A Map-Based Platform for Smart Mobility Services

Pietro Marchetta, Eduard Natale, Antonio Pescapé, Alessandro Salvi, Stefania Santini
University of Napoli Federico II (Italy)
Email: {pietro.marchetta,pescape,alessandro.salvi,stsantin}@unina.it, ed.natale@studenti.unina.it

***Abstract*—Nowadays smart mobility, a new vision of urban mobility, is a reality. To implement smart mobility scenarios a deep integration among citizens, private and public transportation systems and ICT is required. With the S^2-Move project we propose an architecture able to collect, update, and process real-time and heterogeneous information from various sources (tablets, smart-phones, probe vehicles) and actors of the urban scenario (public/private vehicles, pedestrians, infrastructures) in order to provide innovative mobility services. In this paper, we present a core component of the S^2-Move project: the map-based web platform designed and implemented for providing smart mobility services. We present use cases and detail the design and the implementation of the platform. Finally, we evaluate the accuracy and efficiency of the Map Matching and Traffic Monitoring algorithms we implemented in our platform by using realistic urban traffic data generated through simulations in SUMO.**

I. Introduction

The quality of life in the urban environment depends on the interaction between the actors of the city, i.e. citizens, vehicles and transportation systems. In order to improve the mobility quality, the way the urban actors communicate each other so as to create an efficient network where each actor is a node that routes the information, is still an open issue.

This paper describes the map-based platform of the S^2-Move project[1], designed to collect the information from the urban environment, generate new knowledge and share it with the citizens through an interactive map providing a set of mobility services. The S^2-Move project has been previously presented in [12], [13]. More precisely, in [12] we describe the main technological aspects involved in the design and in the implementation of the S^2-Move architecture. Then, in [13] we discuss the main issues related to trustiness and security of the S^2-Move architecture. Rather than describing research activities, the main objective of this paper is to show the practical integration of well known technologies, in order to provide real mobility services in a real mobility scenario. Hence, the novelty and the innovation of this work are in the final services that the S^2-Move platform is able to provide, and not in the technologies themselves adopted to implement these services. Currently pay-by-phone parking systems [16], fleet tracking and management systems [11], [7], [8], and crowd-sourced road traffic evaluation systems [15] do not integrate heterogeneous mobility services but focus on just one particular issue of the urban mobility scenario, i.e. the management of parkings, vehicles, and road traffic. The aim of the S^2-Move platform is developing and integrating Smart Mobility services in order to let them to cooperate with each other and help the authorities to manage urban mobility problems. The S^2-Move platform is able to provide map-based smart parking services, public and private transportation fleets management, road traffic conditions estimation, and warnings management. The motivation of a map-based approach is strictly related to the advantages of the geographic representation of the data, allowing easy manipulation and analysis beyond conventional GIS systems, and allowing the user to contribute to the dataset using intuitive interfaces [2].

The paper is organized as follows: Sec. II briefly presents use cases and users of the S^2-Move map-based platform. Sec. III shows the main aspects of the platform design with specific reference to some Smart Mobility services while Sec. IV details its implementation. Sec. V presents experimental results on the accuracy and efficiency of the Map Matching and Traffic Monitoring algorithms using realistic urban traffic data generated through simulations in SUMO.

II. Use cases and users

The web-based platform accepts input from a generic user (e.g. smartphone, vehicle) through HTTP requests and replies according to the particular service the user is asking for. The general use case of the platform requires a user to first authenticate himself in order to benefit from a Smart Mobility service, e.g. enabling the visualization of the road traffic status. The platform, in turn, verifies the user authentication and retrieves the necessary information to satisfy the user's request. There are four categories of users, according to the possible interaction they can have with the platform. A *guest* is a limited user that can passively benefit of a subset of Smart Mobility services (e.g. view parking lots locations) through the Web GUI (Graphical User Interface). A *logged user* is registered to the platform and can contribute through the GUI, e.g. make reports about mobility inefficiencies. The *smart user* can also contribute through a particular device, namely On-Board Unit (OBU), that is installed on his vehicle and collects, pre-processes, and sends mobility information to the platform. The *administrator* has access to the back end of the platform supporting advanced operations, like assigning roles to the users and interacting with a urban control panel. Figure 1 shows one of the services the platform is able to provide through the Web GUI: the Fleet Management. Each item in the panel on the right is a Smart Mobility service, for which there is a set of options a user can select/configure in order to filter data to show. For example, it is possible to show only vehicles which fuel consumption, speed or pollutants emission level is above a certain threshold.

