Benchmarking real-time vehicle data streaming models for a smart city

Jorge Y. Fernández-Rodríguez [a,*], Juan A. Álvarez-García [a], Jesús Arias Fisteus [b], Miguel R. Luaces [c], Victor Corcoba Magaña [d]

A B S T R A C T

The information systems of smart cities offer project developers, institutions, industry and experts the possibility to handle massive incoming data from diverse information sources in order to produce new information services for citizens. Much of this information has to be processed as it arrives because a real-time response is often needed. Stream processing architectures solve this kind of problems, but sometimes it is not easy to benchmark the load capacity or the efficiency of a proposed architecture. This work presents a real case project in which an infrastructure was needed for gathering information from drivers in a big city, analyzing that information and sending real-time recommendations to improve
driving efficiency and safety on roads. The challenge was to support the real-time recommendation ser- vice in a city with thousands of simultaneous drivers at the lowest possible cost. In addition, in order to estimate the ability of an infrastructure to handle load, a simulator that emulates the data produced by a given amount of simultaneous drivers was also developed. Experiments with the simulator show how recent stream processing platforms like Apache Kafka could replace custom-made streaming servers in a smart city to achieve a higher scalability and faster responses, together with cost reduction.

1. Introduction

Today's cities face a growing demand of real-time services while at the same time new urban sensors produce new valuable information that has to be processed swiftly. In fact, smart cities are evolving into larger interconnected ecosystems with many applications and services that provide real-time information to users, such as the number of parking spaces available or the amount of water needed by plants in gardens [1]. As a consequence, the information systems of smart cities have to deal with massive incoming data from hasty diverse data sources while facing the chal- lenge of minimizing any possible loss of information. To process
data as they arrive, the paradigm has changed from the traditional all at once data handling procedure to stream processing, which grants a continuous and flexible way of processing data. A data stream is defined as a sequence of digitally encoded signals used to represent information in transmission. Smart cities produce huge volumes of varied datasets, which make up big data problems that need to be dealt with new techniques and applications. Those datasets are commonly clustered in data streams in order to get information processed and mined [2]. However, there
are still many challenges in big data applications, such as difficulties in data capture, data storage, data analysis and data visualization [3]. One of the main concerns of smart cities is the difficulty to achieve high-throughput stream processing to support a large number of simultaneous users.

The case study presented in this work explains how the HERMES project (Healthy and Efficient Routes in Massive open-data basEd Smart cities) [4] had to manage the challenging task of han- dling large amounts of real-time vehicle data to make personal- ized safe driving recommendations. In order to develop the best state of the art platform that fulfills a near real-time service for
them, several proposals of smart cities infrastructures were analyzed, including the technologies they used, the parameters to set them up and the real-scale tests performed on them. To the best of our knowledge there is no benchmark to test the efficiency of the smart cities' infrastructures dealing with real-time data pro-

*

cessing, so this paper proposes a case of study consisting on a real use case scenario (a real-time information service for drivers in the city) and a client-side simulator to check the number of concurrent drivers that can be served in near real time by the infrastructure. These two contributions can help future works to benchmark their proposals.

The remainder of this paper is organized as follows. Section 2 analyzes the architectures proposed in some of the most important smart cities that uses real-time processing and storage of data streams. Section 3 presents the SmartDriver case study for this architecture. Section 4 outlines the simulator used in this work. Section 5 exposes the streaming server and the implemented alternatives. Section 6 reports the results of the eval- uated streaming servers and improved configurations to support more simultaneous users. Conclusions and future lines of work are presented in Section 7.

2. State of the art: smart cities infrastructure

In recent years, smart cities are increasingly producing new datasets due to the development of ubiquitous computing and the rise of the Internet of Things (IoT). There are currently 9 billion interconnected devices, and the number is expected to grow to 24 billion devices by 2020 [5]. Therefore, their information systems face a big data challenge because they must manage massive, dynamic, varied, detailed, inter-related, low cost datasets that can be connected and utilized in diverse ways [6]. Moreover, smart cities need to be able to combine services offered by multiple stakeholders and scale to support a large number of users in a reliable and decentralized manner. The success of the smart city depends heavily on the architecture of its information systems, and there have been some successful experiences in this area.

