }
}
```

FileFriendThread.java

```
76 //variabile di controllo
77 boolean check = true;
78
79 //controllo che esista il file nella cartella apposita cercando di
80 //aprire un file channel verso esso
81 FileChannel fileChannel = null;
82 try {
83     fileChannel = FileChannel.open(Paths.get(path+fileName),StandardOpenOption.READ);
84 } catch (IOException e) {
85     //se non esiste il file
86     check = false;
87 }
88
89
90 //se il file che vuole inviare l'utente esiste
91 if(check) {
92     //prendo gli streams per comunicare con il server
93     DataOutputStream writer = ClientMain.WRITER;
94     DataInputStream reader = ClientMain.READER;
95
96     //creo la richiesta di invio file
97     JSONObject request = new JSONObject();
98     request.put("OP", "FILE_FRIEND");
99     request.put("ID", receiver);
100
101    //variabili per gestire la risposta del server
102    String response = null;
103    JSONObject responseJSON = null;
104
105    try {
106        //mando la richiesta
107        writer.writeUTF(request.toJSONString());
108        writer.flush();
109
110        //ricevo la risposta
111        response = reader.readUTF();
112
113        //trasformo in formato JSON la risposta ricevuta dal server
114        responseJSON = (JSONObject) new JSONParser().parse(response);
115
116    } catch (Exception e) {
117        //se c'è qualche problema di comunicazione con il server
118        //o se non è possibile parsare il messaggio ricevuto
119        e.printStackTrace();
120        //termino il client
121        ClientMain.cleanUp();
122    }
123
124
125    //controllo l'esito della risposta
126    Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
127    if (esito==Operations.OP_OK) {
128        //prendo l'indirizzo e la porta del destinatario
129        String address = (String) responseJSON.get ("ADDRESS");
130        long port = (long) responseJSON.get ("PORT_FILE");
131
132        //chiamo il metodo per inviare il file all'address mandato dal server
133    }
134}
```

FileFriendThread.java

```
133     try {
134         sendFile(address, port, fileChannel);
135
136         //comunico all'utente l'invio del file
137         String txt = "["+ClientMain.NICKNAME+"]: you have sent the file \\"+
138             fileName+"\" to \\""+receiver+"\"";
139         chatNicknameGUI.appendTextChat(txt);
140     } catch (IOException e) {
141         //eccezione se l'utente a cui si invia il file si disconnette
142         String msgErr = "ERR: \\""+receiver+"\" is offline now";
143
144         //chiudo la chat verso l'utente
145         chatHandlerGUI.removeChatNickname(receiver);
146
147         //apro l'interfaccia grafica per comunicare l'errore
148         ResponseGUI responseGUI = new ResponseGUI(msgErr);
149         responseGUI.setVisible(true);
150     }
151 }
152 else {
153     //prendo il messaggio di errore trasmesso dal server
154     String msgErr = (String) responseJSON.get ("MSG");
155
156     //chiudo la chat verso l'utente
157     chatHandlerGUI.removeChatNickname(receiver);
158
159     //apro l'interfaccia grafica per comunicare l'errore
160     ResponseGUI responseGUI = new ResponseGUI(msgErr);
161     responseGUI.setVisible(true);
162 }
163 }
164 //se non esiste il file
165 else {
166     String msgErr = "ERR: The file \\""+fileName+
167         "\\" does not exist in "+path;
168
169     //apro l'interfaccia grafica per comunicare l'errore
170     ResponseGUI responseGUI = new ResponseGUI(msgErr);
171     responseGUI.setVisible(true);
172 }
173 }
174
175 /**
176 * Metodo che invia all'address passato da parametro il file collegato
177 * al file channel (passato anch'esso da parametro)
178 * @param: String address: host address del client a cui inviare il file.
179 * @param: long port: porta in cui il ricevente è in ascolto di eventuali file.
180 * @param: FileChannel fileChannel: channel aperto verso il file da inviare.
181 * @throws IOException : se l'utente a cui vuole inviare il file si è disconnesso.
182 */
183
184 @SuppressWarnings("unchecked")
185 private void sendFile(String address, long port, FileChannel fileChannel)
186     throws IOException {
187
188     //prendo il nickname dell'utente che invia il file
189     String nickname = ClientMain.NICKNAME;
```

FileFriendThread.java

```
190
191     //apro la connessione
192     SocketAddress isa = new InetSocketAddress(address,(int) port);
193     SocketChannel socketChannel = SocketChannel.open(isa);
194     socketChannel.configureBlocking(true);
195     ByteBuffer buffer = ByteBuffer.allocate(400);
196
197     long size = fileChannel.size();
198
199     //creo l'oggetto json dove inserisco il nome dell'utente, il nome file
200     //e la size del file
201     JSONObject obj = new JSONObject();
202     obj.put("SENDER", nickname);
203     obj.put("FILE", fileName);
204     obj.put("SIZE", size);
205
206     //invio l'oggetto json al receiver
207     buffer.put(obj.toJSONString().getBytes());
208     buffer.flip();
209     while(buffer.hasRemaining()) socketChannel.write(buffer);
210     buffer.clear();
211
212     //ricevo dal receiver un segnale che ha ricevuto le info
213     socketChannel.read(buffer);
214     buffer.flip();
215
216     //mando il file
217     fileChannel.transferTo(0, size, socketChannel);
218 }
219 }
220 }
```

MsgCRThread.java

```
1 package threads.senders;
2
3 import java.io.DataInputStream;
15
16
17 /**
18 * Classe MsgCRThread.
19 * Thread che si occupa di inviare un messaggio testuale ad una
20 * chatroom.
21 * Per prima cosa controlla che la chat sia ancora attiva
22 * mandando una richiesta di controllo al server.
23 * Se la risposta è positiva spedisce al server un
24 * pacchetto UDP contenente il messaggio e le info sulla
25 * chatroom di destinazione.
26 * Sarà poi compito del server inoltrare il messaggio agli
27 * altri utenti iscritti alla chatroom.
28 * @author Emilio Panti mat:531844
29 */
30 public class MsgCRThread implements Runnable {
31
32     //nome e address della chatroom
33     private String chatroom;
34     private InetAddress ia;
35
36     //messaggio da inviare
37     private String msg;
38
39
40     /**
41      * Costruttore classe MsgCRThread.
42      * @param String chatroom: nome della chatroom.
43      * @param InetAddress ia: inet address del gruppo multicast relativo
44      *                       alla chatroom.
45      * @param String msg: messaggio da inviare.
46      */
47     public MsgCRThread(String chatroom, InetAddress ia, String msg) {
48         this.chatroom = chatroom;
49         this.ia = ia;
50         this.msg = msg;
51     }
52
53
54     /**
55      * Task che esegue l'operazione di invio messaggio ad una chatroom
56      */
57     @SuppressWarnings("unchecked")
58     public void run() {
59         //prendo gli streams per comunicare con il server
60         DataOutputStream writer = ClientMain.WRITER;
61         DataInputStream reader = ClientMain.READER;
62
63         //creo la richiesta di invio messaggio ad una chatroom
64         JSONObject request = new JSONObject();
65         request.put("OP", "MSG_CHATROOM");
66         request.put("ID", chatroom);
67
68         //variabili per gestire la risposta del server
```

MsgCRTThread.java

```
69     String response = null;
70     JSONObject responseJSON = null;
71
72     try {
73         //mando la richiesta
74         writer.writeUTF(request.toJSONString());
75         writer.flush();
76
77         //ricevo la risposta
78         response = reader.readUTF();
79
80         //trasformo in formato JSON la risposta ricevuta dal server
81         responseJSON = (JSONObject) new JSONParser().parse(response);
82
83     } catch (Exception e) {
84         //se c'è qualche problema di comunicazione con il server
85         //o se non è possibile parsare il messaggio ricevuto
86         e.printStackTrace();
87         //termino il client
88         ClientMain.cleanUp();
89     }
90
91
92     //controllo l'esito della risposta
93     Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
94     if (esito==Operations.OP_OK) {
95         //mando il messaggio UDP al server
96         sendMsg();
97     }
98     else {
99         //prendo il messaggio di errore trasmesso dal server
100        String msgErr = (String) responseJSON.get ("MSG");
101
102        //apro l'interfaccia grafica per comunicare l'errore
103        ResponseGUI responseGUI = new ResponseGUI(msgErr);
104        responseGUI.setVisible(true);
105    }
106}
107
108
109 /**
110 * Metodo che invia il messaggio scritto dall'utente al server
111 * con un pacchetto UDP.
112 */
113 @SuppressWarnings("unchecked")
114 private void sendMsg() {
115     //aggiungo il nickname dell'utente al messaggio
116     msg = "[" + ClientMain.NICKNAME + "]: " + msg;
117
118     //creo l'oggetto json che contiene il messaggio e il nome della chatroom
119     JSONObject msgJSON = new JSONObject();
120     msgJSON.put("TO", chatroom);
121     msgJSON.put("MSG", msg);
122
123     //prendo l'host address della chatroom
124     String address = ia.getHostAddress().toString();
125 }
```

MsgCRTThread.java

```
126     //creo l'oggetto json finale da inviare al server
127     JSONObject msgToSendJSON = new JSONObject();
128     msgToSendJSON.put("ADDRESS", address);
129     msgToSendJSON.put("MESSAGE", msgJSON);
130
131     //trasformo in stringa il messaggio da inviare al server
132     String msgToSend = msgToSendJSON.toJSONString();
133
134     //boolean per controllare che nell'invio del messaggio non avvengano errori
135     boolean error = false;
136
137     //apro un datagram socket ed invio il messaggio al server
138     try {
139         //prendo l'inet address del server
140         InetAddress iaServer = InetAddress.getByName(ClientMain.HOSTNAME);
141
142         //apro il datagram socket
143         @SuppressWarnings("resource")
144         DatagramSocket ds = new DatagramSocket();
145
146         byte [ ] data = msgToSend.getBytes();
147
148         //creo il datagram packet e lo invio
149         DatagramPacket dp = new
150             DatagramPacket(data,data.length,iaServer,ClientMain.PORT_UDP);
151         ds.send(dp);
152     }
153     catch (Exception e) {
154         error = true;
155     }
156
157     //se c'è stato un errore lo comunico
158     if(error) {
159         //apro l'interfaccia grafica per comunicare l'errore
160         ResponseGUI responseGUI = new ResponseGUI("ERR: An error occurred");
161         responseGUI.setVisible(true);
162     }
163 }
164 }
```

MsgFriendThread.java

```
1 package threads.senders;
2
3 import java.io.DataInputStream;
4
5
6 /**
7  * Classe MsgFriendThread.
8  * Thread che si occupa di inviare un messaggio testuale ad
9  * un utente.
10 * Manda il messaggio al server che poi lo girerà all'utente
11 * destinatario.
12 * Se l'operazione va a buon fine aggiorna la chat verso l'utente
13 * destinatario, altrimenti la chat viene chiusa.
14 * @author Emilio Panti mat:531844
15 */
16 public class MsgFriendThread implements Runnable {
17
18     //interfaccia della chat verso l'utente
19     private ChatNicknameGUI chatNicknameGUI;
20
21     //interfaccia che gestisce le chat aperte
22     private ChatHandlerGUI chatHandlerGUI;
23
24     //nickname dell'amico a cui si vuole inviare il file
25     private String receiver;
26
27     //messaggio da inviare
28     private String msg;
29
30
31     /**
32      * Costruttore classe MsgFriendThread.
33      * @param ChatNicknameGUI chatNicknameGUI: chat aperta verso l'utente destinatario.
34      * @param ChatHandlerGUI chatHandlerGUI: interfaccia che gestisce le chat.
35      * @param String receiver: nickname dell'utente destinatario.
36      * @param String msg: messaggio da inviare.
37      */
38
39     public MsgFriendThread(ChatNicknameGUI chatNicknameGUI, ChatHandlerGUI chatHandlerGUI,
40                           String receiver, String msg) {
41         this.chatNicknameGUI = chatNicknameGUI;
42         this.chatHandlerGUI = chatHandlerGUI;
43         this.receiver = receiver;
44         this.msg = msg;
45     }
46
47
48     /**
49      * Task che esegue l'operazione di invio messaggio ad un utente amico.
50      */
51     @SuppressWarnings("unchecked")
52     public void run() {
53         //prendo gli streams per comunicare con il server
54         DataOutputStream writer = ClientMain.WRITER;
55         DataInputStream reader = ClientMain.READER;
56
57         //creo la richiesta di invio messaggio
58         JSONObject request = new JSONObject();
59
60
61
62
63
64
65
66
67
```

MsgFriendThread.java

