

---

# SIMPLE SOCIAL

*di Giovanni Graziano Modica*



## 0. PANORAMICA

1. Introduzione
  2. Server
    - a. Avvio
    - b. Threads
    - c. Logica
    - d. Altre funzionalità
  3. Client
    - a. Avvio
    - b. Threads
    - c. Lista Comandi
    - d. Altre funzionalità
  4. Formato dei Messaggi
  5. Struttura Completa
    - a. Server
    - b. Client
  6. Conclusioni
- 

## 1. INTRODUZIONE

**Simple Social** è un software implementato come progetto di laboratorio del corso di “Reti di Calcolatori e Internet” A.A. 2015/2016. Il software consiste in un semplice Social Network il quale permette di aggiungere amici, follower e dare la possibilità a quest’ultimi l’invio e la ricezione di contenuti.

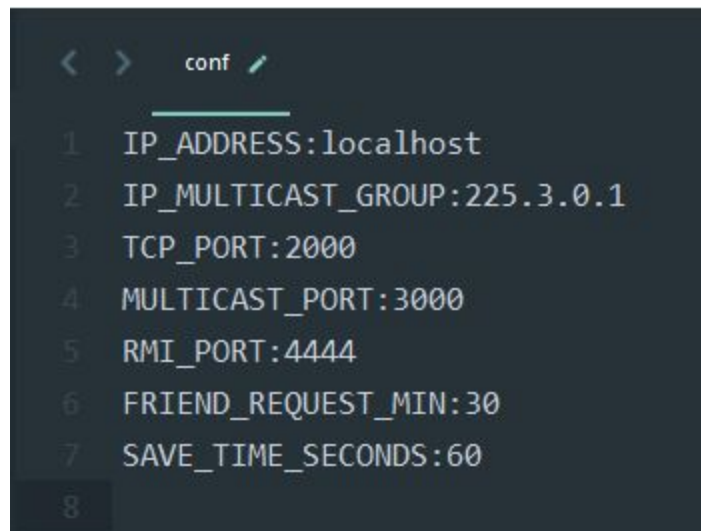
Il sistema SimpleSocial è composto da due componenti: **SocialServer** e **SocialClient**. SocialClient è il client necessario per accedere al social network, si utilizza attraverso utilizzo di riga di comando.

Il presente Software è stato implementato usando Java 8 sull’IDE “Eclipse Mars 2.0”. La macchina, su cui è stato implementato e testato, ha un processore Intel Core i5 di terza generazione con sistema operativo Windows 10 a 64bit.

## 2. SERVER

### A. Avvio

Prima dell'avvio di **SocialServer** è possibile modificare alcune impostazioni cambiando i valori nel file `./conf`. Al suo avvio il server controlla se esiste tale file nella sua cartella di esecuzione, se questa è presente allora il Server carica le impostazioni presenti nel file altrimenti si avvia con valori di default.

A screenshot of a terminal window with a dark background. At the top, there are navigation arrows and the text 'conf' with a small green checkmark icon. Below this, a list of configuration parameters is shown, each preceded by a line number from 1 to 8. The parameters are: IP\_ADDRESS:localhost, IP\_MULTICAST\_GROUP:225.3.0.1, TCP\_PORT:2000, MULTICAST\_PORT:3000, RMI\_PORT:4444, FRIEND\_REQUEST\_MIN:30, and SAVE\_TIME\_SECONDS:60. The line number 8 is at the bottom of the list, followed by a blank line.

```
< > conf ✓  
1 IP_ADDRESS:localhost  
2 IP_MULTICAST_GROUP:225.3.0.1  
3 TCP_PORT:2000  
4 MULTICAST_PORT:3000  
5 RMI_PORT:4444  
6 FRIEND_REQUEST_MIN:30  
7 SAVE_TIME_SECONDS:60  
8
```

Qui sotto la descrizione di ciascuna voce.

