Final Project W205

Loris D'Acunto

Purpose of the Application

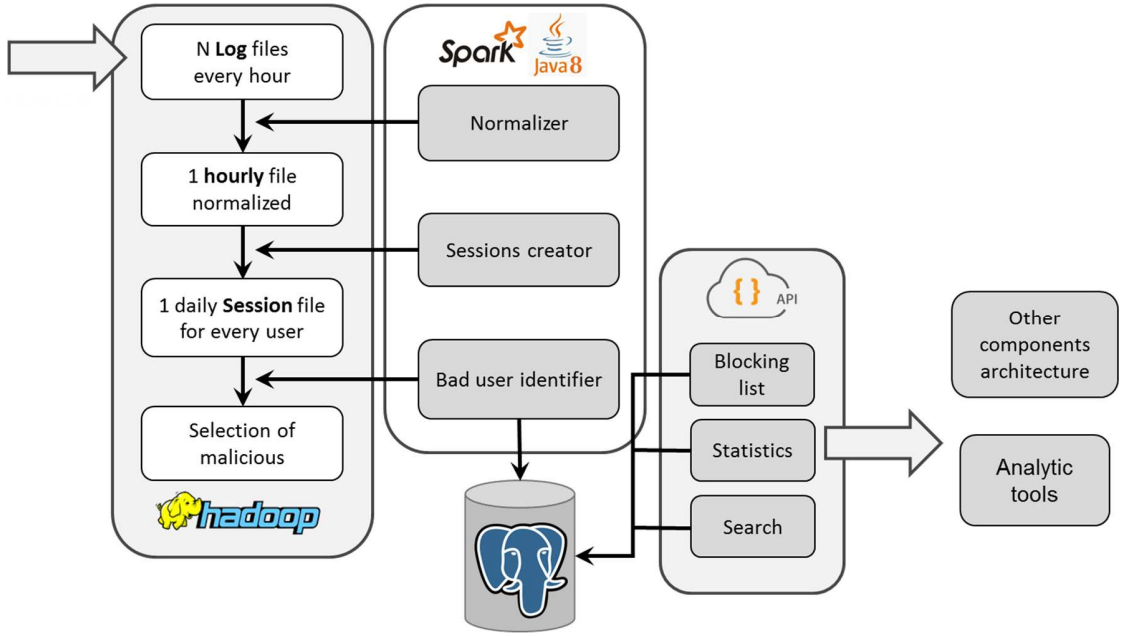
Purpose of this application is to help one of the customers of my company to identify malicious users in order to block them.
My company sells, as a service, the capability to identify a TV show, a movie, a commercial using the audio collected from the microphone of a mobile phone that is close to the TV.
The service is used by several companies like for example a reward app that gives points to the users according to how much TV and commercials they watch. The points are then converted into gift cards.
Only in US this specific customer has 60,000 monthly active users that generate 3.7 billion events, almost 1.4 Terabyte of data in our system.
There are several users that try to cheat the system in order to earn more money. For example, they place several devices in front of a TV or a laptop and they play all the day the same content.
The following pictures show a couple of examples of setups used by malicious users to collect more points.

Because of them, the customer is paying multiple times the same user (that uses multiple accounts to access the app) and, moreover, the service we offer is more expensive because every single user that has several devices in front of a TV, generates a lot of traffic to our servers, day and night.
The goal of my project is to analyze the data we collect in order to identify these malicious users and create a serving layer that can be used by a blacklist manager to block those specific devices.
There are several ways to identify them:

- Looking for users that are watching the same content all the day, like a specific commercial or TV show;
- Searching for people that are watching 24 hours of TV;
- Looking for users which mobile phones are connected to our servers with the same IP address and are watching the same content.

Overview of the Application – Production design

In the following picture I present the final design of the architecture that I will be installed in production.
There is a batch layer composed by Java Spark over Hadoop file system that processes every hour the log files, normalizes the data and creates sessions ordered by time.
At the end of the chain there is a malicious user identifier that populates a Postgres database with the typology of the user.
A JAVA REST API service layer gives access to the data. The goal is to connect the service layer to a blacklist manager and to some dashboards that will allow the customer to control the status of the system.



