Università di Pisa, CdL in Informatica, A.A. 2015-16

Relazione del Progetto di Laboratorio di Reti

Simple-Social: implementazione di una Online Social Network

Giorgio Vinciguerra

5 giugno 2016

Indice

1. Introduzione

Il progetto di fine corso di Laboratorio di Reti richiedeva di realizzare un social network elementare con architettura client-server che utilizzasse i vari meccanismi di comunicazione visti durante le lezioni. Questa relazione è volta a spiegare l'architettura generale della mia soluzione, nonché le scelte progettuali più importanti. I file sorgente sono raccolti nelle appendici.

1.1. Struttura del codice

Il codice sorgente del programma è stato diviso in tre package:

1. `socialnetwork`. Realizza la logica della rete sociale dichiarando classificatori quali `User`, `Post` e metodi per la registrazione di utenti, l'aggiunta di legami di amicizia e di interessi di un utente verso i contenuti di un altro.
2. `server`. Realizza la parte del sistema che memorizza lo stato di una rete sociale e offre ai client canali di comunicazione su cui inviare richieste di registrazione, login, pubblicazione di contenuti ecc. Il sotto-package `server.gui` implementa una semplice finestra che facilita l'avvio del server.
3. `client`. Realizza il lato client, fornendo una classe omonima i cui metodi permettono a un utente di interagire col server. Anche qui è presente un sotto-package di interfaccia grafica.

I tre package verranno nel dettaglio illustrati nelle successive sezioni 2, 3 e 4. Segue in questa sezione una breve descrizione dei modi con cui il server e i client comunicano.

1.2. Protocollo di comunicazione

Tutti i messaggi spediti dal client al server tramite TCP hanno un'intestazione di un byte che indica il tipo di richiesta. I valori possibili sono dati dalle costanti simboliche definite in `server.RequestTypes` e riportate nella seguente tabella:

REGISTER	LOGIN	LOGOUT
FIND_USER	GET_FRIENDS	PUBLISH
FORWARD_FRIEND_REQUEST	ACCEPT_FRIEND_REQUEST	DENY_FRIEND_REQUEST

Ad esclusione dei messaggi di registrazione e login, al tipo di richiesta segue un campo di 4 byte contenente il token dell'utente. La restante parte di un messaggio è variabile in base al tipo di richiesta. La Figura 1.1 mostra in modo sintetico lo scambio di dati tra le parti interessate da una richiesta di amicizia.

Per quando riguarda UDP, utilizzato per la funzionalità di keep-alive, il server invia su un gruppo multicast un datagramma contenente il simbolo '?'; i client connessi rispondono inviando il proprio token direttamente al server.

1.3. Interfacce remote

L'insieme dei metodi remoti che realizzano le funzionalità di registrazione interesse e notifica dei contenuti di un amico è definito nelle interfacce `RemoteNotificationReceiver/Sender`. La prima, implementata da ogni client, oltre al metodo callback prevede un metodo `getToken()`

col quale il server verifica l'identità dell'oggetto remoto. La seconda interfaccia offre invece l'operazione di registrazione d'interesse.

