Teesside University
School of Computing
2011 - 2012

Final Year Project
BSc Computer Science

Social network library development

Thibaut Havel
L1247434

Supervisor: João F. Ferreira
Second reader: Erika Downs

Abstract

The main aim of this project is to develop a low-level library that is able to grab and store data from a social network. This library was designed for embedded devices and allows data to be stored and to be used to provide a simple service. To demonstrate the effectiveness of the final library, I created a demo service that interacts with a known social network (for example, Twitter).

Acknowledgements

I would like to thank my supervisor João Ferreira for his support all over the development of my project.

Contents

1 Introduction

Our society tends to use more and more social networks (for instance, Twitter or Facebook). At the same time, we are increasingly dependent on the use of embedded devices on a day-to-day basis (for instance, home automation). The goal of this project is to develop a software platform that allows an effective and efficient communication between embedded devices and social networks.

The main aim is to develop a low-level library that is able to grab and store data from a social network. This library was designed for embedded devices and allows data to be stored and to be used to provide a simple service. To demonstrate the effectiveness of the final library, I created a demo service that interacts with a known social network: Twitter.

One of my personal objectives was to improve my knowledge in low-level development and become familiar with the C language. Also I was looking forward to improving my software development skill while working with a very specific hardware.

Firstly, this report will present the way I started my initial research and how I designed the library according to my functional choices. Secondly, it will present how I've implemented these functionalities, and what I did to test my library. Finally, this report will end by an evaluation of the final product regarding the design choices, the way I've implemented them, and his reliability.

2 Methodology

2.1 The initial project scheduling

In my project specification, I set my schedule as following:

- January: Research (2 weeks), design (2 weeks).
- Febuary: Design (1 week), Implementation (3 weeks).
- March: Implementation (3 weeks), Testing (1 week).
- April: Evaluation and report compilation (2 weeks).

My supervisor João and I met every week or every two weeks to discuss about the planning and progresses of my project.

2.2 Research plan

My initial approach was to become familiar with the embedded device technology. I had to find an adapted, a small and a simple operating system to work with, thereby I chose **FreeRTOS**[1] supported from my supervisor.

My supervisor had already used this system and he had developed applications before. He provided me one of his own device running with FreeRTOS. So, thanks to João and his knowledge about this operating system, I had a platform in addition to a massive support from him and the online community to develop my library. As I didn't have any knowledge about FreeRTOS, I read several articles which deals with how it works, and how tasks are scheduled in an real-time way.

As a consequence of this chose and because one the goals of this project is to gain knowledge about low-level development, the library has been built using the C language.

Then, I defined every functionality of the final applications. Basically, the library's features are simple: it should allow a user to receive and send text from and to a social network. For instance

[1] FreeRTOS is a light-weight Real-Time Operating System.

a *message on the wall* in **Facebook** or a *tweet* in **Twitter**. Facebook and Twitter are both well known social networks and after some research about them, I choose to build my library suitable for Twitter because of the solid support for **OAuth** which is a secured protocol to access data. Once again, this choice was supported by João.

At this point, I had to defined how to receive and to send tweets, so I've started by looking for any existing solutions. In the next chapter, I will discuss why I chose to build my own library from scratch only using OAuth.

After this key decision, I've learned how OAuth works and how to register an application on Twitter it in order to access to the data.

As I was now aware about what the tools I will use and the way to use them, I designed the library according to the features I wanted to implement.

The next step of the development was to implement the abstract structure of the library. At this stage, I faced lot of issues concerning the use of OAuth, the C language and its requirements. During my previous years of studies I learnt the basics of this language but to build the library I read a lot of articles, books and tutorials along the implementation.

Once I finished the draft version of the library, I tested every functionality by receiving and sending tweets over my own account and I improved the reliability according to the results of these tests.

2.3 The support tools

To help myself into the research and the development of the library, I used some additional appropriate tools:

- A diary: to keep every relevant informations but also as a memory trail of the development chronology.
- Github[2]: to back up the source code and share my progress with my supervisor.
- FreeRTOS POSIX[3] simulator: to develop and to test my library without any embedded device.

