Simple Social

Appunti sulla documentazione del progetto di Reti dei Calcolatori(Laboratorio) in Java.

Bachmann Gherhard 493593
Gennaio 2017

1.Introduzione

Il progetto sviluppato e' una rete sociale caratterizzata da un insieme di funzionalita.

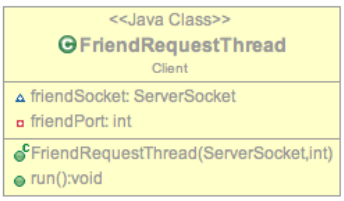
Le funzionalita sono: - Registrazione utente
- Log in utente
- Richiesta nuova amicizia
- Conferma nuova amicizia
- Keep Alive
- Richiesta lista amici
- Ricerca utenti
- Pubblicazione contenuti
- Sottoscrizione ad un utente
- Visualizzazione contenuti
- Log out.

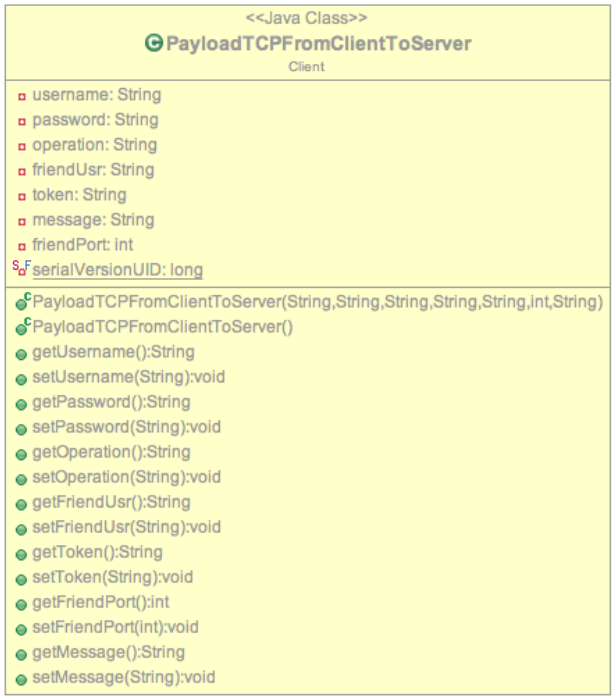
Le due componenti fondamentali del progetto sono SocialClient e SocialServer.
Il SocialClient gestisce l'interazione con l'utente tramite un interfaccia grafica sviluppata in JavaSwing.
Il meccanismo della rete sociale e' semplice, il cliente chiede di eseguire una o piu delle funzionalita elencate precedentemente ed il server le esegue rimandando indietro la risposta al cliente.

Il progetto e' stato consegnato in una cartella contenente:
- Codici sorgente per Client e Server
- File di backup per il Server
- Diagrammi delle classi
- Eseguibili per Client e Server.
- File di costanti (Vengono importate nelle classi interessate)

