Activity-aware geofencing platform: BoloFence

Salvatore Fiorilla[1], *Paola* Persico[1], and *Igor* Iurevici[1]

[1] *University Alma Mater of Bologna*

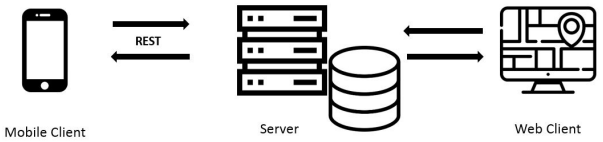

Figure 1. System architecture.

Keywords: Location, Current Activity, Transition, Current Geofence, Time to sleep, REST, API, Web Map, Postgis.

1 Introduction

The platform allows message dissemination according to the user location (location-aware system) and the type of mobility (activity-aware system).

By activating the service on an Android device, the user can receive notifications every time they enter in a specific area (geofence) in the city of Bologna, so that the notification is consistent. There are three types of geofences depending on mobility: bike, walk and car.

When the message contains a link, it is possible to open it in a WebView.

In addition to the mobile client, the platform includes also a back-end, which computes the geofences, saving the path of the user in a database; and a front-end which allows to view the paths, the boundaries of the geofences, and the spatial clusters.

2 Design

The architectural style chosen for the service is REST. The client sends a request in json format through POST method and the server replies with a json response.

In order to identify a user to update the path, serial numbers are used instead of personal information. However, the IP address of the user who makes a request isn't hidden, so it is possible to link it to the path. To improve privacy, a perturbation technique was used.

2.1 Mobile Client

2.1.1 Introduction

In order to improve user privacy, to optimize battery consumption and application performance, our research focused on minimizing client-server interaction. Detecting Activity Recognition is a long running task. For mobile devices, running it periodically can result in an unbearable overhead in terms of computing. These reasons led us to adopt a solution based on Transition Activity Detection.

2.1.2 Client algorithm

Basically the main idea focuses on tracking the paths of our users. The app does so detecting periodically activity transitions and locations. For this reason, we introduced the concept of path identifier. The path identifier is generated when the user clicks on start button and expires when he stops the service.
Actually, the application tracks the user only when he is performing an action of our interest. Therefore, if the app is running in background mode on the device, when the user starts walking, driving or cycling, the app tracks him until the end of the action. In this latter case, the action mode of the system becomes then "unknown" and the service, which has been woken up, goes to sleep until it is not triggered again.
The triggering of the background tracking service occurs thanks to an external service (in our case the Transition API) which triggers a new execution of the job (more details below) or put on the pause of the background service. The trigger message so could be due to a recognition of transition activity which may be of a "start action" or an "end action" event type.
Only on the first start command, a non-valid path identifier is assigned to the client and the current user activity is also detected by sampling instead of transition recognition. If the activity is valid, it is handled as a start Activity Transition event by the algorithm. And the execution starts. Else, if the action is not of our interest, the service goes to wait.
The job executed at every wake up consists in the following steps: the mobility values received are saved as current activity, then location is obtained using GPS and finally, the client prepares and sends all data packed as geoJSON to the server.

Only, meanwhile the user is performing an action, so in other words, only after a transition of a valid stating action is gathered, a timer which periodically wake up the service and sends data to it is set. The lifetime of the timer ends with the action. What rests of the execution time is spent by the app in sleep mode and waiting for a new event occurs.

Here's an example of our geoJSON Format:

Request

```
{"type":"Feature",
 "geometry":{
    "type":"Point",
    "coordinates": [ 11.355, 44.485 ]
       },
 "properties":{
        "activity":"walk",
        "pathId":32,
        "currentGeofence":"Giardini
           Margherita"
        }
}
```

Response

```
{   "pathId": 32,
    "timeToSleep" : "60.8346"
    "message" : "No new Geofence"
}
```

The response message contains three values:

- the path identifier, which is used for the path updates;
- a time to sleep, which is dynamically chosen by the server according to the client position;
- a message, which is linked to the current geofence and that contain useful information for the user.

During the first client-server interaction the path identifier is a new value assigned by the server, which replaces the default value. In order to show the gathered information to the user, the message is displayed as notification and, in addition sometimes, it could have a URL, if it has, this one can also be opened as web view. The message field is ignored only when it is in the form of "No new geofence", in that case receiving it means that the user has already got the new geofence notification and so, the current geofence doesn't need to be updated. Finally the server starts to sleep again according to the time suggested by the server.