Limitations of the prototype

For the prototype I simplified the design, processing 1 week of data of a random sample of 12,000 devices, instead of all the 60,000 devices. The Java application works over Hadoop, and it is designed to use all the cores of the server (when it's possible) to speed up the computation. The service layer is written in Python, and the database has only one table. The system can process the data on a daily basis.

Dependencies and Requirements

The software requires the following libraries/packages in order to work correctly:
For the java application:

- Hadoop
- Postgres
- Java (at least 1.7)

For the Rest API:

- Python 2.7
- flask
- flask-restful
- sqlalchemy
- psycopg2

File Structure

```
/source
|   app.py
|   Architecture.pdf
|   script_project.sql
└──W205
    |   pom.xml
    |
    └──src
        └──main
            └──java
                └──com
                    └──w205
                        |   Configuration.java
                        |   MainExecutor.java
                        |
                        ├──malicious
                        |       Content.java
                        |       ContentLine.java
                        |       MaliciousAnalizer.java
                        |       MaliciousAnalizerMain.java
                        |
                        ├──preprocess
                        |       FilterEvents.java
                        |       Normalizer.java
                        |       PrivacyProtector.java
                        |
                        ├──sessions
                        |       LineUser.java
                        |       OrderByTimeUserFiles.java
                        |       SingleFileOrderer.java
                        |       SplitterByUser.java
                        |
                        └──utils
                                FileAccessFactory.java
                                Reader.java
                                ReadFileLineByLine.java
                                ReadHadoopFileLineByLine.java
                                Writer.java
                                WriterFile.java
                                WriterHadoopFile.java
```

Java Application

To create the Eclipse project to compile the software, it is necessary to have installed Maven in the local pc.
From the command line, run the following command in the directory /source/W205:

```
mvn eclipse:eclipse
```

Maven will install all the dependencies and from the eclipse environment it will be possible to compile and create a runnable jar file (referred in this instruction as *w205_hadoop.jar*).
Be sure to compile the classes using the same version of the JVM installed in the server (in my case 1.7).
The java application *w205_hadoop* is designed to work in both a Linux/Windows file system and in HDFS.
In fact, all the operations performed by the software to the file system like reading and writing from and into a file, creating a directory, listing all the files in a directory, are centralized and managed in the same way.
The only thing required to change in order to switch from one file system to another is to change the value of the static variable HADOOP in the file *com.w205. Configuration*.

List of classes:

Package **util**:

- **FileAccessFactory**: Manager of the file system that allows to easily test the code in a Windows/Linux file system and in a HDFS;
- **Reader**: Abstract class that defines the methods necessary to read a text file, line by line;
- **ReadFileLineByLine**: implementation of the Reader for Windows/Linux;
- **ReadHadoopFileLineByLine**: implementation of the Reader for HDFS;
- **Writer**: Abstract class that defines the methods necessary to write lines in a text file;
- **WriterFile**: implementation of the Writer for Windows/Linux;
- **WriterHadoopFile**: implementation of the Writer for HDFS;

The package **preprocess** filters, normalizes and obfuscates the sensitive information of the user in the log files:

- **FilterEvents**: class that filters the interesting events;
- **Normalizer**: class that normalizes the data, keeping only 8 fields from the original 45;
- **PrivacyProtector**: class that hides the sensitive information of the user like IP address and advertisement id.

The package **sessions** splits the daily normalized files in several files, one for user:

- **LineUser**: class that represents a line of content. Used to order the content inside the file;
- **OrderByTimeUserFiles**: main class that orders the content of all the files by time;
- **SingleFileOrderer**: single worker that orders by time the content inside each file;
- **SplitterByUser**: main class that splits all the files in several files, one for each user.

