SimpleSocial
Progetto di Fine Corso – RCL

Francesco Landolfi
<fran.landolfi@gmail.com>
Matricola: 444151

30 luglio 2016

1 Struttura del progetto

Il progetto è suddiviso nei seguenti package:

server contenente le classi necessarie all'implementazione del server di *SimpleSocial*, ovvero:

- **Server** implementa l'eseguibile ed un'interfaccia basilare; offre alcuni metodi per il savataggio ed il caricamento della configurazione e del database degli utenti;
- **TCPClientHandler** implementa il thread che gestisce una connessione TCP proveniente da un client; esegue le principali funzionalità offerte dal server;
- **OnlineUsers** implementa il database degli utenti online; per ogni utente, memorizza (tramite una classe privata, **LogInRecord**) il tempo di accesso, il token di autenticazione (`oAuth`, un intero positivo generato pseudocasualmente al momento dell'accesso) e l'indirizzo IP del client;
- **UsersDB** implementa il database degli utenti iscritti; una sua istanza potrà essere serializzata e salvata su file, per essere ricaricata in seguito dal server;
- **User** rappresenta un utente di *SimpleSocial*; per ogni utente memorizza username e password, la lista di amici, la lista di richieste di amicizia, la lista di follower e la lista di contenuti ancora non ricevuti;

PostDispatcher notifica agli utenti quando viene pubblicato un nuovo contenuto da un utente da loro seguito (tramite RMI).

client contenente le classi necessarie all'implementazione del client di *SimpleSocial*, ovvero:

Client implementa l'eseguibile ed un'interfaccia utente di tipo testuale e fornisce alcuni metodi per interagire con il server, oltre a quelli per il salvataggio ed il caricamento della configurazione del client, del database delle richieste di amicizia non ancora confermate e del database dei **Post** non ancora letti;

KeepAliveSignalHandler implementa il thread dedicato alla ricezione e alla risposta dei messaggi di *KeepAlive* mandati dal server;

NotificationHandler implementa il thread dedicato alla ricezione di richieste di amicizia inoltrate dal server; delega ogni richiesta ad un'ulteriore thread, che esegue un **Runnable** implementato dalla classe privata **Handler**;

PostNotifier implementa una callback che verrà utilizzata da un'istanza di **PostDispatcher** per mandare al client il contenuto di un **Post** pubblicato da un amico.

simplesocial contenente i precedenti package **server** e **client**, più alcune classi da loro condivise:

Follower stub per RMI (implementata poi da **PostNotifier**);

Updater stub per RMI (implementata poi da **PostDispatcher**);

ErrorCode `enum` utilizzata per la rappresentazione di messaggi di errore (o di successo); offre metodi per la conversione da **ErrorCode** a intero e viceversa;

Operation `enum` utilizzata per la comunicazione da parte del client al server per specificare il tipo di operazione da compiere; offre metodi per la conversione da **Operation** a intero e viceversa;

Post un contenuto pubblicato da un utente di *SimpleSocial*; ogni **Post** memorizza il suo contenuto e l'utente che l'ha pubblicato.

La Figura 1 mostra le varie dipendenze tra le classi del progetto.