III. Platform design

The S^2-Move platform has three architectural layers, namely *Presentation Layer*, *Core Layer*, and *Data Layer*

[1]The S^2-Move project is a 36-months long social innovation project started on June 2012 and co-funded by MIUR (Italian Ministry of Education, University and Research).

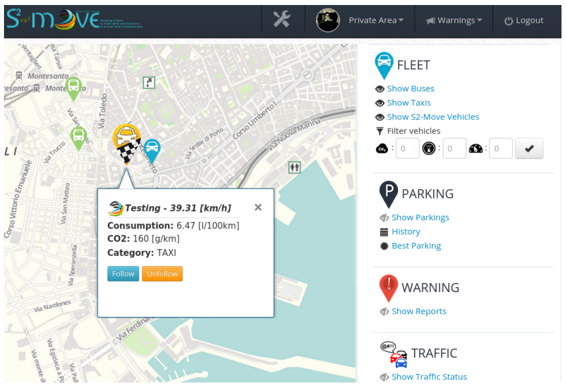


Figure 1. The Fleet Management service through the Web GUI.

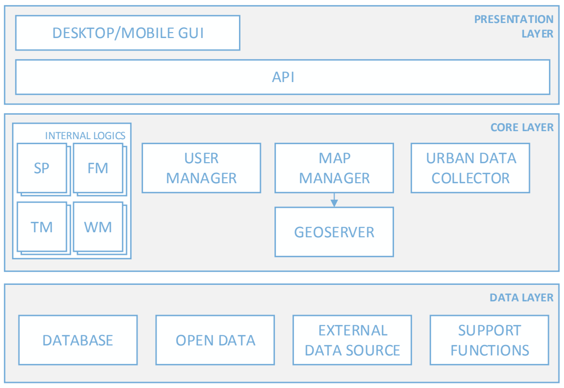


Figure 2. The software architecture allows the user to access the Smart Parking (SP), Traffic Monitoring (TM), Fleet Management (FM), and Warning Management (WM) services.

(Figure 2), implementing the map-based structure introduced in the previous section. The Data Layer communicates with a database in order to store and retrieve the data according to the requests received from the Core Layer. The Core Layer integrates the system logic, i.e. the modules that formally define the Smart Mobility services. Each module performs a particular job (e.g. traffic estimation on urban roads) interacting with the database. The Presentation Layer satisfies user requests. Each request is routed to the specific Core Layer module. Finally, all the data retrieved through the Presentation Layer can be displayed on a geographic map. The platform is able to collect information from Web Contributors and Device Contributors. The *Urban Data Collector* acquires and processes the data sent from device contributors (i.e. vehicles equipped with an OBU), and stores such information through the Data Layer. Figure 3 describes the collection process from a logged user that decides to use a vehicle equipped with the OBU, i.e. he becomes a smart user without loosing the HTTP session and starts to send vehicle's information. The collected information can be used to show the position of the vehicle on the map or to provide Smart Mobility services (e.g. traffic monitoring).

The *Map Manager* is responsible of transforming geospatial information into an image that can be immediately and intuitively understood by humans. The module interacts with a geospatial server implementing mapping protocols, caching systems, and handling several data output formats according to the the client request – from classical image formats (like

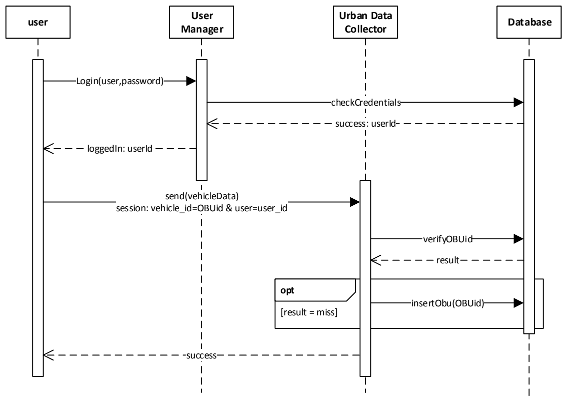


Figure 3. The logged user becomes a smart user.

