Relazione Progetto Simple Social

Alessandro Pagiaro

2 giugno 2016

Indice

1 Descrizione del progetto

Il progetto si divide in tre package:

- SocialServer: contiene tutto ciò che riguarda il server
- SocialClient: contiene tutto ciò che riguarda il client
- SimpleSocial: contiene le strutture condivise tra client e server, le eccezioni e le classi dei messaggi scambiati

Si forniscono di appendice i diagrammi delle classi del Client e del Server

2 Comunicazione

2.1 Protocollo di comunicazione

Lo scambio di messaggi avviene tramite oggetti di tipo `PacketMessage` che vengono serializzati e quindi spediti. Per l'invio ogni `PacketMessage` viene serializzato su un `ByteArrayOutputStream`, si legge quindi la dimensione e viene scritta nella socket. Il ricevente legge i primi 4 byte, alloca un buffer delle dimensioni segnate in quei 4 byte (= int) e quindi legge i byte dell'oggetto. Al termine lo deserializza e lo passa al livello sovrastante.

Il `PacketMessage` al suo interno contiene un valore *type* che indica il tipo del messaggio contenuto al suo interno e un *message* che riferisce il `SimpleMessage`.

2.2 Server

Il server lavora tramite NIO, ricevendo i messaggi sui vari channel. Ogni *key* ha un attachment che contiene i dati ricevuti e ne tiene il conto al fine di eseguire il protocollo di serializzazione e derializzazione sopra descritto. Ogni pacchetto, una volta ricevuto completamente, viene passato al livello successivo (rappresentato dal MainServer) che avvia un thread [1] che gestisce la richiesta e imposta anche il messaggio di risposta da mandare al client [2] L'attachment di ogni *key* è di tipo `ObjectSocketChannel`.

2.3 Client

Il client lavora in maniera più semplice rispetto al server. Utilizza le normali socket con chiamate bloccanti. Il protocollo di lettura e scrittura sopra descritto è attuato tramite l'oggetto `ObjectSocket` ***`extends`*** `Socket` che ridefinisce le funzioni *write()* e *read()*.

2.3.1 `Listener`

Il client all'avvio crea un thread che si mette in ascolto su una porta casuale che verrà comunicata al server al momento del login. Tale thread gestirà le nuove richieste di amicizia in entrata ed **non** sostituisce la funzione RMI di callback per la pubblicazione dei contenuti.

2.4 KeepAlive

Il KeepAlive (da ora KA) viene gestito dal server tramite due thread creati all'avvio e definiti nel file *KeepAliveServerService* [3]. Mentre un thread spedisce un messaggio ogni KEEP_ALIVE_DELAY (vedere sezione 3) e si preoccupa dopo 10 secondi di rimuovere i client che non hanno effettuato iterazioni, l'altro thread ha il compito di ricevere le risposte KA e segnalare in una lista contenuta in `UserDB` l'avvenuta iterazione. Da notare che anche se il pacchetto viene perso (viaggia infatti su UDP), se l'utente effettua una connessione per qualsiasi altra iterazione con il server, questa viene segnalata e quindi non finirà offline non avendo risposto al KA perchè comunque una iterazione è avvenuta.

3 Parametri e costanti

Le costanti sono caricate dinamicamente all'avvio sia nel client che nel server. Il server utilizza il file `config.txt`, mentre il client utilizza il file `client_config.txt`. Entrambi i file devono essere contenuti nelle rispettive working directory.

