CMPE436 Final Project - Follow-it

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

December 2017

Contents

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. 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:

```
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). What this TwitterAgent does it that it makes a request to twitter via Twitter4j[1] with the username it's built with, and gets the tweets associated with that user. Since we already performed the Twitter user check, from then on, we started using Twitter.User class to denote a real Twitter user, instead of a String denoting an unsafe Twitter username.

After we have the tweets, **we mark this point at tweet feed** for further fetchings not to duplicate tweet fetching. If we have > 0 amount of tweets in hand, we are ready to notify the watchers. Remember that we had two parameters for instansiating a TwitterAgent: a $Twitter.User$ and $List < Client >$. We call $notifyUpdate()$ on all of the agents with the tweets we fetched, and hence, we pushed the update to all clients.

3.1.5 Rate Limiting

One of the biggest concerns while making an application that consumes a 3rd party api is rate limiting. And Twitter API, which we consume has rate limiting individual to each of its endpoints. The endpoint we make use of most is the one allows us to fetch tweets, and it has a limit of 900 requests per 15 minutes.

Given that we make request to Twitter API for fetching tweets per 5 seconds for each individual Twitter user, we can only have 5 concurrent Twitter users being watched. This is a hard constraint in the sense that we have no chance of changing it, since it's a third party API. But we can try to do some optimizations on the subject, which we discuss in section 3.1.6.

3.1.6 Optimizations

Because of the hard constraint Twitter API dictates, we need find ways of allowing more clients getting concurrent updates. Hence, we did some optimizations regarding amount of requests being sent to client.

We build internal data structure of Twitter Channel so that is focuses on a single user. This way, even though there are multiple clients watching for a user, we only fetch the tweet data once every 5 seconds for that user, and broadcast this to all clients. This method has a drawback though: we are unable to synchronize newcoming clients with the latest tweets, since we are only allowed to broadcast new updates for the user. We can achieve this behavior **partly** by caching the tweets and then sending them as bulk to newcoming clients, but tweets in that data can be deleted or updated by mentions, so there is no guaranteed way to achieve best of the both worlds.

3.2 Client

Client architecture is different from the one we have in server, because of the nature of distributed applications, clients are the ones who needs to be proactive and tell the server what they want. In Android applications, this is done with buttons and inputs used by users that cause server to take some action.

Before discussing the actual behavior of the Android application, we should talk a little bit about async tasks.

3.2.1 Async Programming

Every IO in this Android application is done asynchronously. This is considered the best practice because making a thread wait until some action is taken means blocking the whole activity, and also means that we are blocking the user. Every action taken, including registering and subscribing has a common interface (we are not talking about the actual Java interface) that causes them to pass a **listener** along with the action being taken. This is called the callback paradigm, which Javascript makes heavy use of.

This listener should obey to the generic type provided when passing it to the task. This generic type dictates the return value of the listener, which constraints to the actual task that returns the value. Hence because of this constraint, we can say we want to get tweet data, and the signature of our data is $List < Tweet >$. This ensures the integrity of our program, which is hard to maintain when programming with callbacks / performing async programming.

We are using these listeners in 3 places throughout the application:

Registration: It being called is enough as evidence that registration is completed, no need for an actual result

```
TaskListener<Void>
```

Subscription: True if subscription succeeded, false otherwise

```
TaskListener<Boolean>
```

Receiving Data: This may seem a little bit odd, but the triple result actually explains much. It stands for: there is data of tweets coming for **Channel** channel for **String** user.