Jpeg) to text-based exchange formats (like GeoJson). In the former case, the server renders an image according to the geospatial information provided by the Data Layer and the client simply downloads and displays the image. In the latter, the server creates a set of features that sends to the client into a text file. In this case, the client itself handles every feature separately and generates an image. In both cases the goal is to produce a map layer with the geospatial information provided by the platform. The most important difference is that in the second case the client is free to handle every single feature composing the layer. Figure 4 shows the interaction between the Map Manager and the geospatial server. The geospatial server, GeoServer, supports WFS (Web Feature Service) and WMS (Web Mapping Service) protocols. The WMS protocol is generally adopted to retrieve the data source as a flat image, while the WFS protocol is used to retrieve the information in text-based formats. The first approach exploits web caching, an advantage of this technique very useful when dealing with huge amount of data to process. Conversely, the advantage of WFS is the possibility to interact on the client side with every feature that compose the layer, which is impossible in the first case. With WFS the data is downloaded in text format and rendered client-side, which can cause performance issues related to the size of the data. The S^2-Move platform adopts WMS for traffic layers, because the only need is showing traffic information as colored lines, and WFS for the other services, allowing the user to interact with the data.

The *Smart Mobility Services* represent the core of the business logic. Each one interacts with the Data Layer in order to retrieve or store partial- or full-processed data. *Traffic Monitoring*, *Smart Parking*, *Warning Management* and *Fleet Management* are the basic services the platform currently provides. When the user is logged into the platform, it is enabled to send information to contribute to the smart mobility services. Each service is thought to be represented as a layer on a geographic map. The Warning Manager collects and shows user reports about public services (inefficiency, damages, unpleasant conditions, etc.). Each report can be classified according to the particular emergency level. Every logged user can make, confirm, and comment reports, and keep trace of reports' history. A marker identifies the exact location of the report on the map. Markers can be of different colors according to the urgency level of the report. The Traffic Manager evaluates the traffic conditions on the roads. It needs Device Contributors to send information (speed and position)

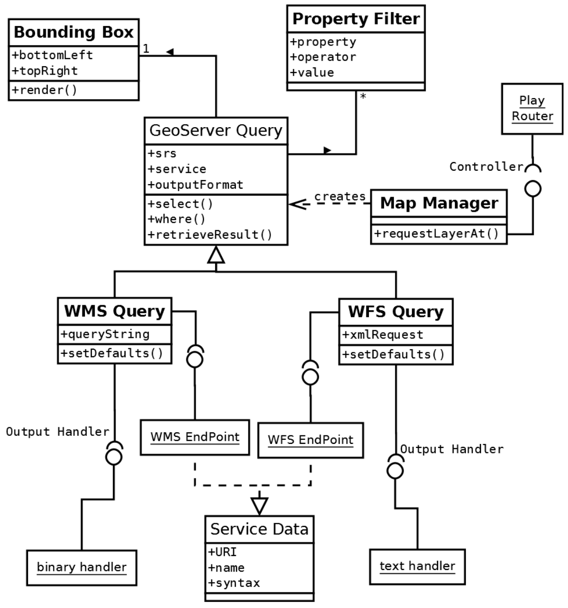
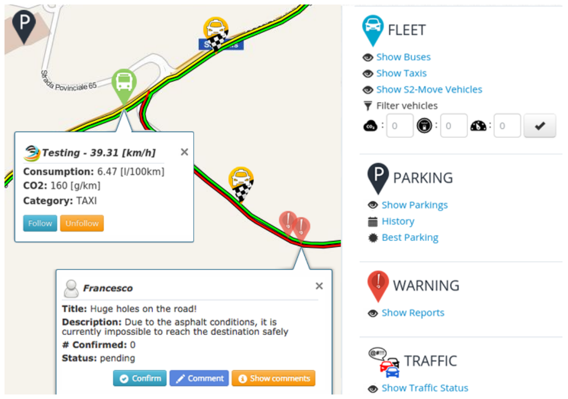


Figure 4. Map Manager simplifies the client-GeoServer interaction.