```
68     request.put("OP", "MSG_FRIEND");
69     request.put("TO", receiver);
70     request.put("MSG", msg);
71
72     //variabili per gestire la risposta del server
73     String response = null;
74     JSONObject responseJSON = null;
75
76     try {
77         //mando la richiesta
78         writer.writeUTF(request.toJSONString());
79         writer.flush();
80
81         //ricevo la risposta
82         response = reader.readUTF();
83
84         //trasformo in formato JSON la risposta ricevuta dal server
85         responseJSON = (JSONObject) new JSONParser().parse(response);
86
87     } catch (Exception e) {
88         //se c'è qualche problema di comunicazione con il server
89         //o se non è possibile parsare il messaggio ricevuto
90         e.printStackTrace();
91         //termino il client
92         ClientMain.cleanUp();
93     }
94
95
96     //controllo l'esito della risposta
97     Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
98     if (esito==Operations.OP_OK) {
99         //scrivo nella chat il messaggio inviato
100        String msgToPost = "[" + ClientMain.NICKNAME + "]: " + msg;
101        chatNicknameGUI.appendTextChat(msgToPost);
102    }
103    else {
104        //prendo il messaggio di errore trasmesso dal server
105        String msgErr = (String) responseJSON.get ("MSG");
106
107        //chiudo la chat verso l'utente
108        chatHandlerGUI.removeChatNickname(receiver);
109
110        //apro l'interfaccia grafica per comunicare l'errore
111        ResponseGUI responseGUI = new ResponseGUI(msgErr);
112        responseGUI.setVisible(true);
113    }
114 }
115
116 }
117 }
```

CloseThread.java

```
1 package threads.usersOp;
2
3 import java.io.DataOutputStream;
4
5
6 /**
7  * Classe CloseThread.
8  * Thread che comunica al server che il client sta per
9  * chiudere la connessione con esso. Successivamente chiama
10 * il metodo di cleanUp della classe ClientMain.
11 * @author Emilio Panti mat:531844
12 */
13 public class CloseThread implements Runnable {
14
15
16
17
18
19
20     /**
21      * Costruttore classe CloseThread.
22      */
23     public CloseThread() {
24
25
26
27
28     /**
29      * Task che esegue l'operazione di chiusura connessione e chiusura client
30      */
31     @SuppressWarnings("unchecked")
32     public void run() {
33
34         //streams per comunicare con il server
35         DataOutputStream writer = ClientMain.WRITER;
36
37         //creo la richiesta di chiusura in formato JSON
38         JSONObject request = new JSONObject ();
39         request.put("OP", "CLOSE");
40
41         try {
42             //mando la richiesta
43             writer.writeUTF(request.toJSONString());
44             writer.flush();
45         } catch (Exception e) {
46             //termino il client se c'è qualche problema di
47             //comunicazione con il server
48             ClientMain.cleanUp();
49             e.printStackTrace();
50         }
51
52         //chiamo la funzione per chiudere connessione e terminare client
53         ClientMain.cleanUp();
54     }
55 }
```

LoginThread.java

```
1 package threads.usersOp;
2
3 import java.io.DataInputStream;
27
28
29 /**
30 * Classe LoginThread.
31 * Thread che richiede al server (e ne gestisce la risposta)
32 * il login dell'utente al servizio SocialGossip.
33 * Se l'operazione va a buon fine vengono create l'interfaccia
34 * operativa (oggetto della classe OperativeGUI) e l'interfaccia
35 * che gestisce le chat aperte (oggetto della classe ChatHandlerGUI).
36 * Il server, in caso di esito positivo, allega alla risposta
37 * anche la lista amici dell'utente e la lista di tutte le chatrooms
38 * aperte (specificando a quali di esse l'utente sia già iscritto).
39 * Infine (sempre in caso di esito positivo) fa partire i thread che
40 * si occupano della ricezione di:
41 * - messaggi testuali provenienti da altri utenti (oggetto della classe
42 * ListenerMsg),
43 * - file inviati da altri utenti (oggetto della classe ListenerFile),
44 * - messaggi provenienti dalle varie chatrooms a cui l'utente è
45 * iscritto (oggetto della classe ListenerCR).
46 * Crea anche uno stub e lo registra sul server per ricevere le notifiche
47 * riguardanti l'utente.
48 * @author Emilio Panti mat:531844
49 */
50 public class LoginThread implements Runnable {
51
52     //nickname e psw
53     private String nickname;
54     private String psw;
55
56     //Interfacce attive al momento in cui viene invocato questo thread
57     private LoginGUI loginGUI;
58
59     //interfacce che verranno create dal thread se l'operazione di login andrà bene
60     private ChatHandlerGUI chatHandlerGUI;
61     private OperativeGUI operativeGUI;
62
63
64     /**
65      * Costruttore classe LoginThread.
66      * @param LoginGUI loginGUI: interfaccia di login che ha fatto partire
67      * questo thread.
68      * @param String nickname: nickname che ha inserito l'utente per loggarsi.
69      * @param String psw: password che ha inserito l'utente per loggarsi.
70      */
71     public LoginThread(LoginGUI loginGUI, String nickname, String psw) {
72         this.loginGUI = loginGUI;
73         this.nickname = nickname;
74         this.psw = psw;
75     }
76
77
78     /**
79      * Task che esegue l'operazione di login
80      */
```

LoginThread.java

```
81  @SuppressWarnings("unchecked")
82  public void run() {
83
84      //prendo gli streams per comunicare con il server
85      DataOutputStream writer = ClientMain.WRITER;
86      DataInputStream reader = ClientMain.READER;
87
88      //creo la richiesta di login in formato JSON
89      JSONObject request = new JSONObject ();
90      request.put("OP", "LOG");
91      request.put("ID", nickname);
92      request.put("PSW", psw);
93
94      //variabili per gestire la risposta del server
95      String response = null;
96      JSONObject responseJSON = null;
97
98      try {
99          //mando la richiesta
100         writer.writeUTF(request.toJSONString());
101         writer.flush();
102
103         //ricevo la risposta
104         response = reader.readUTF();
105
106         //trasformo in formato JSON la risposta ricevuta dal server
107         responseJSON = (JSONObject) new JSONParser().parse(response);
108
109     } catch (Exception e) {
110         //se c'è qualche problema di comunicazione con il server
111         //o se non è possibile parsare il messaggio ricevuto
112         e.printStackTrace();
113         //termino il client
114         ClientMain.cLeanUp();
115     }
116
117
118     //controllo l'esito della risposta
119     Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
120     if (esito==Operations.OP_OK) {
121
122         //salvo il nickname in ClientMain
123         ClientMain.NICKNAME = nickname;
124
125         //creo la operativa interface e l'interfaccia gestore delle chat
126         chatHandlerGUI = new ChatHandlerGUI();
127         operativeGUI = new OperativeGUI(chatHandlerGUI);
128
129         //mostro all'utente l'interfaccia operativa e chiudo quella di login
130         operativeGUI.setVisible(true);
131         loginGUI.dispose();
132
133         //prendo la lista amici e la appendo nell'area di text dell'interfaccia operativa
134         String friends = (String) responseJSON.get ("FRIENDS");
135         if (friends!=null) operativeGUI.setTextFriends(friends);
136
137     }
```

LoginThread.java

```
138     //prendo la lista delle chatrooms,la appendo nell'area di text dell'interfaccia
139     //operativa e mi connetto a quelle a cui sono già iscritto
140     JSONArray chatrooms = (JSONArray) responseJSON.get("CHATROOMS");
141     if (chatrooms!=null) connectToChatrooms(chatrooms);
142
143     //creazione e registrazione dello stub per ricevere le notifiche dal server
144     NotifyEventInterface callbackObj;
145     try {
146         callbackObj = new NotifyEventImpl(chatHandlerGUI,operativeGUI);
147         NotifyEventInterface stub = (NotifyEventInterface)
148             UnicastRemoteObject.exportObject(callbackObj, 0);
149         ServerInterface serverRMI = ClientMain.SERVER_RMI;
150         serverRMI.registerForCallback(stub,nickname);
151     } catch (RemoteException e) {
152         e.printStackTrace();
153         //termino il client
154         ClientMain.cleanUp();
155     }
156
157     //faccio partire il thread che si occupa di ricevere i messaggi dagli altri utenti
158     ListenerMsg listenerMsg = new ListenerMsg(chatHandlerGUI, nickname);
159     Thread thread = new Thread(listenerMsg);
160     thread.start();
161
162     //faccio partire il thread che si occupa di ricevere i file dagli altri utenti
163     ListenerFile listenerFile = new ListenerFile(chatHandlerGUI);
164     Thread thread2 = new Thread(listenerFile);
165     thread2.start();
166
167     //faccio partire il thread che si occupa di ricevere i messaggi dalle chatrooms
168     ListenerCR listenerChatrooms = new ListenerCR(chatHandlerGUI);
169     Thread thread3 = new Thread(listenerChatrooms);
170     thread3.start();
171
172     //comunico all'utente la riuscita dell'operazione
173     ResponseGUI responseGUI = new ResponseGUI("Your login was successful");
174     responseGUI.setVisible(true);
175 }
176 //c'è stato un errore
177 else {
178     //prendo il messaggio di errore trasmesso dal server
179     String msgErr = (String) responseJSON.get ("MSG");
180
181     //riattivo il bottone per il login nella interfaccia di login
182     loginGUI.enabledBtnLogin();
183
184     //apro l'interfaccia grafica per comunicare l'errore
185     ResponseGUI responseGUI = new ResponseGUI(msgErr);
186     responseGUI.setVisible(true);
187 }
188 }
189
190 /**
191 * Metodo che prende una lista di chatrooms in formato JSON.
192 * Per quelle a cui è già iscritto l'utente si registra ai relativi gruppi multicast.
```

LoginThread.java

```
194 * Alla fine appende nell'interfaccia operativa la lista delle chatroom
195 * esistenti al momento del login (specificando a quali è iscritto).
196 * @param JSONArray chatrooms : lista delle chatrooms in formato JSON.
197 */
198 private void connectToChatrooms(JSONArray chatrooms) {
199     //multicast socket per registrare l'utente alle chatrooms a cui è già iscritto
200     MulticastSocket ms = ClientMain.MS;
201
202     //Stringa dove inserisco il nome delle chatrooms (e a quali l'utente è iscritto)
203     String listChatrooms = "";
204
205     //stringa end of line
206     String eol = System.getProperty("line.separator");
207
208     //creo un iteratore per scorrere la lista delle chatrooms
209     @SuppressWarnings("unchecked")
210     Iterator<JSONObject> iterator = chatrooms.iterator();
211
212     while (iterator.hasNext()) {
213         //prendo un oggetto json della lista
214         JSONObject chatroomJSON = iterator.next();
215
216         //prendo il nome della chatroom
217         String chatroom = (String) chatroomJSON.get("ID");
218         //indirizzo multicast della chatroom
219         String address = (String) chatroomJSON.get("ADDRESS");
220         //boolean che mi dice se l'utente è iscritto ad esso o meno
221         Boolean signed = (Boolean) chatroomJSON.get("SIGNED");
222
223         //se l'utente è iscritto
224         if(signed) {
225             //per sapere se la registrazione va bene
226             Boolean check = true;
227
228             //mi registro al gruppo multicast
229             InetAddress ia = null;
230             try {
231                 ia = InetAddress.getByName(address);
232                 ms.joinGroup (ia);
233             } catch (Exception e) {
234                 e.printStackTrace();
235                 check = false;
236             }
237
238             //se la registrazione è andata bene
239             if(check) {
240                 //apro la chat verso tale chatrooms
241                 chatHandlerGUI.addChatroom(chatroom,ia);
242
243                 //aggiorno la stringa della lista chatrooms
244                 listChatrooms = listChatrooms + chatroom + " (signed up)" + eol;
245             }
246         }
247         //se l'utente non è iscritto
248         else {
249             //aggiorno la stringa della lista chatrooms
250             listChatrooms = listChatrooms + chatroom + eol;
```

LoginThread.java

```
251         }
252     }
253
254     //appendo la lista delle chatrooms nell'area di text dell'interfaccia operativa
255     operativeGUI.setTextChatrooms(listChatrooms);
256
257 }
258
```

RegistrationThread.java

```
1 package threads.usersOp;
2
3 import java.io.DataInputStream;
25
26
27 /**
28 * Classe RegistrationThread.
29 * Thread che richiede al server (e ne gestisce la risposta)
30 * la registrazione di un nuovo utente al servizio SocialGossip.
31 * Se l'operazione va a buon fine vengono create l'interfaccia
32 * operativa (oggetto della classe OperativeGUI) e l'interfaccia
33 * che gestisce le chat aperte (oggetto della classe ChatHandlerGUI).
34 * Il server, in caso di esito positivo, allega alla risposta
35 * anche la lista di tutte le chatrooms aperte.
36 * Infine (sempre in caso di esito positivo) fa partire i thread che
37 * si occupano della ricezione di:
38 * - messaggi testuali provenienti da altri utenti (oggetto della classe
39 * ListenerMsg),
40 * - file inviati da altri utenti (oggetto della classe ListenerFile),
41 * - messaggi provenienti dalle varie chatrooms a cui l'utente è
42 * iscritto (oggetto della classe ListenerCR).
43 * Crea anche uno stub e lo registra sul server per ricevere le notifiche
44 * riguardanti l'utente.
45 * @author Emilio Panti mat:531844
46 */
47 public class RegistrationThread implements Runnable {
48
49     //nickname e psw
50     private String nickname;
51     private String psw;
52     private String language;
53
54     //Interfacce attive al momento in cui viene invocato questo thread
55     private RegistrationGUI registrationGUI;
56
57
58     /**
59      * Costruttore classe RegistrationThread.
60      * @param RegistrationGUI registrationGUI: interfaccia di registrazione
61      * che ha fatto partire questo thread.
62      * @param String nickname: nickname che ha inserito l'utente per registrarsi.
63      * @param String psw: password che ha inserito l'utente per registrarsi.
64      * @param String language: lingua selezionata dall'utente.
65      */
66     public RegistrationThread(RegistrationGUI registrationGUI, String nickname,
67                               String psw, String language) {
68         this.registrationGUI = registrationGUI;
69         this.nickname = nickname;
70         this.psw = psw;
71         this.language = language;
72     }
73
74
75     /**
76      * Task che esegue l'operazione di registrazione
77      */
78     @SuppressWarnings("unchecked")
```

RegistrationThread.java