The package **malicious** analyzes the user activity and decides if he is malicious or not and populates the table with the related information:

- **Content**: class that represents the content watched by the user;
- **ContentLine**: class that represents single line of content;
- **MaliciousAnalizer**: single worker that analyzes one user;
- **MaliciousAnalizerMain**: main class of the malicious analyzer;

main package:

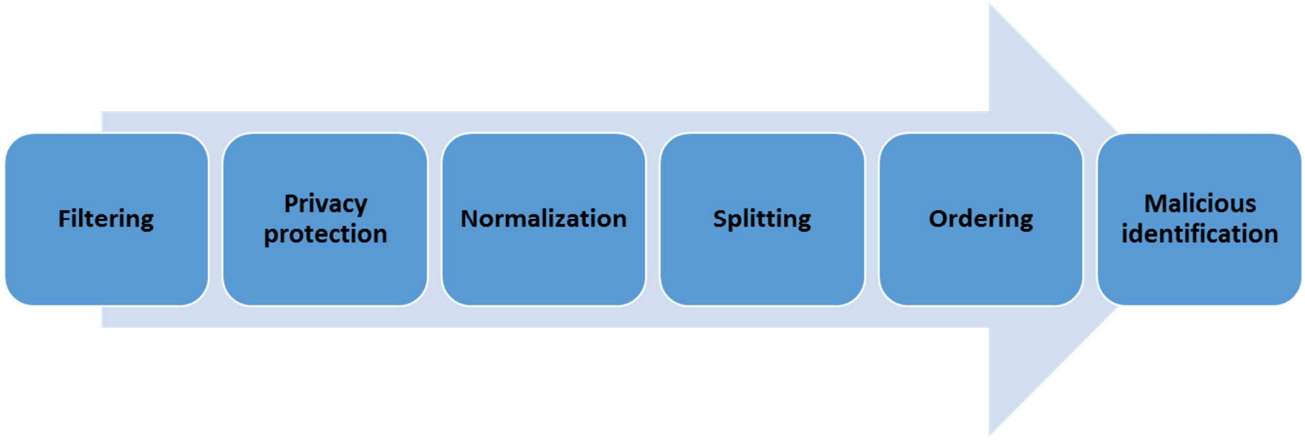
- **Configuration**: configuration class;
- **MainExecutor**: main class that gives access to all the functionalities.

Following the help visualized when the application is run from the command line without arguments:

```
java -jar w205_hadoop.jar
MainExecutor syntax: command souce {dest}
command list
-filter     :filter from the files only the events of match and related to the app of interest
-privacy    :replace sensitive information on the files to protect the privacy of the users
-norm       :normalize the files in order to analyze them
-split      :split the data in different files, one for each user
-order      :order temporarly the data into the files
-malicious  :find the malicious users
```

Steps

Explanation of all the main steps and the way in which the fields and files are shaped across each of them:



1) Filtering: step that filters from the files all the unwanted events like:

1. NOMATCHES: the sound collected from the microphone is non related to a TV content;
2. APPS that are not the target one (everything except the apps *perk* and *discoball*);
3. Errors.

Example of one line of log:

```
1457395200019 LogRequestMoviesSeries d9e6f8cd0e67b46a 566644215-iPciidpH1edC9I0d
1 - - 1457395200019 0.0 0.0 3 null null - 0 76.126.21.242 1457395192360 -8 en US
android 4.4.2 LGLS740 - 56 0.5 19 x5 x5_spr_us com.discoball.aphone 3eb63e10-
a47f-4335-9ece-638902c35f03 false 100.0 true 100 WnlZQUFCSWtEUnNGUlE9PQ== true
ZFU1ZkVCMU9YRnBVQVVsVEYyeGJTMFk9
{"image":"http:\/\/media.axwave.com\/img\/ads\/24258.jpg","iabc":"IAB3, IAB2,
IAB13","id":"24258","type":"ad","hashes":"19","category":"Insurance","brand":"Nat
ionwide Insurance","url":"http:\/\/nationwide.com","iabsc":"IAB13-6"} 3.3 1.0.1
AD H Nationwide Insurance 24258
```

2) Privacy protection: step necessary to obfuscate the information related to the user that can uniquely identify him, like

- IP address: In that case it's randomly generated a new one
- Advertisement id: generated a progressive number uniquely assigned to each user.