to the platform. The Traffic Manager processes the speed and position information and generates aggregated geospatial information ready to be graphically represented. The graphical representation of the traffic is obtained by coloring every lane according to the traffic level. To give an idea of traffic level, the green, yellow, and red colors are used to describe low, medium, and high traffic levels. The Fleet Manager shows the information about fleets moving in the urban roads network. Basic information are instantaneous speed, fuel consumption, pollution level, and vehicle fleet category (e.g. bus, taxi). The service needs Device Contributors to send information to the platform. A different marker identifies each vehicle according to the particular fleet it belongs to. It is possible to filter out vehicles according to several parameters. The Smart Parking service allows to reserve a parking. A guest user can simply view the position of the parking lots, while a logged user can ask for the best parking (according to the distance from the user's position and the price), reserve a parking, see the history of the reservations, and modify previous reservations. The graphical user interface in Figure 5 shows all the services provided by the platform. The moving fleets of vehicles can be filtered according to several options on the right panel. Warnings can be confirmed and commented. Each service is defined as composition of many internal logics, i.e. functional modules belonging to the Core Layer executing basic operations. The platform guarantees **compatibility** among different technologies adopting a universal interface through a powerful API. The API lets devices, sensors, and users having access to the smart mobility services provided. Each service can be **extended** with a higher number of functionalities, thanks to the **modular** architecture. Each module can be **reused** in order to build a new service or a new functionality: for example, it is possible to provide a car pooling service exploiting the Urban Data Collector to acquire vehicles information and the Map Manager to show their position (Sec. IV-A). In order to have a certain degree of **maintainability**, each service can be in a certain operational state: *under construction* or *in execution*. The software is designed to be deployed in a cloud environment, separating the database layer from the business



Testing - 39.31 [km/h]
Consumption: 6.47 [l/100km]
CO2: 160 [g/km]
Category: TAXI

Francesco
Title: Huge holes on the road!
Description: Due to the asphalt conditions, it is currently impossible to reach the destination safely
# Confirmed: 0
Status: pending

Figure 5. Users can show fleets of vehicles, make filtering operations, send, comment, and confirm reports, view and reserve parkings, view traffic conditions.

logic. Finally, **security** requirements have been analyzed in a previous work in vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communications [14].

IV. Platform Implementation

This section describes the implementation of each component of the architectural layers previously introduced. The software framework adopted to develop the architecture is Play! [1], written in Java and Scala. The framework implements the conventional aspects of the platform, allowing the developer to focus just on unconventional aspects and the business logic of the platform. The framework allows to forward every request to the appropriate action of the Core Layer through the router component. The action, in turn, may request persistently stored information to the Data Layer and send HTTP responses to the user. Components are mainly written in Java and Scala, the database uses the PostgreSQL DBMS (DataBase Management System) with PostGIS spatial extension, allowing to efficiently execute spatial operations like geographical distances evaluations, reference systems conversion, map matching, and so on. The mapping procedure takes advantage of GeoServer, an open source server which processes and publishes geospatial data using standard geospatial protocols (like WFS and WMS) and exchange formats (e.g. GeoJson, GML, rasters, and compressed images). In order to access the data-source, the platform uses a Java Persistence API (JPA) for the management of relational data. The JPA implementation adopted is Hibernate Object Relational Mapping (ORM) which supports lazy initialization, several fetching strategies, and optimistic database locking. From a security point of view, the Play! framework supports HTTPS and, in case, it is possible to generate a self-signed certificate, guaranteeing no eavesdropping in the communication between the client and the server.

The Presentation Layer provides the interface adopted by the users to interact with the platform and take advantage of the Smart Mobility services. The Presentation Layer serves requests from users through a GUI or API. In the former case, the interface is accessible through a Web Browser where each service is represented as a layer on a geographic map, while in the latter users retrieve the same information in raw format (e.g. Json or XML). The response obtained through the API can then be handled from any device that is able to issue HTTP requests and read HTTP responses in order to design custom applications. Advanced options, like services parameters modification, services deactivation, users subscription management, are reserved to administrators. The HTTP requests are encapsulated in GET an POST messages.