```
TaskListener<Triple<Channel, String, ArrayList<String>>
```

3.2.2 Application Behavior

When Android is booted, we don't open the socket connection right away, because it would be meaningless since user hasn't done anything yet, and there is nothing for server to consider a client as. User sees a welcome page with just an input box and a button, which is used to register user, and hence client to server. User enters his/her username, then we send server a REGISTER message. Then this activity does absolutely nothing, until a "REGISTER_ACCEPT" message comes from the server. When the message arrives, the callback we provided runs *startActivity*() and we go to dashboard.

In the dashboard, we have a list of subscriptions provided. Dashboard activity is responsible of calling the database task "GetSubscriptions" and asynchronously fill the subscription data with existing persistent storage. The reason behind deciding on using a persistent storage is that the data we pass with the activities turnt out to be too complex for serializing & deserializing and passing as intent extra. Because of this, we decided to use Android's Room library for persistent storage, and we've written our tasks, entities and relationships for our models.

Database tasks are defined in *tasks* package, and in order for one to use them, one should provide necessary parameters to the constructor and provide a callback, since this is a asynchronous task. This is indeed when DashboardActivity does on its *onStart*() call.

Android application has other behaviors as well, but they will be demonstrated in presentation.

4 Experimental Methodology

While working with a 3rd party API, it's always hard to write automated tests, since you need to mock all the behavior of the API, as it's not in your source code. Luckily, we make a very little use of Twitter API, and that's something we can experiment on easily.

We created a new Twitter user for the sake of testing this API, and our server's incremental data pushes. We are the owner of the account `https://twitter.com/fi_debug` and tested the tweet data pushing feature on-hand. Here is the experiment steps:

1. Open the application
2. Write a arbitrary username and click to "Register" button
3. See the empty page, and click to "Add Subscription" button
4. Type the name "wrong_user_name_203120582", click "Add Subscription"
5. Expect a failure popup message
6. Repeat steps from 3 to 4 with username "YigitOzkavci"

7. Expect a successful popup message
8. Repeat steps from 3 to 4 with username “fi_debug”
9. Expect a successful popup message
10. Click on subscription “YigitOzkavci” and expect to see latest tweets of the account @YigitOzkavci and go back in Android navigation
11. Click on subscription “fi_debug” and expect to see latest tweets of the account @fi_debug and go back in Android navigation
12. Post a tweet as user @fi_debug from Twitter UI
13. Click on subscription “fi_debug” and expect to see newly-posted tweet

5 Experimental Results

Experimental results were expected, but it also notified us about the constraint on api rate limiting & not being able to deliver old tweets to new subscribers (see 3.1.6 as a reminder).

6 Appendix

```java
package followit;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

import facebook4j.FacebookException;
import followit.channel.ChannelBuilder;
import twitter4j.Twitter;
import twitter4j.TwitterException;
import twitter4j.TwitterFactory;

/**
 * The entry point for this distributed application. This class is
    responsible of listening
 * clients and spawning client threads.
 *
 * @author yigitozkavci
 */
public class Server {
  /**
   * Message types from server to client.
   */
  public enum Tag {
    /**
```

```
     * Accept the registration of the client
     */
    REGISTER_ACCEPT,
    /**
     * Accept the subscription attempt of the client
     */
    SUBSCRIPTION_ACCEPT,
    /**
     * Reject the subscription attempt of the client
     */
    SUBSCRIPTION_REJECT,
    /**
     * An error has occurred on the server. Details will be given
    in .data.message node
     */
    ERROR, TWEETS
  }

  public static void main(String[] args) throws FacebookException,
    TwitterException, IOException {
    // TODO Auto-generated method stub

    Twitter twitter = new TwitterFactory().getInstance();
    ChannelBuilder channelBuilder = new ChannelBuilder(twitter);
    try (ServerSocket server = new ServerSocket(4444);) {
      while (true) {
        Socket clientSocket = server.accept();
        BufferedReader in = new BufferedReader(new
    InputStreamReader(clientSocket.getInputStream()));
        PrintWriter out = new PrintWriter(clientSocket.
    getOutputStream(), true);
        new Client(in, out, channelBuilder).start();
      }
    }
  }

}
```

../server/src/main/java/followit/Server.java

```
package followit;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

import com.google.gson.Gson;
import com.google.gson.JsonSyntaxException;

import followit.channel.ChannelBuilder;
import followit.command.Command;
import twitter4j.Status;
```

```
/**
 * Client is a big thread which is responsible of only one client.
 *
 * @author yigitozkavci
 */
public class Client extends Thread {
  public enum Tag {
    /**
     * Client wants to register
     */
    REGISTER,
    /**
     * Client wants to subscribe
     */
    SUBSCRIBE
  }

  /**
   * Reader for client's input stream
   */
  private BufferedReader in;

  /**
   * Writer for client's output stream
   */
  private PrintWriter out;

  /**
   * ChannelBuilder gives access to different channel instances,
    since
   * we only instansiate those channels once
   */
  private ChannelBuilder channelBuilder;

  /**
   * If this client is registered or not. Certain actions cannot be
     taken
   * while not registered
   */
  private boolean registered;