The first two steps are applied to the data before sharing with the teacher.

They can be downloaded from this Dropbox link: https://www.dropbox.com/sh/cmvsa16wk96tnp9/AACVpKcHkHCwuni-9AEsqrs4a?dl=0

Next is presented how the previous log is changed:

```
1457395200019 LogRequestMoviesSeries 0 566644215-iPciidpH1edC9I0d 1 - -
1457395200019 0.0 0.0 3 null null - 0 67.120.40.62 1457395192360 -8 en US android
4.4.2 LGLS740 - 56 0.5 19 x5 x5_spr_us com.discoball.aphone 0 false 100.0 true
100 WnlZQUFCSWtEUnNGUlE9PQ== true ZFU1ZkVCMU9YRnBVQVVsVEYyeGJTMFk9
{"image":"http:\/\/media.axwave.com\/img\/ads\/24258.jpg","iabc":"IAB3, IAB2,
IAB13","id":"24258","type":"ad","hashes":"19","category":"Insurance","brand":"Nat
ionwide Insurance","url":"http:\/\/nationwide.com","iabsc":"IAB13-6"} 3.3 1.0.1
AD H Nationwide Insurance 24258
```

3) Normalization: normalization step. All the unnecessary fields are removed, the JSON structure is parsed and it's created a tab separated file with the following structure:

- TIMESTAMP of the event
- ID user (generated in the previous step)
- IP address (generated in the previous step)
- Latitude
- Longitude
- Type of match:
    - AD for advertisement
    - LIVETV for Live TV
    - DVR for Digital Video Recorded
    - MOVIE for movies
    - TVSERIE for TVSHOWs
- Data 1, according to the type of the match, it has different values:
    - Brand name if it's an AD
    - Channel name if it's LIVETV or DVR
    - Title of the Movie
    - Title of the TVSHOW
- Data 2, like the previous one:
    - Empty for AD and Movie
    - Program name for LIVETV or DVR
    - Season number and Episode number for TVSHOW

In the next line is presented how the previous log is modified during this step:

```
1457395200019 0 67.120.40.62 0.0 0.0 AD Nationwide Insurance
```

There are generated 1 file for each day.

4) Splitting: the files are split in several different files by id user. The name of the files is the id of the user.

The structure of the files remains the same as the previous step.

From:

```
1457395200019 0 67.120.40.62 0.0 0.0 AD Nationwide Insurance
1457395200026 1 36.135.218.53 0.0 0.0 LIVETV Headline News Forensic Files
1457395200090 2 228.239.246.222 29.8501077 -90.1087257 LIVETV E! Entertainment
Keeping Up with the Kardashians
1457395200102 3 245.251.95.60 38.8310742 -90.8715725 DVR CMT Pure Country CMT
Music
1457395206837 4 87.170.144.240 39.2840746 -76.512557 DVR Bravo HD The Real
Housewives of Potomac
1457395206843 5 221.164.53.177 37.165983 -76.4404546 DVR Nick Jr. PAW Patrol
1457395206850 6 16.8.6.211 41.6041147 -73.8479929 DVR FXX HD Raising Hope
1457395206853 7 130.155.183.17 0.0 0.0 AD Volvo
1457395206857 8 72.134.127.232 42.3354644 -71.0473624 DVR The Disney Channel
(East) K.C. Undercover
```

To:

```
1457395200019 0 67.120.40.62 0.0 0.0 AD Nationwide Insurance
1457395233715 0 67.120.40.62 0.0 0.0 AD Snickers
```

```
1457395287547 0 67.120.40.62 0.0 0.0 AD Nationwide Insurance
1457395295463 0 67.120.40.62 0.0 0.0 AD Nationwide Insurance
1457395383634 0 67.120.40.62 0.0 0.0 TVSERIE Quantum Leap S5E1
1457395402148 0 67.120.40.62 0.0 0.0 TVSERIE Quantum Leap S5E1
1457395431597 0 67.120.40.62 0.0 0.0 AD Michelob
1457395439514 0 67.120.40.62 0.0 0.0 AD Michelob
1457395449900 0 67.120.40.62 0.0 0.0 AD Michelob
```

5) Ordering: Every single file is ordered by time.