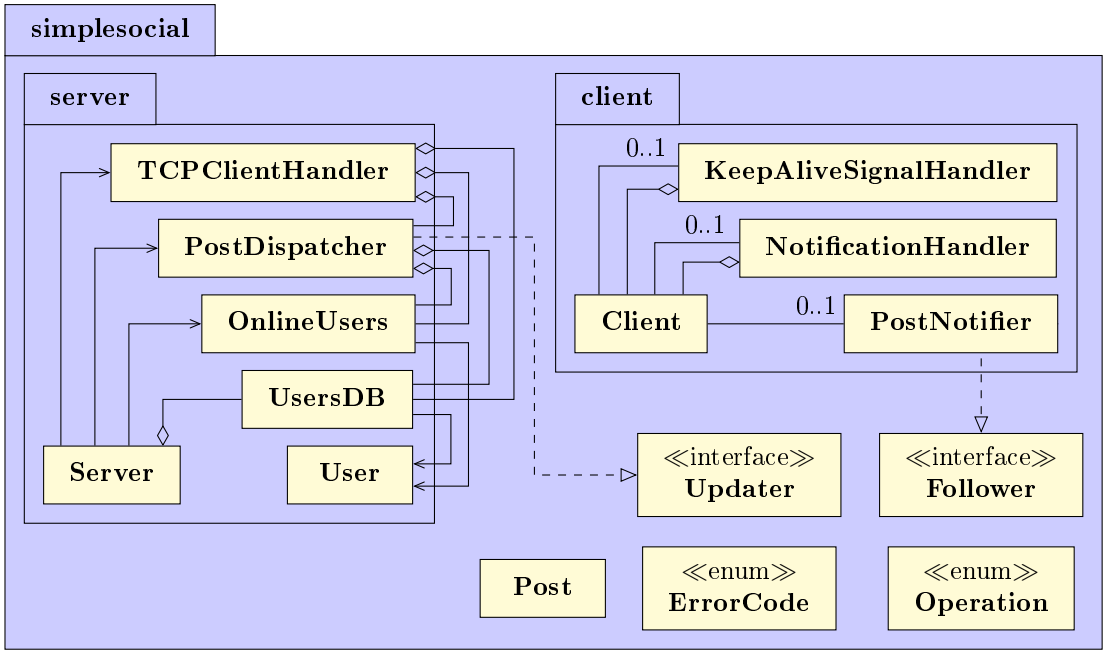

Figura 1: Diagramma di package di *SimpleSocial*

2 Eseguibili

2.1 Server

Il server può essere eseguito dall'archivio `SimpleServer.jar` con il comando

```
$ java -jar SimpleServer.jar [OPTION]...
```

Questo comando accetta varie opzioni che permettono di modificare la configurazione del server (ovvero impostare i valori delle varie porte, degli indirizzi, ecc...). Le opzioni sono:

`-h, -help` mostra le possibili opzioni e termina;

`-d, -default` utilizza i valori di default per tutti i parametri non specificati. Se non viene inserita questa opzione, il server caricherà la precedente configurazione salvata nel file `conf.json`. La configurazione caricata potrà essere comunque modificata dalle altre opzioni, se specificate;

`-s, -save` salva la configurazione generata nel file `conf.json` (al prossimo riavvio del server non sarà necessario passare nuovamente gli stessi argomenti);

`-t, -tcp-port=PORT` imposta `PORT` come porta per le comunicazioni TCP;

`-u, -udp-port=PORT` imposta `PORT` come porta per le comunicazioni UDP;

`-m, -mc-port=PORT` imposta `PORT` come porta per le comunicazioni in multicast;

`-r, -rmi-port=PORT` imposta `PORT` come porta usata dal registro RMI;

`-k, -keep-alive=NUM` imposta `NUM` come intervallo di tempo tra un mesaggio di KeepAlive ed un altro (in millisecondi);

`-g, -mc-group=IP` imposta `IP` come indirizzo del gruppo multicast;

`-L, -log-duration=PERIOD` imposta `PERIOD`[1] come periodo di validità di un accesso (dopo tale periodo l'accesso sarà considerato scaduto e l'utente dovrà rieffettuare il login);

`-R, -req-duration=PERIOD` imposta `PERIOD`[1] come periodo di validità di una richiesta di amicizia (dopo tale periodo la richiesta non viene accettata non sarà più valida e dovrà essere rieffettuata).

Una volta avviato, il server non interagirà più con l'utente: i messaggi che stamperà a video saranno solo quelli di avvenuta connessione da parte di un client e di un'eventuale modifica del database degli utenti (**UsersDB**).

File utilizzati Il processo al suo avvio accede ai file `conf.json` (a meno che non venga passato come argomento `-d`) e `users.db` presenti nella cartella `server_data/`.

Il primo contiene la configurazione del server in formato JSON, con i seguenti attributi:

`tcp_port` che rappresenta il numero di porta utilizzato per le comunicazioni TCP;

`udp_port` che rppresenta il numero di porta utilizzato per le comunicazioni UDP;

`mc_port` che rappresenta il numero di porta utilizzato per le comunicazioni in multicast;

`rmi_port` che rappresenta il numero di porta utilizzato dal registro RMI;

[1] `PERIOD` deve essere rappresentato nel formato `PnDTnHnMn.nS` (standard ISO-8601).