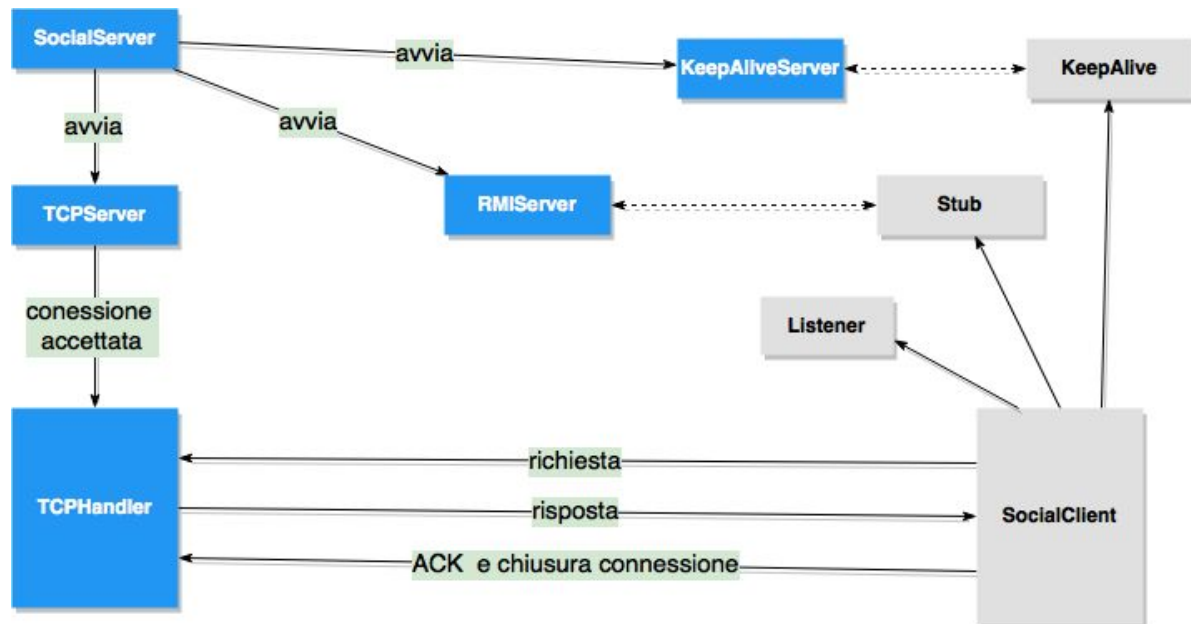
- **IP\_ADDRESS**: indirizzo IP di SocialServer
- **IP\_MULTICAST\_GROUP**: indirizzo IP del gruppo multicast
- **TCP\_PORT**: numero porta di SocialServer
- **MULTICAST\_PORT**: porta di Multicast
- **RMI\_PORT**: porta del Registry RMI
- **FRIEND\_REQUEST\_MIN**: Time out delle richieste d'amicizia (espresso in minuti)
- **SAVE\_TIME\_SECONDS**: Intervallo di tempo espresso in secondi per il salvataggio di stato del server.

Dopo aver provveduto alla sua configurazione per avviare il server basta lanciare sul terminale il comando:

```
java -jar ./SocialServer.jar
```

Se il server è già stato avviato in precedenza, carica lo stato della sua precedente esecuzione salvato in un file `./users`; se il file non fosse presente allora SocialServer provvederà a ricreare uno nuovo stato ove non sono presenti informazioni riguardanti i precedenti utenti.

## B. Threads



Se il login è avvenuto con successo SocialServer avvia tre Tasks, in questo caso sono state scritte tre classi Runnable, i quali oggetti vengono lanciati su un ThreadPool di tre Threads.

- **TCPServer**: rimane in ascolto su una Socket TCP e per ogni richiesta accettata avvia un **TCPHandler**.
- **ServerRMI**: gestisce e avvia l'RMI Registry.
- **KeepAliveServer**: server multicast che gestisce il servizio di Keep Alive.