[2]Github is an online project hosting.
[3]Standards to maintaining compatibility between UNIX operating systems.

3 Research

3.1 Operating System: FreeRTOS

3.1.1 Overview

FreeRTOS is a light-weight and real-time[1] operating system suitable for embedded devices. It is distributed under the GPL[2] even then with certain exceptions: a developer is required to maintain the kernel as open source, whereas he could remain closed source his applications.

Figure 3.1: FreeRTOS logo

The FreeRTOS source code is tiny and simple, thus it is really easy to port. It is mostly written in C and assembly. The kernel has been ported to around thirty micro-controllers.

The system kernel consists of three common main files (*list.c*, *queue.c* and *tasks.c*) and at least one specific to a particular micro-controller (*port.c*).

3.1.2 Real-Time System

One of the main feature provided by FreeRTOS is the way the system performs threads. All tasks are scheduled depending on their priority and sorted according to a round-robin algorithm.

Every files which manage tasks should include the FreeRTOS header *task.h* in order to use the appropriate functions.

A task can be defined as follows and should be of endless loop style:

[1] A system that should fit to time constraints.
[2] General public licence is used for free and open source software.

```c
void vMyTask (void * pvParameters)
{
    for(;;)
    {
        /* Task code content */
    }
}
```

The FreeRTOS thread API[3] enables to manage tasks. For instance, to create or delete them:

```c
xTaskHandle xHandle;

xTaskCreate( vMyTask, "NAME", STACK_SIZE, &parameters, tskIDLE_PRIORITY, &xHandle );
vTaskDelete( xHandle );
```

The `xTaskHandle` element allows to keep a reference to the task that can be then used by other functions

Depending on its presence or its position in the queue, a task can take different states:

- *Ready*: the task is ready to run but it is not currently executing because another task is running. From this state, it might become *Suspended* or *Running*.
- *Running*: the task is currently executing. From this state, it might become *Blocked*, *Ready* or *Suspended*.
- *Blocked*: the task is not available for scheduling until a defined delay period. From this state, it might become *Suspended* or *Ready*.
- *Suspend*: the task is not available for scheduling without any timeout. From this state, it might become *Ready*.

These states can be changed using the appropriate functions from the API, for instance:

```c
/* To suspend a specific task */
vTaskSuspend( xHandle );

/* To resume a suspended task */
vTaskResume( xHandle );

/* To raise a task priority */
vTaskPrioritySet( xHandle, tskIDLE_PRIORITY + 1 );
```

[3]An Application Programming Interface is a set of routines or functions given by a library.

3.1.3 POSIX simulator

To develop and test demo tasks in order to get knowledge about FreeRTOS, I worked with a simulator. This is a ported version of the operating system that enables an embedded application to be simulated on a computer running another system.

As I'm a Linux user, so I chose the POSIX simulator. This simulator consist of a set of files representing the kernel and some demo applications which can all be compiled using GCC[4].

My first attempts to use the simulator were complicated by some undefined references issues at the compilation which came from my operating system. I did managed to fix the problem thanks to my supervisor João. At this point, I have been able to compile the simulator and to play with the demo applications and even create my own one.