Only after the first client-server interaction due to the event of a new performed action, the background service can be triggered up periodically by a timer. When this happens, the activity mode is already known thanks to the previous interaction, so the client is ready to execute, otherwise it goes back to sleep.

2.1.3 Client privacy

Purely the needed information to make the service work has been used and the privacy is handled rounding the coordinates sent to the server to three decimal points (and a random four decimal digit is added) if the user is far from a geofence (depending on time to sleep), or truncating to four decimal points if the user is near a geofence. The former correspond to a precision of about 40 meters near the 45th parallel north (passing just south of Piacenza), while the latter corresponds to a precision of about 8 meters. [1, 2].

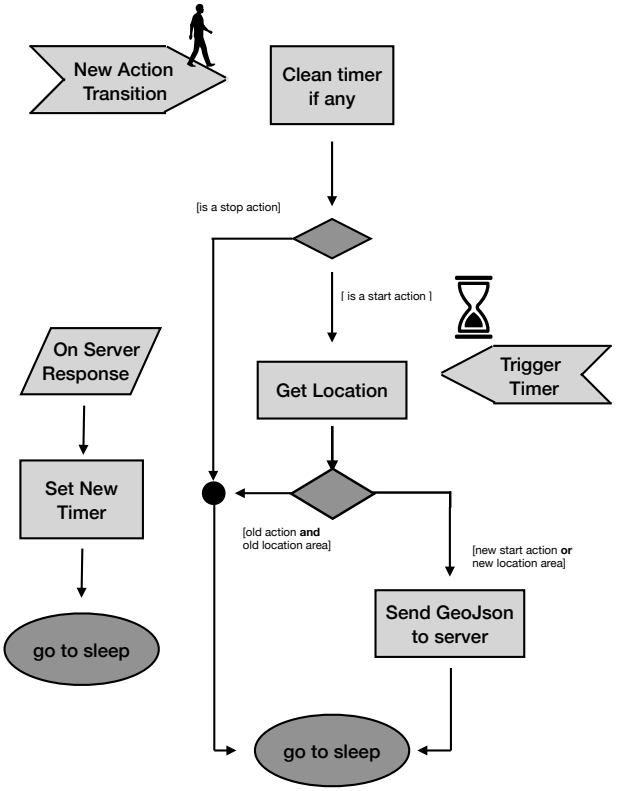

Figure 2. Client Process Flow

2.1.4 Client optimization

There is an optimization check to avoid not necessary client-server interactions when the timer expires and the user is performing the same action and he is in the same detected rounded area as before and no change of privacy precision has occurred.

2.2 Server

2.2.1 Spatial Data Storage

Since our platform is a location-aware system, we chose PostgreSQL with PostGIS extension as DBMS, because it provides spatial data types, spatial queries, and it allows spatial indexing. Our database includes a schema for the three geofence tables (one for each activity: walk, car, bike) and a schema for the three paths tables. Geofences have four fields: id, geom (Polygon), message and intensity (the total number of activations). Paths have three fields: id, geom (LineString) and the date of the last update.

2.2.2 REST service

The idea behind the service is simple. The server implements a RESTful service: it takes as input a geoJSON feature (through POST method) and it can return an error response, if the coordinates or the activity are not valid, or a response with a message, a time to sleep and a user id (which identifies the path for the following updates). Basically, if there is a path associated to that user, that path is updated adding the current location, otherwise a new path is created. The server also checks if there is an associated geofence to that location and it compares it to the current geofence of the user, so that they don't get multiple notifications for the same geofence. The message returned can be of three types: the message of the geofence; a message which reports to the client that the user is not currently in a geofence; an info message which reports to the client that the user is still in the same geofence.

2.2.3 Request optimization

In order to minimize the requests and the battery consumption, we return to the client a "time to sleep", which is an underestimation of the time it takes to get to a new geofence according to the current location and the current activity. An underestimation is used to try to prevent the possibility that the moment that the user enters in a new geofence is skipped. In addition to this, we keep the time to sleep below a certain threshold (which is linked to the mobility speed and the location precision), because if it too short then too many requests could be made and it would become useless.