4 Salvataggio dei dati nel server

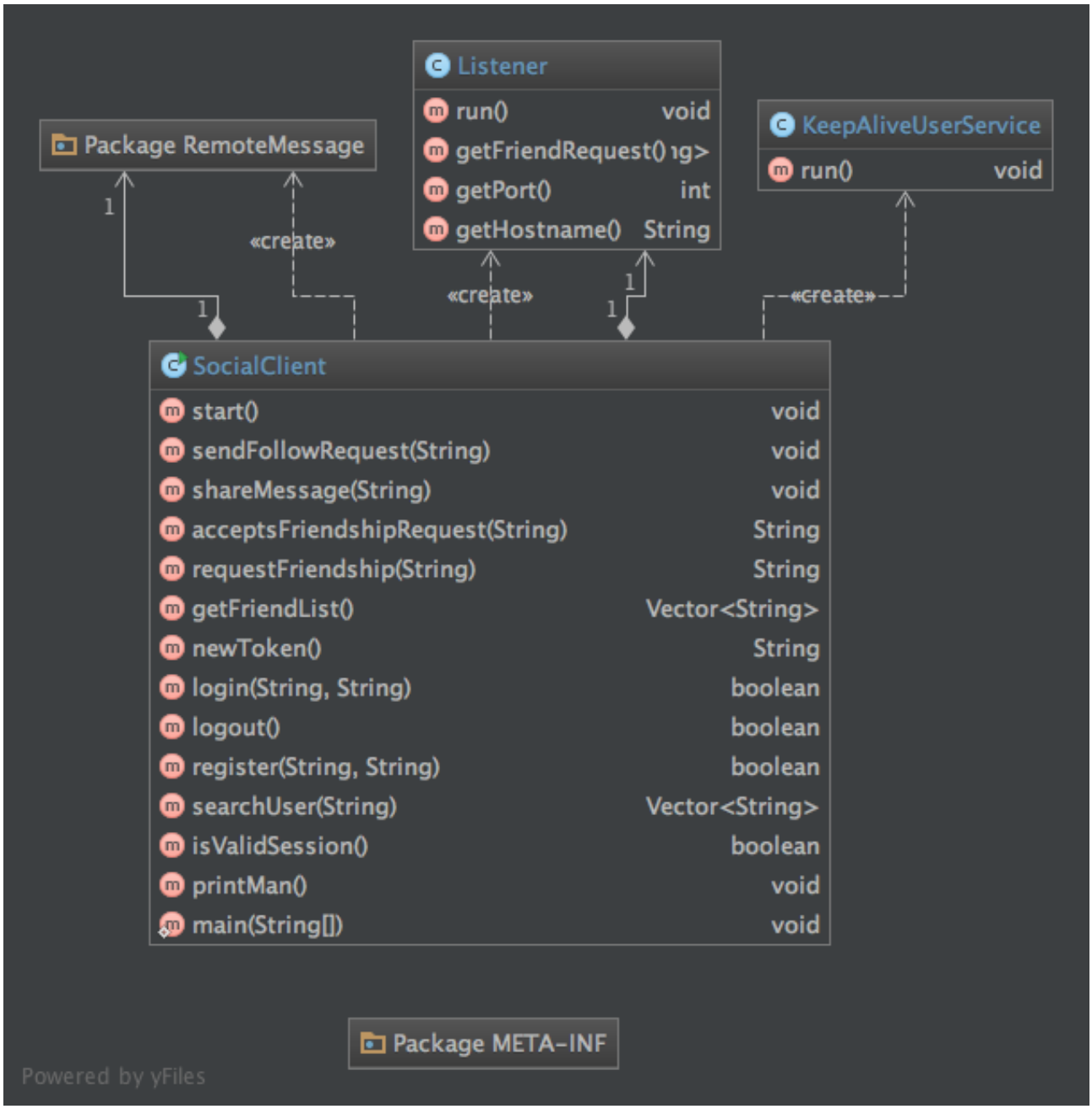
In caso di crash il server è in grado di ricostruire il proprio database. Difatti ad ogni evento fondamentale (aggiunta di un utente, richieste di amicizia, follower...) viene serializzato il database (pulito di liste temporanee come ad esempio la lista degli utenti online) su file. Tale file viene caricato dinamicamente all'avvio.