  /**
   * Even though provided username doesn't involve in business
    logic, we still take
   * value of this field while user registers
   */
  private String username;

  public Client(BufferedReader in, PrintWriter out, ChannelBuilder
    channelBuilder) {
    this.in = in;
    this.out = out;
    this.channelBuilder = channelBuilder;
    this.registered = false;
  }

  public void run() {
    String inputLine;
    try {
      while ((inputLine = in.readLine()) != null) {
        processInput(inputLine).ifPresent((command) -> command.
    perform(this, this.channelBuilder));
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  /**
   * Parser of the client message
   * @param inputLine Message to parse
   * @return Whether a command can be created from this message or
    not
   */
  private Optional<Command> processInput(String inputLine) {
    System.out.println("Got message: " + inputLine);
    Gson gson = new Gson();
    try {
      return Optional.of(gson.fromJson(inputLine, Command.class));
    } catch (JsonSyntaxException e) {
      return Optional.empty();
    }
  }

  /**
   * Called when there are data available to notify the client.
    This is the only interface
   * that is exposed in terms of sending data to client.
   *
   * @param user username that is related to those tweets
   * @param tweets List of tweets to notify client with
   */
  public void notifyUpdate(String user, List<Status> tweets) {
    sendMessage(Server.Tag.TWEETS, "tweets", tweets.stream().map(t
    -> t.getText()).collect(Collectors.toList()), "user", user);
  }

  /**
   * Helper for sending a message with 2 key-values
   *
   * @param tag Tag of the message
   * @param key1 Key of the first key-value
   * @param value1 Value of the first key-value
   * @param key2 Key of the second key-value
   * @param value2 Value of the second key-value
   */
  public void sendMessage(Server.Tag tag, String key1, Object
    value1, String key2, Object value2) {
    HashMap<String, Object> message = new HashMap<>();
    HashMap<String, Object> data = new HashMap<>();
    data.put(key1, value1);
    data.put(key2, value2);

    message.put("tag", tag);
    message.put("data", data);
    sendMessageData(message);
  }

  /**
   * Helper for sending a message with 1 key-value
   *
   * @param tag Tag of the message
   * @param key Key of the key-value
   * @param value Value of the key-value
   */
  public void sendMessage(Server.Tag tag, String key, Object value)
     {
    HashMap<String, Object> message = new HashMap<>();
    HashMap<String, Object> data = new HashMap<>();
    data.put(key, value);

    message.put("tag", tag);
    message.put("data", data);
    sendMessageData(message);
  }

  /**
   * Helper for sending a message with no data associated with it,
    for
   * example, a {@link followit.Server.Tag#REGISTER_ACCEPT} message
    .
   *
   * @param tag Tag of the message
   */
  public void sendMessage(Server.Tag tag) {
    HashMap<String, Object> message = new HashMap<>();
    message.put("tag", tag);
    sendMessageData(message);
  }

  /**
   * Private helper that is used while sending message
   *
   * @param data Data to send message of
   */
  private void sendMessageData(HashMap<String, Object> data) {
    String stringifiedData = new Gson().toJson(data);
    this.out.println(stringifiedData);
    System.out.println("Sent message: " + stringifiedData);
  }

  /**
   * Register the client
   *
   * @param username Username of the client who is registrating
   */
  public void register(String username) {
    this.registered = true;
    this.username = username;
  }
  /**
   * Getter of the {@link followit.Client#username}
   *
   * @return Username of this client
   */
  public String getUsername() {
    return this.username;
  }

  /**
   * Getter of the {@link followit.Client#registered}
   *
   * @return
   */
  public boolean isRegistered() {
    return registered;
  }
}
```

../server/src/main/java/followit/Client.java

```
package followit.channel;

import errors.SubscriptionException;
import followit.Client;

/**
 * A declaration of what should a channel look like.
 *
 * @author yigitozkavci
 */
public abstract class Channel {
  /**
   * Subscribe the client to the channel.
   *
   * @param client Client to subscribe
   * @param username Username that client is subscribing for
   * @return
   * @throws SubscriptionException Thrown when a complication
    happens while subscribing
   */
  abstract public boolean subscribe(Client client, String username)
     throws SubscriptionException;
  protected void clientCheck(Client client) throws
    SubscriptionException {
    if(!client.isRegistered()) {
      throw new SubscriptionException("Client is not registered");
    }
  }
}
```

../server/src/main/java/followit/channel/Channel.java