### 1. TCPServer & TCPHandler

**TCPServer** contiene una **ServerSocket** in ascolto, una volta accettata la richiesta dal Client avvierà un **TCPHandler** che risponderà tutte le richieste TCP del Client.

Il server è di tipo **OneShot** per cui le richieste avvengono con connessioni TCP non persistenti; seguono il protocollo descritto sotto:

- 1) Apertura connessione TCP.
- 2) Dopo l'apertura il server avvia **TCPHandler**.
- 3) il client invia una richiesta al server.
- 4) **TCPHandler** riceve la richiesta, la rielabora e server risponde.
- 5) **TCPHandler** rimane in attesa di un **OK** dal client, infine chiude la connessione.

## 2. KeepAlive

Questo thread si occupa di controllare lo stato degli utenti e di controllarne le relative scadenze. Ad ogni iterazione **KeepAlive** invia un messaggio **Multicast** al gruppo, controlla gli utenti provvisti di **UserID** (quindi online) e controlla chi tra questi hanno risposto al messaggio, infine effettua il logout di tutti gli utenti che non hanno risposto.

Inoltre nella stessa iterazione, **KeepAlive** controlla la validità degli **UserID** degli utenti e le validità delle richieste d'amicizia pendenti.

Se un utente è stato loggato per più di **24 ore** **KeepAlive** mette in logout l'utente, se le richieste d'amicizia pendenti sono in lista per più di un certo quantitativo di tempo (definito nel file **conf**) vengono rimosse dalla lista dell'utente.

## 3. Server RMI

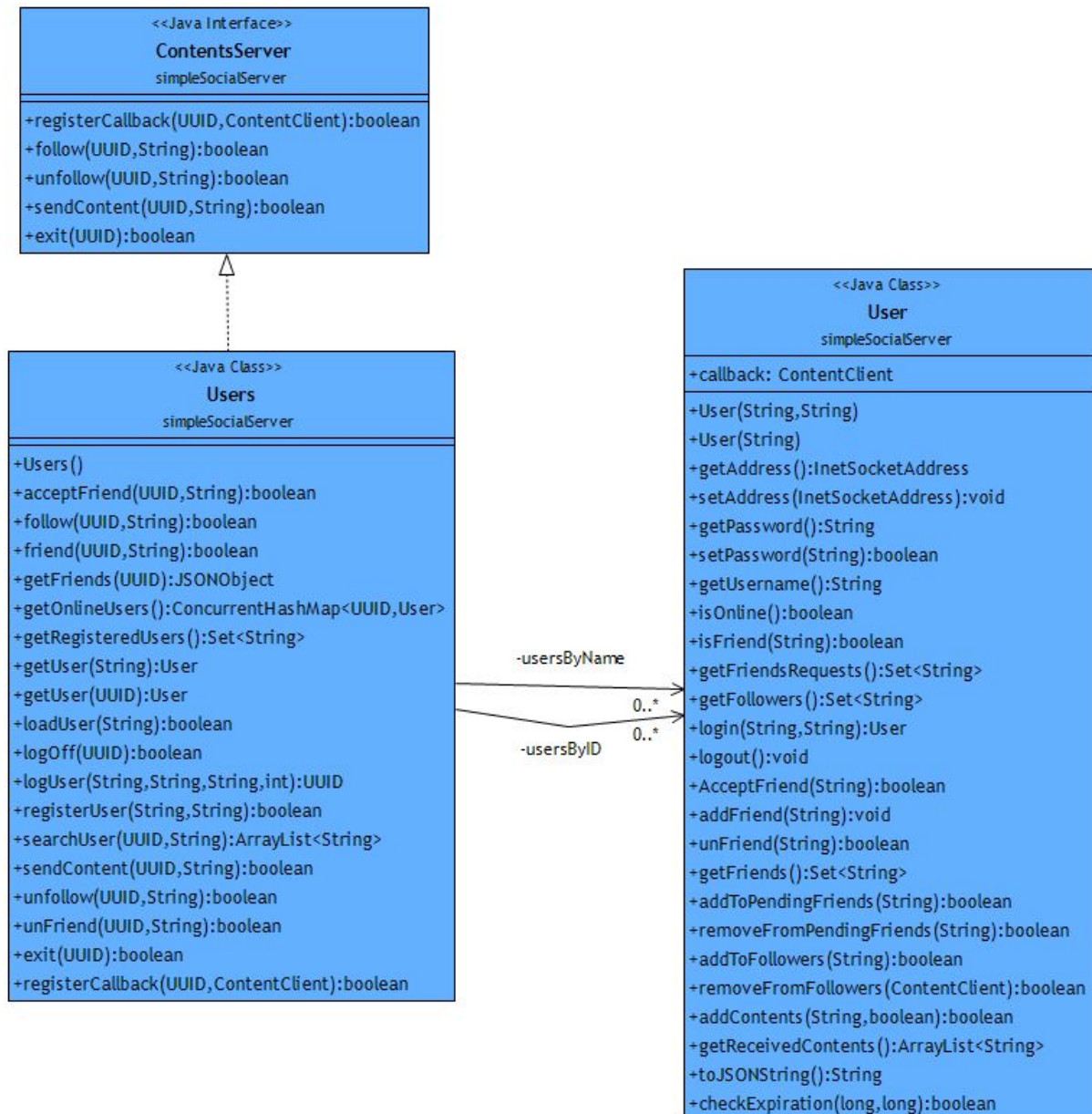
Avvia l'RMI Registry mettendo a disposizione l'interfaccia remota necessaria per l'invio dei contenuti e della registrazione dei followers.

