CMPE436 Final Project - Follow-it

Yiğit Özkavcı
2013400111
yigit.ozkavci@boun.edu.tr

December 2017

Contents

1 Abstract **2**

2 Introduction **2**

3 Approach **3**
3.1 Server 3
3.1.1 Client Thread 3
3.1.2 Messages 4
3.1.3 Commands 5
3.1.4 Channels 6
3.1.5 Rate Limiting 6
3.1.6 Optimizations 6
3.2 Client 6

4 Concepts **6**
4.1 Subscriptions 6

1 Abstract

Follow-it is an Android application that lets you get information on what someone is posting on their Twitter account. Along with that, Follow-it lets you get continuous updates whenever someone tweets something, by making use of sockets.

In the following sections, we will describe the behavior of both server and client, and how we encode the data over the sockets in a useful manner.

2 Introduction

Motivation of creating Follow-it is to be updated whenever someone updates their feed on the Twitter. With the Twitter UI being distracting for this simple achivement, this Android application achieves the task well enough.

The application has a simple interface and user flow: a screen with a text field welcomes the user, user enters his/her name and registers. At this time, application sends the request to server via socket, in FIMessage(see 3.1.2) format.

After registration is accepted by the server, user is ready to add new subscriptions (for more on subscriptions, see 4.1). To do this, user clicks to "Add Subscription" button and fills the "username" information in there. After user's attempt on creating a subscription is completed, it's the time an approval/rejection message comes from the server. At this point, server decides whether it's a valid Twitter username (server makes this test via twitter4j[1] library) and based on this decision, tells the client whether its subscription creation request is to be fulfilled or not.

After subscription creation attempt is completed, user will get an alert box denoting whether the attempt has been successful or not. If successful, a new subscription record is added to both server and client. In server, this subscription info is kept in-memory, and without any databases. In client, though it's different. We use Android's Room[2] persistence library to make sure we don't lose and data while transitioning between activities & opening and closing the application.

In dashboard page, we show user all of his/her subscriptions which s/he can click to find all updates about **that** subscription. In single subscription page, we present all of the tweet updates that has been received by this client until now.

Due to way server handles updates, we have the initial data paged with 20 tweets maximum, and incrementally watch for the updates for every 5 seconds (see section for rate limiting: 3.1.5). And server does some optimizations (see 3.1.6) regarding the fetching for updates, because of the mentioned rate limiting.

3 Approach

We will divide discussing our approach to two different categories: server and client. This is because even though both architectures has many similarities (like message abstraction and connection interface), they also differ in many other areas (like data storage, event handling and type & quantity of threads).

3.1 Server

As entry point in the server, we instansiate Twitter library, ServerSocket and Twitter Channel (see 3.1.4). All server does after this is to wait for client connections and create a new client thread that communicates with that client, personally.

Creating a client thread is simple from server's side:

- Accept a connection from ServerSocket
- Prepare input and output streams
- Start the client thread with these gathered parameters (see 3.1.1)

3.1.1 Client Thread

A client thread is a thread which is responsible of all the communication going on between server and THAT client. Since we don't need a client-to-client communication, client threads are completely independent of each other.

Client thread is where the magic happens. Client thread listens the inputstream Server gave it while creating the socket connection with the client. Whenever something it written into inputstream, client picks it and processes it. We have a function with signature:

```
1 Optional<Command> processInput(String inputLine)
```

As the obvious signature suggests, this function takes the input and **maybe** generates a command (see 3.1.3) out of it. If client thread is indeed able to extract the command, it calls $perform()$ on command instance right away.

Before discussing commands, though, we should talk about handling messages by the server. After that, we will discuss implementation of those messages, namingly commands.

3.1.2 Messages

Server has defined two types of messages, with two distinct enum types for their tags: *Server.Tag* and *Client.Tag*.

A tag describes what that message wants to do. Every message in Follow-it must carry a valid tag along with it, and other fields are not obligatory, even though we have a convention of sending messages along with a data key. We will provide examples for each message type, so our convention will be obvious.

Client to Server Messages:

- **REGISTER**: Sent when client wants to register. A data with key "username" should be sent along with the message, indicating the username that client wants to register with.