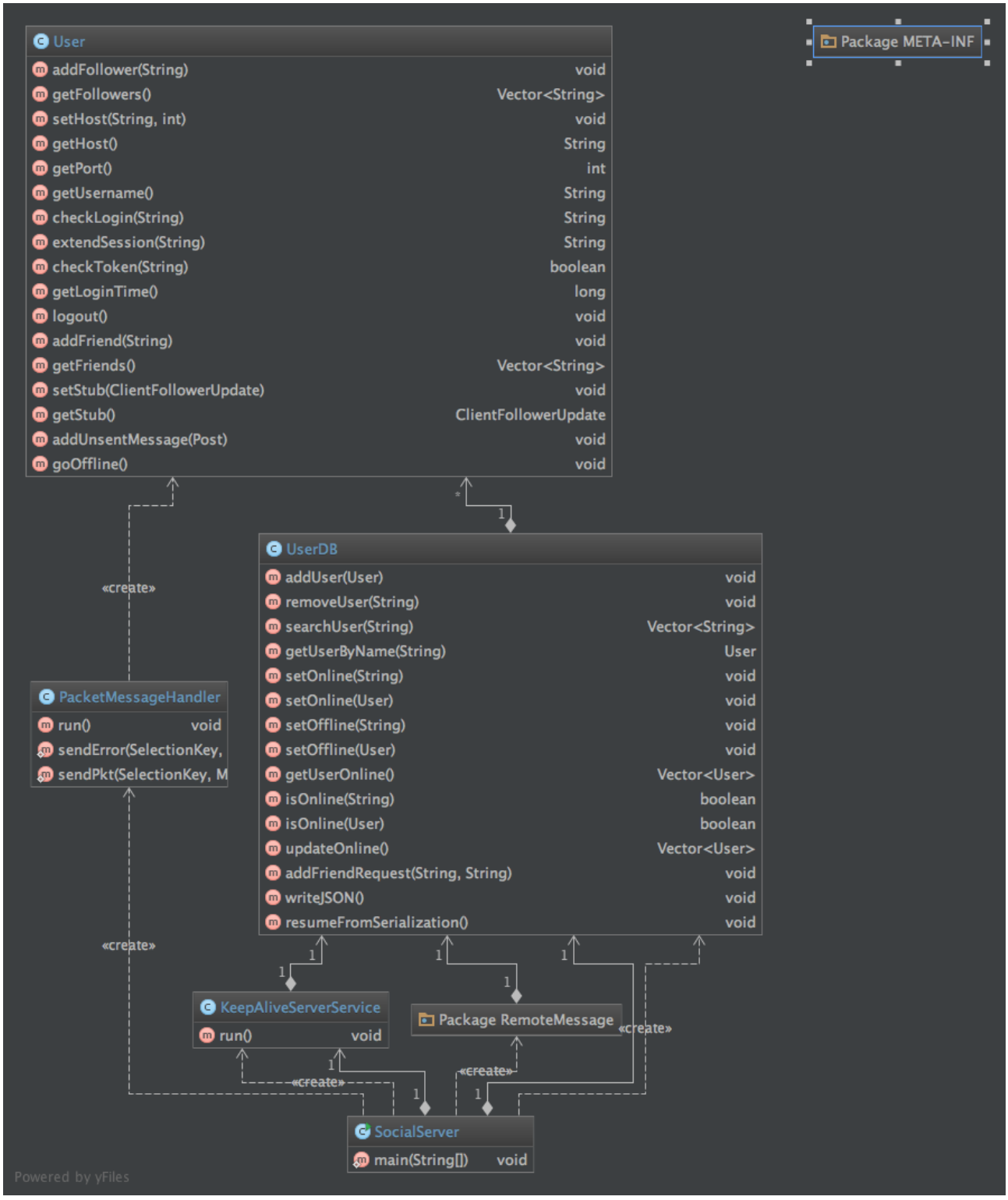
[1] Con eccezione dei messaggi riguardanti lo scambio di richieste di amicizia che poichè devono andare a registrare un nuova channel al selettore non possono che essere gestiti nello stesso thread per evitare situazioni di attesa indeterminata dovuta a lock implicite del selettore

[2] A differenza delle richieste di amicizia, negli altri casi viene usata sempre la stessa *key*, andando a modificare le *interestedOps*, operazione che risulta thread-safe

[3] Un thread è dichiarato inline nel file stesso

5 Appendice


Listener
run() void
getFriendRequest() ıg>
getPort() int
getHostname() String
KeepAliveUserService
run() void
Package RemoteMessage
1
«create»
1
«create»
1
1
«create»
SocialClient
start() void
sendFollowRequest(String) void
shareMessage(String) void
acceptsFriendshipRequest(String) String
requestFriendship(String) String
getFriendList() Vector<String>
newToken() String
login(String, String) boolean
logout() boolean
register(String, String) boolean
searchUser(String) Vector<String>
isValidSession() boolean
printMan() void
main(String[]) void
Package META-INF
Powered by yFiles


User
addFollower(String) void
getFollowers() Vector<String>
setHost(String, int) void
getHost() String
getPort() int
getUsername() String
checkLogin(String) String
extendSession(String) String
checkToken(String) boolean
getLoginTime() long
logout() void
addFriend(String) void
getFriends() Vector<String>
setStub(ClientFollowerUpdate) void
getStub() ClientFollowerUpdate
addUnsentMessage(Post) void
goOffline() void
Package META-INF
UserDB
addUser(User) void
removeUser(String) void
searchUser(String) Vector<String>
getUserByName(String) User
setOnline(String) void
setOnline(User) void
setOffline(String) void
setOffline(User) void
getUserOnline() Vector<User>
isOnline(String) boolean
isOnline(User) boolean
updateOnline() Vector<User>
addFriendRequest(String, String) void
writeJSON() void
resumeFromSerialization() void
PacketMessageHandler
run() void
sendError(SelectionKey,
sendPkt(SelectionKey, M
KeepAliveServerService
run() void
Package RemoteMessage
SocialServer
main(String[]) void
«create»
«create»
«create»
«create»
«create»
1
*
Powered by yFiles