In Spain, SmartSantander [1] is a success case of IoT infrastructure including wireless nodes that measure carbon monoxide, light intensity, noise, temperature, and car presence. To deal with the increasing load of information that is continuously generated by the IoT deployment rolled-out in the city of Santander, a software platform enabling the management of the data has been designed and implemented [7]. The SmartSantander platform follows a three-tiered architecture, where the server tier hosts IoT data repositories and services, and it uses virtualization in a cloud infrastructure in order to ensure high reliability and availability of all the components and services. In this architecture, systems communicate through a topic-based publish-subscribe event bus. This architecture is designed to be asynchronous, distributed and multi-party, but subscribers need to be notified when an event is received, which involves an additional burden on the streaming server. Santander has been used also as a IoT experimental testbed for the platform CiDAP (City Data and Analytics Platform) [8] to set the stage for a big data platform toward smart cities.

Barcelona Smart City [9] is another example of a successful smart city, being recognized as the 2nd world's smartest city, prevailing over New York (USA), London (UK) or San Francisco (USA), by the smart cities top ranking Smart Global City 2016 [10]. Barcelona has a powerful platform with ubiquitous infrastructures. Its technology enables the interconnection of city elements and lets them interact effortlessly with each other and with their administration through electronic means. The Barcelona smart city model identifies 12 areas with initiated projects: environmental, ICT, mobility, water, energy, waste management, nature, built domain, public space, open government, information flows, and services. The infrastructure uses a platform called Sentilo[1] designed following the publish-subscribe pattern. It is responsible for aiding the city in bringing all of its sensor data together. In this case, it uses Redis[2] as the data streaming aggregation system, but although Redis is a mature and widely-used technology, it is based on an in-memory database, which means that it needs more memory than the incoming data requires, and the number of producers and consumers can affect its performance. Additionally, if a Redis instance restarts or crashes, all data between consecutive snapshots will be lost.

A smart city event-driven architecture is presented in [11]. It allows the management and cooperation of heterogeneous sensors for monitoring public spaces. Its design is structured in knowledge processors and semantic information brokers, implementing a publish-subscribe paradigm. A knowledge processor receives notifications from a semantic information broker on the subscribed events. It also provides composite events, which are published when a certain pattern of events occurs, preventing subscribers from being overwhelmed by a large number of raw event publications. The authors use a subway station scenario as a testbed for the architecture, enhancing the detection of anomalous events and simplifying both the operators' tasks and the communications to passengers in case of emergency.

Oulu Smart City [12] has created a middleware for ubiquitous-computing researchers, offering opportunities to enhance and facilitate communication between citizens and government. This middleware uses asynchronous communications based on the publish-subscribe model. An important part of the communications solution is the content-based routing of messages, which enables the accurate allocation of information for subscribers. For example, a message can be directed into a certain logical or physical space, such as to all users in a marketplace who have been there for ten minutes. This event-based communication and content-based routing was implemented with the open source Fuego-architecture.[3] However, Fuego had not matured enough to be used in real world deployments. Client support in Fuego was limited, and the stability issues it suffered did not generate enough trust among applica- tion developers, so it was abandoned and a new middleware was implemented using the RabbitMQ message broker model, as explained in [13].

Within the Asian continent, one of the most important smart cities is Songdo, in South Korea, which was built from scratch to be a ubiquitous eco-city. Ubiquitous cities (or U-cities) are considered an evolution of digital cities where all information systems are linked, and virtually everything is linked to an information system [14]. It creates an environment that connect citizens to any services through any device. Songdo is known in the urban studies literature as a model smart city and an example of testbed urbanism on a grand scale. However, this top-down infrastructure is not free from problems, as it is considered to promote private business interests while ignoring society's needs [15]. Asian emerging u-city projects have their own ad-hoc service platforms in which sensors and devices are connected to servers dedicated to a particular application domain, and networks are separated from each other.