```
79  public void run() {
80
81      //prendo gli streams per comunicare con il server
82      DataOutputStream writer = ClientMain.WRITER;
83      DataInputStream reader = ClientMain.READER;
84
85      //creo la richiesta di registrazione in formato JSON
86      JSONObject request = new JSONObject ();
87      request.put("OP", "REG");
88      request.put("ID", nickname);
89      request.put("PSW", psw);
90      request.put("LANGUAGE", language);
91
92      //variabili per gestire la risposta del server
93      String response = null;
94      JSONObject responseJSON = null;
95
96      try {
97          //mando la richiesta
98          writer.writeUTF(request.toJSONString());
99          writer.flush();
100
101         //ricevo la risposta
102         response = reader.readUTF();
103
104         //trasformo in formato JSON la risposta ricevuta dal server
105         responseJSON = (JSONObject) new JSONParser().parse(response);
106
107     } catch (Exception e) {
108         //se c'è qualche problema di comunicazione con il server
109         //o se non è possibile parsare il messaggio ricevuto
110         e.printStackTrace();
111         //termino il client
112         ClientMain.cleanUp();
113     }
114
115
116     //controllo l'esito della risposta
117     Operations esito = Operations.valueOf((String) responseJSON.get("OP"));
118     if (esito==Operations.OP_OK) {
119
120         //salvo il nickname in ClientMain
121         ClientMain.NICKNAME = nickname;
122
123         //creo la operative interface e l'interfaccia gestore delle chat
124         ChatHandlerGUI chatHandlerGUI = new ChatHandlerGUI();
125         OperativeGUI operativeGUI = new OperativeGUI(chatHandlerGUI);
126
127         //prendo la lista delle chatrooms e la appendo nell'interfaccia operativa
128         JSONArray chatrooms = (JSONArray) responseJSON.get("CHATROOMS");
129         if (chatrooms!=null) getListChatrooms(chatrooms,operativeGUI);
130
131         //creazione e registrazione dello stub per ricevere le notifiche dal server
132         NotifyEventInterface callbackObj;
133         try {
134             callbackObj = new NotifyEventImpl(chatHandlerGUI,operativeGUI);
135             NotifyEventInterface stub =
```

RegistrationThread.java

```
136             (NotifyEventInterface) UnicastRemoteObject.exportObject(callbackObj,
137             0);
138             ServerInterface serverRMI = ClientMain.SERVER_RMI;
139             serverRMI.registerForCallback(stub,nickname);
140             } catch (RemoteException e) {
141                 e.printStackTrace();
142                 //termino il client
143                 ClientMain.cleanUp();
144             }
145             //faccio partire il thread che si occupa di ricevere i messaggi dagli altri utenti
146             ListenerMsg listenerMsg = new ListenerMsg(chatHandlerGUI, nickname);
147             Thread thread = new Thread(listenerMsg);
148             thread.start();
149
150             //faccio partire il thread che si occupa di ricevere i file dagli altri utenti
151             ListenerFile listenerFile = new ListenerFile(chatHandlerGUI);
152             Thread thread2 = new Thread(listenerFile);
153             thread2.start();
154
155             //faccio partire il thread che si occupa di ricevere i messaggi dalle chatrooms
156             ListenerCR listenerChatrooms = new ListenerCR(chatHandlerGUI);
157             Thread thread3 = new Thread(listenerChatrooms);
158             thread3.start();
159
160             //mostro all'utente l'interfaccia operativa e chiudo quella di registrazione
161             operativeGUI.setVisible(true);
162             registrationGUI.dispose();
163
164             //comunico all'utente la riuscita dell'operazione
165             ResponseGUI responseGUI = new ResponseGUI("Your registration was successful");
166             responseGUI.setVisible(true);
167
168         }
169         //c'è stato un errore
170         else {
171             //prendo il messaggio di errore trasmesso dal server
172             String msgErr = (String) responseJSON.get ("MSG");
173
174             //riattivo il bottone per la registrazione nella interfaccia di registrazione
175             registrationGUI.enabledBtnRegister();
176
177             //apro l'interfaccia grafica per comunicare l'errore
178             ResponseGUI responseGUI = new ResponseGUI(msgErr);
179             responseGUI.setVisible(true);
180         }
181     }
182
183
184 /**
185 * Metodo che prende una lista di chatrooms in formato JSON.
186 * Ne ricava una lista da appendere nella text area relativa alle chatroom
187 * nella interfaccia operativa
188 * @param JSONArray chatrooms : lista delle chatrooms in formato JSON.
189 * @param OperativeGUI operativeGUI : interfaccia dove appendere la lista.
190 */
191 private void getListChatrooms(JSONArray chatrooms,OperativeGUI operativeGUI) {
```

RegistrationThread.java

```
192     //Stringa dove inserisco il nome delle chatrooms
193     String listChatrooms = "";
194
195     //stringa end of line
196     String eol = System.getProperty("line.separator");
197
198     //creo un iteratore per scorrere la lista delle chatrooms
199     @SuppressWarnings("unchecked")
200     Iterator<JSONObject> iterator = chatrooms.iterator();
201
202     while (iterator.hasNext()) {
203         //prendo un oggetto json della lista
204         JSONObject chatroomJSON = iterator.next();
205
206         //prendo il nome della chatroom
207         String chatroom = (String) chatroomJSON.get("ID");
208
209         //aggiorno la stringa della lista chatrooms
210         listChatrooms = listChatrooms + chatroom + eol;
211     }
212
213     //appendo la lista delle chatrooms nell'area di text dell'interfaccia operativa
214     operativeGUI.setTextChatrooms(listChatrooms);
215 }
216 }
217 }
```

CODICE LATO SERVER

(Cartella SocialGossipServer/src)

NB: il codice dei seguenti files (presenti anche nel lato client) è già stato riportato in precedenza, pertanto non verrà fatto nuovamente:

- Operations.java (nel package enumerations),
- NotifyEventInterface.java (nel package notificationRMI),
- ServerInterface.java (nel package notificationRMI).

ServerMain.java

```
1 package server;
2
3 import java.net.ServerSocket;
16
17
18 /**
19 * Classe ServerMain.
20 * Contiene il main che gestisce il server dell'applicazione
21 * SocialGossip.
22 * Crea una hash map per gli utenti registrati (ConcurrentHashMap
23 * <String,User>) e una per le chatrooms (oggetto della classe
24 * HashChatrooms).
25 * Offre un servizio remoto per permettere ai clients di registrarsi
26 * e ricevere le notifiche che li riguardano.
27 * Fa partire un thread (oggetto della classe HandlerMsgChatrooms)
28 * che si occupa di smistare alle chatrooms di competenza i messaggi
29 * ricevuti dagli utenti destinati ad esse.
30 * Infine si mette in attesa ciclica di nuove connessioni; che vengono
31 * passate e gestite da un thread pool (oggetto della classe ThreadPool,
32 * creato precedentemente).
33 * @author Emilio Panti mat:531844
34 */
35 public class ServerMain {
36
37     //porta per ricevere le connessioni TCP
38     public static int PORT_TCP = 6012;
39
40     //porta per ricevere i messaggi da girare alle chatrooms
41     public static int PORT_UDP = 6013;
42
43     //porta dove creare il registry ed esporre oggetti remoti
44     public static int PORT_RMI = 6014;
45
46     //porta dove i client aprono il multicast socket per
47     //ricevere i messaggi dalle chatrooms
48     public static int PORT_CLIENT = 6000;
49
50     //numero max delle connessioni contemporanee
51     public static int MAX_CONN = 12;
52
53     public static void main(String[] args) {
54
55         //hash map per gli utenti
56         ConcurrentHashMap <String,User> hashUsers = new ConcurrentHashMap <String,User>();
57
58         //hash map per le chatrooms
59         HashChatrooms hashChatrooms = new HashChatrooms();
60
61         //creo il thread pool che gestirà le connessioni
62         ThreadPool threadPool = new ThreadPool(hashUsers,hashChatrooms);
63
64         //faccio partire il thread che si occupa di smistare i messaggi alle chatrooms
65         HandlerMsgChatrooms handlerMsgChatrooms = new HandlerMsgChatrooms();
66         Thread thread = new Thread(handlerMsgChatrooms);
67         thread.start();
68
69         //listening socket
```

ServerMain.java

```
70     ServerSocket serverSocket;
71
72     try {
73         //registrazione del servizio remoto per le notifiche presso il registry
74         ServerImpl server = new ServerImpl(hashUsers);
75         ServerInterface stub = (ServerInterface) UnicastRemoteObject.exportObject
76             (server,39000);
77         LocateRegistry.createRegistry(PORT_RMI);
78         Registry registry = LocateRegistry.getRegistry(PORT_RMI);
79         registry.bind ("Server", stub);
80
81         //apro il server socket
82         serverSocket = new ServerSocket(PORT_TCP);
83
84         System.out.println("Server ready");
85
86         //ciclo di vita del server
87         while(true){
88
89             //active socket verso un client
90             Socket socket = serverSocket.accept();
91
92             //passo la nuova connessione al thread pool
93             threadPool.newConnection(socket);
94         } catch (Exception e) {
95             // TODO Auto-generated catch block
96             e.printStackTrace();
97             System.exit(0);
98         }
99     }
100 }
101 }
```

Chatroom.java

```
1 package dataStructures;
2
3 import java.net.InetAddress;
4
5
6
7
8 /**
9  * Classe Chatroom.
10 * Contiene i metodi per creare ed eseguire operazioni
11 * sulle chatroom.
12 * @author Emilio Panti mat:531844
13 */
14 public class Chatroom {
15
16     //nome chatroom
17     private String id;
18
19     //creatore della chatroom
20     private String creator;
21
22     //utenti connessi
23     private int usersConn;
24
25     //lista iscritti
26     private ListUsers subscribers;
27
28     //multicast address della chatroom
29     private InetAddress address;
30
31     //variabile per stabilire se una chatroom è attiva o in chiusura
32     private boolean active;
33
34
35 /**
36  * Costruttore classe Chatroom.
37  * @param String id: nome della chatroom.
38  * @param User user: utente creatore della chatroom.
39  */
40     public Chatroom(String id, User user) {
41         this.id = id;
42         creator = user.getNickname();
43
44         active = true;
45         usersConn = 0;
46
47         subscribers = new ListUsers();
48         //aggiungo il creatore agli iscritti
49         addSubscriber(user);
50     }
51
52
53 /**
54  * Override del metodo equals.
55  * @param: Object obj: oggetto da confrontare.
56  * @return true: se le due chatrooms hanno lo stesso id,
57  *             false: altrimenti.
58  */
59     @Override
```

Chatroom.java

```
60  public boolean equals(Object obj) {
61      if (obj==null) return false;
62      Chatroom chatroom = (Chatroom) obj;
63      return id.equals(chatroom.getId());
64  }
65
66
67 /**
68 * Override del metodo hashCode.
69 * Ho effettuato l'override di hashCode solo per formalità
70 * dato che ho fatto l'override di equals.
71 * @return int: restituisce il risultato del metodo hashCode chiamato
72 *             sull'id della chatroom (anche il metodo equals di cui ho
73 *             fatto l'override sopra si basa sul confronto del campo id)
74 */
75 @Override
76 public int hashCode() {
77     return id.hashCode();
78 }
79
80
81 /**
82 * Metodo per settare l'indirizzo multicast della chatroom.
83 * @param InetSocketAddress: indirizzo multicast della chatroom
84 */
85 public void setAddress(InetAddress address) {
86     this.address = address;
87 }
88
89
90 /**
91 * Metodo che restituisce l'address della chatroom
92 * @return InetAddress: address della chatroom
93 */
94 public InetAddress getAddress() {
95     return address;
96 }
97
98
99 /**
100 * Metodo che restituisce l'id della chatroom
101 * @return String: nome della chatroom
102 */
103 public String getId() {
104     return id;
105 }
106
107
108 /**
109 * Metodo per incrementare gli utenti online nella chatroom.
110 */
111 public synchronized void increaseUsersConn() {
112     usersConn++;
113 }
114
115
116 /**
```

Chatroom.java