From:

```
1457998060675 0 205.139.31.92 0.0 0.0 DVR KBCWDT (KBCW-DT) PIX11 Morning News
1457998066877 0 205.139.31.92 0.0 0.0 DVR ABC (WABC) Jeopardy!
1457998489514 0 205.139.31.92 0.0 0.0 DVR Telemundo Network (East) Un nuevo da
1457998921201 0 205.139.31.92 0.0 0.0 AD KFC
1457999000536 0 205.139.31.92 0.0 0.0 DVR The Golf Channel Morning Drive
1457999171437 0 205.139.31.92 0.0 0.0 AD Hyundai
1457999171598 0 205.139.31.92 0.0 0.0 DVR Fox Sports 1 College Basketball
1457999177601 0 205.139.31.92 0.0 0.0 DVR CBS (WCBS) College Basketball
```

To:

```
1457999006801 0 205.139.31.92 0.0 0.0 DVR ESPN SportsCenter
1457999171408 0 205.139.31.92 0.0 0.0 DVR NBC (WNBC) NHL Hockey
1457999171437 0 205.139.31.92 0.0 0.0 AD Hyundai
1457999171598 0 205.139.31.92 0.0 0.0 DVR Fox Sports 1 College Basketball
1457999177442 0 205.139.31.92 0.0 0.0 DVR USA Network Modern Family
1457999177601 0 205.139.31.92 0.0 0.0 DVR CBS (WCBS) College Basketball
1457999348294 0 205.139.31.92 0.0 0.0 TVSERIE The Big Bang Theory S6E3
1457999683248 0 205.139.31.92 0.0 0.0 TVSERIE The Big Bang Theory S6E3
```

6) Malicious identification: In this step the software analyzes the data in every single file and decides if the user is malicious or not and populate a Postgres table with that information.

Postgres table

All the data analyzed are inserted in the following table:

FIELD	TYPE	DESCRIPTION
ID	SERIAL	Unique id of the row
USERID	VARCHAR	User id
IPADDRESS	VARCHAR	IP Address of the mobile phone of the user
LON	VARCHAR	Longitude of the mobile phone
LAT	VARCHAR	Latitude of the mobile phone
TIME_FIRST	TIMESTAMP	First time that the user activity is saved in the logs
CALLS	INTEGER	Number of total calls collected during the period
TYPE	VARCHAR	Type of user (GOOD| SAME CONTENT) *
MOST_WATCHED_CONTENT	VARCHAR	Name of the most watched content
TOTAL_TIME	INTEGER	Total time related of usage of the service
MOST_WATCHED_MATCHES	INTEGER	Number of matches related to the most watched content

*: GOOD means a regular users, SAME CONTENT means an user that is watching the same content all the day

In the source directory there are 2 sql scripts:

- script_project.sql: takes care of creating the table necessary for the software
- users-data.sql: data that are generated by the software. I exported the final result of the work, in case there are some problems running the software.

Python Rest API

The python REST API allows to access to the data using a set of APIs that accept a date in the format of "yyyy-MM-dd" (example: 2016-03-07) or a value "ALL".

1. Count the number of users grouped by type: **/boats/count/<string:date>**
   This api is used for statistical purpose.
2. Number of queries done to the service: **/boats/queries/<string:date>**
   This api is used to understand the number of calls done by malicious users across the time, versus the ones done by the good users. The purpose of this api is to share with the customer the results of the blocking.
3. List of all the malicious users with ids: **/boats/list/<string:date>**
   The purpose of this api is to feed a black list manager in order to block the malicious users that use the service.
4. Complete set of data: **/boats/data/<string:id>**
   The purpose of this api is to give access in read only mode to all the data of the user, for analysis in external tools.

The software that implements the REST API is app.py.

Run the software

To run correctly the software, it's necessary to login into the server with the w205 account.