Singapore is another blooming smart city. Singapore's Smart Nation project, launched in November 2014 [16] and relies heavily on cloud computing in its infrastructure. The aim is to get Smart Nation ready in less than ten years, in the Prime Minister's wishes.

Focusing on smart transportation challenges, in [17], it was de- veloped an evaluation data streaming framework, including a traf- fic simulation, to check the influence of parameters of road net- works and traffic scenarios as well as data mining algorithms in order to estimate the state of the traffic. However, there is no in-

[1] http://www.sentilo.io/xwiki/bin/view/Sentilo.About.Product/Whatis 2017-03-13).

[2] https://redis.io/ (Visited 2017-03-13).

[3] http://www.ubioulu.fi/en/UBI-middleware (Visited 2017-03-15).

formation about the underlying infrastructure used to support the stream management or the stream rates from vehicles sending and requesting information that could support.

Traffic jams is one of the most common problems in modern cities. Thanks to new urban sensors scattered over the city, developers could subscribe to these real time data streaming sources to predict traffic hazards in routing. Dynamic route planning systems are able to give user alternative routes in real time. In [18], it is used the data provided by SCATS sensors in Smart Dublin, to make traffic predictions and suggest different routes to avoid congested streets.

Problems appear when it is increased the rate of data produced by sensors, or when the number of sensors arise. In the recent paradigm of Social Sensing, it is proposed an integrated model in which citizens them-selves are turned into sensors, thanks to the use of smart phones and social networks [19], but unfortu- nately, there are no details on how many social streams of human- generated data are able to process in real-time or the system ar- chitecture used to tackle the problem.

A particularly interesting case is the real-time tracking of dangerous good transport. In [20], it is proposed an architecture for real-time collection of telemetry and event data conveyed by the vehicle on-board system, allowing to monitor up to 4600 oil trucks sending data every 5 s, but although it is a good solution to solve their truck fleet monitoring, its middleware does not seem easily scalable to a higher number of vehicles, as it is not a fully distributed solution.

The importance of real-time data management in transport could be also seen in accident prevention works like [21], where having real-time traffic variables like traffic volume, average speed, standard deviation of detector occupancy or volume difference between adjacent lanes could prevent crashes, or even after a first ac- cident, help taking actions to decrease the likelihood of secondary crashes [22].

Parallel and distributed systems are needed in smart cities in order to address massive datasets and provide efficient real-time services [23]. Distributed publish-subscribe architectures are one of the most used paradigm in smart cities projects. A message-oriented middleware is essential for this kind of platforms that involve asynchronous data exchange, decoupling senders from receivers and providing the flexibility to defer tasks to separate pro- cesses. Furthermore, there is no need to maintain any in-memory information about senders and receivers [24], hence the hardware requirements are affordable. However, as far as we know, no proposal describes a method to benchmark the capacity of the system architecture in terms of the number of users that can be supported simultaneously, as well as the scalability of the architecture.

Therefore, we implemented and benchmarked two publish-subscribe architectures for our case study, described in Section 5.

In many cases it is difficult to evaluate the effectiveness of proposed solutions, because only specific parts of the infrastructure are exposed. Other works explore big data platforms for smart cities, but they only introduce high level platform architecture designs [25,26]. There are also works which introduce an empirical-based framework to offer a holistic picture of how smart cities can be analyzed. In [27], it is taken into account the Seoul and San Francisco smart cities to understand the process of building a smart city.