`mc_group` che rappresenta l'indirizzo IP del gruppo multicast;

`keep_alive_millis` che rappresenta l'intervallo di tempo tra un messaggio di KeepAlive ed un altro;

`log_duration` che rappresenta il tempo di validità di un accesso;[2]

`request_duration` che rappresenta il tempo di validità di una richiesta di amicizia.[2]

Il secondo contiene un oggetto **UsersDB** serializzato, salvato su file al termine di ogni esecuzione e ripristinato ad ogni avvio del server, per garantire che i dati di ogni utente non vadano persi tra un'esecuzione e l'altra (viene salvato anche in caso di terminazione improvvisa).[3]

Thread e concorrenza Ogni richiesta ricevuta dal server da parte di un client viene affidata ad un nuovo thread **TCPClientHandler**. Questo crea un alto grado di concorrenza nelle strutture dati condivise, ovvero **UsersDB**, **OnlineUsers** e **PostDispatcher**. Questa concorrenza viene gestita nei metodi delle classi stesse, mediante l'utilizzo di blocchi di mutua esclusione alle strutture dati interne.

2.2 Client

Il client può essere eseguito dall'archivio `SimpleClient.jar` con il comando

```
$ java -jar SimpleClient.jar
```

Questo comando non accetta ulteriori argomenti, ma una volta eseguito chiede all'utente quali azioni eseguire mediante la selezione di alcune opzioni. Appena avviato il programma chiede se effettuare il login, registrarsi, configurare il client o uscire. Nei primi due casi, dopo aver inserito username e password, se il login o la registraizione[4] vanno a buon fine, mostra le possibili interrogazioni che possono essere effettuate al server (ovvero ricercare utenti, aggiungere amici, accettare richieste di amicizia, mostrare la lista amici,

[2] Rappresentato nel formato `PnDTnHnMn.nS` (standard ISO-8601).

[3] Questo è anche il motivo principale per cui **UsersDB** e **OnlineUsers** sono due entità separate: poteva essere infatti conveniente creare un'unica classe che gestisse sia gli utenti registrati che quelli online (ed evitare così delle ridondanze), ma una volta termitata l'esecuzione, i dati degli utenti online non sarebbero stati più rilevanti, rendendo così inutile il salvataggio e ancor di più il ripristino di tali informazioni.

[4] Per motivi pratici, la registrazione esegue in automatico anche il login.

pubblicare un contenuto o seguire un amico) o in locale (mostrare i contenuti pubblicati dagli amici seguiti dall'utente o mostrare le richieste in sospeso[5]). Se si decide di configurare il client, invece, il programma offre un altro menù a scelta multipla con cui offre la possibilità di modificare i valori dell'IP del server o del gruppo multicast, delle varie porte utilizzate dal server (TCP, UDP, RMI, multicast), o selezionare l'interfaccia di rete da utilizzare per la ricezione dei messaggi in multicast[6]

File utilizzati Il processo al suo avvio accede al file `conf.json` presente nella cartella `client_data/`, contenente la configurazione del client in formato JSON, con i seguenti attributi:

`tcp_port` che rappresenta il numero di porta utilizzato dal server per le comunicazioni TCP;

`udp_port` che rppresenta il numero di porta utilizzato dal server per le comunicazioni UDP;

`mc_port` che rappresenta il numero di porta utilizzato dal server per le comunicazioni in multicast;

`rmi_port` che rappresenta il numero di porta utilizzato dal registro RMI del server;

`mc_group` che rappresenta l'indirizzo IP del gruppo multicast;

`server_address` che rappresenta l'indirizzo IP del server;

`network_interface` che rappresenta l'interfaccia di rete utilizzata dal client.

Inoltre, ad ogni login, il programma accede ai file `requests-<`*`username`*`>.db` e `posts-<`*`username`*`>.db`, che contengono rispettivamente il database delle

[5]Questa funzione viene effettuata automaticamente quando viene richiesto di accettare una richiesta amicizia: prima mostra le richieste in sospeso (in locale) poi chiede di inserire il nome dell'amico da accettare e invia la richiesta al server.