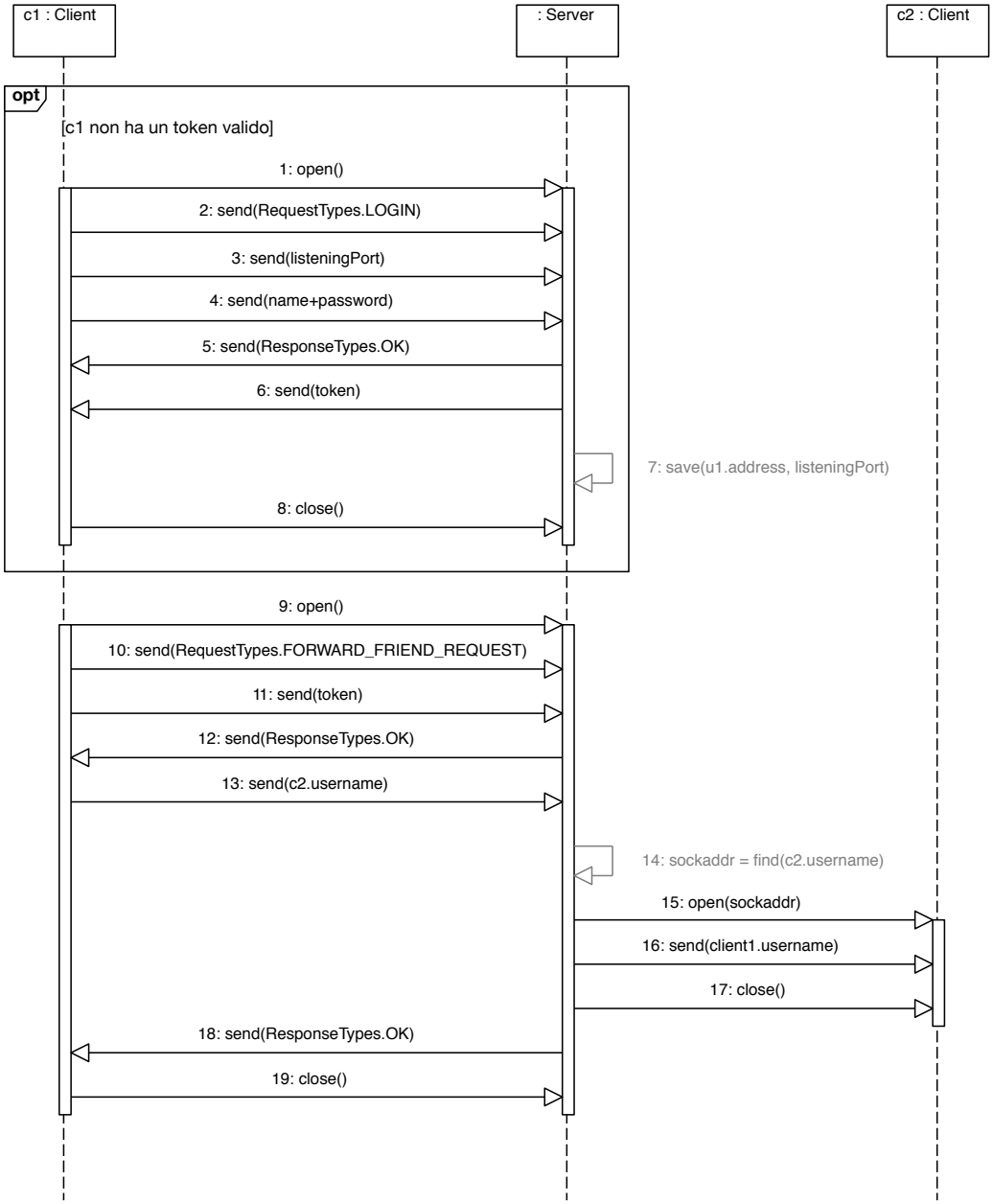
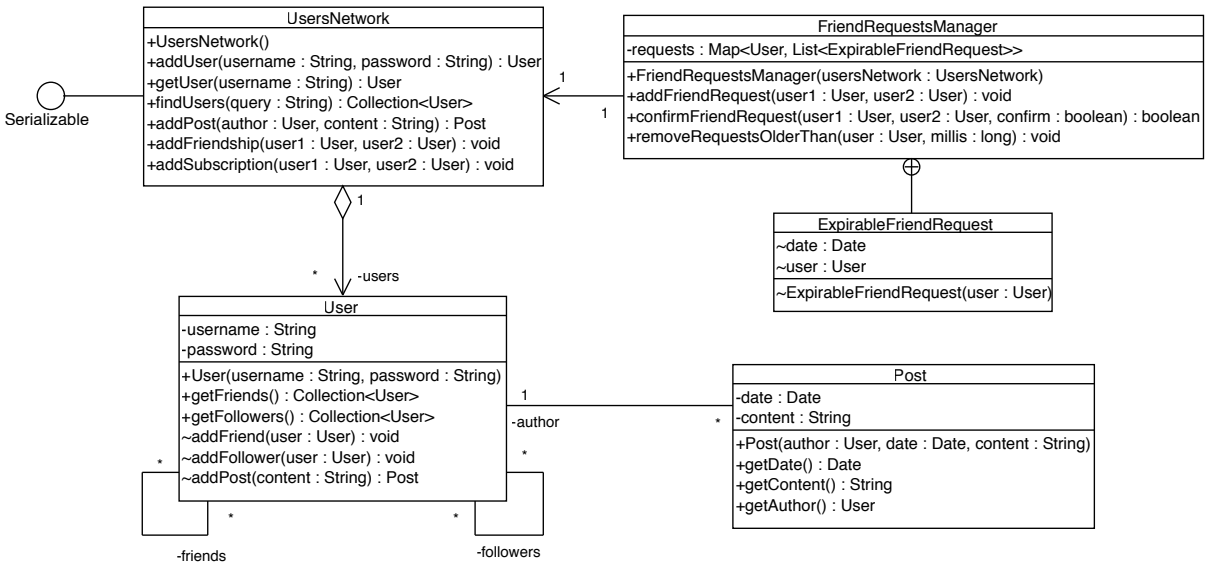


