Laboratorio Reti di Calcolatori a.a. 2015-16

Relazione `SimpleSocial`

Mario Leonardo Salinas

Corso B

Matricola 502795

Indice

1 Introduzione

Il progetto è stato testato con successo su *Ubuntu 14.02* e *Ubuntu 15.04*. Nell'implementazione delle varie funzioni e procedure l'obiettivo principale è stato quello di massimizzare la modularità del codice per migliorarne comprensibilità e manutenibilità. A questo proposito il progetto è sparso su 16 files che verranno discussi nel dettaglio in seguito. Questi 16 files implementano le funzionalità generali richieste e sono eseguibili attraverso i due `.jar` inclusi nella consegna: `Server.jar`, che esegue un'istanza del SimpleSocialServer.class, e `Client.jar` che esegue un'istanza di SimpleSocialClient.class. E' possibile configurare porte ed indirizzi delle socket e dei server RMI utilizzati modificando il file `configFile`. Per un corretto funzionamento `configFile` deve trovarsi nella stessa directory in cui i `.jar` vengono eseguiti. Di seguito è riportato un esempio di corretta formattazione del configFile:

```
tcpServerIP:localhost
tcpServerPort:2000
mcPort:2001
mcIP:225.1.1.1
rmiServerPort:2002
udpPort:2003
udpIP:localhost
```

In questo documento verranno discusse le scelte implementative architetturali principali (classi e thread attivati) e, per ogni classe e interfaccia utilizzate, verrà fornita un descrizione delle responsabilità affidategli. Per la descrizione dell'implementazione di ogni funzione e metodo utilizzati si rimanda alla documentazione redatta in `javaDoc`, inclusa nella consegna e situata nella directory `SALINAS_javaDoc`, o al codice sorgente incluso nella cartella `SALINAS_src`.

2 Architettura

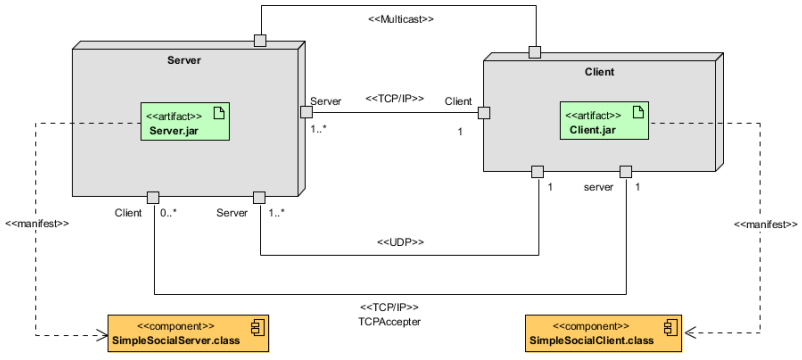
SimpleSocial è logicamente scomposto in un primo insieme di classi che implementa il comportamento richiesto al server, e un secondo insieme di classi che implementa il comportamento richiesto al generico client. I due eseguibili quindi comunicano fra di loro attraverso:

a. due socket TCP, una per servire le richieste del generico client e un'altra per effettuare controlli sullo stato di `online` dell'utente quando gli viene inviata una richiesta di amicizia;

b. una socket UDP dove il server riceve le risposte di *keepAlive*;

c. un gruppo multicast al quale aderiscono tutti i client per ricevere i messaggi di *keepAlive*;

Di seguito è riportata una vista di deployment con componenti dell'architettura del sistema che considera solamente le componenti e gli artefatti principali:



1: Vista di deployment con componenti dell'architettura del sistema

2.1 Architettura server

Il server è strutturato in 4 componenti: SimpleSocialServer, KeepAliveServer, ServerStateUpdater e SimpleSocialClientHandler. Il componente principale è SimpleSocialServer che implementa il main del server provvedendo a:

1. ripristinarne lo stato dal backup più recente, se esiste;
2. inizializzare ed eseguire il keepAlive, il server RMI e ServerStateUpdater, il thread incaricato di eseguire il backup con frequenza costante (5 s);
3. mettersi in attesa dell'arrivo di client;

Il componente KeepAliveServer ha il compito di mantenere aggiornata la lista degli utenti online in base ai messaggi ricevuti su una socket UDP dedicata, mentre SimpleSocialClientHandler riceve e processa le richieste dei client.