2.Diagramma delle classi

Diagramma client





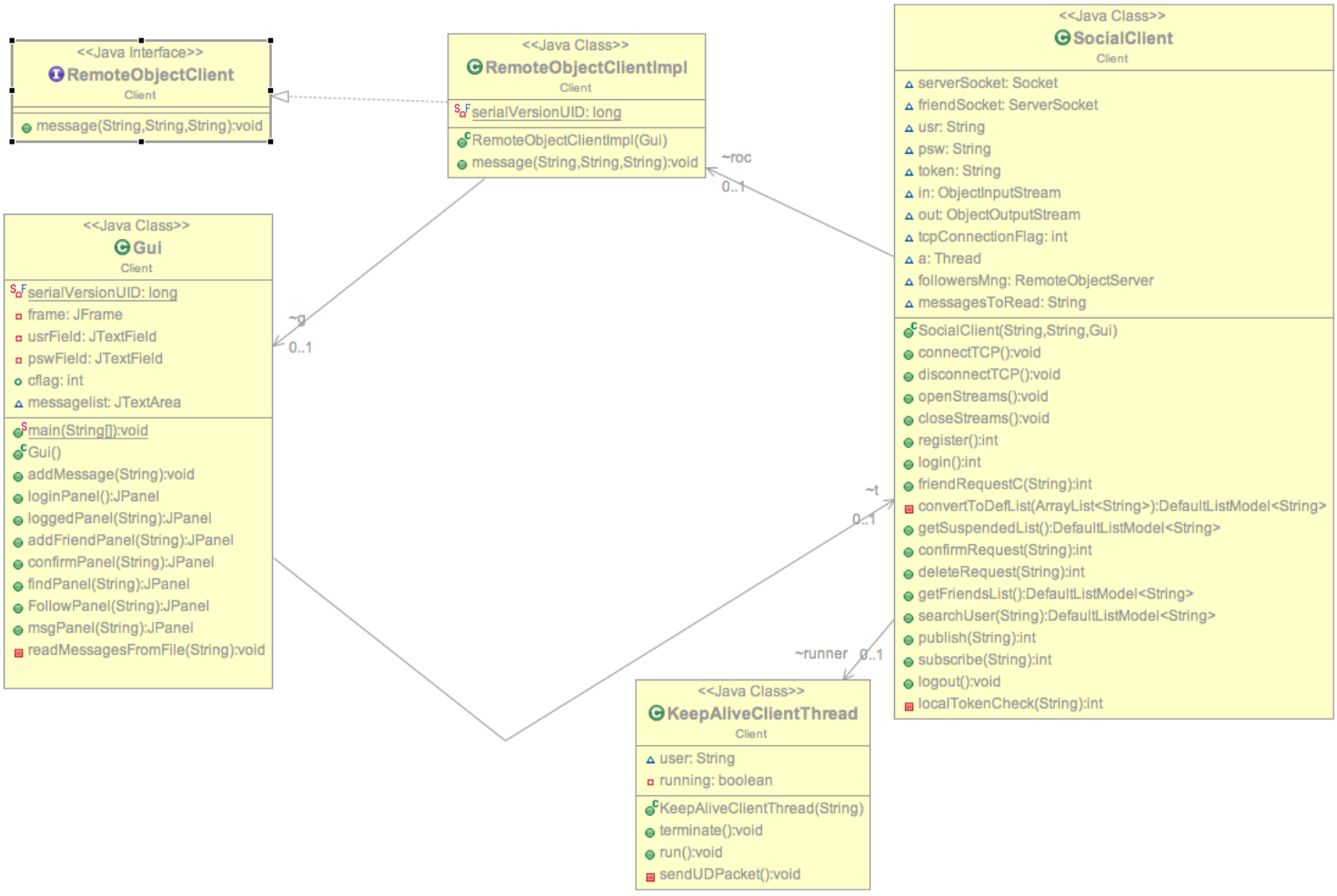


Diagramma Server

3.Descrizione dettagliata del funzionamento di SimpleSocial

————————————————Server————————————————

ServerMain e' la classe che fa iniziare il Server. L'inizializzazione del Server, l'avvio del thread di KeepAlive del server, l'inizializzazione del RMI(*) e in fine l'avvio della connessione TCP.

SocialServer e' la classe fondamentale del Server.
Abbiamo le 2 collezioni:
static ConcurrentHashMap<String,User> registeredUsers
static ConcurrentHashMap<String,User> onlineUsers
registeredUsers contiene tutti gli utenti registrati sul Server, mentre onlineUsers contiene tutti gli utenti the hanno eseguito il log in. Le due collezioni sono thread safe.

```
public void initializeServer()
In questo metodo vengono caricati gli utenti registrati dal file
di backup. Se il file e' vuoto viene stampato una stringa con
scritto "backUp file empty".

public void keepAliveThread()
Semplicemente esegue il thread di KeepAlive del Server.

public void startRMI()
Avvio del servizio RMI

public void startTCP()
Avvia la connessione TCP e si mette in attesa di clienti, appena
un cliente arriva viene creato ed eseguito un nuovo thread
(ServerWorker).
```

ServerWorker si occupa di eseguire tutte le operazioni richieste dal client e di restituire risultati attraverso la connessione TCP.

All'inizio nel suo metodo run, apre i due stream di lettura e scrittura dopodiche legge il messaggio che conterra l'operazione da eseguire.

`private void checkMessage(PayloadTCPFromClientToServer msg)`
Estrae il messaggio dal pacchetto ricevuto ed in uno switch controlla l'operazione che deve svolgere e richiama il metodo corrispondente.