3. The SmartDriver case study

In order to test smart cities infrastructures, a real-world use-case scenario has been considered: the SmartDriver mobile application [28,29], which is used to monitor the location and a collection of driving parameters (speed, acceleration, etc.) as well as a series of biometric information (e.g., heart rate) to analyze the stress level of drivers and assist them to improve their driving efficiency, minimizing the waste of fuel, reducing their stress by notifying them about the profiles of the surrounding drivers (i.e., aggressive, calm, etc.), and warning them when speed limits are exceeded. In order to provide this information, SmartDriver has to send periodically the current location and user driving information to the server. Then, it requests information about the surrounding drivers and the road (e.g., road type and speed limit). During the driving, SmartDriver sends two types of events:

- Vehicle Location. It contains information about the location of the vehicle and driving variables measurements (600 bytes on average). It is posted every 10 s in order to reduce battery usage. It includes:
  - Timestamp of the sample.
  - Latitude and longitude of the vehicle in that moment.
  - An estimation of the accuracy of the location.
  - Instantaneous vehicle speed.
- Data Section. It contains detailed information about the vehicle and the driver in the last 500 m of road (5000 bytes on average). It includes one sample per second containing the vehicle location, the vehicle speed and certain information from driver heart rate, such as R-R intervals.[4] It also includes a summary with the following statistics computed for the whole 500 m driving section:
  - Maximum, minimum, average and standard deviation of vehicle speed.
  - Number of times a sharp acceleration or deceleration was detected.
  - Average acceleration and deceleration.
  - Average and standard deviation of R-R intervals.
  - Average and standard deviation of heart rate.
  - PKE (Positive Kinetic Energy) [30], which is an indication of the aggressiveness of driving and depends on the frequency and intensity of positive accelerations, computed as:

$$\frac{\sum(v_f^2 - v_s^2)}{x} \quad when \quad \frac{\Delta v}{\Delta t} > 0 \tag{1}$$

    where:
    - v_f: Final speed
    - v_s: Initial speed
    - $\frac{\Delta v}{\Delta t} > 0$: For positive acceleration only
    - x: Distance travelled

The SmartDriver application encodes each event using JSON and compresses the result with ZIP before transmission in order to eliminate the high internal redundancy these events exhibit. Similarly, every second during the driving, SmartDriver requests from the server information about the surrounding environment:

- Surrounding drivers. Inside the streaming server, all the vehicles' locations are analyzed to determine moving vehicles close to each SmartDriver connected to the platform. Two SmartDrivers are considered to be close if they are separated no more than 100 m, so that the SmartDriver application has enough time to advise the driver if required. In order to save resources, the current position of all drivers is hold in a memory database in the streaming server only for those that keep moving. Drivers that remain still for more than 60 s are removed from the database.

- Current road. It contains information about the road type and the speed limit. It is used by SmartDriver to warn drivers that exceed the speed limit and to evaluate their driving style. This message is 100 bytes long on average.

[4] https://courses.kcumb.edu/physio/ecg%20primer/normecgcalcs.htm#TheR-Rinterval (Visited 2017-03-19).

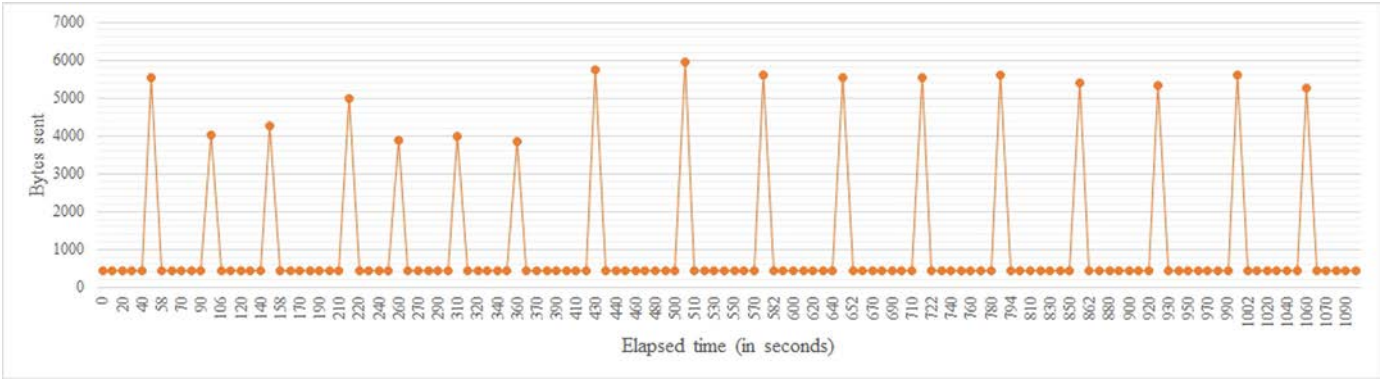


Fig. 1. Bytes sent covering a pre-defined route at 50 Km/h.