[6]Questa opzione è stata inserita poichè `NetworkInterface.getByInetAddress( InetAddress.getLocalHost() )` può restituire `null` a seconda della macchina su cui viene eseguito il programma. Il valore di default di questo attributo è `wlan0`. Se questo valore non dovesse andar bene, il thread **KeepAliveSignalHandler** non potrebbe ricevere i messaggi di KeepAlive dal server, causando così il logout automatico dopo 10 secondi (default). Si consiglia quindi di modificare questo valore al primo avvio, provando con `eth0` o con una delle interfacce mostrate col comando `$ ifconfig -a` (su sistema UNIX).

richieste di amicizia non ancora accettate (un oggetto **CopyOnWriteArrayList<String>** serializzato) e il database dei contenuti ricevuti non ancora letti (un oggetto **BlockingQueue<Post>** serializzato) appartenenti all'utente che ha effettuato il login, il cui nome utente è *username*. Questi file verranno poi sovrascritti al momento del logout. L'utilizzo del nome utente nel nome dei file garantisce la sua unicità, poichè non possono esistere due utenti con lo stesso nome. I file inoltre vengono modificati solamente se il login ha successo, quindi un utente che non conosce la password non può accedere ai contenuti dei file mediante l'utilizzo di questo programma.

Thread e concorrenza Al momento del login, il processo crea due nuovi thread: un **KeepAliveSignalHandler** ed un **NotificationHandler**. Il primo ha come unico compito quello di rispondere ai messaggi di KeepAlive da parte del server e non crea quindi grossi problemi di concorrenza. Il secondo, invece, deve attendere da parte dal server l'arrivo di una richiesta di amicizia da parte di un altro utente. Una volta arrivata, il thread crea a sua volta un altro mini-thread e torna in attesa di un'altra richiesta. Il mini-thread, nel frattempo, accetta la connessione dal server e salva la richiesta di amicizia nell'apposita struttura dati del client. Per risolvere il problema della concorrenza, viene utilizzata come struttura dati una **CopyOnWriteArrayList**, che è thread-safe.

Al momento del login, inoltre, viene creato un oggetto **PostNotifier** che verrà utilizzato come callback dal server per inviare al client i contenuti degli utenti seguiti. Questo oggetto modifica il database dei contenuti non ancora letti dal client, creando così un'altra concorrenza. Anche in questo caso viene risolta con l'utilizzo di una struttura dati thread-safe, ovvero **BlockingQueue** (viene utilizzata una coda per mantenere l'ordine di arrivo dei contenuti).

2.3 Test Suite

È possibile inoltre eseguire una suite di test sulle classi usate dal client e dal server tramite l'utilizzo dell'archivio eseguibile `TestSuite.jar` con il comando

```
$ java -jar TestSuite.jar
```

Questo comando esegue tutte le classi di test presenti nella cartella `test/`, ovvero

ErrorCodeTest esegue un test di conversione dell'`enum` **ErrorCode**;

OperationTest esegue un test di conversione dell'`enum` **Operation**;

UserTest esegue il test sui vari metodi offerti dalla classe **User**;

UsersDBTest esegue il test sui vari metodi offerti dalla classe **UsersDB**;

PostDispatcherTest esegue il test sui vari metodi offerti dalla classe **PostDispatcher**;

ServerTest esegue il test di salvataggio della configurazione e della serializzazione del database degli utenti;

ClientTest esegue un test generale di interazione[7] tra più istanze di **Client** ed una di **Server**; vengono eseguite tutte le principali funzioni.

L'esecuzione di questo processo produrrà una serie di messaggi di errore (causati di proposito). Al termine dell'esecuzione, il programma stamperà a schermo `All test were successful!` se tutti i test sono stati superati, oppure `FAIL:` seguito da un messaggio di errore se alcuni test non sono stati superati.

3 Comunicazione tra processi

3.1 Canali di comunicazione

Le comunicazioni client-server avvengono tramite **SocketChannel** per l'esecuzione delle funzionalità principali (login, registrazione, ricerca degli utenti, ecc...), **DatagramChannel** per l'invio dei messaggi di KeepAlive in multicast e per la ricezione delle eventuali risposte (client-server UDP), e **Socket** per l'inoltro delle richieste di amicizia dal server al destinatario.

Nel server, i vari **Channel** sono in modalità non-bloccante e vengono smistati con l'utilizzo di un **Selector**. Quando arriva una richiesta da parte di un client, il server accetta la connessione la delega ad un nuovo thread **TCPClientHandler**. Il canale risultante viene lasciato in modalità bloccante, per permette di implementare più facilmente un protocollo di comunicazione tra il server ed il client.