## C. Logica

Nella figura sottostante è possibile vedere il diagramma delle classi che compongono la logica del Server. Tutte le operazioni vengono gestite in modo sincrono dal tipo **Users** il quale al suo interno sono presenti due **ConcurrentHashMap**:

- **UsersByName**: contiene **User** registrati, permette l'accesso a tutti gli utenti registrati ma non necessariamente online.
- **UserByID**: contiene **User** a cui è stato assegnato uno **UserID** quindi a tutti gli utenti online.

Usando due **ConcurrentHashMap** è più facile individuare quale utente è online e quale offline, inoltre permette al Thread di **KeepAlive** di scandire una struttura più ridotta ed individuare più velocemente quali utenti vanno mantenuti online e quali no.



## 1. User & Users:

Nella struttura User si ha accesso a tutte le funzionalità che coinvolgono il singolo utente registrato in “Social Server” al suo interno ci sono tre HashSet tutti ThreadSafe:

- Set delle richieste di amicizia non ancora accettate.
- Set degli amici.
- Set degli utenti che hanno espresso interesse ai contenuti dello User (followers).

I metodi di User sono utilizzati dalla classe **Users**. La classe **Users** implementa anche i metodi dell'interfaccia **ContentServer**: interfaccia remota utilizzata dall'**RMI Registry**.

## 2. ContentServer:

ContentServer è l'interfaccia remota utilizzata da RMI, è composta da cinque metodi:

- **registerCallback**: per registrare l'interfaccia remota del client
- **follow** e **unfollow**: permettono di aggiungere o rimuovere un follower
- **sendContent**: per inviare un contenuto al SocialClient

## D. Altre funzionalità

Il server salverà il proprio stato in nel file `./users` in formato **JSON** ogni 60 secondi o in base al numero settato prima dell'avvio del server.

Il login di un utente dura non più di 24 ore, alla scadenza lo UserID verrà revocato e l'utente dovrà effettuare nuovamente la procedura di login.