3.2 Twitter authentication: OAuth protocol

3.2.1 Overview

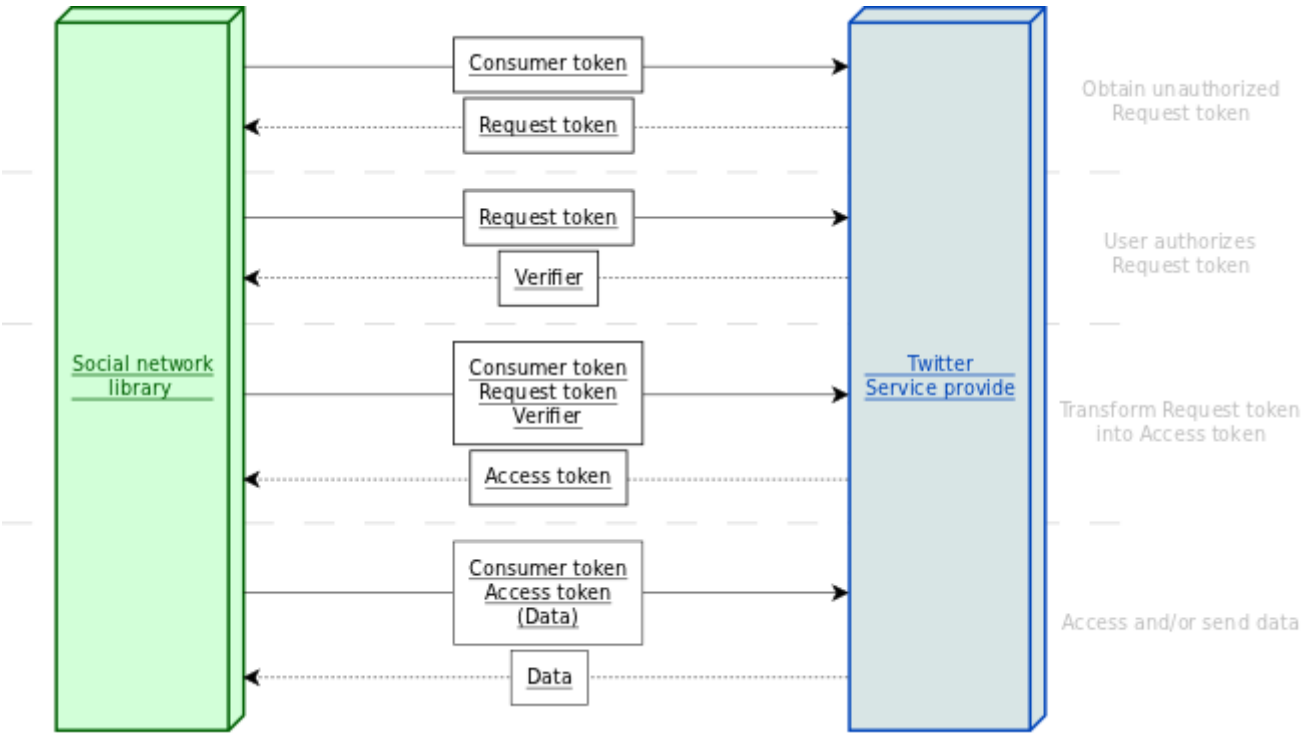

Figure 3.2: OAuth protocol: communication process

[4]GNU Compiler Collection is a compiler system which can compile several programming languages including C and C++.

OAuth is the secured protocol used by Twitter to enable developers to access and share data from a third-party application. The authentication process is based on the public and private key system. Each step of the process (from the authentication to the data access) is represented by a granted token. The authentication consists three different tokens:

- The *Consumer token* is provided by the Twitter developers website at the application referencing.
- The *Request token* is given by the service provider as a temporary token.
- The *Access token* is given as a proof of authenticity that must be use to access and share data.

3.2.2 Existing library in C

The Twitter developers documentation provides several implementations of OAuth protocol in many languages including C++, Java or Python, but for some reason not in C. However, I found a tiny C library under GPL, written by Robin Gareus and called *liboauth*. His repository gives few examples, thus it was easy to get to know how to use it.

Basically, this library provides a set of functions to encode parameters according to OAuth specifications and to implement the protocol using HTTP requests (*GET* and *POST* methods). For instance, these functions could be used useful to perform the granting access rights: `oauth_sign_url2()` , `oauth_http_get()` , `oauth_http_post()` . However, the whole Twitter authentication process is really complex and could be packaged as only one main routine.

3.2.3 Register an application on Twitter

The first step is the account registration: I created a personal account which is the same no matter I use Twitter as a simple user sending tweets and following profiles, or as a developer creating applications using the API features.

Then, I registered a new application in the Twitter developers website to get the first required token to use the Twitter API via OAuth. The main informations are the name of the application and the access level: read and/or write which means that the application will be allowed to receive and/or send tweets.