Figura 1.1: Le interazioni tra due client e il server durante l'invio di una richiesta d'amicizia. Il server memorizza per ogni client l'indirizzo IP e la porta su cui è in ascolto di richieste d'amicizia.

2. La rete sociale

2.1. Descrizione delle classi

Le classi che rappresentano lo stato della rete sociale sono raffigurate nel seguente diagramma:



`UsersNetwork` offre metodi che manipolano la rete, garantendo nel contempo il rispetto dei vincoli di univocità dei nomi degli utenti e di simmetria della relazione d'amicizia. Al di fuori del package `socialnetwork`, i soli metodi visibili sono quelli pubblici. Pertanto non è possibile cambiare la rete in modo incontrollato (ad esempio chiamando il metodo `User.addFriend()` che è package-private), se non passando per un'istanza di `UsersNetwork`.

Oltre all'insieme degli `User`, ho voluto aggiungere in `UsersNetwork` una variabile d'istanza `HashMap⟨String, User⟩` per mantenere costante la complessità delle operazioni di recupero utente per nome, che nel server sono *molto* frequenti.

`FriendRequestsManager` aggiunge a `UsersNetwork` la funzionalità delle richieste d'amicizia con scadenza.

2.2. Concorrenza e sincronizzazione

L'istanza di `UsersNetwork` presente nel server necessita sia di protezione da accessi concorrenti, che di un alto livello di concorrenza. A tale scopo è stata usata una `ReadWriteLock` in modo tale che più thread possano leggere lo stato della rete sociale concorrentemente, ma soltanto uno vi può scrivere.

3. Il server

3.1. Paradigma blocking-I/O

Il server adotta il modello blocking-I/O, con un thread per connessione. Ho preferito questa tecnica rispetto a un server single-thread con non-blocking I/O per i seguenti motivi:

- tutte le interazioni tra client e server sono di brevissima durata, quindi i thread non rimangono bloccati a lungo sulle operazioni di lettura/scrittura; di conseguenza, usando un thread pool l'overhead di spazio e tempo dovuto alla creazione e la gestione di tanti thread è basso;
- per sfruttare l'architettura multicore della macchina su cui viene eseguito il server;
- per semplicità di codice e riduzione di spazio usato, in quando col non-blocking I/O andrebbe salvato e ripristinato lo stato di ogni interazione.

3.2. Descrizione delle classi

Con riferimento alla Figura 3.1, le classi che partecipano alla realizzazione della componente server sono le seguenti:

- `UsersNetwork` e `FriendRequestsManager`. Mantengono lo stato della rete sociale (utenti, amicizie, post, ...) e le richieste d'amicizia non ancora confermate.
- `SessionsManager`. Ha la responsabilità di gestire le sessioni degli utenti nella rete sociale. Una sessione viene creata quando un utente fa il login e viene distrutta quando lo stesso fa il logout o dopo un periodo definito (v. parametro del costruttore). Ogni sessione ha associato un *token di autenticazione*, spedito al client all'inizio della sessione e utilizzato dallo stesso in ogni comunicazione col server; un oggetto *socket address* (indirizzo IP, numero di porta) che permette al server di inoltrare al client le richieste di amicizia ricevute da un altro client; una *data di ultima attività*, aggiornata con ogni interazione client→server.
- `NotificationManager`. È un'implementazione dell'interfaccia `RemoteNotificationSender` e si occupa di invocare una callback RMI registrata da utenti che vogliono essere avvisati quando un loro amico pubblica un nuovo contenuto. A tale scopo memorizza per ogni utente un'associazione `User`→`RemoteNotificationReceiver` e un'associazione `User`→`Collection<Post>`, cioè una collezione di contenuti non ancora notificati perché l'utente è offline.
- `Server`. Oltre ad aggregare le classi appena descritte:
  - avvia i thread che inviano periodicamente le richieste di keep-alive ai client e ne raccolgono le risposte;
  - realizza la funzionalità di backup/ripristino dello stato della rete sociale in caso di spegnimento del server, sfruttando la serializzabilità di `UsersNetwork`;
  - a ogni connessione di un client sottomette un nuovo `ServerTask` a un thread pool. `ServerTask.run()` legge l'intestazione del pacchetto inviato dal client e lo smista a uno dei metodi privati della classe (es. `login()`, `findUser()`, ...).

3.3. Concorrenza e sincronizzazione

Ogni singola richiesta da parte di un client viene eseguita dal server su un thread separato e accede alle strutture dati dello *stato della rete sociale* e/o delle *sessioni* in modo concorrente con le altre richieste. La sincronizzazione degli accessi su `UsersNetwork` è stata già discussa nella sezione 2.2; quella su `SessionsManager` adotta la stessa soluzione: si fa uso di una `ReadWriteLock`.