2.2 Architettura client

Il client è strutturato in 3 componenti: SimpleSocialClient, KeepAliveClient e ClientTCPAccepter. Il componente principale è SimpleSocialClient che, una volta che l'utente ha effettuato il login, provvederà ad avviare:

- il server RMI per gestire la ricezione e il salvataggio di contenuti provenienti da utenti per i quali è stato registrato interesse;
- il KeepAliveClient che risponderà ai messaggi di keepAlive;
- ClientTCPAccepter che, su una socket TCP nota al server solo dopo il login o l'aggiornamento del token di autorizzazione, attenderà richieste d'amicizia per memorizzarle localmente.

3 Classi lato server

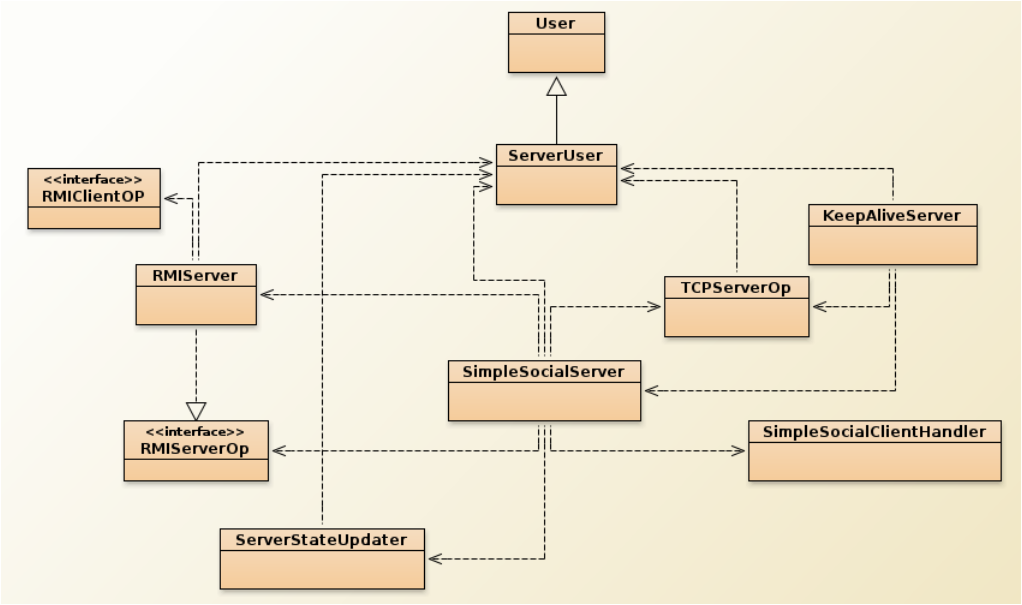
Per l'implementazione del server sono state sviluppate 8 classi e 2 interfacce:

- `SimpleSocialServer`, classe che contiene il main del server che inizializza lo stato, avvia i threads principali e si mette in attesa di nuovi client.
- `SimpleSocialClientHandler`, classe che implementa `Runnable`, viene avviata dal server all'arrivo di un nuovo client e ne gestisce le richieste;
- `ServerStateUpdater`, è una classe che implementa `Runnable` e, oltre ad eseguire il backup dei dati del server nel suo metodo `run()`, fornisce il metodo statico `readUsers()` che legge e ritorna il backup del server salvato su file;
- `KeepAliveServer`, classe che implementa `Runnable`, avviata nel main di `SimpleSocialServer`, che gestisce l'invio dei messaggi di multicast e la ricezione dei messaggi di keepAlive degli utenti su una socket UDP dedicata;
- `TCPServerOp`, classe che contiene l'insieme dei metodi per gestire le richieste di un generico client sulla socket TCP (registrazione, login, post, ecc.);
- `RMIClientOP`, interfaccia, implementata nel client dalla classe `RMIClient`, che contiene le operazioni offerte dal client via RMI: `addFollowcontent(String post)` che memorizza localmente il *post* e `addFollowed(Vector<String> followed)` che inizializza la lista degli utenti seguiti dell'utente corrente;

- `RMIServerOp`, interfaccia implementata dalla classe `RMIServer`, contiene le operazioni consentite via RMI: `follow` che permette di aggiungere un *follower* ad un utente, e `login` che completa il login dell'utente che la invoca impostando il riferimento al suo server RMI, la porta sulla quale attenderà le richieste d'amicizia e inviandoli i contenuti non ancora ricevuti e utenti seguiti;

- `User`, classe che implementa il generico utente e le funzionalità di base disponibili anche nel lato client. Per le funzionalità specifiche del server `User` è stata estesa con la classe `ServerUser` che, oltre a fornire metodi aggiuntivi utili alle operazioni che il server deve poter eseguire, estende la *superclasse* anche con variabili d'istanza che faranno parte dello stato dell'utente sul server come il suo *Online status*, i suoi *followers*, i suoi amici, la sua password e i *post* pubblicati dagli utenti seguiti mentre era offline che gli saranno inviati al nuovo login.

Di seguito è riportata un vista strutturale d'uso e generalizzazione delle classi del server:



2: Vista strutturale d'uso e generalizzazione delle classi del server

4 Classi lato client

Le classi `RMIClient`, `User` e le interfacce `RMIClientOP` e `RMIServerOP`, sebbene utilizzate anche nel client, sono state già discusse nella sezione precedente, quindi verranno omesse nella sezione corrente. Per l'implementazione del client sono state sviluppate 7 classi e 2 interfacce:

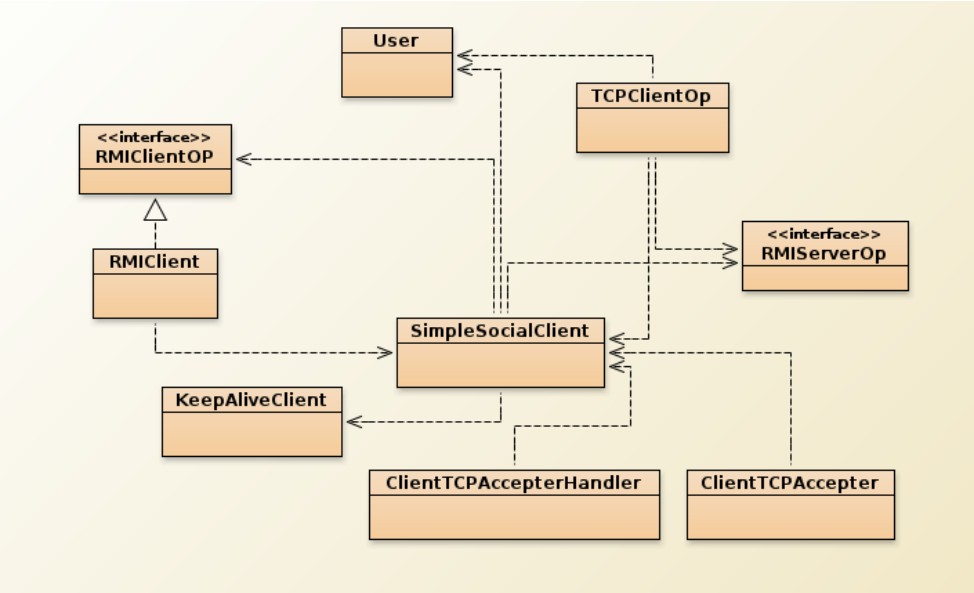
- `SimpleSocialClient`, classe contenente il main del client che, finchè non viene effettuato il login, mostra come opzioni disponibili la registrazione, il login e l'uscita dal programma. Una volta che il login viene eseguito vengono inizializzati e avviati il server RMI, `ClientTCPAccepter` e `KeepAliveClient` e le richieste dell'utente vengono inoltrate al server, o processate localmente, quando possibile.