The Graphical User Interface (GUI) allows user authentication, provides the interactive geographic map to visualize the geo-referenced mobility services, allows data filtering, allows to perform client-side aggregations of the data and users' management.The OpenLayers open source JavaScript library allows to render on the map the geo-referenced data obtained from the server in form of points, lines or polygons. For example, a parking lot may be identified by a marker, the traffic level by two lines of different colors, one per lane, according to the average speed. Each layer is placed on a base layer provided by a tile server, allowing zooming and panning operations (slippy map). OpenLayers interacts with geospatial servers (like GeoServer) through WMS and WFS protocols. Actually, in order to guarantee information hiding and to allow third party applications to easily communicate with the platform, the API *hides* the classical WFS and WMS formalism, exposing just a subset of important characteristics. The client-side implementation, in addition, introduces the user strategies which allow to (i) periodically refresh the layer (so the service) and (ii) manage user events (e.g. on click, on zoom).

The API allows to create third party applications that can issue to the server the same requests a user can perform through the GUI. The API system is generated exploiting the route files in the Play! framework. In particular, the route file contains a set of rows where each row is composed of three elements. For example, in order to send the mobility information to the platform for a vehicle equipped with an OBU, there's the following syntax:

```
POST /acquire controllers.Acquire.index()
```

where `POST` is the HTTP method, `/acquire` is the URI (Uniform Resource Identifier) pattern, and the rest is the specific action of the Controller. A first form of control can be made directly in the routes file, which supports regular expressions to validate any dynamic part of the URI. Third party applications can use the API to integrate smart mobility services. For example, tour operators, which develop their own application, can call the API to allow parking reservation near touristic amenities, a mobile navigation system can adopt traffic information to estimate the time to reach the destination or to provide alternative paths.

A. Core Layer

This section describes the business logic modules of the platform, describing how geospatial data is collected, stored, and represented. The Map Manager is responsible of the data geographic mapping operations. The module sends and retrieve geospatial information interacting with GeoServer, an open source server that can share and process geospatial information, exposing an API in the Presentation Layer. The two main protocols adopted by the Map Manager to communicate with GeoServer are WMS and WFS. In addition, the GeoServer caching system makes the platform more efficient in serving user requests. Figure 4 describes the communication between the Map Manager and GeoServer. The API hides useless details for which it is possible to (i) use default values and (ii) avoid undesired access to certain features (e.g. delete layers). The Map Manager (i) receives the user query through a Restful interface, (ii) translates the request into one of the GeoServer compatible formats (WMS or WFS), (iii) forwards the request to GeoServer and (iv) forwards the GeoServer's answer back to the user. The Map Manager interface adopts the following structure:

```
/mapmanager/:srs/:x1/:y1/:x2/:y2/:srv.:frmt
```

where `srs` is the coordinates systems (e.g. `EPSG:900913` or `EPSG:4326`), the variables $x1$, $y1$, $x2$ ed $y2$ specify the map bounding box, `srv` specifies one of the Smart Mobility services, and `frmt` specifies the format: an image (`.jpg`, `.png`) or a text file (`.json`, `.xml`).

The Urban Data Collector (UDC) is a controller module that collects and store the information acquired from vehicles. Each vehicle is equipped with an OBU. The OBU is connected to a smart device (i.e. smartphone, tablet) which rules the authentication of the user that is driving the vehicle in that particular moment, allowing a *many-to-many* relationship between users and vehicles. The vehicle authenticates through the Login procedure, and sends to the platform the information collected from the vehicle itself and the external environment.

The User Manager is responsible for registration, authentication, and accounting operations (e.g. password changing, activation emails). The user authentication is performed through the Login procedure. The procedure is composed of two basic operations: the former is requesting the login page, the latter is submitting the login information. The first one is optional, since the API allows to directly authenticate the user when there's no need to load a login form (e.g. authentication through third party applications). The authentication procedure can also be performed using Social Networks accounts through the OAuth protocol. The platform keeps track of the session through HTTP cookies. In this way, it is possible to control whether the user is equipped with an On-Board Unit (OBU). The OBU is a smart device installed on a vehicle that is able to collect information from the vehicle itself or the urban environment. The device is able to communicate with the platform in order to send the collected data, like speed, geographic coordinates, pollution levels, fuel consumption, etc.