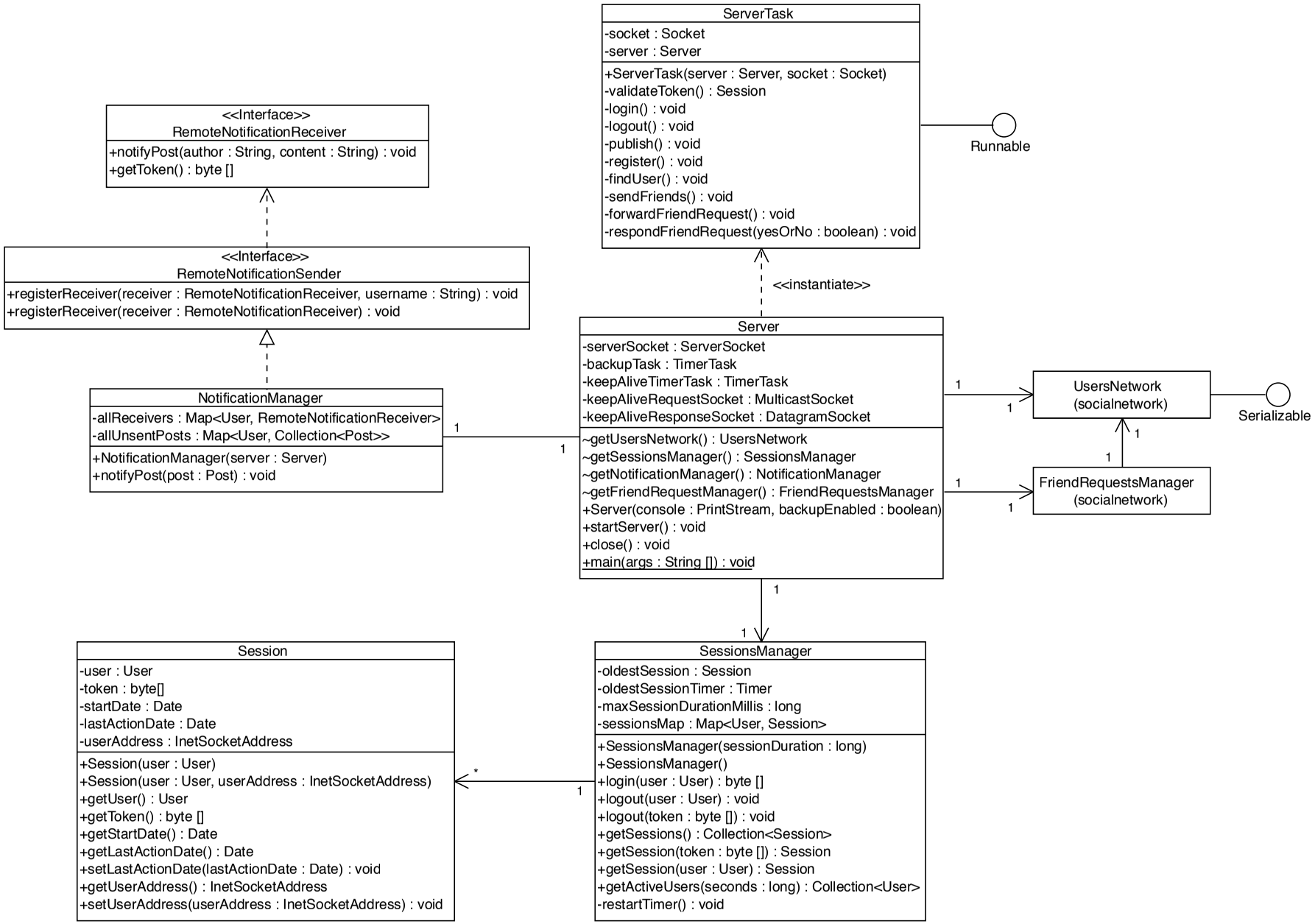


Figura 3.1: Diagramma UML delle classi che realizzano la componente server.

4. Il client

4.1. Descrizione delle classi

Similmente al server, ho isolato la logica della componente client dall'interfaccia grafica. Come mostrato nella Figura 4.1 è presente una classe centrale `Client` che offre tutte le operazioni che un utente può eseguire nella rete sociale, nascondendo le comunicazioni col server.

Le classi di supporto sono le seguenti:

- `ShortConnectionFactory`. Costruisce socket connessi a un hostname e un numero di porta fissati (quelli del server).
- `AuthenticationManager`. Gestisce la connessione di un singolo utente in un SimpleSocial server. In particolare si occupa della registrazione e del login di un utente, nonché della richiesta di nuovi token quando questi scadono. Un oggetto `Client`, chiamando il metodo `makeAuthenticatedConnection()` di questa classe, ottiene una connessione col server su cui è già stato spedito il token di autenticazione e ricevuta una risposta positiva: un'operazione necessaria prima di ogni richiesta al server e che quindi imponeva un'astrazione procedurale.
- `FriendRequestsReceiverTask`. Si mette in ascolto su una porta casuale per ricevere le richieste di amicizia inoltrate da un server. Il numero di porta scelto viene comunicato al server durante la procedura di login.
- `KeepAliveResponderTask`. Riceve e risponde ai messaggi di keep-alive. Le richieste di keep-alive vengono fatte dal server su un gruppo multicast; le risposte (che contengono il token) avvengono tramite una connessione UDP diretta col server.

A queste si aggiungono due classi immutabili, `FriendWithStatus` e `PostWithAuthor`, definite per incapsulare il risultato delle richieste `retrieveFriends` e `retrieveUnreadPosts`.

Il package definisce anche un'interfaccia `ClientEventListener`: un oggetto che la implementa, registrandosi con `Client.setClientEventListener()`, verrà notificato degli eventi asincroni "richiesta di amicizia" e "nuovo contenuto pubblicato da un amico". Le classi che realizzano l'interfaccia grafica utente utilizzano proprio questo meccanismo di notifica per aggiornare dinamicamente la lista delle richieste e quella dei contenuti, senza che l'utente debba ogni volta premere tasti per ricaricarle.

4.2. Concorrenza e sincronizzazione

Nel package, le uniche due risorse che richiedono un accesso in mutua esclusione sono le variabili d'istanza `pendingFriendRequests` e `unreadPosts` in `Client`: la prima viene acceduta in scrittura da `FriendRequestsReceiverTask` (eseguito su un thread ausiliario); la seconda viene modificata dal metodo *remoto* `notifyPost()`. In entrambe i casi sono stati utilizzati i meccanismi synchronized method/statement di Java.