- `KeepAliveClient`, classe che implementa `Runnable`, avviata nel main di `SimpleSocialClient`, alla quale è affidata la ricezione e l'invio dei messaggi di keepAlive rispettivamente sul gruppo multicast e sulla socket UDP dedicata;

- `ClientTCPAccepter`, classe che implementa `Runnable`, viene avviata nel main di `SimpleSocialClient` e si mette in attesa di connessioni TCP da parte del server. All'arrivo di una richiesta di connessione, questa viene passata ad un thread implementato da `ClientTCPAccepterHandler` che provvede a comunicare col server, ricevere la richiesta di amicizia e memorizzarla localmente;

- `TCPClientOp`, classe che contiene l'insieme dei metodi per gestire le operazioni lato client sulla socket TCP (registrazione, login, post, ecc.);

Di seguito è riportata un vista strutturale d'uso e generalizzazione delle classi del client:



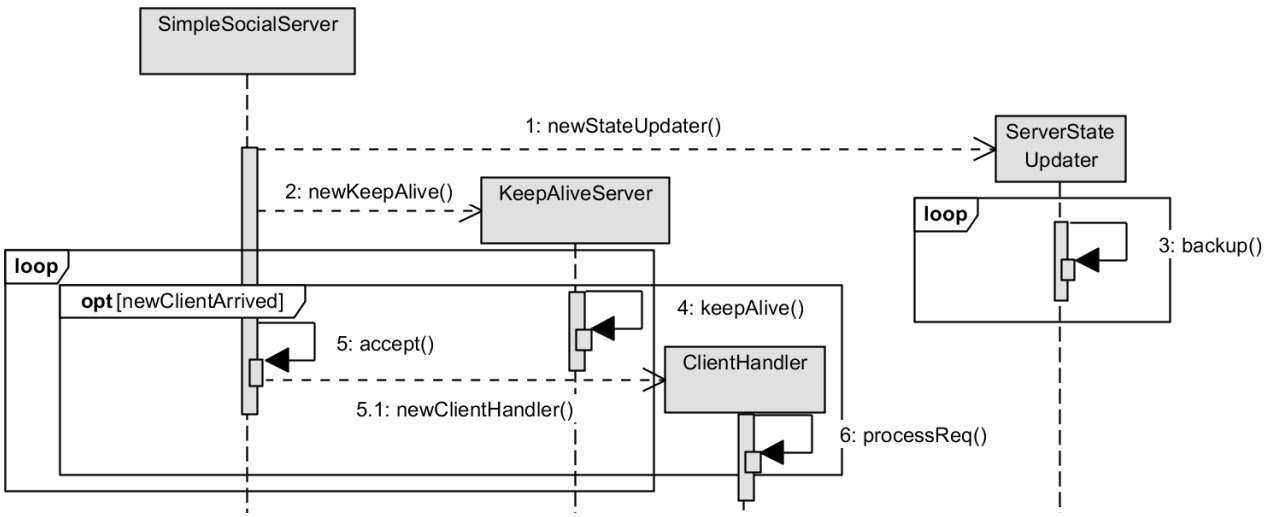
3: Vista strutturale d'uso e generalizzazione delle classi del client

5 Threads e strutture dati

Nella realizzazione di SimpleSocial, per garantire consistenza ai dati acceduti in modo concorrente da più threads, sono stati usati `Vector<>` per ogni lista sulla quale si potesse verificare una *race condition* e `Reentrantlock` con diversa granularità (lista utente, singolo utente, variabile d'istanza del singolo utente) a seconda delle esigenze. Di seguito verranno analizzati i thread del client e del server e verrà descritta la gestione della concorrenza sulle strutture dati condivise fra essi.