At the end of the process, Twitter gives the *Consumer token* (a set of a public and a private key) and also the useful URL to send requests. These URL are static so they might be stored into the main file that gather the whole authentication.

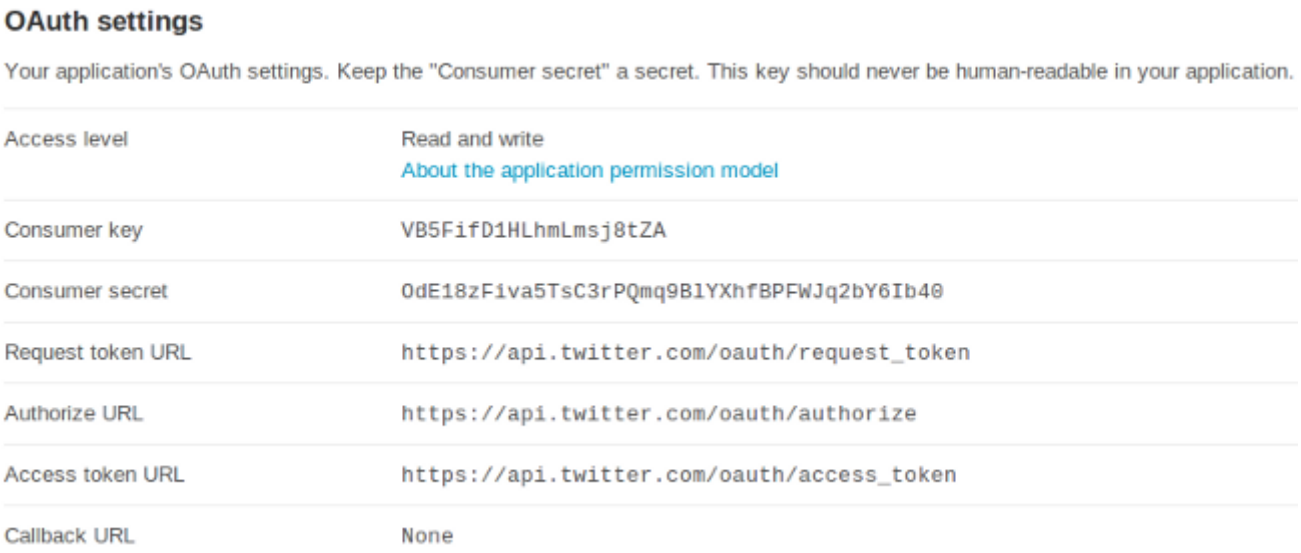

Figure 3.3: Application informations from Twitter developers website

3.2.4 Required libraries

The liboauth required two libraries: **OpenSSL** and **libcurl**, thereby to build and library that could communicate with Twitter from an embedded device, these library must be ported and included.

OpenSSL is a open source library implementing the *Secure Sockets Layer* and the *Transport Layer Security* protocols and including several encryption algorithms. Libcurl is a library that perform several kind of requests including FTP, IMAP, HTTP and HTTPS.

These two libraries are free open source and portable. As they could be compiled to be used with an embedded device, they are suitable for my project.

I downloaded both of them and I checked the source code. I even tried to quickly port and compile them to the FreeRTOS POSIX simulator, but without success. The investigation about how to fix the compilation issues should be put aside until the library fully works on the simulator.

4 Design of the library

4.1 Functional design

4.1.1 Interactions with Twitter

Basically, the final library should enables a developer:

- To authenticate its application to Twitter.
- To send tweets to Twitter.
- To receive tweets from Twitter.

And these features have to work whatever the operating system and the hardware architecture.

I chose not implement any storing properties because it is more flexible to let the developer chose the way he wants to store tweets.

4.1.2 A user-friendly layer

This library has to be a user-friendly layer. The developer does not have to know how OAuth works to build its own application to access to Twitter. He just have to give the basic informations about the registered application (the public and secret key provided by Twitter) and informations about its Twitter account (the login and password).

The first main functionality is the authentication process which gathers all OAuth operations and returns an authentication entity (for instance, typed as a C structure) which could be use by the developer in a further step to send and to receive tweets. This entity contains every required informations needed to allow OAuth to connect to Twitter.