[7] In alcuni punti del test vengono utilizzate delle `Thread.sleep()`. Queste non servono a risolvere una concorrenza non gestita, bensì ad attendere che il client (o il server) abbia eseguito l'operazione richiesta (ad esempio: quando il client esegue una `post()`, il server risponde con `SUCCESS` quando ha ricevuto correttamente l'istruzione ma non quando ha terminato di eseguirla. Questo può causare un'errore se si eseguono un'operazione di `post()` e una operazione di lettura dei contenuti in rapida successione, ma non se viene utilizzato con tempi di reazione *umani*).

3.2 Protocollo di comunicazione

I client ed il server si scambiano messaggi tramite l'utilizzo di **ByteBuffer**. I messaggi sono composti per lo più da interi e stringhe, convertiti in array di byte. Ogni stringa è sempre preceduta dalla lunghezza che occupa nel buffer (quindi, da un intero). Il contenuto dei messaggi varia a seconda della funzionalità da eseguire:

`login()`, `register()` invia la codifica dell'operazione da eseguire in intero, seguita dall'username, la password, ed il numero di porta[8] in cui il client attende connessioni per l'inoltro di una richiesta di amicizia. Il server risponde con un intero che rappresenta un **ErrorCode** se negativo, altrimenti l'`oAuth` se positivo (il client ha effettuato il login con successo);

`addFriend()` invia la codifica dell'operazione da eseguire in intero, seguita dall'username, l'`oAuth`, ed il nome dell'utente a cui inviare la richiesta di amicizia. A questo punto, se l'autenticazione è valida e l'utente è online, il server apre una socket al suo indirizzo tramite la porta da lui specificata e gli invia una stringa contenente l'username dell'utente che ha effettuato la richiesta. Dopodichè, in ogni caso, il server risponde con un intero che rappresenta la codifica di un **ErrorCode**. La Figura 2 mostra il diagramma di sequenza relativo allo svolgimento di questa funzionalità;

`confirmFriendship()` invia la codifica dell'operazione da eseguire in intero, seguita dall'username, l'`oAuth`, ed il nome dell'utente a cui confermare la richiesta di amicizia. Il server, dopo aver confermato la richiesta, risponde con un intero che rappresenta la codifica di un **ErrorCode**.

`getFriendsList()` invia la codifica dell'operazione da eseguire in intero, seguita dall'username e dall'`oAuth`. Il server, dopo aver verificato l'autenticazione, manda un intero che rappresenta un `ErrorCode` se negativo, altrimenti il numero di utenti trovati nella lista di amicizie. Dopodiché, seguiranno tutti i nomi degli amici trovati in coppia con il loro rispettivo stato, codificato ad intero: 0 se online, -1 se offline. Se il numero degli utenti trovato è alto, le coppie utente-stato possono essere distribuite più su buffer. Il client saprà che ci saranno altri buffer da ricevere finche non avrà ricevuto un numero di utenti pari a quello specificato all'inizio della serie.

[8] La porta viene generata dal thread **NotificationHandler** in modo dinamico. Questo permette di poter utilizzare più client su una stessa macchina.