`private void register(String password, String username)`
Registrazione dell'utente, si controlla subito se l'utente e' gia registrato e in tal caso viene mandato un pacchetto d'errore al Client. Altrimenti viene mandato un pacchetto con dentro un messaggio di conferma. In quest'ultimo caso bisogna anche aggiungere il nuovo utente registrato nella collezione registeredUsers. Come ultima operazione in questo metodo viene salvata la collezione nel file di backUp, visto che e' stato aggiunto un nuovo utente.

`private void login(String password, String username,int friendPort)`
Viene subito controllato il nome utente per verificare che e' un utente registrato e che la sua password e' giusta. Verifica anche che l'utente non sia gia online.
Se la verifica e' andata bene viene creato un token per l'utente con il metodo createToken. Il token e' semplicemente una stringa contenente il currentTime--username. Viene aggiunto il token all'utente nella collezione del server. Dopodiché viene creato il pacchetto con dentro il token e un messaggio di conferma di log in. Nel caso in cui la verifica del username non e' andata bene allora viene mandato il pacchetto di errore ed il log in non verra effettuato. Un fatto importante e' che ne in login ne in register non vengono controllati username e password se sono null perché l'interfaccia grafica non permette di premere il pulsante di login o register se i due campi non sono stati riempiti.

`private int checkToken(String token)`
Controlla che il token non sia scaduto e mette il feedback a true per il keepAliveThread di quell'utente.

`private void friendRequest(String password, String username, String friendUsr, String token)`
Controlla il token e se l'esito e' negativo manda un pacchetto di errore al Client altrimenti passa alla fase di connessione TCP con

il possibile nuovo amico. Ovviamente prima di aprire una connessione TCP sulla friendPort dell'utente che vuole avere come amico deve fare dei controlli. Quindi verifica che l'utente che vuole aggiungere agli amici e' online, che non cerchi di avere se stesso come amico, che non siano gia amici o che non ha gia inviato una richiesta d'amicizia a quel utente. Se questo controllo va male viene mandato un pacchetto d'errore al client. Se il controllo e' andato bene viene aperta una connessione TCP con l'utente amico e verifica che la connessione sia aperta. Se la connessione e' aperta significa che l'utente che vuole aggiungere e' online e quindi invia un pacchetto di conferma richiesta al client. Inoltre aggiunge la richiesta in una lista di richieste inviate (SentRequests) e in una lista di richieste sospese (SuspendedRequests). Infine sulla nuova TCP aperta invia il suo username all'utente che vuole come amico, dopodiché la connessione viene chiusa. Se la connessione non e' aperta significa che l'utente desiderato come amico e' offline e quindi viene inviato un pacchetto di errore al client.

`private void sendSuspendedList(String username,String token)`
Controlla il token e come sempre se negativo invia un pacchetto d'errore altrimenti crea un pacchetto e inserisce la lista delle richieste sospese del utente.

`private void confirmRequests(String usr,String token)`
Controlla token. Dopodiché il server aggiunge l'amico nella lista amici dell'utente e cancella le richieste sospese ed inviate dalle opportune liste. Se il token e' valido crea un pacchetto e lo invia al client con una conferma della aggiunta di un amico(richiesta confermata). Altrimenti come sempre manda un pacchetto d'errore

`private void deleteRequest(String usr,String token)`
Controlla token, se non e' valido invia il pacchetto d'errore al client. Se il token e' valido crea un nuovo pacchetto mette un messaggio di conferma e cancella la richiesta dalle liste dei sospesi e inviate. Infine invia il pacchetto di conferma al client.

`private void friendsList(String usr,String token)`
Solito controllo token. Crea un pacchetto da inviare al client dentro al quale ci inserisce pure la lista degli amici.Oltre al username degli amici viene aggiunto pure lo status come stringa separato da due dash. Infine invia il pacchetto.

```
private void searchList(String u,String token)
```
Solito controllo token. Crea un pacchetto da inviare al client dentro al quale ci inserisce pure la lista delle persone che ha trovato con il nome cercato.

```
private void publishMsg(String usrname,String token,String pmsg)
```
Controlla il token e in caso di token non valido invia il solito pacchetto d'errore. Altrimenti inizia a scorrere la lista del followers dell'utente e verifica chi e' online e chi no. Se un utente e' online usa l'RMI per inviargli il messaggio. Ma se un utente e' offline il messaggio viene salvato in un file con il nome di tale utente (MESSAGES_nomeutente). Alla fine viene inviato un messaggio di conferma indietro al client che serve soltanto all'interfaccia grafica.

```
private void logout(String username,String token)
```
Cancella l'utente dagli utenti online, mette il suo token a null e il suo feedback a false. Inoltre fa anche il salvataggio sul backUpFile.