The S^2-Move platform provides the following Smart Mobility services: Fleet Management, Warning Management, Traffic Monitoring, and Smart Parking. This section focuses on the Core Layer logics related to the Smart Mobility Services. Traffic Monitoring uses the information periodically sent by vehicles and stored into the database to estimate road traffic conditions. The overall procedure is asynchronous, in fact once the procedure starts, the algorithm is executed regardless the interaction between the user and the platform. The idea is to have threads, managed by the Akka framework [19] supported by Play!, which periodically wakes up and processes the data. Akka is a distributed and scalable architecture which can execute concurrent operations and is extremely fault tolerant. Akka introduces an evolution of the Java thread concept, called *actor*, which automatically manage concurrency and mutual exclusion. The actors can asynchronously communicate each other through the *mailbox*. When an actor receives an update message, it immediately starts to process the data. The Map Matching module reads the information sent by vehicles and finds the actual road segment the vehicle is moving on. The Traffic Manager reconstructs the full path of a vehicle on every road. Afterwards, the manager identifies the vehicle's lane by comparing the position of two subsequent points. Algorithm 1 describes the traffic estimation procedure supported by the Map Matching algorithm (Section IV-B) which discovers the road's name and identifier, given the vehicle's position.

Algorithm 1: TrafficEstimator

```
input  : vehiclesList on the current road
output: speedA, speedB, according to the road's lane
speedA, speedB, countA, countB ← $0$;
foreach vehicle ∈ vehiclesList do
    samples ← OrderByTimestamp(vehicle.samples);
    for $i \leftarrow 0$ to SizeOf(samples)-1 do
        current ← samples [$i$];
        next ← samples [$i+1$];
        if Orientation(current, next) $== 1$ then
            speedA ← speedA + current.speed;
            countA ← countA +1;
        else
            speedB ← speedB + current.speed;
            countB ← countB +1;
        end
    end
end
speedA ← speedA /countA; speedB ← speedB /countB;
```

B. Data Layer

The Data Layer offers a lower level interface to interact with the database and related functions. The database is managed through PostgreSQL DBMS with PostGIS geospatial extension. In order to comply with efficiency requirements, the Data Layer implements low-level algorithms like Map Matching. Besides all the tables needed by the Core Layer, the database contains a dump of the OpenStreetMap [9] roads tables imported through the Imposm tool.

The Map Mathing module maps the position of the vehicle to the road segment it is moving on. The platform may use this information to provide several services. One of them, in particular, is the Traffic Monitor which evaluates road traffic conditions. Generally, positioning errors affect the GPS information, due to the instrument accuracy and obstacles (e.g. low coverage, multi-path effect [6]) that, in the urban environment, may appear in case of tall buildings [5]. In literature there exist many different Map Matching implementations. The *point-to-point* is the simplest algorithm, which associates the geographical location to the closest node or link [4], [10], [20], while the *point-to-curve* algorithm associates the point to the closest edge [4], [20], [18]. The point-to-point algorithms are less precise than point-to-curve ones, especially when errors affect the GPS measurement. White et al. road intersections make Map Matching procedures more difficult [20]. The Map Matching algorithm implemented considers only urban paths, leading to worst performances compared to highways because of lower speed values of vehicles and sudden changes in the travel direction. Algorithm 2 describes the Map Matching procedure which assigns a point to a specific road section. The OpenStreetMap database provides the road segments. The algorithm is a point-to-curve implementation which firstly builds a bounding box surrounding the actual point, then

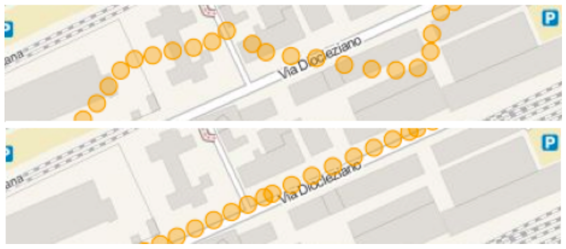


Figure 6. Map Matching corrects the positioning error.

considers only the roads which intersects this bounding box, and finally all the closest road segments (according to a certain threshold) are considered: the first is most likely to be the actual road section. The `st_expand` function builds a bounding box surrounding the point in order to intersect neighbor roads. The algorithm then considers the roads within the intersection and, through the PostGIS operator <#>, calculates the distance between the point and the bounding box of each road segment. This distance estimation, at this point, is not precise, but it's fast. In order to make the matching operation more reliable, the point-to-curve algorithm is then called through the `st_distance` PostGIS function.