5.1 Threads e strutture dati lato server

Di seguito è mostrato un diagramma di sequenza che mostra l'ordine di attivazione dei vari threads nel server:



4: Diagramma di sequenza dell'attivazione dei threads nel server

Nel server la struttura dati condivisa da tutti i threads è la lista degli utenti `Vector<ServerUser> users`. Essa è acceduta essenzialmente in due modi:

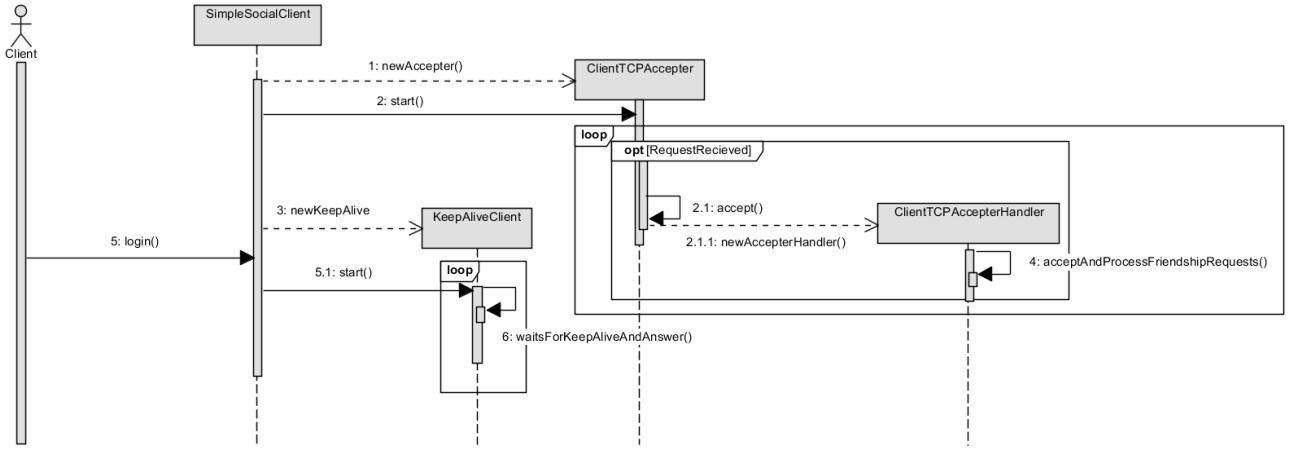
1. accesso in lettura/scrittura a tutta la lista, in questo caso il thread che ha bisogno di un accesso esclusivo all'intera lista - e.g. thread che effettua il backup - tenterà di prendere la lock definita come variabile statica in `SimpleSocialServer`;

2. accesso in lettura/scrittura ai dati di un preciso utente o ad un preciso dato di un preciso utente, in questo caso il thread dovrà acquisire la lock specifica.

Di seguito si riporta un esempio dell'acquisizione granulare delle lock nel thread che effettua il backup dello stato del server:

```
SimpleSocialServer.lock.lock();
int sizeU = SimpleSocialServer.users.size();
 for(int i = 0; i < sizeU; i++)
  size += SimpleSocialServer.users.get(i).getSize();
 if(sizeU > 0){
  MappedByteBuffer mappedFile= inChannel.map(MapMode.READ_WRITE, 0, size);
  mappedFile.putInt(sizeU);
  for(ServerUser user : SimpleSocialServer.users){
   locks = new Vector<>();
   locks.add(0, user.getPostLock());
   locks.add(1, user.getFriendLock());
   locks.add(2, user.getFollowedLock());
   locks.add(3, user.getFollowLock());
   locks.add(4, user.getReqLock());
   //Writing username, password, UID and authorization token
   locks.get(0).lock();
   //Writing toBeSentPosts after acquiring the specific lock
   locks.get(0).unlock();
   //In the same way writing the ramaining user status
  }
 }
SimpleSocialServer.lock.unlock();
```