```
package followit.channel;

import twitter4j.Twitter;
```

```java
/**
 * Keeps the instance of twitter channel, since we only instansiate
     it once.
 *
 * @author yigitozkavci
 */
public class ChannelBuilder {
  private TwitterChannel twitterChan;

  public ChannelBuilder(Twitter twitter) {
    this.twitterChan = new TwitterChannel(twitter);
  }

  public TwitterChannel getTwitterChannel() {
    return this.twitterChan;
  }
}
```

../server/src/main/java/followit/channel/ChannelBuilder.java

```java
package followit.channel;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Optional;
import java.util.Timer;
import java.util.TimerTask;

import errors.SubscriptionException;
import followit.Client;
import twitter4j.Paging;
import twitter4j.Status;
import twitter4j.Twitter;
import twitter4j.TwitterException;
import twitter4j.User;

/**
 * Watcher of the tweets. This object is spawned for each
    individual Twitter user
 * we want to watch. Performs its action in per 5 seconds.
 *
 * @author yigitozkavci
 */
class TwitterAgent extends TimerTask {
  /**
   * Twitter user that this agent tracks
   */
  private User user;

  /**
   * Clients that this agent will notify occasionally
   */
  private List<Client> watchers;

  /**
   * Twitter API instance
   */
  private Twitter twitter;

  /**
   * Tweet ID for caching tweets
   */
  private Optional<Long> sinceId;

  public TwitterAgent(User user, ArrayList<Client> watchers,
    Twitter twitter) {
    this.user = user;
    this.watchers = watchers;
    this.twitter = twitter;
    this.sinceId = Optional.empty();
  }

  @Override
  public void run() {
    // TODO Auto-generated method stub
    try {
      Paging paging = this.sinceId.isPresent() ? new Paging(this.
    sinceId.get()) : new Paging();
      List<Status> tweets = twitter.getUserTimeline(user.getId(),
    paging);
      if (tweets.isEmpty())
        return;
      this.sinceId = Optional.of(tweets.get(0).getId());
      watchers.forEach((watcher) -> {
        watcher.notifyUpdate(user.getScreenName(), tweets);
      });
    } catch (TwitterException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }
}

/**
 * Twitter channel. See {@link followit.channel.Channel} for more
    on channels
 *
 * @author yigitozkavci
 */
public class TwitterChannel extends Channel {
  private Twitter twitter;
  private HashMap<User, List<Client>> subscriptions;

  public TwitterChannel(Twitter twitter) {
    this.twitter = twitter;
    subscriptions = new HashMap<>();
  }

  public boolean subscribe(Client client, String username) throws
    SubscriptionException {
    super.clientCheck(client);
    System.out.println("Subscribing to twitter user " + username);
    try {
      Optional<User> mb_user = findUser(username);
      if (mb_user.isPresent()) {
        User user = mb_user.get();
        if (!subscriptions.containsKey(user)) {
          ArrayList<Client> watcherClients = new ArrayList<>();
          subscriptions.put(user, watcherClients);
          new Timer().scheduleAtFixedRate(new TwitterAgent(user,
    watcherClients, twitter), 0, 5000);
        }
        subscriptions.get(user).add(client);
        return true;
      } else {
        return false;
      }
    } catch (TwitterException e) {
      throw new SubscriptionException(e);
    }
  }

  /**
   * Find Twitter user with the given username
   *
   * @param username Username of the Twitter user we are looking
    for
   * @return A user if we can find him/her, or an empty optional if
     we cannot
   * @throws TwitterException
   */
  private Optional<User> findUser(String username) throws
    TwitterException {
    return twitter.searchUsers(username, 5).stream().findFirst();
  }

  public static final String getName() {
    return "Twitter";
  }
}
```

../server/src/main/java/followit/channel/TwitterChannel.java

```
package followit.command;

import java.util.HashMap;
import java.util.Optional;

import errors.SubscriptionException;
import followit.Client;
import followit.Server;
import followit.channel.Channel;
import followit.channel.ChannelBuilder;

/**
 * An abstraction over what to do with a valid client message.
 * Details regarding commands are in the project report.
 *
 * @author yigitozkavci
 */
```

```
public class Command {
  /**
   * Tag of this message
   */
  private Client.Tag tag;

  /**
   * Data contained within this message
   */
  private HashMap<String, String> data;

  /**
   * Client that this command has came by
   */
  private transient Client client;