After that, all the data files must be copied in HDFS in the following directories:

/user/w205/project/dataObfuscated/20160309
/user/w205/project/dataObfuscated/20160310
/user/w205/project/dataObfuscated/20160311
/user/w205/project/dataObfuscated/20160312
/user/w205/project/dataObfuscated/20160313
/user/w205/project/dataObfuscated/20160314

In every single directory there are 2 files for each hour in the following form:

/user/w205/project/dataObfuscated/20160308/data-20160308_00_sdr-us-sdk-grp-v2-kvxi.txt
/user/w205/project/dataObfuscated/20160308/data-20160308_00_sdr-us-sdk-grp-v2-ud64.txt
/user/w205/project/dataObfuscated/20160308/data-20160308_01_sdr-us-sdk-grp-v2-kvxi.txt
/user/w205/project/dataObfuscated/20160308/data-20160308_01_sdr-us-sdk-grp-v2-ud64.txt
….

Then the sequence of commands to run are:

Normalization

java -jar w205_hadoop.jar -norm /user/w205/project/dataObfuscated /user/w205/project/normalized

Output:

Processing: data-20160314_12_sdr-us-sdk-grp-v2-kvxi.txt
Processing: data-20160314_12_sdr-us-sdk-grp-v2-ud64.txt
…

All the files are normalized and grouped in N files, one for every single day

hdfs dfs -ls /user/w205/project/normalized

-rw-r--r-- 3 w205 supergroup 6219473 2016-04-27 14:15 /user/w205/project/normalized/20160308
-rw-r--r-- 3 w205 supergroup 5388565 2016-04-27 14:17 /user/w205/project/normalized/20160309
-rw-r--r-- 3 w205 supergroup 5606575 2016-04-27 14:20 /user/w205/project/normalized/20160310
-rw-r--r-- 3 w205 supergroup 5487535 2016-04-27 14:22 /user/w205/project/normalized/20160311
-rw-r--r-- 3 w205 supergroup 6244483 2016-04-27 14:25 /user/w205/project/normalized/20160312
-rw-r--r-- 3 w205 supergroup 7616021 2016-04-27 14:28 /user/w205/project/normalized/20160313
-rw-r--r-- 3 w205 supergroup 5668028 2016-04-27 14:30 /user/w205/project/normalized/20160314

Splitting

java -jar w205_hadoop.jar -split /user/w205/project/normalized /user/w205/project/users

log4j:WARN No appenders could be found for logger (org.apache.hadoop.conf.Configuration.deprecation).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Processing: hdfs://127.0.0.1:8020/user/w205/project/normalized/20160308
Processing: hdfs://127.0.0.1:8020/user/w205/project/normalized/20160309
Processing: hdfs://127.0.0.1:8020/user/w205/project/normalized/20160310
Processing: hdfs://127.0.0.1:8020/user/w205/project/normalized/20160311
Processing: hdfs://127.0.0.1:8020/user/w205/project/normalized/20160312
Processing: hdfs://127.0.0.1:8020/user/w205/project/normalized/20160313
Processing: hdfs://127.0.0.1:8020/user/w205/project/normalized/20160314

After this command, there are generated N files, one for every single user

hdfs dfs -ls /user/w205/project/users | more
Found 10387 items
-rw-r--r-- 3 w205 supergroup 1156 2016-04-27 15:20 /user/w205/project/users/0
-rw-r--r-- 3 w205 supergroup 6885 2016-04-27 15:20 /user/w205/project/users/1
-rw-r--r-- 3 w205 supergroup 3029 2016-04-27 15:14 /user/w205/project/users/10
-rw-r--r-- 3 w205 supergroup 6911 2016-04-27 15:25 /user/w205/project/users/100
-rw-r--r-- 3 w205 supergroup 3948 2016-04-27 15:20 /user/w205/project/users/1000
-rw-r--r-- 3 w205 supergroup 70 2016-04-27 14:45 /user/w205/project/users/10000
-rw-r--r-- 3 w205 supergroup 81 2016-04-27 14:45 /user/w205/project/users/10003
….