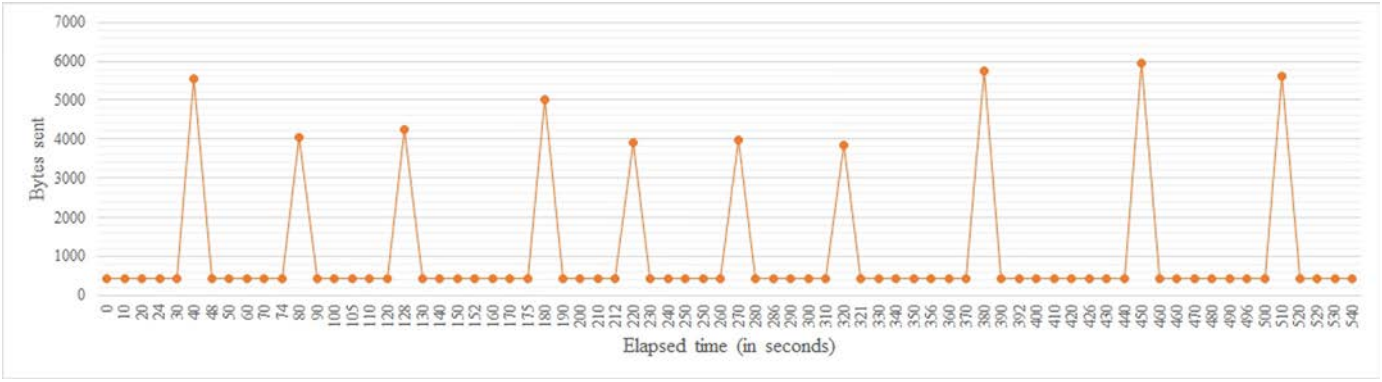


Fig. 2. Bytes sent covering a pre-defined route at 100 Km/h.

Table 1
Relationship between speed and data traffic.

Driving speed	No. of messages	Bytes sent
100 Km/h	73	74,784
50 Km/h	129 (+76.71%)	134,017 (+79.20%)

It is important to show that variations on the speed of the driver affect the amount of data transmitted, so the server-side of the infrastructure must resist any driver's scenario. In a study with real users, it was examined a range of journeys from 1 Km to 30 Km, bearing in mind that all drivers have their own driving behaviour that affects the speed. Hence, the realm of data transmitted for each SmartDriver varied from almost 23 kilobytes for the shortest journeys to about 1 megabyte for the longest journeys. Figs. 1 and 2 represent the amount of data sent along the same pre-defined route of approximately 10 Km at different constant speeds (50 Km/h and 100 Km/h). The faster driver finishes before and sends less data than the slower one. In the figures, the lower marks correspond to vehicle location messages which are far more numerous but smaller, while higher marks correspond to data section messages, which are less frequent but larger.