4.3. Esempio d'uso della classe `Client`

Qui si riporta un breve esempio di codice che registra un utente sulla rete sociale ed effettua due operazioni prima di uscire. Per semplicità e stata omessa la gestione degli errori.

```
1 ShortConnectionFactory factory = new ShortConnectionFactory("localhost", 11234);
2 AuthenticationManager.register(factory, "Alan", "pwd");
3 Client client = new Client(new AuthenticationManager(factory, "Alan", "pwd"));
4 client.publish("qwerty␣uiop");
5 client.friendRequest("Ada");
6 client.close();
```

La prima riga istanzia la classe che crea le connessioni col server all'indirizzo localhost:11234. La seconda usa il metodo statico `AuthenticationManager.register()` per inviare al server una richiesta di registrazione di un utente con le credenziali specificate. La terza istanzia un `AuthenticationManager` e lo passa a un nuovo oggetto `Client`, che lo userà per ottenere connessioni autenticate. Seguono le operazioni di pubblicazione di un nuovo contenuto e l'invio di una richiesta d'amicizia a un utente.

Il codice si conclude con una chiamata a `client.close()` che effettua il logout dell'utente e termina i thread ausiliari attivati dal costruttore.

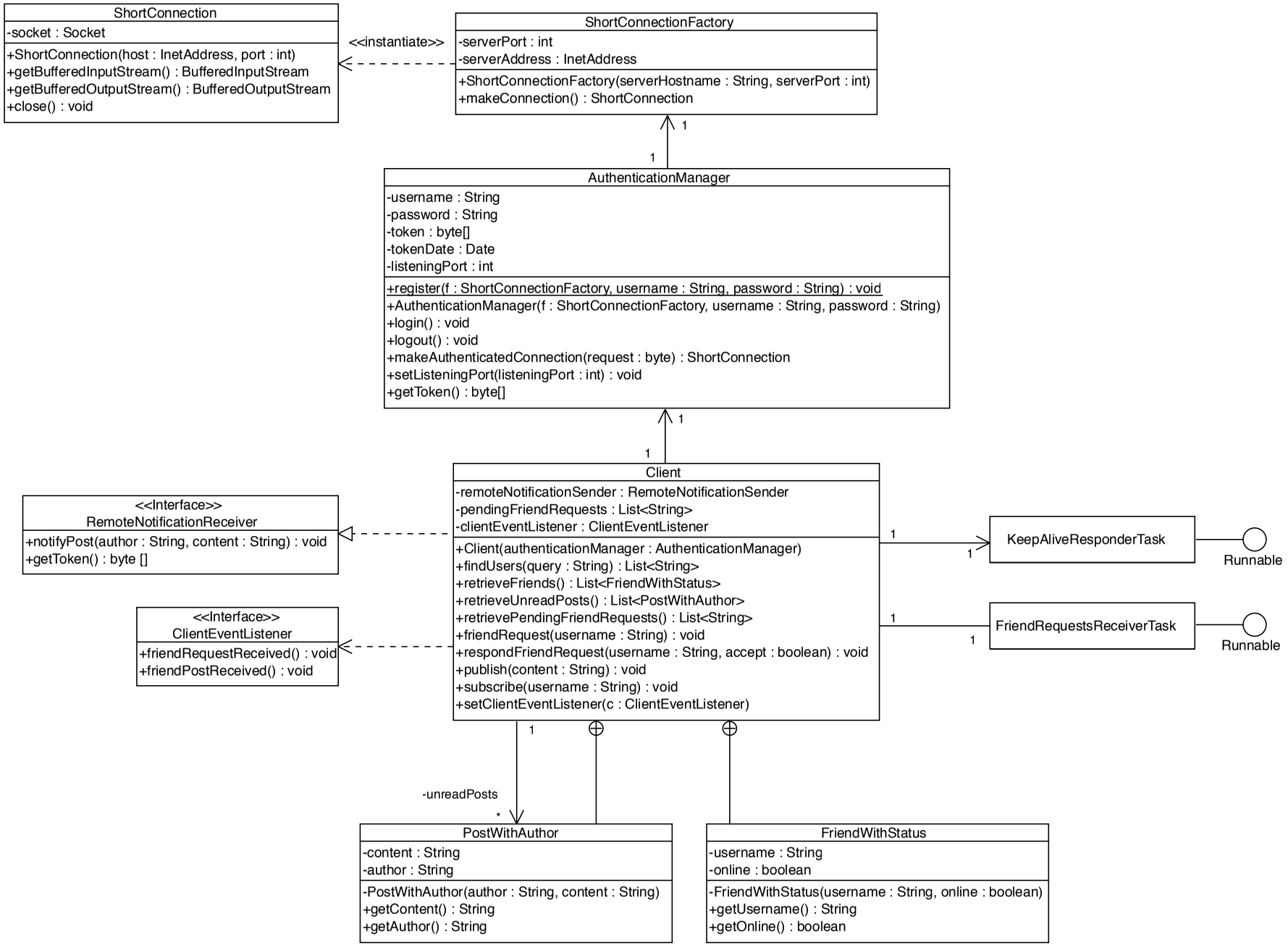


Figura 4.1: Diagramma UML delle classi che realizzano la componente client.

5. Manuale utente