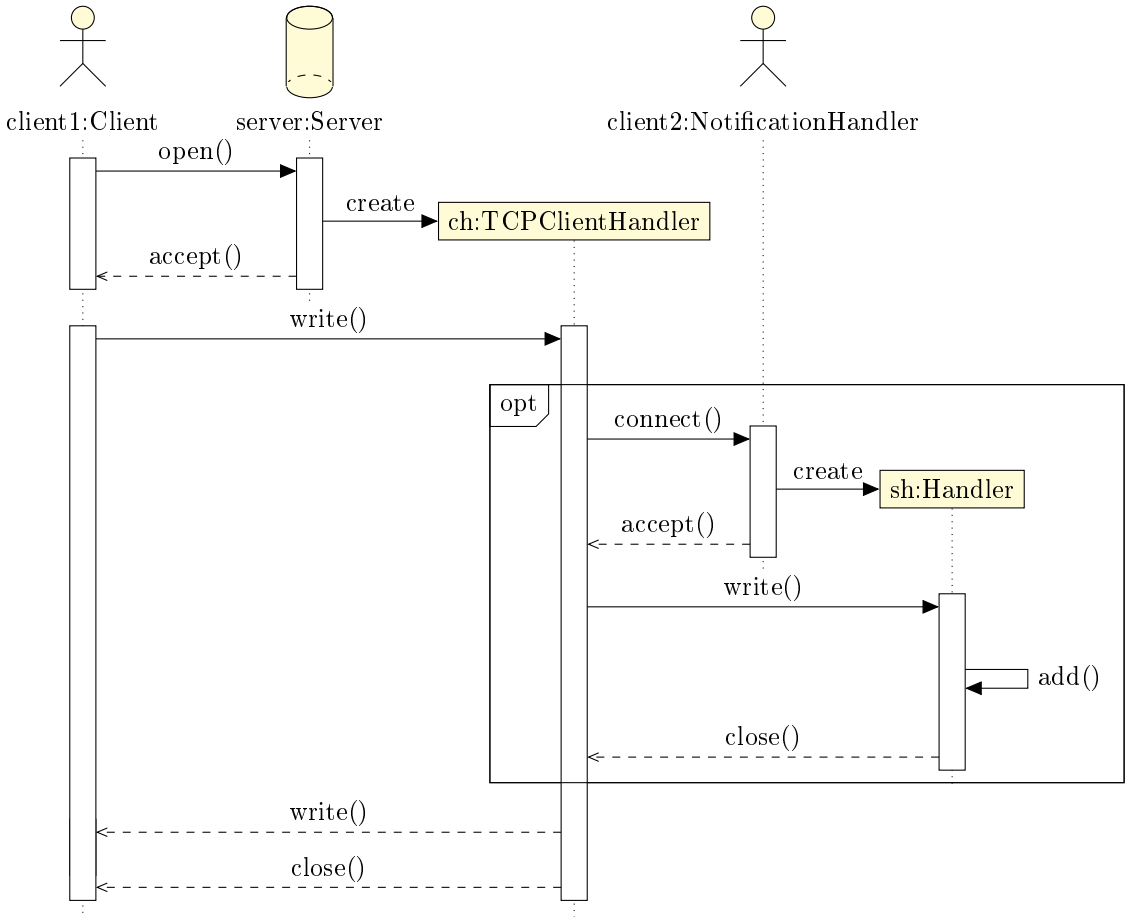

Figura 2: Diagramma di sequenza di `addFriend()`

`search()` come `getFriendList()`, ma senza la codifica dello stato attuale;

`post()` invia la codifica dell'operazione da eseguire in intero, seguita dall'username e dall'`oAuth`. Il server, dopo aver verificato l'autenticazione, manda un intero che rappresenta un `ErrorCode`. Se quest'ultimo è uguale a `SUCCESS`, il client invia al server la stringa del contenuto da pubblicare (sempre preceduta dalla sua lunghezza). Il contenuto può essere diviso su più buffer all'occorrenza;

`logout()` invia la codifica dell'operazione da eseguire in intero, seguita dall'username e dall'`oAuth`. Il server, dopo aver eseguito il logout, risponde con un intero che rappresenta la codifica di un **ErrorCode**.

4 Note finali

Il progetto è stato sviluppato con IntelliJ IDEA 2016.2 (è possibile trovare i file relativi al progetto all'interno della cartella `SimpleSocial/`). Sono state utilizzate le librerie standard Java (JDK 8 – Oracle), JSON-Simple 1.1.1, e JUnit4 per lo sviluppo della suite di test.

Ulteriori informazioni riguardo al progetto possono essere trovate nei vari commenti all'interno dei file sorgente presenti in `src/simplesocial/`, oppure consultando il JavaDoc presente in `doc/html/index.html`.

Possibili errori Eseguendo il test può capitare (in modo apparentemente casuale) che nei punti dei metodi di **Client** dove vengono effettuate più letture dal server o più scritture al server in successione (ad esempio in `getFriendList()`, `search()` o `post()`) venga lanciata un'eccezione del tipo **BufferUnderflowException**[9]. Pare quindi che qualche volta il server (o il client) non invii tutti i byte necessari a completare il messaggio, causando così il lancio di questa eccezione.

[9] Questa eccezione viene lanciata quando, utilizzando un **ByteBuffer**, l'indicatore `position` va oltre l'indicatore `limit`.