Ordering

java -jar w205_hadoop.jar -order /user/w205/project/users /user/w205/project/finals

hdfs dfs -ls /user/w205/project/finals | more
Found 10387 items
-rw-r--r-- 3 w205 supergroup 1156 2016-04-27 15:56 /user/w205/project/finals/0
-rw-r--r-- 3 w205 supergroup 6885 2016-04-27 15:56 /user/w205/project/finals/1
-rw-r--r-- 3 w205 supergroup 3029 2016-04-27 15:56 /user/w205/project/finals/10
-rw-r--r-- 3 w205 supergroup 6911 2016-04-27 15:56 /user/w205/project/finals/100
-rw-r--r-- 3 w205 supergroup 3948 2016-04-27 15:56 /user/w205/project/finals/1000
…

After this command, every single file is ordered by timestamp ascending.

Malicious finding
java -jar w205_hadoop.jar -malicious /user/w205/project/finals

psql -U postgres
select count(*) from users;
count

 10386
(1 row)

After this command, all the files are analyzed and are marked as malicious or not in a PostgreSQL table.
Example of data in the database:

```
postgres=# select * from users limit 5;
  id   | userid |   ipaddress    |    lon     |     lat     |       time_first        |  calls  |     type     |   mostwatchedcontent   | totaltime | mostwatchedmateches
-------+--------+----------------+------------+-------------+-------------------------+---------+--------------+------------------------+-----------+---------------------
 20451 | 0      | 26.196.255.15  | 0.0        | 0.0         | 2016-03-08 00:00:00.019 | 1346195 | SAME CONTENT | AD_AT&T                | 348973934 |               41299
 20452 | 1      | 199.76.81.186  | 0.0        | 0.0         | 2016-03-08 00:00:00.026 |    5487 | GOOD         | LIVETV_Cartoon Network | 162584902 |                 775
 20453 | 10     | 103.37.209.11  | 0.0        | 0.0         | 2016-03-08 00:00:06.88  |    2855 | GOOD         | LIVETV_MSNBC           |  93087325 |                 280
 20454 | 100    | 10.104.226.182 | 0.0        | 0.0         | 2016-03-08 00:00:05.755 |    6941 | GOOD         | LIVETV_CNN             | 203420492 |                1300
 20455 | 1000   | 110.252.82.216 | 33.7286918 | -84.3894391 | 2016-03-08 00:01:17.442 |    5894 | GOOD         | LIVETV_CNN             | 193797550 |                 738
(5 rows)
```

Run REST API

Run
`source api/bin/activate`
and
`python app.py`
in the directory where is located the app.py file.
Starting from now, the data will be available to be used by other applications or software analysis.

Output from the REST API Layer

/boats/count/ALL

```
{
    "boats": [
        {
            "count": 1004,
            "type": "SAME CONTENT"
        },
        {
            "count": 11049,
            "type": "GOOD"
        }
    ]
```

```
}
```


/boats/queries/ALL


```
{
    "boats": [
        {
            "queries": 6385348,
            "type": "SAME CONTENT"
        },
        {
            "queries": 15485016,
            "type": "GOOD"
        }
    ]
}
```


/boats/list/ALL


```
{
    "boats": [
        {
            "id": "0"
        },
        {
            "id": "10004"
        },
        {
            "id": "10007"
        },
        {
            "id": "1001"
        },
…
```