Per lanciare le applicazioni server o client è sufficiente fare doppio click sul rispettivo file con estensione .jar.

L'applicazione server è fatta di una sola finestra che mostra un'area con i messaggi di log e due pulsanti per l'avvio e l'interruzione della componente server: dopo che è stato premuto il pulsante *Start*, i client possono cominciare a connettersi e a utilizzare i servizi offerti dal server.

L'applicazione client è composta da due finestre: quella di login e la home. Nella finestra di login è possibile configurare l'indirizzo del server (localhost per default), registrarsi e connettersi alla rete sociale. La finestra home (v. Figura 5) mostra:

- sulla sinistra i contenuti generati dall'utente e dagli amici che segue, una casella per pubblicare un nuovo messaggio e un tasto per tornare alla finestra di login;
- sulla destra tre pannelli:
  - *Search*, che realizza la funzionalità di ricerca di un utente per nome;
  - *Friends*, che mostra gli amici dell'utente, colorati in verde se sono online. Premendo su un amico si attiva il tasto *Follow* col quale l'utente si iscrive ai contenuti pubblicati dall'amico;
  - *Requests*, che elenca le richieste d'amicizia ricevute. Premendo su un nome in questa lista si attiveranno i tasti *Accept* e *Deny* per rispondere alla richiesta.

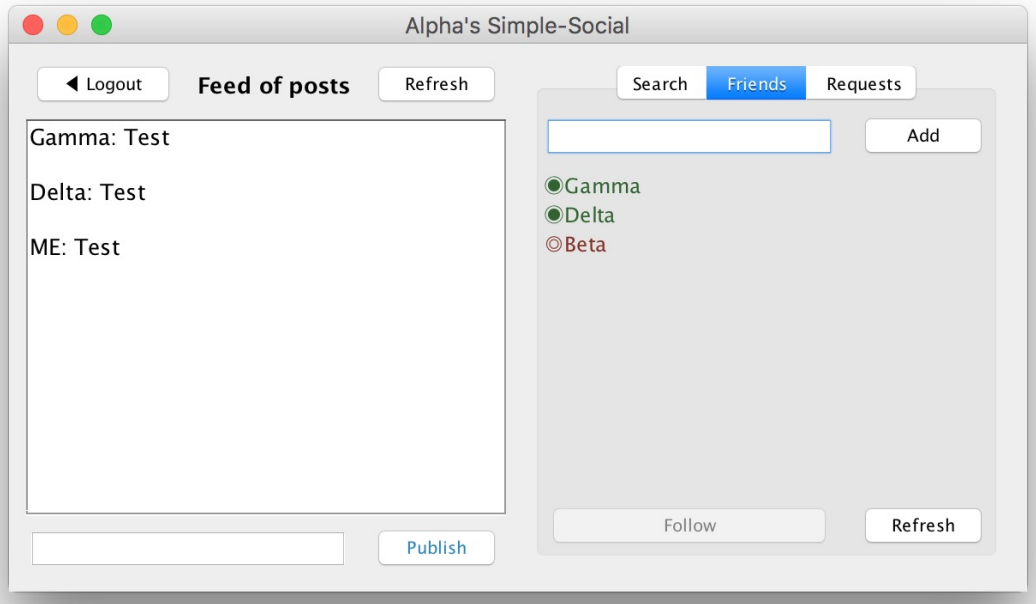


Figura 5.1: La finestra home dell'applicazione client.