User e' la classe dell'utente. Tutto cio di cui un utente ha bisogno. Le variabili sono elencate dentro la classe ed ognuna ha il suo set e get.

RemoteObjectServer e RemoteObjectServerImpl
Sono le classi per il servizio RMI e hanno 3 metodi:

```
public int addCallback(String myUsr ,RemoteObjectClient c)
```
Controlla se l'utente esiste come utente registrato e se esiste aggiunge la callback all'utente usando la setMyRMI dell'utente. Altrimenti invia un -1 che verra gestito come errore.

```
public int follow(String myUsr ,String usr)
```
Controlla se sono amici per poter registrare interesse. Se si lo aggiunge come follower nella collezione del server, altrimenti ritorna -1.

```
public int removeCallback(String usr)
```
Controlla il nome utente e rimuove la callBack.

PayloadTCPFromServerToClient e ErrorPayload
Sono i due possibili tipi di pacchetti che vengono mandati dal Server al client, tutte le loro variabili (stringe e liste ed interi) sono dentro le classi e non vengono elencate qui.

KeepAliveServerThread
Come prima cosa apre una connessione multicast su una porta nota (caricata da Constants). Per la connessione UDP su cui verrano mandati i messaggi di keep alive viene impostato un timeout (10sec). La prima volta il server invia sulla multicast e ovviamente non ricevera risposte non essendoci nessuno online. Dopodiche non inviera piu messaggi finche non avra ricevuto qualcosa sulla UDP. Questo funziona grazie ad un flag(timetosend) che controlla quando il server deve inviare il messaggio di keepalive. Inoltre viene fatto un controllo anche sul timeout e quindi il flag va positivo soltanto quando ha ricevuto e il timeout e' 0 (cioe sono passati i 10 secondi necessari). Il timeout viene calcolato con delle operazioni di sottrazioni basate sul tempo d'inizio e il vecchio timeout.
Alla fine di ogni iterazione viene chiamato un metodo checkOnline che restituisce il numero di persone che hanno risposto al messaggio keep alive e per tutti quelli viene messo a true il campo feedback.

SocialClient

La classe che parla con il server via TCP e con l'interfaccia grafica. Per ogni operazione il client invia al server un pacchetto PayloadTCPFromClientToServer che oltre agli attributi fondamentali (token) contiene anche l'operazione che il client sta facendo e questo e' il modo che il client e il server usano per comunicare l'uno con l'altro.

```
public void connectTCP()
public void disconnectTCP()
public void openStreams()
public void closeStreams()
```

Sono i quattro metodi usati per tutte le operazione per aprire e chiudere una connessione TCP e per aprire e chiudere gli stream.

```
public int register()
```

Come in tutte le operazioni da qui in poi all'inizio viene aperta la connessione TCP e gli stream di in ed out. Invia al server il pacchetto ed aspetta una risposta. A seconda della risposta manda un intero alla Gui che proseguirà di conseguenza. Alla fine chiude gli stream e la connessione TCP.

```
public int login()
```

Il metodo login inizia facendo partire il KeepAliveThread e il thread per le richieste dell'amicizia.
Dopo manda il pacchetto al server ed attende la risposta. Controlla se e' un pacchetto di conferma oppure di errore.Se tutto e' andato bene fa anche la parte RMI del client e in fine ritorna il risultato.

```
public int friendRequest(String friendUsername)
```

Invia al server un pacchetto con il nome dell'utente a cui vuole inviare la richiesta ed attende la risposta.

```
public DefaultListModel<String> getSuspendedList()
```