The send functionality is one of the two behaviours which could use the authentication entity in order to send a tweet on a Twitter profile.

Finally, the receive functionality use the authentication entity as well in order to receive tweets from a Twitter account's **timeline**[1]. This functionality include parsing functions which enables to return a set of tweets entities.

Each functionality is represented by a single function. Nevertheless, the content of a functionality could be split into several operation each represented by another function.

4.2 Implementation design

4.2.1 Functions implementations

Every steps of the authentication process are gather into the main authentication function. Basically each use of the OAuth library for a specific stage of the synchronisation is surrounded by a set of operations, for this very reason each stage is defined into a distinctive function. As explained above in the chapter Research, to authenticate an application to Twitter and then be able to access to the timeline or to send a tweet is simple but it requires few steps:

- 1- Consumer token: it is the first token given by the Twitter service provider.
- 2- Request token: it requests the token needed to obtain user authorization.
- 3- Verifier: it uses the request token in order to get the PIN code (or verifier).
- 4- Access token: it uses the verifier to request the final token which will be use to send or receive tweets.

This main function gives to the user an authentication entity, that is the one he provides to the behaviour functions (send and receive). This entity is typed as a C structure.

As every behaviour functions, the send function need the access token to be able to send a text message over Twitter. The main function retrieves the needed informations from the authentication entity which are given as parameters in sub-functions[2]. Whatever the sub-function, no field of the authentication entity is directly used, only the main function holds this responsibility.

To get the tweets from the user's timeline, a request is firstly send to the Twitter service provider. The result is a XML content which is parsed by some sub-functions. These parsing functions determine how many tweets are there in the timeline, and for each of them a new structure is created. Thus, the main function gives to the developer a collection of tweets each represented by a C structure which contains the most significant informations about it (e.g. the date, the text content).

[1]The timeline is the part of a Twitter profile which contains all tweets sent by a user.
[2]A sub-function is used by the main function in a distributed way to perform the goal.

4.2.2 Functional diagram of the library

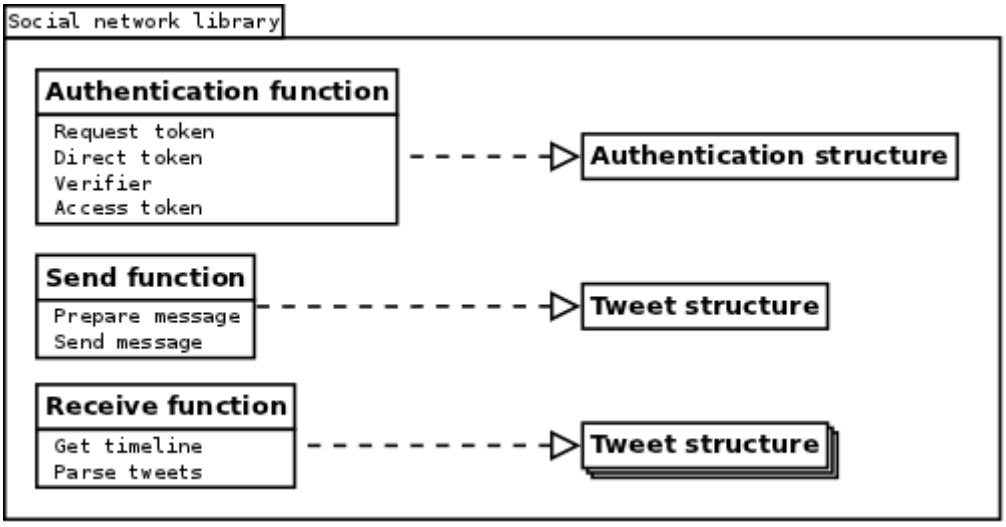

Figure 4.1: Functional diagram representing the implemented design

5 Implementation

5.1 Required libraries

To be able to write the library, I first decided to implement the dependencies: *liboauth*, *OpenSSL* and *libcurl*.