```
117     * Metodo per decrementare gli utenti online nella chatroom.  
118     */  
119     public synchronized void decreaseUsersConn() {  
120         usersConn--;  
121     }  
122  
123  
124     /**  
125      * Metodo per sapere se ci sono almeno due utenti online  
126      * iscritti alla chatroom.  
127      * @return true: se ci sono,  
128      *          false: se non ci sono o se la chatroom è in chiusura  
129      */  
130     public synchronized boolean checkUsersConn() {  
131         if(active && usersConn>1) return true;  
132         else return false;  
133     }  
134  
135  
136     /**  
137      * Metodo che restituisce un oggetto in formato JSON contenente l'id  
138      * e l'indirizzo della chatroom.  
139      * @return JSONObject: se la chatroom è ancora attiva  
140      *                      null: se la chatroom è in chiusura.  
141      */  
142     @SuppressWarnings("unchecked")  
143     public synchronized JSONObject getInfoJSON() {  
144         if(active) {  
145             JSONObject info = new JSONObject();  
146             info.put("ID", id);  
147             info.put("ADDRESS", address.getHostAddress().toString());  
148             return info;  
149         }  
150         else return null;  
151     }  
152  
153  
154     /**  
155      * Metodo per controllare se il nickname passato da parametro  
156      * corrisponde al nome del creatore della chatroom.  
157      * @param String nickname: nickname da controllare.  
158      * @return true: se i due nomi corrispondono,  
159      *          false: altrimenti.  
160      */  
161     public boolean checkCreator (String nickname) {  
162         return (creator.equals(nickname));  
163     }  
164  
165  
166     /**  
167      * Metodo per aggiungere un utente alla lista iscritti.  
168      * @param User user: user da inserire nella lista iscritti.  
169      * @return true: se non era già presente tale utente nella lista,  
170      *          false: se era già iscritto oppure se la chatroom  
171      *                  non è più attiva ma ancora non è stata eliminata  
172      *                  dalla hash map.  
173      */
```

Chatroom.java

```
174     public boolean addSubscriber (User user) {
175         boolean check = false;
176
177         synchronized (this) {
178             if (active) check = true;
179         }
180
181         if (check) {
182             if(subscribers.add(user)) {
183                 //aumento di 1 gli utenti connessi
184                 increaseUsersConn();
185                 return true;
186             }
187             else return false;
188         }
189         else return false;
190     }
191
192
193     /**
194      * Metodo per disattivare una chatroom e notificare a tutti i suoi
195      * iscritti che sta per essere chiusa.
196      * @return InetAddress: l'address che utilizzava.
197      */
198     public InetAddress close() {
199         //"disattivo" la chatroom in attesa che venga eliminata dalla hash
200         //map che la contiene
201         synchronized (this) {
202             active = false;
203         }
204
205         //notifco a tutti i suoi iscritti che sta per essere cancellata
206         subscribers.notifyAllClose(this);
207
208         return address;
209     }
210 }
```

HashChatrooms.java

```
1 package dataStructures;
2
3 import java.net.InetAddress;
10
11
12 /**
13 * Classe HashChatrooms.
14 * Struttura dati per la gestione delle chatrooms, che
15 * utilizza due 'sotto-strutture dati':
16 * - una hash map (oggetto della classe HashMap <String,Chatroom>),
17 * - una lista di InetAddress (oggetto della classe LinkedList <InetAddress>).
18 * Nella hash map vengono inserite e rimosse le chatrooms. I metodi
19 * che operano su di essa sono sincronizzati in modo manuale per
20 * evitare inconsistenze. E' previsto un numero massimo di chatrooms
21 * esistenti, deciso dalla variabile statica MAX.
22 * La lista contiene MAX InetAddress di indirizzi multicast.
23 * Ogni volta che viene inserita una chatroom nella hash map le viene
24 * assegnato un indirizzo multicast, prelevato dalla lista.
25 * Se la lista è vuota e se la hash map ha raggiunto MAX elementi
26 * non è possibile inserire nuove chatrooms.
27 * Quando una chatroom viene chiusa e rimossa dalla hash map,
28 * viene reinserito nella lista degli InetAddress l'indirizzo
29 * che utilizzava, così da renderlo nuovamente disponibile.
30 * @author Emilio Panti mat:531844
31 */
32 public class HashChatrooms {
33
34     //numero massimo di chatrooms
35     public static int MAX = 256;
36
37     //base address (le possibili 256 chatrooms hanno una parte di indirizzo
38     //multicast uguale)
39     public static String baseAddress = "224.0.0.";
40
41     //Lista che contiene gli indirizzi multicast disponibili per le chtrooms
42     private LinkedList<InetAddress> listAddress;
43
44     //hash map che contiene le chatrooms
45     private HashMap <String,Chatroom> hc;
46
47
48     /**
49      * Costruttore classe HashChatrooms.
50      */
51     public HashChatrooms() {
52         hc = new HashMap <String,Chatroom>();
53         listAddress = new LinkedList<InetAddress>();
54
55         //inserisco MAX address nella lista address
56         int i = 0;
57         for(i=0;i<MAX;i++) {
58             String addressStr = baseAddress + i;
59             try {
60                 InetAddress address = InetAddress.getByName(addressStr);
61                 //aggiungo il nuovo address alla lista
62                 listAddress.add(address);
63             } catch (UnknownHostException e) {
```

HashChatrooms.java

```
64             e.printStackTrace();
65         }
66     }
67 }
68
69
70 /**
71 * Metodo per aggiungere una nuova chatroom.
72 * @param Chatroom chatroom: chatroom da aggiungere.
73 * @return InetAddress: l'indirizzo della chatroom appena aggiunta
74 *         null : se è già stato raggiunto il numero max o
75 *                 se già esiste una chatroom con lo stesso nome.
76 */
77 public synchronized InetAddress add(Chatroom chatroom) {
78     //se ho raggiunto il numero max
79     if ((hc.size() >= MAX) || (listAddress.isEmpty())) return null;
80
81     //altrimenti prendo il nome della chatroom
82     String id = chatroom.getId();
83
84     //se esiste già una chatroom con tale nome
85     if (hc.containsKey(id)) return null;
86     else {
87         //prelevo un indirizzo disponibile dalla list address
88         //e lo assegno alla chatroom
89         InetAddress address = listAddress.remove();
90         chatroom.setAddress(address);
91
92         //la aggiungo alla hash map
93         hc.put(id,chatroom);
94
95         return address;
96     }
97 }
98
99
100 /**
101 * Metodo per cercare e restituire una chatroom.
102 * @param String id: stringa che contiene il nome della chatroom.
103 * @return Chatroom: la chatroom cercata
104 *         null : se non esiste una chatroom con tale nome.
105 */
106 public synchronized Chatroom get(String id) {
107     return hc.get(id);
108 }
109
110
111 /**
112 * Metodo per rimuovere e cancellare una chatroom.
113 * @param String id: nome della chatroom da cancellare .
114 * @param String id: nome dell'utente che ha chiesto di cancellarla.
115 * @return true: se la chatroom viene cancellata correttamente
116 *         false: se la chatroom non esiste o se il creatore è un altro.
117 */
118 public boolean remove(String id, String nickname) {
119     //cerco la chatroom
120     Chatroom chatroom = this.get(id);
```

HashChatrooms.java

```
121
122     //se esiste
123     if(chatroom!=null) {
124         //se il creatore è lo stesso che vuole cancellarla
125         if(chatroom.checkCreator(nickname)) {
126
127             synchronized (this) {
128                 //chiamo la funzione di chiusura nella chatroom
129                 //che mi restituisce l'address che utilizzava
130                 InetAddress address = chatroom.close();
131
132                 //rimuovo la chatroom dalla hash map
133                 hc.remove(id);
134
135                 //rendo nuovamente disponibile l'address
136                 listAddress.add(address);
137             }
138
139             return true;
140         }
141         //se non è il creatore
142         else return false;
143     }
144     else return false;
145 }
146
147 /**
148 * Metodo per rimuovere e cancellare una chatroom.
149 * @return JSONArray: se la hashmap ha almeno un elemento,
150 *         null: altrimenti.
151 */
152 @SuppressWarnings("unchecked")
153 public synchronized JSONArray getListChatrooms() {
154     //controllo che non sia vuota la hash map
155     if(hc.isEmpty()) return null;
156
157     JSONArray listChatrooms = new JSONArray();
158
159     //inserisco tutte le chatroom nel JSONarry
160     for(Chatroom chatroom : hc.values()) {
161         //prendo le info della chatroom
162         JSONObject infoJSON = chatroom.getInfoJSON();
163
164         //lo aggiungo al JSONArray
165         if(infoJSON != null) listChatrooms.add(infoJSON);
166     }
167
168     return listChatrooms;
169 }
170 }
171 }
172 }
```

ListChatrooms.java

```
1 package dataStructures;
2
3 import java.util.ArrayList;
4
5
6
7
8
9
10 /**
11  * Classe ListChatrooms.
12  * Contiene i metodi per creare, gestire ed eseguire
13  * operazioni sulle liste di chatrooms.
14  * La struttura base utilizzata è un oggetto della classe
15  * ArrayList <Chatroom>.
16  * @author Emilio Panti mat:531844
17 */
18 public class ListChatrooms {
19
20     private ArrayList<Chatroom> listChatrooms;
21
22     /**
23      * Costruttore classe ListChatrooms.
24      */
25     public ListChatrooms() {
26         listChatrooms = new ArrayList<Chatroom>();
27     }
28
29
30     /**
31      * Metodo che aggiunge una chatroom alla lista.
32      * @param Chatroom chatroom: chatroom da aggiungere.
33      * @return true: se non era già presente tale chatroom nella lista,
34      *          false: altrimenti.
35      */
36     public synchronized boolean add(Chatroom chatroom) {
37         if(!listChatrooms.contains(chatroom)) {
38             listChatrooms.add(chatroom);
39             return true;
40         }
41         else return false;
42     }
43
44
45     /**
46      * Metodo restituisce la chatroom che ha come id la stringa
47      * passata da parametro.
48      * @param String id: id della chatroom da cercare.
49      * @return Chatroom: se viene trovato tale chatroom,
50      *          null: altrimenti.
51      */
52     public synchronized Chatroom get(String id) {
53         //creo un iteratore per scorrere la lista chatroom
54         Iterator<Chatroom> iterator = listChatrooms.iterator();
55
56         //chatroom che restituisco
57         Chatroom found = null;
58
59         while (iterator.hasNext() && found==null) {
60             //prendo una chatroom dalla lista
61             Chatroom chatroom = iterator.next();
```

ListChatrooms.java

```
62
63         //se ha l'id che cercavo
64         if(id.equals(chatroom.getId())) found = chatroom;
65
66     }
67
68     return found;
69 }
70
71 /**
72 * Metodo che rimuove la chatroom passata come parametro dalla lista.
73 * @param Chatroom chatroom: chatroom da rimuovere.
74 */
75 public synchronized void remove(Chatroom chatroom) {
76     listChatrooms.remove(chatroom);
77 }
78
79
80 /**
81 * Metodo che prende come argomento un JSONArray contenente oggetti JSON.
82 * Ogni oggetto JSON contiene il nome e l'indirizzo di una chatroom.
83 * A tali oggetti JSON viene aggiunto un booleano in base al fatto o meno
84 * che la chatroom sia presente anche in listChatrooms.
85 * @param JSONArray allChatrooms: JSONArray contenente le info su tutte
86 * le chatroom registrate sul server.
87 * @return JSONArray: contiene allChatrooms aggiornata come sopra descritto
88 * null: se il allChatrooms è null
89 */
90 @SuppressWarnings("unchecked")
91 public synchronized JSONArray listChatroomSigned(JSONArray allChatrooms) {
92
93     if(allChatrooms==null) return null;
94
95
96     //creo un iteratore per scorrere il JSONArray
97     Iterator<JSONObject> iterator = allChatrooms.iterator();
98
99     while (iterator.hasNext()) {
100         //prendo un oggetto json della lista
101         JSONObject objJSON = iterator.next();
102
103         //prendo il nome della chatroom
104         String id = (String) objJSON.get("ID");
105
106         //se la chatroom è presente in listChatrooms
107         if(this.get(id) != null) {
108             objJSON.put("SIGNED", true);
109         }
110         //altrimenti
111         else objJSON.put("SIGNED", false);
112     }
113
114     return allChatrooms;
115 }
116
117
118 /**
```

ListChatrooms.java

```
119     * Metodo che incrementa il numero degli utenti connessi di 1 per ogni
120     * chatroom presente nella lista.
121     */
122     public synchronized void increaseAll() {
123         //creo un iteratore per scorrere la lista chatroom
124         Iterator<Chatroom> iterator = listChatrooms.iterator();
125
126         while (iterator.hasNext()) {
127             //prendo una chatroom dalla lista
128             Chatroom chatroom = iterator.next();
129             chatroom.increaseUsersConn();
130         }
131     }
132
133
134 /**
135     * Metodo che decrementa il numero degli utenti connessi di 1 per ogni
136     * chatroom presente nella lista.
137     */
138     public synchronized void decreaseAll() {
139         //creo un iteratore per scorrere la lista chatroom
140         Iterator<Chatroom> iterator = listChatrooms.iterator();
141
142         while (iterator.hasNext()) {
143             //prendo una chatroom dalla lista
144             Chatroom chatroom = iterator.next();
145             chatroom.decreaseUsersConn();
146         }
147     }
148 }
```

ListUsers.java