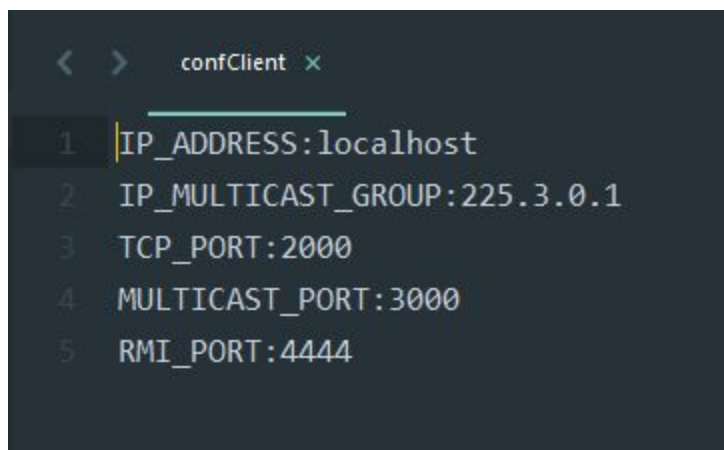
Le richieste d'amicizia non durano per più di un tempo definito nel file `./conf`.

---

## 3. CLIENT

### A. Avvio

Come nel caso di SocialServer anche **SocialClient** è provvisto di un file di configurazione chiamato `./confClient` ed ha una formattazione simile a quella di SocialServer ma con qualche voce in meno. Il significato delle voci non varia.



```
< > confClient x
1 IP_ADDRESS:localhost
2 IP_MULTICAST_GROUP:225.3.0.1
3 TCP_PORT:2000
4 MULTICAST_PORT:3000
5 RMI_PORT:4444
```

E' possibile lanciare **SocialClient** usando uno tra i parametri: **register** o **login**.  
per effettuare la registrazione dell'utente digitare:

```
java -jar ./SocialClient.jar register [username] [password]
```

Una volta digitato SocialClient risponderà con un messaggio di successo e social client terminerà.

Dopo aver provveduto alla registrazione sarà possibile effettuare login digitando:

```
java -jar ./SocialClient.jar login [username] [password]
```

Ricevuto il messaggio di benvenuto il client sarà collegato al sistema e potrà utilizzare tutte le funzionalità del Social Network.

Se il login è stato effettuato con successo:

1. Il client riceve uno **UserID** dal server.
2. il client sincronizza i contenuti salvati localmente con quelli ricevuti dal server durante lo stato offline.
3. avvio dei Task: **KeepAlive**, **ClientRMI** e **Listener**.
4. inserimento dei **comandi** per interagire con il sistema.
5. ogni volta che l'utente invia dei comandi, viene inviato soltanto l' UserID dell'utente invece che effettuare sistematicamente il login.

## B. Lista comandi

Qui sotto una lista di comandi necessari per interagire con **SocialClient**:

**help**: per ricevere un messaggio di aiuto.

**find** [nome utente]: per cercare un utente nella rete.

**friend** [nome utente]: per inviare una richiesta di amicizia.

**unfriend** [nome utente]: per rifiutare una richiesta di amicizia o eliminare un'amicizia già esistente.

**friends**: per visualizzare i propri amici ed il loro stato online/offline.

**getreq**: per visualizzare tutte le richieste di amicizia ricevute.

**accept** [nome utente]: per accettare un utente tra le richieste ricevute.

**follow** [nome amico]: per esprimere interesse nei contenuti dell'amico.

**send**: per inviare un contenuto scrivere **send** seguito da **invio**, successivamente scrivere il messaggio che si desidera inviare, infine premere di nuovo invio per inviare il contenuto.

**contents**: per visualizzare i contenuti ricevuti dagli utenti seguiti.

**logout**: per effettuare il logout dal sistema.



## C. Tasks

Come visto nell'immagine del paragrafo 2.B, il client avvia tre threads: **Listener**, **KeepAlive** e **ClientRMI**.

### Listener:

Il task **Listener** consiste in una `ServerSocket` che rimane in ascolto su un port generato in modo random, l'IP ed il port vengono poi comunicati a **SocialServer** che li utilizzerà per inviare eventuali notifiche.