2.3 Web map application

2.3.1 Overview

The implemented server is also responsible for the webpage hosting. Our CodeIgniter project has been adapted in order to do the job for both RESTful service and web map visualizer. As already stated before, the framework is based on an MVC template, so the web app has been developed accordingly.

2.3.2 Front-end

The visualizer implementation has not been written from scratch, in fact for the initial spatial data representation we relied on QGIS software, which provides an intuitive GUI and a collection of tools for spatial data visualization and manipulation. The obtained QGIS project has been then exported (qgis2web) in a web code format (HTML/JS/CSS) that exploits the Openlayers library functions. Starting from there we implemented the clustering algorithms and updated the UI (using Bootstrap components) in order to fulfill our purposes.
For each activity we can distinguish the following layers:

- Initially loaded layers:
  - Geofences;
  - Paths;
  - Arrival points.
- On-demand layers:
  - K-means area;
  - Geofence suggestions (DBSCAN).

In addition to that the navbar provides tools in order to show/hide/generate the respective layers.

2.3.3 Back-end

Since everything lays under the same server, the web backend shares also the same database. On page loading, the controller makes available through Postgis queries all the necessary data for initial layers rendering, while the clustering functions are requested using AJAX POST requests. To not overwhelm the client computational resources, the clustering algorithms are performed on the server-side in the form of spatial queries. Intuitively, this approach may also affect significantly the server performance due to the high complexity of clustering algorithms, but since the web map is supposed to be a tool available only to the staff, it's very unlikely that its usage saturates the server.

3 Implementation

In this section we describe the platform implementation details.

3.1 Mobile Client

The client has been implemented for Android OS. The target version is 9 or higher and Android Studio IDE has been used. To detect the user activity modes we chose Android Activity Recognition Transition API. It allows to notify the application only when it is needed. Since we use the activity transition event, when the service is started it can happen that none of them occurs and therefore, in order for the algorithm to work correctly, the recognition of the first user activity is required. The Activity Recognition Sampling API does the job. Furthermore, to retrieve the last location we opted for the Fused Location API which is connected to the Google Play services location API. To gather client properties, generate the geoJSON request and send it to the server, Retrofit 2 Library has been used. Basically, the main work of the application is done in a foreground service class of Android which implements the algorithm described in the Design section. The creation of the geoJSON involves a set of wrapper classes. Memory is managed by Shared Private Preferences.

3.2 Server

3.2.1 Framework

The framework chosen for the server is CodeIgniter, a MVC-based PHP web application framework which is

reliable, lightweight and simplifies operations such as database access and debugging. To implement the RESTful service, a library called CodeIgniter RestServer was used, which can easily handle POST JSON request.

3.2.2 Service

The database queries make use of several PostGIS functions, such as:

- ST_GeomFromText, to convert coordinates to a geometry;
- ST_Within and ST_Buffer, to check if the location is in the center of Bologna;
- ST_AddPoint, to add a location to the path;
- ST_Contains, to check if a location is in a geofence;
- ST_MakeLine, to create a new path;
- ST_DistanceSphere, to compute the distance between the location and the geofence;
- ST_ExteriorRing, to get the perimeter of a geofence;

In order to compute an estimate of the time it takes to get to a new geofence, the average walking speed and cycling speed are used [3, 4], as well as the driving speed limit in the urban center of Bologna.

3.3 Web map application

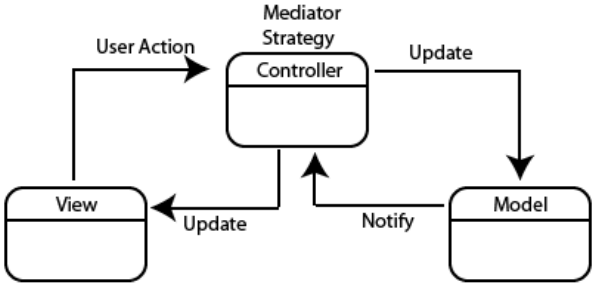

Figure 3. model-view-control.

3.3.1 Model

The model is the brain of the web-app and in our case consists mostly in querying the database. As soon as the page is loaded, the model is involved by triggering simple queries necessary for the initial layers rendering. To obtain a JSON file directly from the queries, we have exploited the Postgres built-in function json_build_object() and json_agg() which fulfilled the job. We have also taken advantage of Postgis built-in clustering functions in order to compute K-means and DBSCAN. The last-mentioned has been used to determine an area outside any geofence with, potentially, the highest intensity. The whole approach is explained in detail down below.