```
1 package dataStructures;
2
3 import java.net.InetAddress;
4
5
6
7
8 /**
9  * Classe ListUsers.
10 * Contiene i metodi per creare, gestire ed eseguire
11 * operazioni sulle liste di utenti.
12 * La struttura base utilizzata è un oggetto della classe
13 * ArrayList <User>.
14 * @author Emilio Panti mat:531844
15 */
16 public class ListUsers {
17
18     private ArrayList<User> listUsers;
19
20
21     /**
22      * Costruttore classe ListUsers.
23      */
24     public ListUsers() {
25         listUsers = new ArrayList<User>();
26     }
27
28
29     /**
30      * Metodo che aggiunge un utente alla lista.
31      * @param User user: utente da aggiungere.
32      * @return true: se non era già presente tale utente nella lista,
33      *             false: altrimenti.
34      */
35     public synchronized boolean add(User user) {
36         if(!listUsers.contains(user)) {
37             listUsers.add(user);
38             return true;
39         }
40         else return false;
41     }
42
43
44     /**
45      * Metodo restituisce l'utente che ha per nickname la stringa
46      * passata da parametro.
47      * @param String nickname: nickname dell'utente da cercare.
48      * @return User: se viene trovato tale utente,
49      *             null: altrimenti.
50      */
51     public synchronized User get(String nickname) {
52         //creo un iteratore per scorrere la lista utenti
53         Iterator<User> iterator = listUsers.iterator();
54
55         //utente che restituisco
56         User found = null;
57
58         while (iterator.hasNext() && found==null) {
59             //prendo un utente dalla lista
```

ListUsers.java

```
60         User user = iterator.next();
61
62         //se ha il nickname che cercavo
63         if(nickname.equals(user.getNickname())) found = user;
64
65     }
66
67     return found;
68 }
69
70
71 /**
72 * Metodo che restituisce una stringa contenente i nomi degli utenti in lista.
73 * @return String: se ha almeno un utente la lista,
74 *         null, altrimenti.
75 */
76 public synchronized String getList() {
77     //se la lista amici è vuota ritorno null
78     if(listUsers.isEmpty()) return null;
79
80     //altrimenti creo un iteratore per scorrere la lista utenti
81     Iterator<User> iterator = listUsers.iterator();
82
83     //Stringa contiene i nickname
84     String str = "";
85
86     //stringa end of line
87     String eol = System.getProperty("line.separator");
88
89     while (iterator.hasNext()) {
90         //prendo un utente dalla lista
91         User user = iterator.next();
92
93         //aggiungo il nickname
94         str = str + user.getNickname() + eol;
95     }
96
97     return str;
98 }
99
100
101 /**
102 * Metodo per notificare a tutti gli utenti della lista che
103 * l'utente passato da parametro è adesso online.
104 * @param String name: nome dell'utente.
105 */
106 public synchronized void notifyAllOnline(String name) {
107     //creo un iteratore per scorrere la lista utenti
108     Iterator<User> iterator = listUsers.iterator();
109
110     while (iterator.hasNext()) {
111         //prendo un utente dalla lista
112         User user = iterator.next();
113
114         //notifico l'evento all'utente
115         user.notifyFriendOnline(name);
116     }
```

ListUsers.java

```
117     }
118
119
120    /**
121     * Metodo per notificare a tutti gli utenti della lista che
122     * l'utente passato da parametro è adesso offline.
123     * @param String name: nome dell'utente.
124     */
125    public synchronized void notifyAllOffline(String name) {
126        //creo un iteratore per scorrere la lista utenti
127        Iterator<User> iterator = listUsers.iterator();
128
129        while (iterator.hasNext()) {
130            //prendo un utente dalla lista
131            User user = iterator.next();
132
133            //notifico l'evento all'utente
134            user.notifyFriendOffline(name);;
135        }
136    }
137
138
139    /**
140     * Metodo per notificare a tutti gli utenti della lista che
141     * la chatroom passata da parametro è stata chiusa e rimuoverla
142     * dalla loro lista chatrooms.
143     * @param Chatroom chatroom: chatroom.
144     */
145    public synchronized void notifyAllClose(Chatroom chatroom) {
146        //prendo il nome e l'address della chatroom
147        String name = chatroom.getId();
148        InetAddress address = chatroom.getAddress();
149
150        //creo un iteratore per scorrere la lista utenti
151        Iterator<User> iterator = listUsers.iterator();
152
153        while (iterator.hasNext()) {
154            //prendo un utente dalla lista
155            User user = iterator.next();
156
157            //rimuovo la chatroom dalla sua lista
158            user.removeChatroom(chatroom);
159
160            //notifico l'evento all'utente
161            user.notifyCloseChatroom(name,address);
162        }
163    }
164 }
```

User.java

```
1 package dataStructures;
2
3 import java.io.DataInputStream;
4
5 /**
6  * Classe User.
7  * Contiene i metodi per creare ed eseguire operazioni
8  * sugli utenti.
9  * @author Emilio Panti mat:531844
10 */
11 public class User {
12
13     //nickname, password e lingue dell'utente
14     private String nickname;
15     private String psw;
16     private String language;
17
18     //status utente
19     private boolean online;
20
21     //lista amici dell'utente
22     private ListUsers friends;
23
24     //lista delle chatrooms a cui l'utente è iscritto
25     private ListChatrooms chatrooms;
26
27     //stub per notificare gli eventi al client
28     private NotifyEventInterface stub;
29
30     //socket e stream per inviare messaggi all'utente
31     private Socket socketMsg;
32     private DataOutputStream writerMsg;
33     private DataInputStream readerMsg;
34
35     //porta dove il client dell'user è in ascolto per ricevere file
36     private long portFile;
37
38
39     /**
40      * Costruttore classe User.
41      * @param String nickname: nickname scelto dall'utente
42      * @param String psw: password scelta dall'utente.
43      * @param String language: lingua dell'utente.
44      */
45     public User(String nickname, String psw, String language) {
46         this.nickname = nickname;
47         this.psw = psw;
48         this.language = language;
49         online = false;
50         friends = new ListUsers();
51         chatrooms = new ListChatrooms();
52         stub = null;
53         socketMsg = null;
54         writerMsg = null;
55         portFile = 0;
56     }
57 }
```

User.java

```
68
69 /**
70 * Override del metodo equals.
71 * @param Object obj: oggetto da confrontare.
72 * @return true: se i due utenti hanno lo stesso nickname,
73 *         false: altrimenti.
74 */
75 @Override
76 public boolean equals(Object obj) {
77     if (obj==null) return false;
78     User user = (User) obj;
79     return nickname.equals(user.getNickname());
80 }
81
82 /**
83 * Override del metodo hashCode.
84 * Ho effettuato l'override di hashCode solo per formalità (non verrà
85 * utilizzato) dato che ho fatto l'override di equals.
86 * @return int: restituisce il risultato del metodo hashCode chiamato
87 *             sul nickname dell'utente (anche il metodo equals
88 *             di cui ho fatto l'override sopra si basa sul confronto
89 *             del campo nickname)
90 */
91 @Override
92 public int hashCode() {
93     return nickname.hashCode();
94 }
95
96
97 /**
98 * Metodo che restituisce lo status dell'utente.
99 * @return boolean: true se l'utente è online
100 *                  false altrimenti.
101 */
102 public synchronized boolean getOnline() {
103     return online;
104 }
105
106 /**
107 * Metodo che restituisce la lingua dell'utente
108 * @return String: la lingua dell'utente
109 */
110 public String getLanguage() {
111     return language;
112 }
113
114
115 /**
116 * Metodo che restituisce il nickname dell'utente
117 * @return String: il nickname dell'utente
118 */
119 public String getNickname() {
120     return nickname;
121 }
122
123
124
```

User.java

```
125 /**
126 * Metodo che restituisce un oggetto json contenente l'inet address e
127 * la porta dove il client è in ascolto per eventuali file.
128 * @return JSONObject: contenente l'inet address e la port file,
129 *         null: se è offline.
130 */
131 @SuppressWarnings("unchecked")
132 public synchronized JSONObject getInfoFile() {
133     if (online) {
134         JSONObject infoFile = new JSONObject();
135         infoFile.put("ADDRESS", socketMsg.getInetAddress().getHostAddress().toString());
136         infoFile.put("PORT_FILE", portFile);
137         return infoFile;
138     }
139     else return null;
140 }
141
142 /**
143 * Metodo per controllare se la password passata da parametro corrisponde
144 * a quella salvata nella struttura dell'user.
145 * @param String password: password da controllare.
146 * @return true: se le due password corrispondono,
147 *         false: altrimenti.
148 */
149 public boolean checkPsw (String password) {
150     return (psw.equals(password));
151 }
152
153
154 /**
155 * Metodo per settare il socket e gli streams dedicati all'invio di messaggi all'user.
156 * Se lo stub per le notifiche è già stato settato cambio lo stato dell'user a online.
157 * @param Socket socketMsg: socket aperto per inviare i messaggi all'user.
158 * @param DataOutputStream writerMsg: stream per inviare i messaggi all'user.
159 */
160 public void setSocketMsg (Socket socketMsg, DataOutputStream writerMsg,
161                         DataInputStream readerMsg, long portFile) {
162
163     //variabile per sapere se è cambiato di stato l'utente
164     boolean check = false;
165
166     synchronized (this ) {
167         this.socketMsg = socketMsg;
168         this.writerMsg = writerMsg;
169         this.readerMsg = readerMsg;
170         this.portFile = portFile;
171         if (stub != null) {
172             online = true;
173             check = true;
174         }
175     }
176
177     //se l'utente è online
178     if(check) {
179         //notifico a tutti gli amici dell'utente che adesso è online
180         friends.notifyAllOnline(nickname);
```

User.java

```
182
183         //aumento di 1 il numero degli utenti connessi a tutte le chatroom
184         //a cui l'utente è iscritto
185         chatrooms.increaseAll();
186     }
187 }
188
189
190 /**
191 * Metodo per settare lo stub che permette di notificare i vari eventi all'user.
192 * Se il socket per i messaggi è già stato settato cambio lo stato
193 * dell'user a online.
194 * @param NotifyEventInterface stub: stub per le notifiche.
195 */
196 public void setStubNotify (NotifyEventInterface stub) {
197
198     //variabile per sapere se è cambiato di stato l'utente
199     boolean check = false;
200
201     synchronized (this ) {
202         this.stub = stub;
203         if (socketMsg!=null) {
204             online = true;
205             check = true;
206         }
207     }
208
209     //se l'utente è online
210     if(check) {
211         //notifico a tutti gli amici dell'utente che adesso è online
212         friends.notifyAllOnline(nickname);
213
214         //aumento di 1 il numero degli utenti connessi a tutte le chatroom
215         //a cui l'utente è iscritto
216         chatrooms.increaseAll();
217     }
218 }
219
220
221 /**
222 * Metodo per aggiungere un amico alla lista amici.
223 * @param User friend: user da inserire nella lista amici.
224 */
225 public void addFriend (User friend) {
226     //aggiungo friend alla lista amici
227     if (friends.add(friend)) {
228         //notifico l'utente dell'avvenimento
229         notifyNewFriend(friend.getNickname());
230     }
231 }
232
233
234 /**
235 * Metodo per stabilire una nuova amicizia.
236 * @param User friend: utente con cui vuole stringere una nuova amicizia
237 * @return true: se l'utente viene aggiunto alle amicizie,
238 *         false: se è già amico con esso.
```

User.java

```
239  /*
240  public boolean doFriendship (User friend) {
241      //se l'inserimento va a buon fine
242      if (friends.add(friend)) {
243          //notifico l'utente della nuova amicizia
244          notifyNewFriend(friend.getNickname());
245
246          //aggiorno anche la lista amici dell'amico aggiunto
247          friend.addFriend(this);
248
249          return true;
250      }
251      //se i due utenti erano già amici
252      else return false;
253  }
254
255
256 /**
257 * Metodo che restituisce una stringa contenente i nomi degli amici dell'utente.
258 * @return String: se ha almeno un amico,
259 *         null, altrimenti.
260 */
261 public String getFriendList () {
262     return friends.getList();
263 }
264
265 /**
266 * Metodo che restituisce l'amico cercato (se è nella lista amici).
267 * @param String nickname: nickname dell'utente da cercare.
268 * @return User: se viene trovato tale utente,
269 *         null: altrimenti.
270 */
271 public User getFriend(String nickname) {
272     return friends.get(nickname);
273 }
274
275
276 /**
277 * Metodo per aggiungere una chatroom alla lista chatrooms.
278 * @param Chatroom chatroom: chatroom da aggiungere.
279 * @return true: se la chatroom viene aggiunta alla lista.
280 *         false: se l'utente ha già tale chatroom nella propria lista.
281 */
282 public boolean addChatroom (Chatroom chatroom) {
283     //aggiungo la chatroom alla lista chatrooms
284     return chatrooms.add(chatroom);
285 }
286
287
288 /**
289 * Metodo per rimuovere una chatroom dalla lista chatrooms.
290 * @param Chatroom chatroom: chatroom da rimuovere.
291 */
292 public void removeChatroom (Chatroom chatroom) {
293     //rimuovo la chatroom alla lista chatrooms
294     chatrooms.remove(chatroom);
295 }
```

User.java