### Keep Alive:

Keep Alive consiste in una `MulticastSocket` in ascolto del Server: attende i messaggi del **KeepAliveServer** e risponde ogni 10 secondi attraverso una `UDP Socket`.

Una volta ricevuto il messaggio Multicast dal Server, KeepAlive invia un `Datagram UDP` contenente l'ID dello User online in modo tale che il Server possa riconoscerlo e mantenere lo stato dell'utente online.

### ClientRMI:

Estende `RemoteObject`, è la callback che usa il server per inviare i contenuti ai follower di un dato utente.

## D. Altre funzionalità

Quando l'utente effettua il login, invia una richiesta di sincronizzazione con i contenuti remoti, in questa operazione vengono sincronizzate le richieste d'amicizia pendenti, la lista di amici, i followers, i contenuti **non** vengono scaricati seguendo questo procedimento, per la sincronizzazione viene utilizzato **JSON**.

Quando un utente riceve dei **contenuti** dai follower questi vengono salvati localmente in un file e rimossi dal Server.

Se l'utente è offline ma riceve altri contenuti dai follower questi vengono mantenuti nel `SocialServer` finché l'utente non effettua nuovamente login, poi vengono rimossi.

## 4. FORMATO DEI MESSAGGI

Per l'invio/ricezione dei messaggi si utilizza la classe **SocialMessage** che contiene due campi: il codice di operazione ed un array di parametri.

Per avere un modo intuitivo per utilizzare il codice di operazione è presente anche una classe **Operations**, ogni valore intero implica un tipo di operazione da effettuare.

Alcune tra le più importanti sono:

- OK: inviare conferma
- REG\_USER: richiesta di registrazione dell'utente
- LOG\_USER: richiesta di login dell'utente e dello UserID
- FRIEND\_REQ: richiesta d'amicizia
- SYNC\_USER: sincronizzazione con il profilo remoto dell'utente.