3.3.2 View

The view section is mostly responsible for front-end visualization. In fact, it is composed of an index.html file and all its support files (assets). Most of these files are the result of qgis2web export operation. Some of these have been retouched in a significant way through edits, additions and code cleaning. But the overall structure and the core files that define qgis2web have been maintained in order to be able to import easily new layers and styles. For a more intuitive and comprehensive UI, we used Bootstrap components in addition to jQuery and font-awesome (CDN source).

When it comes to the projection CRS we used the "WGS84 / Pseudo-Mercator" standard EPSGs which are:

- EPSG:4326 for geoJSON data;
- EPSG:3857 for Openlayers map projection.

3.3.3 Control

Last but not least, the controller acts as a man in the middle between the view and the model by providing the model's computed data to the view. The request happens both in a direct way (when the page is loaded) and indirect (through AJAX POST requests).

3.3.4 Geofence suggestion

Among the web-app tools, it is important to focus a little bit more in detail when it comes to the "Generate suggestion" one. In fact, instead of applying directly built-in and well-known algorithms, we had to find out how to pre-compute existing data in order to meet our needs. "Generate suggestion" tool idea is to suggest to the user (staff in our case) the area, outside the existing geofences, where it is most likely that a person goes through. In other terms, it is potentially the best area where to place a new geofence. To do that the obvious thing was to consider the already existing paths and work on their density distribution throughout the map. So, we decided to use DBSCAN algorithm in order to generate the cluster with the highest density. Since the goal is to suggest an area for a new geofence, we cut off those paths fragments that lay inside an existing geofence by using ST_Difference(path, geofence) function. The POSTGIS built-in DBSCAN works both with points and linestrings. Applying the algorithm on the lines didn't work quite well since we don't really care about the path in its whole, but only about those fragments that tend to be close enough to other fragments of other paths. On the other side, using fixed points (start/middle/arrival) that lay on the linestring wasn't enough to represent its trajectory, so it was clear that those "points-online" had to be dynamic. In order to do that, the idea was to exploit ST_ClosestPoint(path1, path2) (Figure 4) function in order to compute the point on the line that is the closest to the other line. By applying this function we were able to split each line into its more relevant parts: the closest points to other lines. During this operation, we also

kept track of the distance between each ST_ClosestPoint result. This allowed us to have an overall picture of how the points distribution is, and by applying a user-chosen percentile on that collection, we had, as a result, a legit eps value to feed to DBSCAN function. Now we were missing the last DBSCAN parameter needed: minPoints per cluster. Since we care only about the cluster with highest density, minPoints is far from being relevant in this operation, so we kept it at 1. From the resulting clusters, we extracted the one that intersects most of the initial paths and returned its geoJSON to the view.

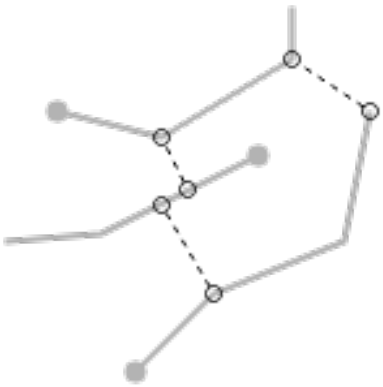

Figure 4. ST_ClosestPoint() example.

Table 1. Lockito car (35 km/h) delay evaluation

Test	Delay (s)
Test 1	2.5
Test 2	4
Test 3	5
Test 4	2.5
Test 5	2
Test 6	1 (instant)
Test 7	1 (instant)
Test 8	5
Test 9	4
Test 10	3.5
Overall average	3.05

Table 2. In-site notification evaluation

Geofence	Type	Dev (m)	Delay (s)
Santo Stefano	Walk	7	5
Due Torri	Walk	1	29
Piazza Maggiore	Walk	8	10
Colonnina marciapiede	Car	110	10
Colonnina San Felice	Car	70	9
Viale XII Giugno	Car	1	85
Rastrelliere Zona Torri	Bike	5	2
Rastrelliere San Petronio	Bike	8	3
Rastrelliere Nosadella	Bike	7	3

4 Results

4.1 Evaluation

Several tests were carried out after the implementation phase and in this section we are going to show the most interesting ones. Basically, tests were done in two different environment: using a real phone outside Bologna and using another phone in the city center of Bologna. The reason why we chose to do so comes from the need to have many different evaluations in different environments, and furthermore, because of current quarantine restrictions. Android 10 was the running version in every test. The technical characteristics of the testing devices are the following: a One Plus 6T with Snapdragon 845 and 8GB RAM and a Huawei Mate 20, with processor Huawei Kirin 980 and 4 GB RAM.