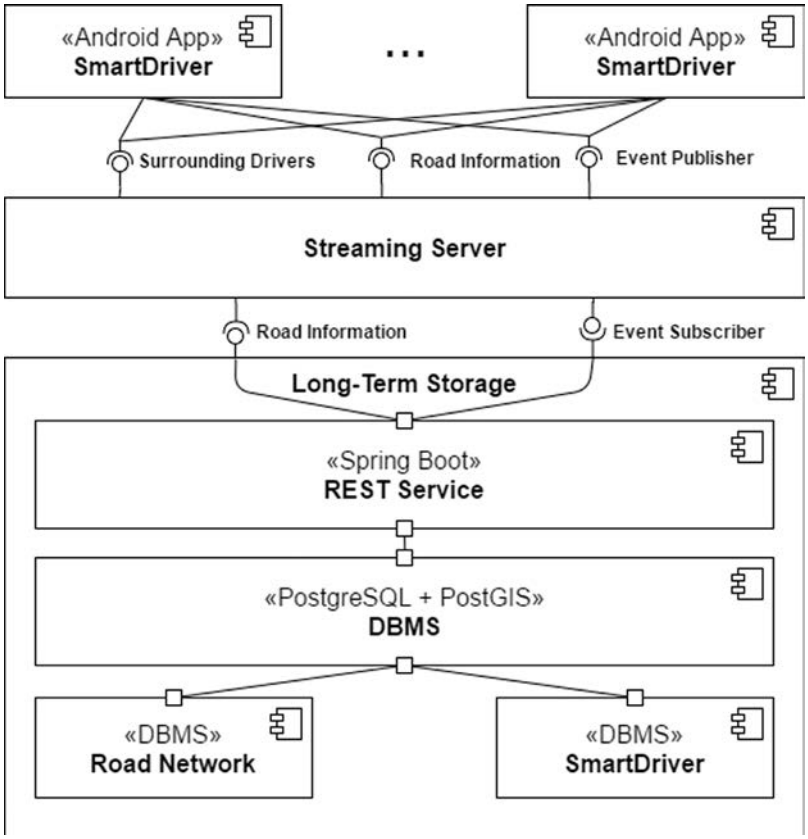
Table 1 shows the number of messages and total bytes sent considering the same pre-defined path, but two different driver speeds.

Regarding response time when sending data to the server-side, and taking into consideration current high-speed network coverage and the fact that mobile data communications are also limited in order to save battery, a reasonable time delay of 5 s has been considered between the instant a SmartDriver location is sent to the server and the instant the ACK response is received.

The server-side of the infrastructure is composed of two layers (Fig. 3): a streaming server component that is responsible of receiving all the events from all the SmartDriver users and answering real-time requests regarding the surrounding driver, and a long-term storage component that is responsible for storing the information to be used on offline analysis and answering requests regarding the road network. The bottleneck of the server-side is the streaming server component, because it has to deal with data streams and real-time requests from many SmartDriver applications. Two different approaches has been implemented for this component, which are described in Section 5.

The long-term storage component is not considered a bottle- neck in this scenario because delaying the final data storage of the events is not considered harmful. Furthermore, the long-term storage does not require to consume the vehicle location events because their information can be recovered from the data section events, and waiting some seconds until a data section event arrives from the SmartDriver does not affect the system. Thus, in our tests, the workload on the long-term storage component has been around a 10% of the streaming server processing. Therefore, it was resolved to use a traditional relational database management system (PostgreSQL[5]) with a spatial data management extension (PostGIS[6]). PostgreSQL manages two databases: a road network database created from OpenStreetMap[7] data and a SmartDriver application database that stores the information received in the data section events. The requests are handled by a REST service developed and deployed using Spring Boot.[8] Even though

[5] https://www.postgresql.org/ (Visited 2017-03-19).
[6] http://postgis.net/ (Visited 2017-03-19).
[7] https://www.openstreetmap.org/ (Visited 2017-03-18).
[8] https://projects.spring.io/spring-boot/ (Visited 2017-03-19).



this component has all the advantages of a relational DBMS, it was observed that this component does not scale horizontally easily. Therefore, in order to keep all the system layers equally scalable, a non-relational database alternative should be considered for the long-term storage component.

4. The simulator

In order to test the performance and the strength of the developed SmartDriver platform, a large number of people using the mobile application was necessary. However, despite the application being free of charge at Google Play[9], it was difficult to achieve enough engaged Android users that actively used the application. To overcome this problem, a configurable simulator has been implemented, which is able to create thousands of simulated SmartDriver users, each of them generating data with the same pattern as the original Android application.