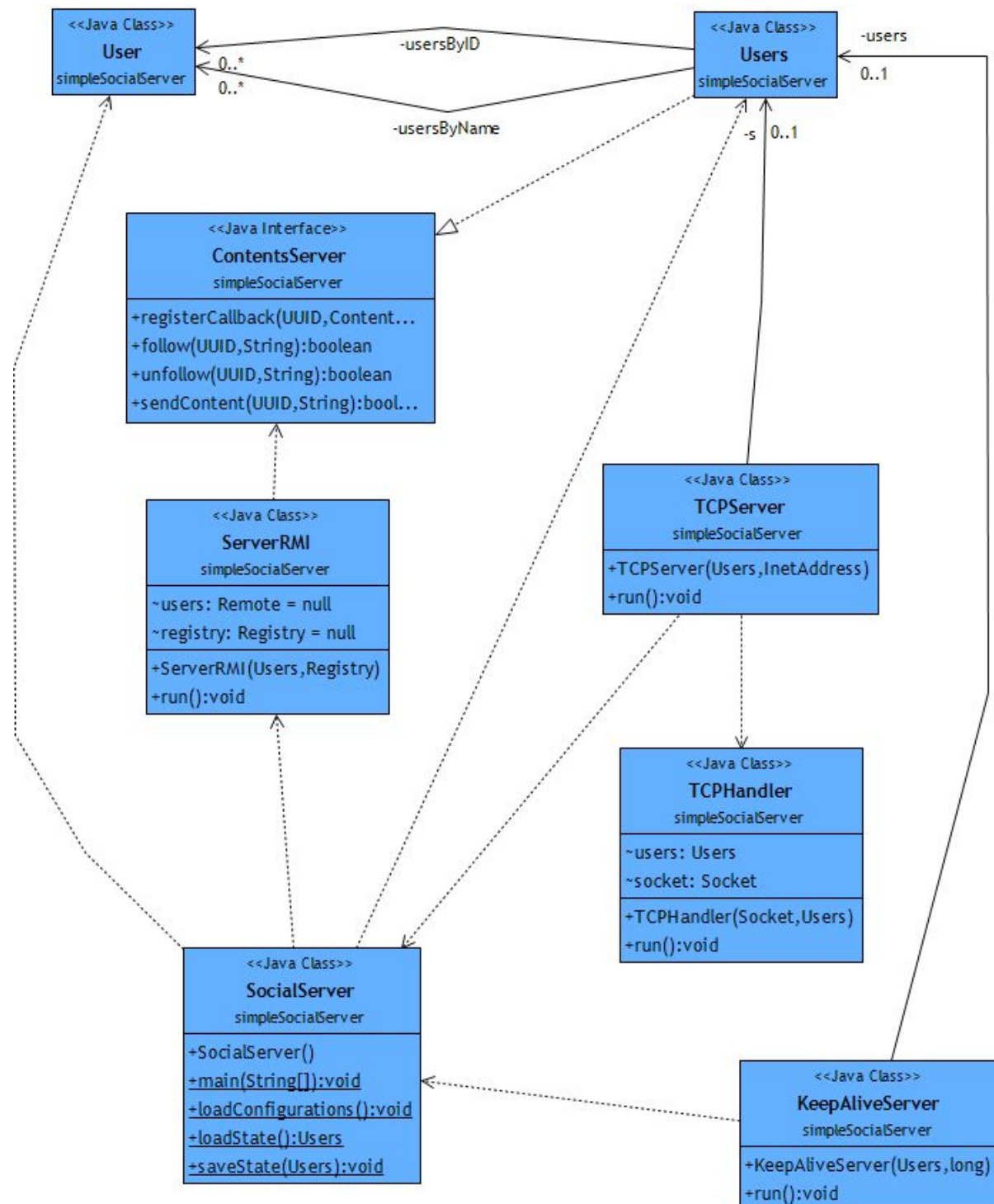
<<Java Class>> SocialMessage simpleSocialShared	
-ops: long	
-parameters: String[]	
+SocialMessage() +parseFriends(String):HashMap<String,Boolean> +parseMessage(String):SocialMessage +createMessage(int,String[]):SocialMessage +getOpCode():int +getParameters():String[] +toString():String +toJSONString():String	

<<Java Class>> Operations simpleSocialShared	
+OK: int	
+BAD_REQUEST: int	
+REG_USER: int	
+LOGIN_USER: int	
+LOGOUT_USER: int	
+SYNC_USER: int	
+SEARCH: int	
+FRIEND_REQ: int	
+FRIEND_ACCPT: int	
+FRIEND_SEARCH: int	
+FRIENDS_LIST: int	
+GET_FRIENDS_REQS: int	
+UNFRIEND: int	
+FRIEND_DENY: int	
+GENERIC_ERROR: int	
+USER_NOT_FOUND_ERROR: int	
+CREDENTIALS_ERROR: int	
+ACCESS_DENIED: int	
+CONNECTION_ERROR: int	
+ALREADY_EXISTS: int	
+Operations()	