Algorithm 2: MapMatching

```
input  : Geometric Point $P$, Bounding Box size $S$,
         Road Sections $sections$, Maximum Distance $D$
output: Road Identifier $RoadId$
bbox ← st_expand($P$, $S$);
closestBbox,closestSections ← set();
foreach $section \in sections$ do
    if intersects($section$, bbox) then
        closestBbox.add(section);
    end
end
foreach $section$ ∈ closestBbox do
    if distance($P$, $section$) $\leq D$ then
        closestSections.add(section);
    end
end
$RoadId$ ← sortByDistance(closestSections)[0];
```

The procedure described by the Algorithm 2 has been developed as a SQL function, making it more efficient from a computational point of view. Figure 6 shows the vehicle positions before and after the application of the Map Matching algorithm.

V. Simulation and Results

In this section, we present the results on the accuracy and efficiency of the Map Matching and Traffic Monitoring algorithms: we exploited realistic urban traffic data generated through simulations in the SUMO (Simulator of Urban MObility) environment [3]. A *good* Map Matching algorithm is responsible for *good* traffic estimation. Traffic Monitoring efficiency can be evaluated on time and accuracy parameters. In SUMO, we generated a fleet of vehicles moving in the urban environment using random trips. The current simulation assumes there are no obstacles that can interfere with the normal traffic flow. SUMO takes as input the road network and produces an `xml` file which describes the vehicles characteristics at different timestamps, i.e. speed, position (latitude and longitude), road identifier (according to the OpenStreetMap data source). The final step adds some noise to the data in order

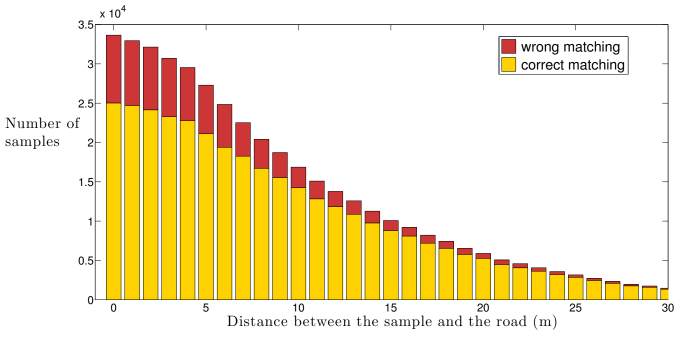


Figure 7. Map Matching accuracy: more than 78% of the speed samples are mapped to the correct road.

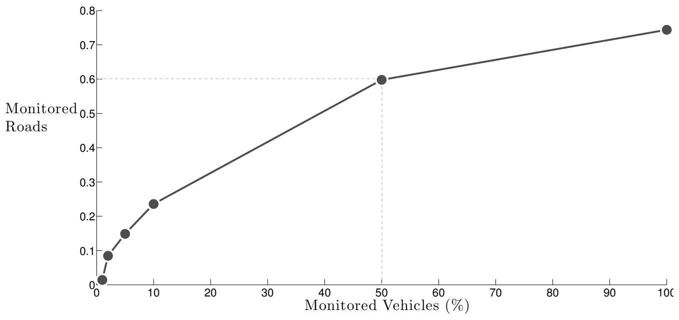


Figure 8. Traffic monitoring coverage: monitoring 50% of the vehicles provides enough information to traffic status estimation for 60% of the roads.

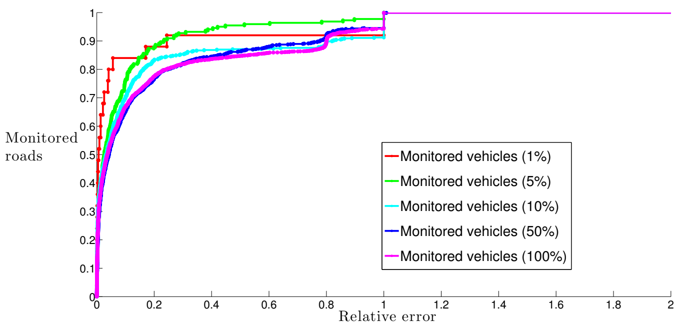


Figure 9. Traffic monitoring accuracy: CDF of the relative error.