As far as the simulated tests are concerned, a fake location generated by Lockito application was used [5]. As well known, Lockito can't simulate a activity transition action because it cannot generate enough sensors values. Therefore we carried out the Lockito tests only with "car" mobility type.

4.1.1 Service Performance

We have compared the notifications to the user context simulating the paths using Lockito and testing in the city center of Bologna. In the first case, there were no false positive or false negative geofence recognized, and the delay (the difference between the time the user enters in a geofence and the time the notification is received) was always below 5 seconds, as shown in Table 1.

In the second case, we have also evaluated the deviation (that is the distance between the border of the geofence and the user location when the notification is received). The results are shown in Table 2, and have been clustered depending on the type of geofence.

Our testing results gives us a general idea about the good functionality of the service, but these results are not immediate and need a few words of explanation. The ideal feedback of a good functionality is given by a low deviation combined to low delay, which as we can see occurs in many cases. However our concerns point to the few cases were there is not a direct correlation between deviation and delay, hence we have studied more in detail these numbers. By analyzing those paths we have found out that these anomalies occur when the trajectory of the user is parallel to the border of the geofence. In fact, those cases are the trade-off of the cloaking technique applied to the location coordinates computed by the client. Finally, even though the deviation distance for car may result high in certain tests, these values are easily explained by the fact that the driving speed (30-40 km/h) is way higher that the other two mobilities' speed, so we have to keep in mind those proportions.

Table 3. In-site notification evaluation statistics

Type	Dev (m)	Delay (s)
Walk average	5.33	14.67
Walk median	7	10
Car average	60.33	34.67
Car median	70	10
Bike average	6.67	2.67
Bike median	7	3
Overall average	24.11	17.33
Overall median	7	9

Table 4. Activity recognition evaluation

Activity	Recognition	Average Delay (s)
Walk	Sampling	10
Bike	Sampling	16
Car	Sampling	5
Walk	Transition	10
Bike	Transition	13
Car	Transition	67
Average		20.17

4.1.2 Activity Recognition

. Even though we used an Android API for activity recognition, we tested the first activity detection and the activity transition recognition to better understand how they affect the service. The results are shown in Table 4.

4.1.3 Battery Consumption

The in-site tests were carried out on a device with no energy saving setting with a 30% battery level, and the average energy consumption was of 0.05% every 30 minutes.

4.2 Conclusion

In conclusion we believe that BoloFence respects every assignment request. As for everything, there is always room for improvement, and we have figured out a few points which may be implemented in future works:

- the activity recognition, which now relies on Android services, is not ideal for our purpose, and so, an ad-hoc Activity Recognition API may be developed in order to accomplish our urban context needs;
- our web app clustering tools exploits Postgis built-in functions which, as for the previous case, may be implemented using some more accurate tools provided by Machine Learning dedicated Libraries (e.g. Python's Scikit Learn Library);
- in order to achieve a better time-to-sleep estimation, a trajectory prediction may be computed exploiting the device compass and Bologna city's streets topology, available either on Google services or through Geoserver datasets.

However, as previously mentioned, the provided tool performance and reliability is satisfactory enough for real use and therefore for public distribution.

References

[1] Wikipedia, *Decimal degrees* (2020), `https://en.wikipedia.org/wiki/Decimal_degrees`

[2] Wikipedia, *45th parallel north* (2020), `https://en.wikipedia.org/wiki/45th_parallel_north`

[3] R.V. Levine, A. Norenzayan, *The pace of life in 31 countries* (1999)

[4] C.E. of Denmark, *Copenhagen's bicycle account* (2013), `http://www.cycling-embassy.dk/2013/06/03/6995/`

[5] Bootstraptaste, *Lokito - faking android gps itinerary* (2020), `https://lockito-app.com/`