```
296     }
297
298
299     /**
300      * Metodo che confronta la lista di chatrooms passata da parametro con
301      * la lista chatrooms dell'utente.
302      * Restituisce la lista che ha preso come argomento specificando per ogni
303      * chatrooms al suo interno se l'utente vi è iscritto o meno.
304      * @return JSONArray: se la lista passata da parametro non è null,
305      *         null, altrimenti.
306      */
307     public JSONArray getChatroomsList (JSONArray allChatrooms) {
308         return chatrooms.listChatroomSigned(allChatrooms);
309     }
310
311
312     /**
313      * Metodo per inviare un messaggio all'utente se online.
314      * @param String msg: stringa da inviare sullo stream.
315      * @return true: se il messaggio viene inviato correttamente.
316      *             false: se l'utente non è online oppure se si verifica un errore.
317      */
318     public synchronized boolean sendMsg (String msg) {
319         //se l'utente è online
320         if (online) {
321             boolean check = true;
322             //mando il messaggio
323             try {
324                 writerMsg.writeUTF(msg);
325                 writerMsg.flush();
326             } catch (IOException e) {
327                 //se si verifica un errore
328                 e.printStackTrace();
329                 check = false;
330             }
331             return check;
332         }
333         else return false;
334     }
335
336
337     /**
338      * Metodo per notificare all'utente (se è online) che un suo amico è adesso online.
339      * @param String name: nome dell'amico che adesso è online.
340      */
341     public synchronized void notifyFriendOnline (String name) {
342         //se l'utente è online
343         if (online) {
344             //chiamo il metodo dello stub notifiche
345             try {
346                 stub.notifyFriendOnline(name);
347             } catch (RemoteException e) {
348                 e.printStackTrace();
349             }
350         }
351     }
352 }
```

User.java

```
353
354 /**
355 * Metodo per notificare all'utente (se è online) che un suo amico è adesso offline.
356 * @param String name: nome dell'amico che adesso è offline.
357 */
358 public synchronized void notifyFriendOffline (String name) {
359     //se l'utente è online
360     if (online) {
361         //chiamo il metodo dello stub notifiche
362         try {
363             stub.notifyFriendOffline(name);
364         } catch (RemoteException e) {
365             e.printStackTrace();
366         }
367     }
368 }
369
370 /**
371 * Metodo per notificare all'utente (se è online) che ha stretto una nuova amicizia.
372 * @param String name: nome del suo nuovo amico.
373 */
374
375 public synchronized void notifyNewFriend (String name) {
376     //se l'utente è online
377     if (online) {
378         //chiamo il metodo dello stub notifiche
379         try {
380             stub.notifyNewFriend(name);
381         } catch (RemoteException e) {
382             e.printStackTrace();
383         }
384     }
385 }
386
387 /**
388 * Metodo per notificare all'utente (se è online) che una chatroom a cui era
389 * iscritto è stata chiusa.
390 * @param String name: nome della chatroomm chiusa.
391 * @param InetAddress address: address della chatroom.
392 */
393
394 public synchronized void notifyCloseChatroom (String name, InetAddress address) {
395     //se l'utente è online
396     if (online) {
397         //chiamo il metodo dello stub notifiche
398         try {
399             stub.notifyCloseChatroom(name,address);
400         } catch (RemoteException e) {
401             e.printStackTrace();
402         }
403     }
404 }
405
406 /**
407 * Metodo per rendere l'utente offline.
408 */
409
```

User.java

```
410     public void setOffline () {
411         synchronized (this) {
412             online = false;
413             stub = null;
414
415             //chiudo writerMsg e socketMsg
416             try {
417                 writerMsg.close();
418                 readerMsg.close();
419                 socketMsg.close();
420             } catch (IOException e) {}
421
422             socketMsg = null;
423             writerMsg = null;
424             readerMsg = null;
425             portFile = 0;
426         }
427
428         //comunico agli amici che l'utente è offline adesso
429         friends.notifyAllOffline(nickname);
430
431         //decremento di 1 il numero degli utenti connessi a tutte le chatroom
432         //a cui l'utente è iscritto
433         chatrooms.decreaseAll();
434     }
435 }
436 }
```

ServerImpl.java

```
1 package notificationRMI;
2
3 import java.rmi.RemoteException;
4
5
6 /**
7  * Classe ServerImpl.
8  * Contiene le implementazioni dei metodi dichiarati nell'interfaccia
9  * ServerInterface.
10 * @author Emilio Panti mat:531844
11 */
12 public class ServerImpl extends RemoteObject implements ServerInterface{
13
14     private static final long serialVersionUID = 1L;
15
16     //hash map degli utenti
17     private ConcurrentHashMap <String,User> hashUsers;
18
19
20     /**
21      * Costruttore classe ServerImpl.
22      * @param ConcurrentHashMap hashUsers: hash map degli utenti.
23      */
24     public ServerImpl(ConcurrentHashMap<String,User> hashUsers)
25         throws RemoteException {
26         this.hashUsers = hashUsers;
27     }
28
29
30
31     /**
32      * Metodo che può essere richiamato dal client per registrarsi alla callback
33      * e poter ricevere le notifiche che lo riguardano.
34      * @param NotifyEventInterface clientInterface: stub del client utilizzato
35      *          dal server per richiamare i metodi che notificano l'eventi.
36      * @param String nickname: nickname dell'utente che ha richiesto la registrazione
37      *          alla callback
38      * @throws RemoteException
39      */
40     public void registerForCallback (NotifyEventInterface clientInterface,
41                                     String nickname) throws RemoteException {
42
43         //cerco l'utente nella hash table utenti
44         User user = hashUsers.get(nickname);
45
46         //salvo lo stub nella sua struttura dati
47         if (user!=null) user.setStubNotify(clientInterface);
48     }
49
50 }
51
52
53 }
54 }
```

HandlerMsgChatrooms.java

```
1 package threads;
2
3 import java.net.DatagramPacket;
11
12 /**
13 * Classe HandlerMsgChatrooms.
14 * Thread che apre un DatagramSocket e in cui successivamente
15 * si mette in ascolto di messaggi destinati alle chatrooms
16 * (inviai dagli utenti).
17 * Per ogni pacchetto ricevuto preleva l'inet address della
18 * chatroom di destinazione e spedisce il messaggio a tale
19 * indirizzo multicast.
20 * @author Emilio Panti mat:531844
21 */
22 public class HandlerMsgChatrooms implements Runnable {
23
24
25 /**
26 * Costruttore classe HandlerMsgChatrooms.
27 */
28 public HandlerMsgChatrooms() {
29
30 }
31
32
33 /**
34 * Task che riceve messaggi dai client per poi smistarli
35 * alle chatrooms.
36 */
37 public void run() {
38     //in che porta aprire il datagram socket
39     int portDs = ServerMain.PORT_UDP;
40
41     //porta dove sono in ascolto i clients
42     int sendPort = ServerMain.PORT_CLIENT;
43
44     //utilizzato per ricevere i messaggi UDP dai clients
45     //e rispedirli alle chatroom
46     DatagramSocket ds = null;
47
48     //apro il datagram socket
49     try {
50         ds = new DatagramSocket(portDs);
51     }
52     catch (Exception e) {
53         e.printStackTrace();
54         ds.close();
55         System.exit(0);
56     }
57
58     //variabile per gestire i messaggi che ricevo
59     String msgReiceved = null;
60     JSONObject msgJSON = null;
61
62     //variabili dove salvo l'indirizzo della chatrooom e il messaggio
63     //(in formato json) da inviare ad essa
64     String address = null;
```

HandlerMsgChatrooms.java

```
65     InetAddress ia = null;
66     JSONObject msg = null;
67
68     while(true){
69         try {
70             //buffer
71             byte [ ] buffer = new byte[8000];
72
73             //creo il datagram packet
74             DatagramPacket dpReceived = new DatagramPacket(buffer,buffer.length);
75
76             //aspetto un messaggio
77             ds.receive(dpReceived);
78             msgReiceved = new String(dpReceived.getData());
79
80             //prendo solo la parte della stringa che mi interessa
81             int lastIndex = msgReiceved.lastIndexOf('}');
82             String msgCorrect = msgReiceved.substring(0, lastIndex+1);
83
84             //trasformo in formato JSON il messaggio corretto
85             msgJSON = (JSONObject) new JSONParser().parse(msgCorrect);
86
87             //prendo il messaggio e l'indirizzo della chatroom destinataria
88             address = (String) msgJSON.get ("ADDRESS");
89             msg = (JSONObject) msgJSON.get ("MESSAGE");
90
91             //prendo l'inet address della chatroom
92             ia = InetAddress.getByName(address);
93
94             //creo ed invio il pacchetto a tutta la chatroom
95             byte [ ] data = (msg.toJSONString()).getBytes();
96             DatagramPacket dpToSend = new DatagramPacket(data,data.length,ia,sendPort);
97             ds.send(dpToSend);
98         }
99         catch (Exception e) {
100             //se occorre un'eccezione nel blocco sopra, non faccio niente, così da
101             //far continuare il lavoro al thread
102             e.printStackTrace();
103         }
104     }
105 }
106 }
107 }
```

ThreadPool.java

```
1 package threads;
2
3 import java.net.Socket;
11
12
13 /**
14 * Classe ThreadPool.
15 * Crea un thread pool di N threads (dove N è il
16 * numero massimo di connessioni accettate dal server).
17 * Ogni thread del pool (oggetti della classe Worker)
18 * gestisce una connessione verso un client finchè non
19 * viene chiusa.
20 * @author Emilio Panti mat:531844
21 */
22 public class ThreadPool {
23
24     //thread pool esecutore di task
25     private ThreadPoolExecutor executor;
26
27     //hash map per gli utenti
28     private ConcurrentHashMap <String,User> hashUsers;
29
30     //hash map per le chatrooms
31     private HashChatrooms hashChatrooms;
32
33
34     /**
35      * Costruttore classe ThreadPoolContatore.
36      * @param ConcurrentHashMap<String,User> hashUsers: hash map degli utenti.
37      * @param HashChatrooms hashChatrooms: hash map delle chatrooms.
38      */
39     public ThreadPool(ConcurrentHashMap<String,User> hashUsers,
40                     HashChatrooms hashChatrooms){
41         this.hashUsers = hashUsers;
42         this.hashChatrooms = hashChatrooms;
43         executor=(ThreadPoolExecutor)Executors.newFixedThreadPool(ServerMain.MAX_CONN);
44     }
45
46
47     /**
48      * Metodo che dà in gestione ad un thread del pool una nuova connessione.
49      * @param Socket socket: socket della nuova connessione.
50      */
51     public void newConnection(Socket socket){
52         try{
53             executor.execute((Runnable)new Worker(socket,hashUsers,hashChatrooms));
54         }
55         catch (Exception e) {
56             e.printStackTrace();
57         }
58     }
59 }
60
```

Worker.java

Worker.java

```
77     writer = new DataOutputStream(new BufferedOutputStream(socket.getOutputStream()));
78     reader = new DataInputStream(new BufferedInputStream(socket.getInputStream()));
79 } catch (IOException e) {
80     // TODO Auto-generated catch block
81     e.printStackTrace();
82     close = true;
83 }
84
85
86 //gestisco le richieste del client
87 while(!close) {
88     try {
89         //ricevo la richiesta
90         String request = reader.readUTF();
91
92         //trasformo in formato JSON la richiesta ricevuta dal server
93         JSONObject requestJSON = (JSONObject) new JSONParser().parse(request);
94
95         //conterrà la risposta da inviare al client
96         JSONObject response = null;
97
98         //prendo l'operazione richiesta
99         Operations requestOp = Operations.valueOf((String) requestJSON.get("OP"));
100
101        //in base all'operazione richiesta
102        switch(requestOp) {
103
104            case REG:
105                //chiamo la funzione che esegue la registrazione
106                response = registration(requestJSON);
107                break;
108
109            case LOG:
110                //chiamo la funzione che esegue il login
111                response = login(requestJSON);
112                break;
113
114            case CONN_MSG:
115                //chiamo la funzione che apre la seconda connessione TCP verso il
116                //client
117                response = secondConnection(requestJSON);
118                //il thread che esegue questa operazione dopo termina
119                close = true;
120                secondConn = true;
121                break;
122
123            case LISTFRIEND:
124                //chiamo la funzione che resituisce la lista amici dell'utente al
125                //client
126                response = friendsList(requestJSON);
127                break;
128
129            case FRIENDSHIP:
130                //chiamo la funzione che esegue la richiesta di amicizia
131                response = friendship(requestJSON);
132                break;
133 }
```

Worker.java