/boats/data/0

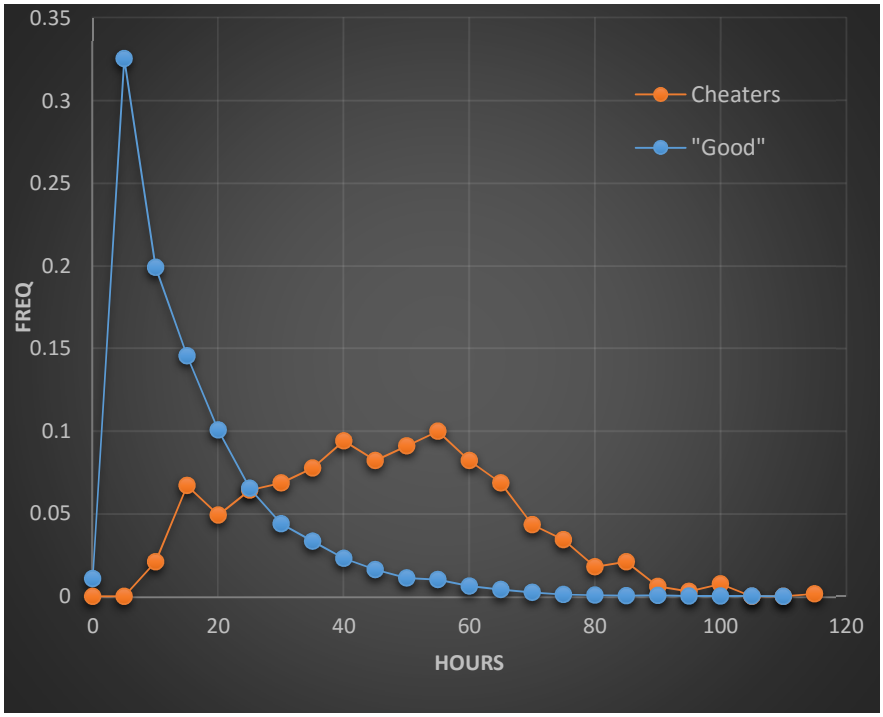

```
{
    "boats": [
        {
            "calls": 1346195,
            "ipaddress": "26.196.255.15",
            "lat": "0.0",
            "lon": "0.0",
            "matches": 41299,
            "queries": "0",
            "total": 348973934,
            "type": "SAME CONTENT",
            "watched": "AD_AT&T"
        }
    ]
}
```

Results

From the data analyzed, 8% of the devices are malicious and they generate 30% of the total queries. The majority of the malicious users have only 1 device that is listening continuously to the same content. There are also some powerful users with 10 to 15 devices like in the table below.

Devices	Users
1	644
2	56
3	17
4	14
5	3
6	5
7	3
9	2
10	2
11	2
15	1

In this graph, as expected, we can see that a normal user (blue line) spends less time in front of a TV compared to a malicious one (orange line).



Another interesting thing is presented in the following table:

DATE	TYPE	TOTAL CALLS
2016-03-07	GOOD	15,070,386
2016-03-07	SAME CONTENT	6,339,811
2016-03-08	GOOD	414,630
2016-03-08	SAME CONTENT	45,537

For the nature of the app that is using our service, the number of users doesn't change a lot across the time and in 1 week of analysis of the sample, there are not many new entries of malicious and good users.

That means that the system that blocks the activity of the malicious users can be implemented as a batch process that runs periodically, like for example once a day, or even once a week. This has a positive effect on the scalability of the system, because it's a process that doesn't require a lot of computational power to be completed thanks to the fact that can take even 1 day of analysis.

Next steps:

In order to install the system in production

- I have to scale to all the 60,000 users and analyze the data on a daily base;
- use Spark for the computational part;
- move from Pythoin to JAVA REST API;
- add more algorithms to better identify the malicious users;
- refine the software for a production environment (handling the exceptions, probes to verify that there are not blocking events);

The intermediate files can be removed as soon as the overall process is completed. This saves a lot of space.

The space currently used for the solution is the following (evaluated on a NFTS, on HDFS should be considered the replication factor):

Stage	Prototype (1 week 12k users)	Production (1 day 60k users)
0 – Input data	28.1 Gbyte	20 Gbyte
1 - Filtering	13.7 Gbyte	9.8 Gbyte
3 – Normalization	1.65 Gbyte	1.18 Gbyte
4 – Splitting	1.65 Gbyte	1.18 Gbyte
5 - Ordering	1.65 Gbyte	1.18 Gbyte

As you can see, the numbers involved are pretty low.

N.B. The step for the privacy protection is not necessary in the production environment (the only purpose is to share real data with the class and the teacher).