5.2 Threads e strutture dati lato client



5: Diagramma di sequenza dell'attivazione dei threads nel server

Il diagramma di sequenza sopra descrive l'ordine di attivazione dei vari threads nel client. Nel client l'unica struttura dati condivisa fra i vari threads in esecuzione è `User user` che rappresenta l'utente che ha effettuato l'accesso nella sessione corrente e viene acceduto con modi analoghi a quelli presentati nella sezione precedente.

6 Login e token

Il login dell'utente è diviso in due parti:

1. nella prima parte il client e il server comunicano mediante socket TCP e si scambiano le informazioni per il login. Se le informazioni sono corrette il server invia al client un token che ha validità 24 ore e che servirà al server per capire se la sessione dell'utente è scaduta e deve essere rinnovata;

2. la seconda parte inizia non appena la funzione `login(BufferedReader r, KeepAliveThread kaThread)` contenuta nella classe `TCPClientOp` restituisce un oggetto di tipo `User` valido: vengono prima lette le richieste di amicizia non confermate dal file locale, se esiste, e successivamente viene invocata la funzione di login offerta dall'RMI del server che provvede a configurare correttamente i dati utente sul server - *userStub*, porta TCP e *online status* - e ad inviargli i post non letti e la lista degli utenti seguiti.

Per ogni richiesta effettuata, il client invia al server il suo `UID`, generato dal server stesso, che funge da identificatore unico. Il server cerca l'utente nella sua lista interna e controlla se la sua sessione è scaduta o meno, se sì, prima di soddisfare la richiesta del client, richiede all'utente di effettuare nuovamente il login così da potergli fornire un token valido, altrimenti la richiesta viene processata normalmente. La funzione che gestisce il nuovo login lato server e lato client è `postLogin()` che rieseguirà entrambe le fasi del login così da assicurare che sia il server che il client abbiamo dati consistenti circa la sessione corrente.

7 Persistenza dati server

La persistenza dei dati del server (lista utenti, rete amicizie, follower e post non inviati) è garantita dal thread `ServerStateUpdater` che ogni 5 secondi salva lo stato del server prima su un file temporaneo e, una volta terminata la scrittura, sostituisce il file temporaneo al file contenete il backup più aggiornato. Questa accortezza evita la perdita completa dei dati nel caso si verifichi un malfunzionamento durante le operazioni di backup.

8 Persistenza dati client

La persistenza dei dati del client riguarda i post ricevuti e non ancora letti e le richieste di amicizia non ancora confermate o rifiutate. Per quanto riguarda i post non ancora letti, se l'utente che deve ricevere il post è online la persistenza è garantita dalla funzione `addFollowcontent(String post)` fornita dall'RMI del client che provvede a scrivere su file il post appena pubblicato, se l'utente è offline il post sarà salvato in una lista locale sul server che provvederà ad inviarlo non appena il client avrà completato il login invocando la funzione `login(String uName, RMIClientOP userStub, int port)` offerta dall'RMI del server. Le richieste d'amicizia invece vengono inviate all'utente solo se è online: il server prova a connettersi alla socket TCP dedicata, se ci riesce gli invia la richiesta e, nel client, il thread `ClientTCPAccepterHandler` provvederà a salvarla sia in una lista locale all'oggetto di tipo `User`, sia su file, in modo da garantirne la persistenza. Questo file sarà letto al successivo login cosicché l'utente non perda le richieste ricevute e non ancora visualizzate.