```
132
133     case LOOKUP:
134         //chiamo la funzione che esegue l'operazione di look up
135         response = lookUp(requestJSON);
136         break;
137
138     case STARTCHAT:
139         //chiamo la funzione che gestisce la richiesta di apertura di una chat
140         response = startChat(requestJSON);
141         break;
142
143     case MSG_FRIEND:
144         //chiamo la funzione che manda un messaggio ad un amico
145         response = msgFriend(requestJSON);
146         break;
147
148     case FILE_FRIEND:
149         //chiamo la funzione che manda un file ad un amico
150         response = fileFriend(requestJSON);
151         break;
152
153     case CHATLIST:
154         //chiamo la funzione che restituisce la lista delle chatrooms al client
155         response = chatroomsList(requestJSON);
156         break;
157
158     case CREATE_CHATROOM:
159         //chiamo la funzione che crea una nuova chatroom
160         response = createChatroom(requestJSON);
161         break;
162
163     case ADDME_CHATROOM:
164         //chiamo la funzione che aggiunge l'utente ad una chatroom
165         response = addToChatroom(requestJSON);
166         break;
167
168     case CLOSE_CHATROOM:
169         //chiamo la funzione che chiude una chatroom
170         response = closeChatroom(requestJSON);
171         break;
172
173     case MSG_CHATROOM:
174         //chiamo la funzione che dà l'ok per inviare un messaggio ad una
175         //chatroom
176         response = msgChatroom(requestJSON);
177         break;
178
179     case CLOSE:
180         //setto close a true
181         close = true;
182         //setto response a null così non viene inviata nessuna risposta
183         response = null;
184         break;
185
186     default:
187         close = true;
188         response = null;
189         break;
```

Worker.java

```
189         }
190
191         //spedisco la risposta al client
192         if(response != null) {
193             writer.writeUTF(response.toJSONString());
194             writer.flush();
195         }
196
197     } catch (Exception e) {
198         //se viene rilevato qualche problema di connessione con il client
199         close = true;
200     }
201 }
202
203 //se il thread non ha gestito l'operazione di apertura della seconda
204 //connessione verso il client chiudo la connessione verso il client
205 if (!secondConn) closeConnection();
206 }
207
208 /**
209 * Metodo che esegue l'operazione di registrazione.
210 * @param: JSONObject request: richiesta fatta dal client.
211 * @return: JSONObject: contiene la risposta (con esito positivo o
212 *                     negativo) da inviare al client.
213 */
214
215 @SuppressWarnings("unchecked")
216 private JSONObject registration(JSONObject request) {
217     //response da restituire
218     JSONObject response = new JSONObject();
219
220     //prendo i vari campi di interesse dalla richiesta del client
221     String nickname = (String) request.get("ID");
222     String psw = (String) request.get("PSW");
223     String language = (String) request.get("LANGUAGE");
224
225     //creo un nuovo utente
226     User us = new User(nickname, psw, language);
227
228     //se non esiste già un utente registrato con tale nickname
229     if(hashUsers.putIfAbsent(nickname, us)==null) {
230         //salvo la struttura dell'utente per le successive richieste
231         user = us;
232
233         //prendo la lista di tutte le chatrooms
234         JSONArray allChatrooms = hashChatrooms.getListChatrooms();
235
236         response.put("OP", "OP_OK");
237         response.put("CHATROOMS", allChatrooms);
238     }
239     //se esiste già un utente registrato con tale nickname
240     else {
241         response.put("OP", "OP_ERR");
242         response.put("MSG", "ERR: This nickname is already used");
243     }
244
245 return response;
```

Worker.java

```
246     }
247
248
249     /**
250      * Metodo che esegue l'operazione di login.
251      * @param: JSONObject request: richiesta fatta dal client.
252      * @return: JSONObject: contiene la risposta (con esito positivo o
253      *                      negativo) da inviare al client.
254      */
255     @SuppressWarnings("unchecked")
256     private JSONObject login(JSONObject request) {
257         //response da restituire
258         JSONObject response = new JSONObject();
259
260         //prendo i vari campi di interesse dalla richiesta del client
261         String nickname = (String) request.get("ID");
262         String psw = (String) request.get("PSW");
263
264         //cerco un utente con tale nickname
265         User us = hashUsers.get(nickname);
266
267         //se esiste un utente con tale nickname
268         if(us != null) {
269             //controllo che la password sia corretta
270             if(us.checkPsw(psw)) {
271                 //controllo se l'utente è già loggato su un altro client
272                 if(us.getOnline()) {
273                     response.put("OP", "OP_ERR");
274                     response.put("MSG", "ERR: This user is already logged in");
275                 }
276                 //se non è già online
277                 else {
278                     //salvo la struttura dell'utente per le successive richieste
279                     user = us;
280
281                     //prendo la lista amici dell'utente
282                     String listFriends = us.getFriendList();
283
284                     //prendo la lista chatrooms dell'utente
285                     JSONArray allChatrooms = hashChatrooms.getListChatrooms();
286                     JSONArray listChatrooms = us.getChatroomsList(allChatrooms);
287
288                     //inserisco lista amici e lista chatrooms nella risposta
289                     response.put("OP", "OP_OK");
290                     response.put("FRIENDS",listFriends);
291                     response.put("CHATROOMS",listChatrooms);
292                 }
293             }
294             //se la psw non coincide
295             else {
296                 response.put("OP", "OP_ERR");
297                 response.put("MSG", "ERR: The password is incorrect");
298             }
299         }
300         //se non esiste un utente con tale nickname
301         else {
302             response.put("OP", "OP_ERR");
303         }
304     }
```

Worker.java

```
303         response.put("MSG", "ERR: There is no user with this nickname");
304     }
305
306     return response;
307 }
308
309
310 /**
311 * Metodo che esegue l'operazione di apertura della seconda connessione
312 * TCP verso il client per permettere agli altri utenti di inviargli messaggi.
313 * @param: JSONObject request: richiesta fatta dal client.
314 * @return: JSONObject: contiene la risposta (con esito positivo o
315 *                     negativo) da inviare al client.
316 */
317 @SuppressWarnings("unchecked")
318 private JSONObject secondConnection(JSONObject request) {
319     //response da restituire
320     JSONObject response = new JSONObject();
321
322     //prendo i vari campi di interesse dalla richiesta del client
323     String nickname = (String) request.get("ID");
324     long portFile = (long) request.get("PORT_FILE");
325
326     //cerco un utente con tale nickname
327     User us = hashUsers.get(nickname);
328
329     //se esiste un utente con tale nickname
330     if(us!=null && writer!=null && reader!=null) {
331         //salvo nella struttura dell'utente il socket e lo stream
332         //di output per mandare i messaggi all'utente e la porta
333         //dove il client è in ascolto per eventuali file
334         us.setSocketMsg(socket, writer, reader, portFile);
335         response.put("OP", "OP_OK");
336     }
337     //se non esiste un utente con tale nickname
338     else {
339         response.put("OP", "OP_ERR");
340         response.put("MSG", "ERR: There is no user with this nickname");
341     }
342
343     return response;
344 }
345
346
347 /**
348 * Metodo che restituisce la lista amici dell'utente.
349 * @param: JSONObject request: richiesta fatta dal client.
350 * @return: JSONObject: contiene la risposta (con esito positivo o
351 *                     negativo) da inviare al client.
352 */
353 @SuppressWarnings("unchecked")
354 private JSONObject friendsList(JSONObject request) {
355     //response da restituire
356     JSONObject response = new JSONObject();
357
358     //se l'utente è loggato
359     if(user != null) {
```

Worker.java

```
360         //prendo la lista amici dell'utente
361         String listFriends = user.getFriendList();
362
363         //inserisco lista amici nella risposta
364         response.put("OP", "OP_OK");
365         response.put("FRIENDS",listFriends);
366     }
367     //se non è loggato (cosa che non dovrebbe verificarsi)
368     else {
369         response.put("OP", "OP_ERR");
370         response.put("MSG", "ERR: An error occurred");
371     }
372
373     return response;
374 }
375
376
377 /**
378 * Metodo che esegue l'operazione di richiesta amicizia.
379 * @param: JSONObject request: richiesta fatta dal client.
380 * @return: JSONObject: contiene la risposta (con esito positivo o
381 *                     negativo) da inviare al client.
382 */
383 @SuppressWarnings("unchecked")
384 private JSONObject friendship(JSONObject request) {
385     //response da restituire
386     JSONObject response = new JSONObject();
387
388     //prendo il nickname dell'utente con cui vuole diventare amico
389     String nickname = (String) request.get("ID");
390
391     //cerco un utente con tale nickname
392     User friend = hashUsers.get(nickname);
393
394     //eventuale messaggio di errore
395     String msgErr = null;
396
397     //se l'utente è loggato
398     if(user != null) {
399         //se esiste l'utente con cui vuole stringere amicizia
400         if(friend != null) {
401             //se i due non erano già amici
402             if(user.doFriendship(friend)) {
403                 response.put("OP", "OP_OK");
404             }
405             //se erano già amici
406             else {
407                 response.put("OP", "OP_ERR");
408                 msgErr = "ERR: You and \'"+nickname+"\' are already friends";
409                 response.put("MSG", msgErr);
410             }
411         }
412         //se non esiste
413         else {
414             response.put("OP", "OP_ERR");
415             msgErr="ERR: \'"+nickname+"\' does not exists";
416             response.put("MSG", msgErr);
417         }
418     }
419 }
```

Worker.java

```
417         }
418     }
419     //se non è loggato (cosa che non dovrebbe verificarsi)
420     else {
421         response.put("OP", "OP_ERR");
422         response.put("MSG", "ERR: An error occurred");
423     }
424
425     return response;
426 }
427
428 /**
429 * Metodo che esegue l'operazione di look up.
430 * @param: JSONObject request: richiesta fatta dal client.
431 * @return: JSONObject: contiene la risposta (con esito positivo o
432 *                     negativo) da inviare al client.
433 */
434 @SuppressWarnings("unchecked")
435 private JSONObject lookUp(JSONObject request) {
436     //response da restituire
437     JSONObject response = new JSONObject();
438
439     //prendo il nickname dell'utente che vuole cercare il client
440     String nickname = (String) request.get("ID");
441
442     //cerco un utente con tale nickname
443     User us = hashUsers.get(nickname);
444
445     //messaggio che il client comunicherà all'utente
446     String msg = null;
447
448     //se l'utente è loggato
449     if(user != null) {
450         //se esiste l'utente cercato
451         if(us != null) {
452             response.put("OP", "OP_OK");
453             msg = "\"" + nickname + "\" exists";
454             response.put("MSG", msg);
455         }
456         //se non esiste
457         else {
458             response.put("OP", "OP_OK");
459             msg = "\"" + nickname + "\" does not exists";
460             response.put("MSG", msg);
461         }
462     }
463
464     //se non è loggato (cosa che non dovrebbe verificarsi)
465     else {
466         response.put("OP", "OP_ERR");
467         response.put("MSG", "ERR: An error occurred");
468     }
469
470     return response;
471 }
472
473 }
```

Worker.java

```
474 /**
475 * Metodo che esegue l'operazione di start chat.
476 * @param: JSONObject request: richiesta fatta dal client.
477 * @return: JSONObject: contiene la risposta (con esito positivo o
478 * negativo) da inviare al client.
479 */
480 @SuppressWarnings("unchecked")
481 private JSONObject startChat(JSONObject request) {
482     //response da restituire
483     JSONObject response = new JSONObject();
484
485     //prendo il nickname dell'utente con cui vuole aprire una chat
486     String nickname = (String) request.get("ID");
487
488     //eventuale messaggio di errore
489     String msgErr = null;
490
491     //se l'utente è loggato
492     if(user != null) {
493         //cerco il nickname nella lista amici dell'utente
494         User us = user.getFriend(nickname);
495
496         //se l'utente con cui vuole aprire la chat è suo amico
497         if(us != null) {
498             //se l'utente con cui vuole aprire la chat è online
499             if(us.getOnline()) {
500                 response.put("OP", "OP_OK");
501             }
502             //se non è online
503             else {
504                 response.put("OP", "OP_ERR");
505                 msgErr = "ERR: \\""+nickname+"\\ is not online";
506                 response.put("MSG", msgErr);
507             }
508         }
509         //se non sono amici
510         else {
511             response.put("OP", "OP_ERR");
512             msgErr="ERR: You and \\""+nickname+"\\ are not friends";
513             response.put("MSG", msgErr);
514         }
515     }
516     //se non è loggato (cosa che non dovrebbe verificarsi)
517     else {
518         response.put("OP", "OP_ERR");
519         response.put("MSG", "ERR: An error occurred");
520     }
521
522     return response;
523 }
524
525
526 /**
527 * Metodo che esegue l'operazione di invio messaggio.
528 * NB: non viene controllato che il nickname a cui l'utente vuole
529 *      inviare il messaggio sia suo amico perchè questo viene
530 *      controllato nella operazione di start chat.
```

Worker.java