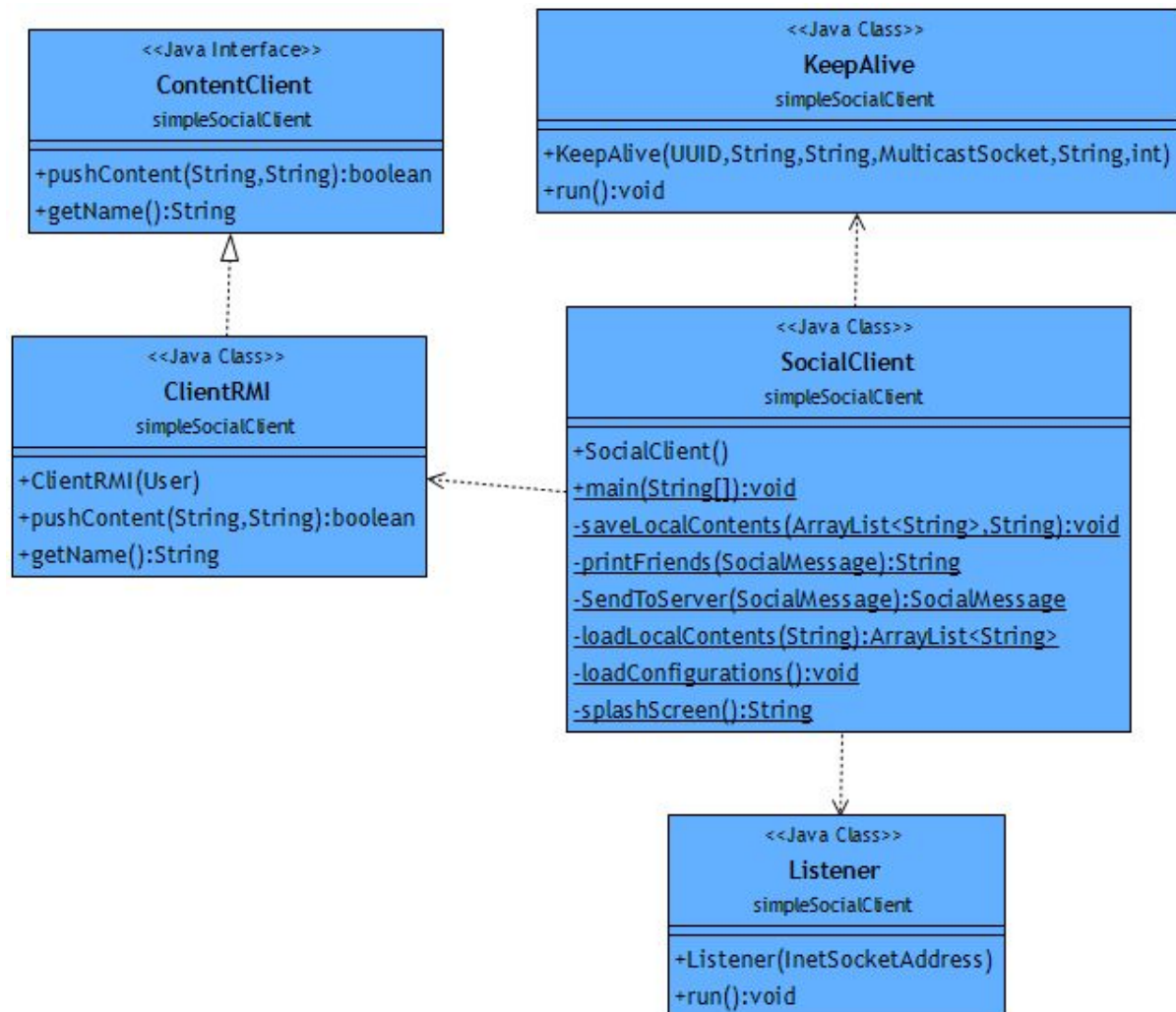
## 5. STRUTTURA COMPLETA

Diagrammi delle classi del Server e del Client

### A. SERVER



## B. CLIENT



## 6. CONCLUSIONI

Ho ritenuto molto valido l'utilizzo di **JSON** sia per lo scambio di messaggi ma anche per il salvataggio degli utenti del server le file locale **users**; per i file di configurazione ho preferito utilizzare invece una notazione più semplice **IMPOSTAZIONE:VALORE** poichè solo in questo caso ho ritenuto eccessivo l'utilizzo di JSON.

Differentemente dalla specifica le richieste di amicizia pendenti non vengono scaricate localmente, rimangono invece sul server per un certo periodo di tempo fino alla scadenza e relativa rimozione.

In questo modo è possibile loggare da più macchine diverse senza perdere le richieste di amicizia pendenti, si perderanno invece i contenuti ricevuti e non letti.