  Server then determines whether registration is successful or not (it always is, as we don't have a predicate for registration-checking, unlike subscription. See REGISTER_ACCEPT for more.). An example:

```
{
  "tag": "REGISTER",
  "data": {
    "username": "Yigit"
  }
}
```

- **SUBSCRIBE**: Sent when client wants to subscribe to a channel, for a user. Server then determines whether user is eligible to subscribe to the provided username for the provided channel. An example:

```
{
  "tag": "SUBSCRIBE",
  "data": {
    "channel": "TWITTER",
    "username": "LondraGazete"
  }
}
```

Server to Client Messages:

- **REGISTER_ACCEPT**: Sent by the server if server decides that client can be registered. This is always the case, since we don't check eligibility of the registration. We could have rules like username being unique though.
- **SUBSCRIPTION_ACCEPT**: Sent by the server if server decides that client's subscription attempt is valid and accepted. Server decides whether a subscription attempt is valid by checking the Twitter username via Twitter4j API library. If user doesn't exist, a SUBSCRIPTION_REJECT message is sent instead.

- **SUBSCRIPTION_REJECT**: Sent by the server if server decides that client's subscription attempt is not valid. This case only happens if the provided username is wrong or a Twitter user with that username doesn't exist. This decision requires us to make a HTTP request to Twitter API, via Twitter4j library.

- **ERROR**: As we stated earlier, we have a pre-defined protocol message format, which includes having a valid json structure and having a valid key "tag" and a valid TAG value associated with it, which should be either "REGISTER" or "SUBSCRIBE". If client fails to send message with this format, server catches this error and immediately notifies the client about its mistake.

  Message format being invalid is not the only case for this ERROR message. In specific points such as subscription, server can throw SubscriptionException or ProtocolException. All of these exceptions are always caught and turnt into a valid "ERROR" message to notify client properly. Server also logs all of these exception occurences, just in case of debugging.

- **TWEETS**: This message denotes that server wants to send client tweets data, associated with an existing username. When client sees the message, it can be sure that there are tweet updates for the user with username associated with this message. Server doesn't send any empty updates to client if there are no tweet updates. We will discuss scheduling and pollig for tweet messages, as well as the rate limiting optimization in the following sections.

3.1.3 Commands

Command is an abstraction over what to be done with a valid client message. If you want to perform a command, it takes action based on the message data it has, which consists of a tag and a hashmap with keys being Strings and values being Objects.

For now, only commands server should handle are the messages with tags "REGISTER" and "SUBSCRIBE". Their behavior highly involves our task management in server, so we are discussing them in depth in the following sections.

3.1.4 Channels

Channels are essentially managers for the particular social media (only Twitter for our case). I mentioned that unlike client, we keep our data in-memory in server-side, and TwitterChannel is the object that keeps all the subscription data in its instance variable "subscriptions", which is just a mapping from users to list of clients.

This hashmap's structure may seem a bit strange at first, but it has a good reason behind it. We will discuss on it at section 3.1.5, since it involves the mindset evolved for rate limiting optimization.

TwitterChannel is instansiated only once through all the server application. This is because we need only one manager to provide support for us for one social media. After that, the only thing one could do with this instance is to call *subscribe*() on it. This is the major and only api this object exposes.

Once one invokes subscribe on twitter channel, first we find the Twitter user with that username. Then there are two options:

- If a subscription for that twitter user has been made before, we add that client to that user's watchers list
- Else, we create a new entry on *subscriptions* hashmap, and spawn a new TwitterAgent for that user. TwitterAgent will be discussed just below.

In order to watch updates on a certain user, we use Java's Timer class, along with TimerTask for scheduling tasks. The spawned TwitterAgents are subclasses of TimerTasks, and this inheritance allows us to invoke *run*() method on each TwitterAgent every 5 seconds (we define this interval ourselves, and this is subject to change, see 3.1.6).

3.1.5 Rate Limiting

3.1.6 Optimizations

(fetching once for each user)

3.2 Client

labelclient

4 Concepts

4.1 Subscriptions

- Actual rate limiting

References

[1] Twitter4j: A Java library for the Twitter API http://twitter4j.org/en/index.html

[2] Room: Persistence Library for Android https://developer.android.com/topic/libraries/architecture/room.html