I started by downloading liboauth. The content of the library is really simple because it consists of eight C source code and header files. I managed to compile the FreeRTOS simulator including the OAuth library: in order to do so, I first edited the simulator *makefile*[1] by hand.

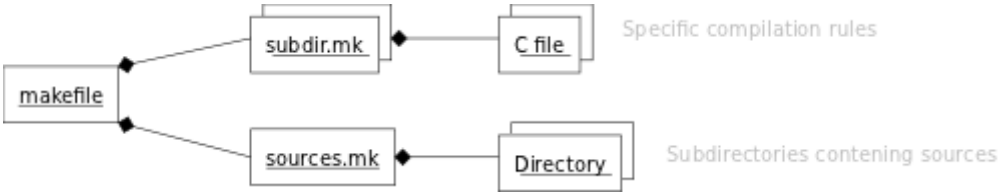

Figure 5.1: FreeRTOS POSIX simulator, makefile architecture

As shown in the figure above, the compilation instructions are gather into a *Debug* folder where rules are split into sub-files. Thus, I added a *subdir.mk* listing the C source codes files of the library and I referenced the name of the sub-directory into *sources.mk*.

To compile liboauth the system requires *OpenSSL* and *libcurl*. Therefore I added some GCC options that references these libraries into the makefile: *-lssl -lcrypto -lcurl*. Nevertheless, the libraries used are those installed on my own system, not those specific to simulated FreeRTOS system, which is not suitable for an embedded environment.

The best possibility could be to port OpenSSL and libcurl to FreeRTOS in order to make liboauth totally independent of my system. These libraries are massive and require many research to be ported, moreover that was not an immediate priority under the plan, so I chose to set aside this issue until my Twitter library works.

[1] A makefile is a file containing compilation instructions.

5.2 Authentication process

Prior to determining the authentication process, I defined the required informations to perform it as follows:

- The *Consumer token* allows a specific application to access data from Twitter.
- The user login and password prove this user allows the application to access to his timeline.
- The URL to send requests are static informations given by Twitter.

I chose to defined the *Consumer token* and the user informations as parameters of the main authentication function. In this way, the developer will be able to chose which Twitter application and which Twitter account would used by the library to access data. The URL are defined as static fields into the library.

Then, the first step of the authentication is the use of the *Consumer token* given by Twitter to get the *Request token*. Whatever the request, the URL and sometimes the parameters are generated into a specific function according to the Twitter specifications. For this first stage:

```
1 twitter_request_token_url(consumer_key, consumer_secret, &request_token_url);
2 twitter_request_token(request_token_url, &request_token_key, &request_token_secret, &
    callback);
```

Basically, the output parameters begin by the `&` symbol because I assign the result values to the specified variables, so I need to know its references. For instance, the first function use the `consumer_key` and the `consumer_secret` to get the generated `request_token_url`. And the second one use the `request_token_url` value to return the `request_token_key`, the `request_token_secret` and the `callback`.

The *Verifier* step is probably the most important but also the more complex. As usual, the URL is generated by a function and then the request is sent by another one:

```
1 twitter_direct_token_url(request_token_key, &direct_token_url);
2 twitter_verifier(direct_token_url, request_token_key, &verifier);
```

But the `twitter_verifier` function is divided into several steps:

- Send a HTTP GET request using OAuth: `oauth_http_get()`.
- Parse the HTML result of the request and get the authenticity code using `twitter_direct_token_authenticity()`.
- Regenerate another URL and some parameters using the authenticity code, and the user account informations through `twitter_direct_token_url2()`.
- Send a HTTP POST request using the OAuth with the new URL and the generated parameters: `oauth_http_post()`.

- Parse the HTML result of the request and get the final PIN code (or verifier) with `twitter_direct_token_pin()` .

The final step of the authentication is to obtaining the *Access token*:

```
1 twitter_access_token_url(consumer_key, consumer_secret, request_token_key,
    request_token_secret, verifier, &access_token_url);
2 twitter_access_token(access_token_url, &access_token_key, &access_token_secret, &
    access_token_user_name, &access_token_user_id);
```

These functions use the two first tokens (*Consumer* and *Request*) and the *Verifier* to obtain the final token which could be use as a proof of user authenticity and user allowance about data accessing from his timeline.