to consider any possible GPS positioning errors, using an ad-hoc Java tool sending noisy data to the Urban Data Collector. The GPS position error can be approximated by a bi-variate Gaussian distribution with no correlation between the two variables [17] with a variance of 1.12 for the latitude (north-south variance) and 0.82 for the longitude[2] (east-west variance) according to positioning errors experimentally measured for smartphones and GPS devices [21], [5], [17]. Figure 7 shows the Map Matching accuracy: a correct matching is always over 78%, and the wrong matching decreases w.r.t. the distance between the sample and the actual road. Moreover, Figure 8 describes the relationship between the percentage of monitored vehicles and monitored roads. Then, according to both the monitored vehicles and the monitored road, we compute the relative error, as depicted in Figure 9: for about 80% of the monitored roads, we observed a relative error lower than 20%.

[2]Opportunely normalized considering the Java Random class specifics.

VI. Conclusions

In this paper we have presented the S^2-Move map-based platform, its use cases, design and implementation. We also provided some results to show the accuracy and efficiency of the Map Matching and Traffic monitoring algorithms implemented. Thanks to S^2-Move map-based platform we can provide smart mobility services to final users aiming at improving the urban mobility and city life.

VII. Acknowledgments

This work is partially funded by PLATINO (PON01_01007), SMART HEALTH (PON04a2_C), SIRIO (PON01_02425,) and S^2-MOVE (PON04a3_00058) projects funded by MIUR.

References

[1] The play! framework. https://www.playframework.com/.

[2] Batty et al. Map mashups, web 2.0 and the gis revolution. *Annals of GIS*, 16(1):1–13, 2010.

[3] Behrisch et al. Sumo-simulation of urban mobility-an overview. In *SIMUL 2011, The Third International Conference on Advances in System Simulation*, pages 55–60, 2011.

[4] Bernstein et al. An introduction to map matching for personal navigation assistants. 1996.

[5] Blum et al. Smartphone sensor reliability for augmented reality applications. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 127–138. Springer, 2013.

[6] Braasch et al. Multipath effects. In *Global positioning system: theory and applications*. Citeseer, 1996.

[7] M. di Bernardo et al. Distributed consensus strategy for platooning of vehicles in the presence of time-varying heterogeneous communication delays. IEEE, 2014.

[8] M. di Bernardo et al. Design, analysis and experimental validation of a distributed protocol for platooning in the presence of time-varying heterogeneous delays. IEEE, 2015.

[9] Haklay et al. Openstreetmap: User-generated street maps. *Pervasive Computing, IEEE*, 7(4):12–18, 2008.

[10] J. Kim. Node based map matching algorithm for car navigation system. In *International Symposium on Automotive Technology*, 1996.

[11] Köhler et al. Platforms for the internet of things–an analysis of existing solutions. *5th Bosch Conference on Systems and Software Engineering*.

[12] Marchetta et al. S2-move: Smart and social move. In *Global Information Infrastructure and Networking Symposium (GIIS), 2012*, pages 1–6. IEEE, 2012.

[13] Marchetta et al. Social and smart mobility for future cities: the s 2-move project. In *50th Annual Congress*. AICA, 2013.

[14] Marchetta et al. Trusted information and security in smart mobility scenarios: The case of s2-move project. In *Algorithms and Architectures for Parallel Processing*, pages 185–192. Springer, 2013.

[15] S. E. Matthews. How google tracks traffic. www.ncta.com/platform/broadband-internet/how-google-tracks-traffic/.

[16] Nawaz et al. Parksense: A smartphone based sensing system for on-street parking. In *Proceedings of the 19th annual international conference on Mobile computing & networking*. ACM, 2013.

[17] Singh et al. Stochastic analysis and behavior modeling of errors associated with global positioning sensor. In *Proceedings of Conference on Advances In Robotics*, pages 1–6. ACM, 2013.

[18] Taylor et al. Gps positioning using map-matching algorithms, drive restriction information and road network connectivity. In *NavSat 2001 Conference, Nice, France*, 2001.

[19] M. Thurau. Akka framework. *University of Lübeck*, 2012.

[20] White et al. Some map matching algorithms for personal navigation assistants. *Transportation Research Part C: Emerging Technologies*, 8(1):91–108, 2000.

[21] P. A. Zandbergen. Accuracy of iphone locations: A comparison of assisted gps, wifi and cellular positioning. *Transactions in GIS*, 2009.