```
531     * @param: JSONObject request: richiesta fatta dal client.  
532     * @return: JSONObject: contiene la risposta (con esito positivo o  
533                 negativo) da inviare al client.  
534     */  
535     @SuppressWarnings("unchecked")  
536     private JSONObject msgFriend(JSONObject request) {  
537         //response da restituire  
538         JSONObject response = new JSONObject();  
539  
540         //prendo i campi di interesse della richiesta fatta dal client  
541         String receiverNickname = (String) request.get("TO");  
542         String msg = (String) request.get("MSG");  
543  
544         //prendo l'utente destinatario del messaggio  
545         User receiver = hashUsers.get(receiverNickname);  
546  
547         //eventuale messaggio di errore  
548         String msgErr = null;  
549  
550         //se l'utente è loggato e il destinatario esiste  
551         if(user != null && receiver != null) {  
552             //oggetto json da inviare al receiver  
553             JSONObject msgToSendObj = new JSONObject();  
554  
555             //prendo le lingue dei due utenti  
556             String l1 = user.getLanguage();  
557             String l2 = receiver.getLanguage();  
558  
559             String msgToSend = null;  
560  
561             //se le due lingue solo uguali  
562             if(l1.equals(l2)) {  
563                 //non effettuo nessuna traduzione  
564                 msgToSend = "[" + user.getNickname() + "]: " + msg;  
565             }  
566             else {  
567                 //tento di tradurre il messaggio  
568                 msgToSend = translate(msg, l1, l2);  
569  
570                 //se la traduzione è andata a buon fine  
571                 if(msgToSend!=null) {  
572                     msgToSend = "[" + user.getNickname() + "]: " + msgToSend;  
573                 }  
574                 //altrimenti mando il messaggio originale senza traduzione  
575                 else {  
576                     msgToSend = "[" + user.getNickname() + "]: " + msg;  
577                 }  
578             }  
579  
580             msgToSendObj.put("OP", "MSG_FRIEND");  
581             msgToSendObj.put("FROM", user.getNickname());  
582             msgToSendObj.put("MSG", msgToSend);  
583  
584             //se il messaggio viene spedito correttamente  
585             if(receiver.sendMsg(msgToSendObj.toJSONString())) {  
586                 response.put("OP", "OP_OK");  
587             }  
588         }
```

Worker.java

```
588         //se il messaggio non viene inviato
589         else {
590             response.put("OP", "OP_ERR");
591             msgErr="ERR: '"+receiverNickname+"\\ is offline now";
592             response.put("MSG", msgErr);
593         }
594     }
595     //se non è loggato o se il receiver non esiste (cosa che non dovrebbe verificarsi)
596     else {
597         response.put("OP", "OP_ERR");
598         response.put("MSG", "ERR: An error occurred");
599     }
600
601     return response;
602 }
603
604
605 /**
606 * Metodo che traduce un messaggio da una lingua ad un'altra utilizzando
607 * il servizio rest api del sito mymemory.
608 * @param String msg: messaggio da tradurre.
609 * @param String l1: lingua originale del messaggio.
610 * @param String l2: lingua in cui deve essere tradotto il messaggio.
611 * @return String: il messaggio tradotto,
612 *         null: se non è possibile tradurlo o se si è verificato un errore.
613 */
614 private String translate(String msg, String l1, String l2) {
615     //url base per fare la richiesta di traduzione a mymemory
616     String pathName = "https://api.mymemory.translated.net/get?";
617
618     //messaggio da tradurre
619     String message = msg.replace(" ", "%20");
620
621     //come inserire il messaggio nella richiesta
622     String msgToTranslate = "q=" + message;
623
624     //come inserire le lingue nella richiesta
625     String languages = "&langpair=" + l1 + "|" + l2;
626
627     try {
628         //creo l'url per la richiesta e apro la connessione
629         URL url = new URL(pathName + msgToTranslate + languages);
630         URLConnection uc = url.openConnection();
631         uc.connect();
632
633         //prendo la risposta
634         BufferedReader in = new BufferedReader(new
635             InputStreamReader(uc.getInputStream()));
636         String line=null;
637         StringBuffer sb=new StringBuffer();
638         while((line=in.readLine())!=null){
639             sb.append(line);
640         }
641
642         //trasformo in formato json la risposta e prendo il campo di interesse
643         JSONObject response = (JSONObject) new JSONParser().parse(sb.toString());
644         JSONObject responseData = (JSONObject) response.get("responseData");
```

Worker.java

```
644         String msgTranslated = (String) responseData.get("translatedText");
645
646         return msgTranslated;
647
648     } catch (Exception e) {
649         //se il sito non è reperibile o se non è traducibile il messaggio
650         return null;
651     }
652 }
653
654
655 /**
656 * Metodo che esegue l'operazione di richiesta di invio file.
657 * NB: il worker restituisce al client solo il numero di porta e l'address
658 * dell'amico a cui l'utente vuole trasferire il file.
659 * @param: JSONObject request: richiesta fatta dal client.
660 * @return: JSONObject: contiene la risposta (con esito positivo o
661 *                     negativo) da inviare al client.
662 */
663 @SuppressWarnings("unchecked")
664 private JSONObject fileFriend(JSONObject request) {
665     //response da restituire
666     JSONObject response = new JSONObject();
667
668     //prendo i campi di interesse della richiesta fatta dal client
669     String receiverNickname = (String) request.get("ID");
670
671     //prendo l'utente destinatario del messaggio
672     User receiver = hashUsers.get(receiverNickname);
673
674     //eventuale messaggio di errore
675     String msgErr = null;
676
677     //se l'utente è loggato e il destinatario esiste
678     if(user != null && receiver != null) {
679         //prendo le info del receiver da inviare al client
680         JSONObject infoFile = receiver.getInfoFile();
681
682         //se il receiver è online
683         if(infoFile!=null) {
684             response = infoFile;
685             response.put("OP", "OP_OK");
686         }
687         //se è offline
688         else {
689             response.put("OP", "OP_ERR");
690             msgErr="ERR: '"+receiverNickname+"' is offline now";
691             response.put("MSG", msgErr);
692         }
693     }
694     //se non è loggato o se il receiver non esiste (cosa che non dovrebbe verificarsi)
695     else {
696         response.put("OP", "OP_ERR");
697         response.put("MSG", "ERR: An error occurred");
698     }
699
700     return response;
```

Worker.java

```
701     }
702
703
704     /**
705      * Metodo che restituisce la lista delle chatrooms all'utente.
706      * @param: JSONObject request: richiesta fatta dal client.
707      * @return: JSONObject: contiene la risposta (con esito positivo o
708      *                      negativo) da inviare al client.
709      */
710     @SuppressWarnings("unchecked")
711     private JSONObject chatroomsList(JSONObject request) {
712         //response da restituire
713         JSONObject response = new JSONObject();
714
715         //se l'utente è loggato
716         if(user != null) {
717             //prendo la lista chatrooms dell'utente
718             JSONArray allChatrooms = hashChatrooms.getListChatrooms();
719             JSONArray listChatrooms = user.getChatroomsList(allChatrooms);
720
721             //inserisco lista amici e lista chatrooms nella risposta
722             response.put("OP", "OP_OK");
723             response.put("CHATROOMS",listChatrooms);
724         }
725         //se non è loggato (cosa che non dovrebbe verificarsi)
726         else {
727             response.put("OP", "OP_ERR");
728             response.put("MSG", "ERR: An error occurred");
729         }
730
731         return response;
732     }
733
734
735     /**
736      * Metodo che crea una nuova chatroom.
737      * @param: JSONObject request: richiesta fatta dal client.
738      * @return: JSONObject: contiene la risposta (con esito positivo o
739      *                      negativo) da inviare al client.
740      */
741     @SuppressWarnings("unchecked")
742     private JSONObject createChatroom(JSONObject request) {
743         //response da restituire
744         JSONObject response = new JSONObject();
745
746         //prendo i vari campi di interesse dalla richiesta del client
747         String id = (String) request.get("ID");
748
749         //se l'utente è loggato
750         if(user != null) {
751             //creo una nuova chatroom
752             Chatroom chatroom = new Chatroom(id,user);
753
754             //tento di inserire la chatroom nella hash chatrooms
755             InetAddress address = hashChatrooms.add(chatroom);
756
757             //se l'inserimento è andato bene
```

Worker.java

```
758     if(address!=null) {
759         //aggiorno la lista chatroom dell'utente
760         user.addChatroom(chatroom);
761
762         //prendo l'host address in formato String
763         String hostAddress = address.getHostAddress().toString();
764
765         response.put("OP", "OP_OK");
766         response.put("ADDRESS", hostAddress);
767     }
768     //se esiste già una chatroom con tale id
769     else {
770         response.put("OP", "OP_ERR");
771         String msgErr = "ERR: There is already a chatroom called '"+id+
772                         "' or the maximum number of open chatrooms has been reached";
773         response.put("MSG", msgErr);
774     }
775 }
776 //se non è loggato (cosa che non dovrebbe verificarsi)
777 else {
778     response.put("OP", "OP_ERR");
779     response.put("MSG", "ERR: An error occurred");
780 }
781
782 return response;
783 }
784
785
786 /**
787 * Metodo che aggiunge l'utente ad una chatroom.
788 * @param: JSONObject request: richiesta fatta dal client.
789 * @return: JSONObject: contiene la risposta (con esito positivo o
790 *                     negativo) da inviare al client.
791 */
792 @SuppressWarnings("unchecked")
793 private JSONObject addToChatroom(JSONObject request) {
794     //response da restituire
795     JSONObject response = new JSONObject();
796
797     //prendo i vari campi di interesse dalla richiesta del client
798     String id = (String) request.get("ID");
799
800     //se l'utente è loggato
801     if(user != null) {
802         //cerco la chatroom
803         Chatroom chatroom = hashChatrooms.get(id);
804
805         //se la chatroom esiste
806         if(chatroom!=null) {
807             //se l'utente non era già iscritto alla chatroom
808             if(user.addChatroom(chatroom)) {
809                 //aggiungo l'utente alla lista iscritti della chatroom
810                 chatroom.addSubscriber(user);
811
812                 //prendo l'address della chatroom
813                 InetAddress address = chatroom.getAddress();
814             }
815         }
816     }
817 }
```

Worker.java

```
815         //prendo l'host address in formato String
816         String hostAddress = address.getHostAddress().toString();
817
818         response.put("OP", "OP_OK");
819         response.put("ADDRESS", hostAddress);
820     }
821     //se era già iscritto a tale chatroom
822     else {
823         response.put("OP", "OP_ERR");
824         String msgErr = "ERR: You are already registered to the chatroom
825             \"+id+\"";
826         response.put("MSG", msgErr);
827     }
828     //se non esiste nessuna chatroom con tale id
829     else {
830         response.put("OP", "OP_ERR");
831         String msgErr = "ERR: There is no chatroom called \"+id+\"";
832         response.put("MSG", msgErr);
833     }
834 }
835 //se non è loggato (cosa che non dovrebbe verificarsi)
836 else {
837     response.put("OP", "OP_ERR");
838     response.put("MSG", "ERR: An error occurred");
839 }
840
841     return response;
842 }
843
844
845 /**
846 * Metodo che chiude una chatroom.
847 * @param: JSONObject request: richiesta fatta dal client.
848 * @return: JSONObject: contiene la risposta (con esito positivo o
849 *                     negativo) da inviare al client.
850 */
851 @SuppressWarnings("unchecked")
852 private JSONObject closeChatroom(JSONObject request) {
853     //response da restituire
854     JSONObject response = new JSONObject();
855
856     //prendo i vari campi di interesse dalla richiesta del client
857     String id = (String) request.get("ID");
858
859     //se l'utente è loggato
860     if(user != null) {
861         //provo a rimuovere la chatroom
862         if(hashChatrooms.remove(id, user.getNickname())) {
863             response.put("OP", "OP_OK");
864         }
865         //se la chatroom non esiste o l'utente non è il creatore
866         else {
867             response.put("OP", "OP_ERR");
868             String msgErr = "ERR: The chatroom \"+id+
869                 "\" does not exist or you are not the creator";
870             response.put("MSG", msgErr);
```

Worker.java

```
871         }
872     }
873     //se non è loggato (cosa che non dovrebbe verificarsi)
874     else {
875         response.put("OP", "OP_ERR");
876         response.put("MSG", "ERR: An error occurred");
877     }
878
879     return response;
880 }
881
882 /**
883 * Metodo che controlla che ci siano almeno due utenti online
884 * in ascolto su una chatroom.
885 * @param: JSONObject request: richiesta fatta dal client.
886 * @return: JSONObject: contiene la risposta (con esito positivo o
887 * negativo) da inviare al client.
888 */
889 @SuppressWarnings("unchecked")
890 private JSONObject msgChatroom(JSONObject request) {
891     //response da restituire
892     JSONObject response = new JSONObject();
893
894     //prendo i vari campi di interesse dalla richiesta del client
895     String id = (String) request.get("ID");
896
897     //se l'utente è loggato
898     if(user != null) {
899         //cerco la chatroom
900         Chatroom chatroom = hashChatrooms.get(id);
901
902         //se nella chatroom ci sono almeno due utenti online
903         if(chatroom.checkUsersConn()) {
904             response.put("OP", "OP_OK");
905         }
906         //se non ci sono
907         else {
908             response.put("OP", "OP_ERR");
909             String msgErr = "ERR: You are the only one online";
910             response.put("MSG", msgErr);
911         }
912     }
913     //se non è loggato (cosa che non dovrebbe verificarsi)
914     else {
915         response.put("OP", "OP_ERR");
916         response.put("MSG", "ERR: An error occurred");
917     }
918 }
919 return response;
920 }
921
922 /**
923 * Metodo che esegue la chiusura della connessione verso il client.
924 */
925 private void closeConnection() {
926     //setto offline l'utente (se si era loggato)
```

Worker.java

```
928     if(user!=null) user.setOffline();  
929  
930     //chiudo il socket e gli streams della connessione  
931     try {  
932         if(writer!=null) writer.close();  
933         if(reader!=null) reader.close();  
934         if(socket!=null) socket.close();  
935     } catch (Exception e) {}  
936 }  
937 }  
938 }
```