  /**
   * The ChannelBuilder instance for using Twitter channel
   */
  private transient ChannelBuilder channelBuilder;

  /**
   * These are the keys when we are performing search in incoming
    data (see {@link followit.command.Command#data})
   */
  private final String USERNAME_KEY = "username";
  private final String DATA_CHAN_KEY = "channel";
  private final String TWITTER_NAME = "Twitter";
  private final String CHAN_USER_KEY = "user";

  public Command() {}

  /**
   * Perform the action contained in the command
   * @param client Client to perform action for
   * @param channelBuilder see {@link followit.command.Command#
    channelBuilder}
   */
  public void perform(Client client, ChannelBuilder channelBuilder)
     {
    this.client = client;
    this.channelBuilder = channelBuilder;

    if(tag == null) sendErrToClient("Cannot resolve message tag.");

    switch(this.tag) {
    case REGISTER:
      findDataFromKey(Client.Tag.REGISTER, USERNAME_KEY)
        .ifPresent((username) -> {
          client.register(username);
          client.sendMessage(Server.Tag.REGISTER_ACCEPT);
        });
      break;
    case SUBSCRIBE:
      findSubscriptionChannel()
        .ifPresent((chan) ->
          findDataFromKey(Client.Tag.SUBSCRIBE, CHAN_USER_KEY)
          .ifPresent((user) ->
            {
              try {
                final Server.Tag msgTag = chan.subscribe(client,
  user) ? Server.Tag.SUBSCRIPTION_ACCEPT : Server.Tag.
  SUBSCRIPTION_REJECT;
                client.sendMessage(msgTag, "channel", chan.
  getClass().getName());
              } catch (SubscriptionException e) {
                System.out.println(e.getMessage());
                sendErrToClient(e.getMessage());
              }
            }
          )
      );
    break;
  default:
    sendErrToClient("Tag " + this.tag + " is not understood.");
  }
}

/**
 * Find data by key contained in this instance's data
 *
 * @param tag Tag of the message
 * @param key Key to look for value of
 * @return
 */
private Optional<String> findDataFromKey(Client.Tag tag, String
  key) {
  if(this.data == null || !this.data.containsKey(key)) {
    sendErrToClient("Message tag is " + tag + ", but data doesn't
   contain key " + key);
    return Optional.empty();
  } else {
    return Optional.of(this.data.get(key));
  }
}

/**
 * Find the subscription channel just by looking at internal
  state
 *
 * @return Channel if it's found, empty optional else
 */
private Optional<Channel> findSubscriptionChannel() {
  // Uuu, monadic bind, love it
  return findDataFromKey(Client.Tag.SUBSCRIBE, DATA_CHAN_KEY).
  flatMap((chanName) -> {
    switch(chanName) {
    case TWITTER_NAME:
      return Optional.of(channelBuilder.getTwitterChannel());
    default:
      sendErrToClient("Message tag is SUBSCRIBE, but value of key
   " + DATA_CHAN_KEY + " doesn't match any of the channels.");
      return Optional.empty();
    }
    });
  }

  private void sendErrToClient(String message) {
    client.sendMessage(Server.Tag.ERROR, "message", message);
  }
}
```

../server/src/main/java/followit/command/Command.java

```
package followit.Utils;

/**
 * A helper wrapper for data that contains two elements within
 * @author yigitozkavci
 *
 * @param <Fst> Type of the first data
 * @param <Snd> Type of the second data
 */
public class Tuple<Fst, Snd> {
  /**
   * First data
   */
  public Fst fst;

  /**
   * Second data
   */
  public Snd snd;

  public Tuple(Fst fst, Snd snd) {
    this.fst = fst;
    this.snd = snd;
  }
}
```

../server/src/main/java/followit/Utils/Tuple.java

```
package errors;

import facebook4j.FacebookException;
import twitter4j.TwitterException;

public class SubscriptionException extends Exception {
  private static final long serialVersionUID = 1L;
  private String message;

  public SubscriptionException(String details) {
    this.message = "A subscription error has occured: " + details;
  }

  public SubscriptionException(TwitterException e) {
    this.message = "Twitter exception occured: " + e.getMessage();
  }

  public SubscriptionException(FacebookException e) {
    this.message = "Facebook exception occured: " + e.getMessage();
  }

  public String getMessage() {
    return this.message;
  }
}
```

../server/src/main/java/errors/SubscriptionException.java

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