Invia una richiesta per avere la lista delle amicizie sospese. DefaultListModel e' un tipo di lista che serve nella Gui(per JList) e quindi la ArrayList che il server restituisce viene convertita in DefaultListModel con un metodo convertToDefList(ArrayList<String> list). La lista finale viene restituita alla Gui.

```
public int confirmRequest(String susr)
public int deleteRequest(String susr)
```
Entrambe inviano il nome scelto dalla lista al server e restituiscono il risultato alla Gui.

```
public DefaultListModel<String> getFriendsList()
public DefaultListModel<String> searchUser(String usr)
```
Inviano l'operazione da fare al server e ricevono le liste desiderate. Le liste possono anche essere vuote. Vengono comunque inoltrate alla Gui.

```
public int publish(String pmsg)
```
Siamo dalla parte del client e questa funzione invia soltanto il messaggio al server via TCP, aspetta una risposta che servirà soltanto per la Gui.

```
public int subscribe(String text)
```
Questo metodo non comunica con il server via TCP e quindi bisogna controllare la validità del token localmente, perciò ce un metodo checkTokenLocally(String token). Se il token e' valido allora chiama il metodo follow del server via RMI, che e' stato descritto nella parte del server.

```
public void logout()
```
Per prima cosa termina il thread di keep alive del client con un metodo terminate(). Il metodo terminate ferma il ciclo del thread di keep alive e ferma anche la spedizione di altri pacchetti keep alive con un flag. Perché potrebbe succedere che prima di chiudersi il thread riesce comunque a inviare un pacchetto UDP al keep alive del server.

Una parte importante che ce in tutti i metodi ma che non e' stata specificata e' quello che succede quando dal server arriva un ErrorPayload. In questo caso si controlla il messaggio del pacchetto d'errore e se il token e' scaduto allora il token del client viene messo a null, l'informazione viene inoltrata alla Gui e il keep alive thread viene terminato. Altri messaggi di errore vengono gestiti correttamente.

PayloadTCPFromClientToServer il pacchetto inviato dal Client al Server, tutti gli attribuiti sono elencati nella classe.

RemoteObjectClient e RemoteObjectClientImpl
Sono le classi per il servizio RMI del client. Contiene soltanto il metodo message che serve per aggiungere un messaggio nella lista della nostra Gui.

FriendRequestsThread Thread che accetta le richieste d'amicizia sulla TCP secondaria (sulla friendPort).

KeepAliveClientThread
Ha due flag, running e sendPacket. Che vengono messi a false quando il client fa il logout. Nel metodo run il thread si connette alla multicast e aspetta di ricevere un messaggio di keep alive dal server. Se l'ha ricevuto controlla se lo puo inviare, perche puo essere stato interrotto nel frattempo. Se sendPacket e' true allora invia al server un messaggio di keep alive. Quando running e' false significa che il thread deve finire e quindi vengono chiuse le connessione e il thread lascia pure il gruppo multicast. La terminazione avviene grazie al metodo terminateThread del SocialClient che aspetta affinche il thread finisca e solo allora fa il log out.

Gui
L'interfaccia grafica che lavora con il SocialClient.
E' costituita da un insieme di pannelli ed un solo frame.
I pannelli fondamentali sono i loginPanel e il loggedPanel.
Il loginPanel e' il panello iniziale mentre il loggedPaned e' il pannello dopo un login avvento con successo. Tutti gli altri sono pannelli secondari che aiutano per le funzioni.
I campi di testo dei vari pannelli sono fatti in modo che se vuoti allora i pulsanti collegati ai campi non sono attivi.
Quando la Gui riceve un errore di token non valido dal SocialClient il pannello cambia al panello di login. Mentre per altri errori vengono mostrati delle finestre di dialogo.

Esecuzione del programma:
(Importante) Il backUp file non deve essere cancellato.Il suo contenuto si ma non il file stesso.
Per primo va eseguito il server, aprendo il file Server.jar con un doppio click oppure dal terminale sfogliando nella cartella del progetto. Una volta dentro la cartella si deve inserire il commando: “java -jar Server.jar”. Questo maniera in esecuzione il Server. L’esecuzione piu conveniente del server e’ quella dal terminale perche cosi si possono vedere le stampe degli utenti online.
Il Client.jar va eseguito dopo l’avvio del server, con un semplice doppio click comparira un interfaccia grafica molto intuitiva.

Possibili Bug: La funzione logOut impiega un po di tempo visto che deve terminare un thread. Bisogna aspettare che faccia il logOut senza premere nessun altro pulsante.
Un altro possibile bug e’ l’avvio di tanti client, il file Client.jar non sembra aprire un nuovo client se ce gia altri client aperti ma semplicemente ci porta alla finestra di uno dei client gia aperti. Quindi conviene aprire con il terminale con la stessa procedura con cui si avvia il server.