Finally, the main function gives to the developer a C entity (typed as a structure) called `twitterAuthEntity` and defined with the following fields:

- The user identifier and screen name.
- The *Consumer token* key and secret.
- The *Access token* key and secret.

5.3 Send a tweet

As all operations allowed by the final library, the main function require the authentication entity. Thus, this entity is one of the parameter and the fields of the structure which store the details of the authentication are used inside the function in order to perform the send operation.

The second parameter is the content of the tweet, indeed. Twitter allows to send text-based posts of up 140 characters. Other extra-details might be attached to the tweet (e.g. the geo-localisation) but this library only supports the text content.

To perform the submission, the library first generate the URL and the parameters of the HTTP POST request, and then send the tweet:

```
1 twitter_tweet_url(tweet_content, consumer_key, consumer_secret, access_token_key,
    access_token_secret, &tweet_url, &tweet_param);
2 twitter_tweet(tweet_url, tweet_param, &post_result);
```

If the last function succeed, it gives the XML value of the submitted tweet. This content is parsed and returned as a `tweet` which is typed as a structure with the following fields:

- The tweet identifier.
- The creation date.

- The user screen name.
- The tweet text content.

5.4 Receive a tweet

The main function gets the entire timeline which means all tweets are received using one routine:

```
1 twitter_timeline_user(consumer_key, consumer_secret, access_token_key,
    access_token_secret, access_token_user_name, &timeline_user);
```

The received content is in XML format and the data structure is defined as follows:

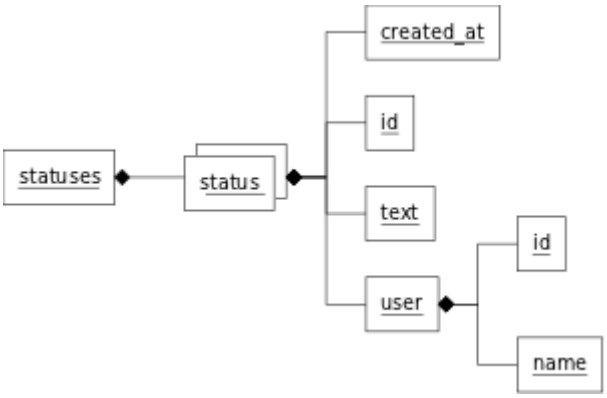

Figure 5.2: Tweets XML data structure

In the figure above, I chose to display only useful elements of the structure: those I use to fulfil `tweet` entities.

To parse XML data I written some tool functions to count, give a value or all values from specific elements:

```
1 char * xml_parser_get (const char * xml_content, const char * element_key)
2 int xml_parser_count (const char * xml_content, const char * element_key)
3 void xml_parser_getall (const char * xml_content, const char * element_key, char *
    element_value[])
```

Using these functions, the main one enables the developer to get an array of `tweet` .

6 Testing of the library

7 Evaluation of the project

7.1 Project aim and personal goal

The main aim of this project was to develop library that is able to grab data from a social network, Twitter. This part of the goal has been achieved, as a mater of fact my library enables a developer to get a user timeline and send tweets after granted access using the appropriate authentication protocol.

Regarding the portability, the library might be used by any device or any system. Nevertheless, two dependencies are required: OpenSSL and libcurl which are both also portable.

This library has also been build in order to be reused or integrated to another project. Every piece of the project has been documented and this report has also been made to provide a support as regards of the way it works. The source code and the report are freely available on Github.

7.2 Compliance with schedule

7.3 Possible improvements

8 Conclusion

Bibliography

[1] Robin Gareus. liboauth. http://liboauth.sourceforge.net.

[2] GitHub. Github, social coding. https://github.com.

[3] Real Time Engineers ltd. Freertos website. http://www.freertos.org.

[4] The OpenSSL Project. Openssl. http://www.openssl.org.

[5] Daniel Stenberg. libcurl. http://curl.haxx.se.

[6] Twitter. Developers documentation. https://dev.twitter.com/docs.