There are already some spatio-temporal simulators in the literature. In [31], it is dealt with the generation of network-based mov- ing objects and tries to be very precise on the capacity of roads and speed (external objects decrease the speed of the moving objects in their vicinity), whereas in [32], it is described a microscopic traffic simulation package called SUMO, aimed to help to investigate urban mobility research topics. However, none of them focus on simulating a high volume of simultaneous drivers.

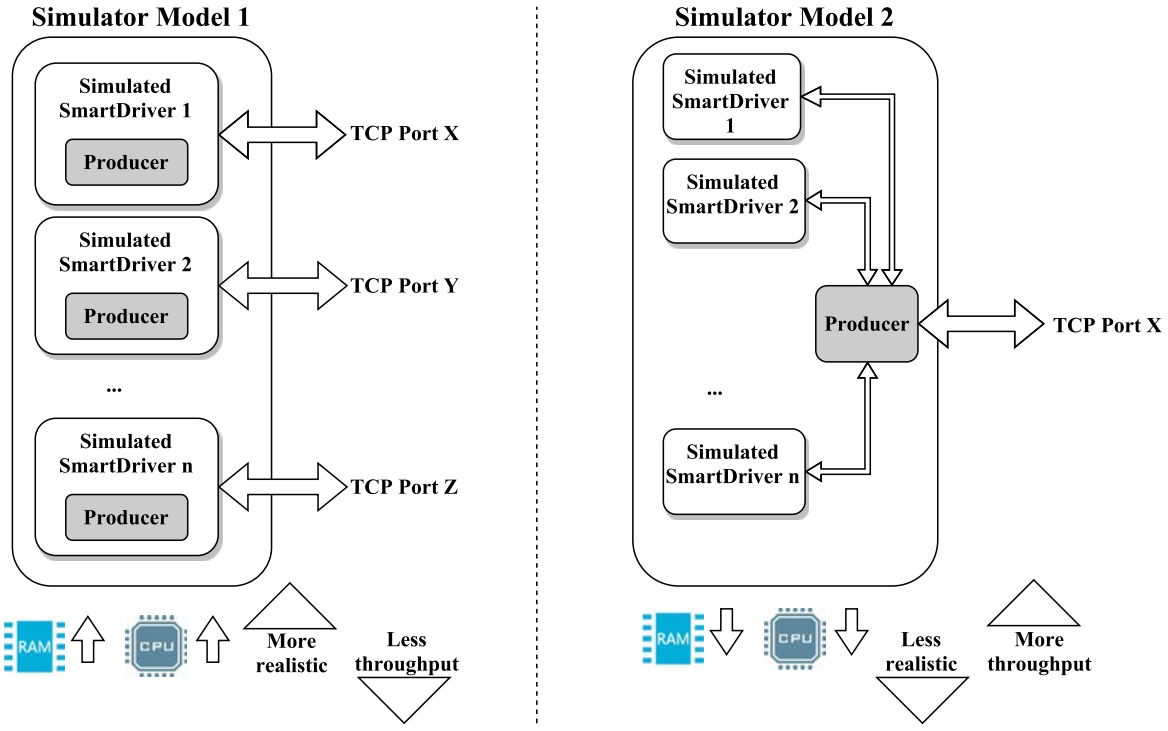
Within the framework of HERMES project, HERMES-Simulator, which is open source and available at https://github.com/hermes-smartcity/hermes-simulator , generates concurrent SmartDriver users driving along a real road network extracted from Google Maps[10] or OpenStreetMap, although not with microscopic traffic or a very realistic road capacity model. Each simulated driver runs in a separate thread within the simulator and its behavior can be considered, from the point of view of the data it sends and receives, equivalent to an actual driver using the Smart- Driver application. Although it involves a significant overhead in terms of CPU, memory and use of network connections in an attempt to be closer to reality, the simulator was designed to replicate the publisher/producer, as well as the subscriber/consumer modules on each simulated SmartDriver. The two simulator architecture alternatives are shown in Fig. 4. Simulator Model 1 was chosen for the benchmarks because it emulates the real scenario better, where SmartDrivers cannot share a TCP connection since each driver uses its own mobile device.

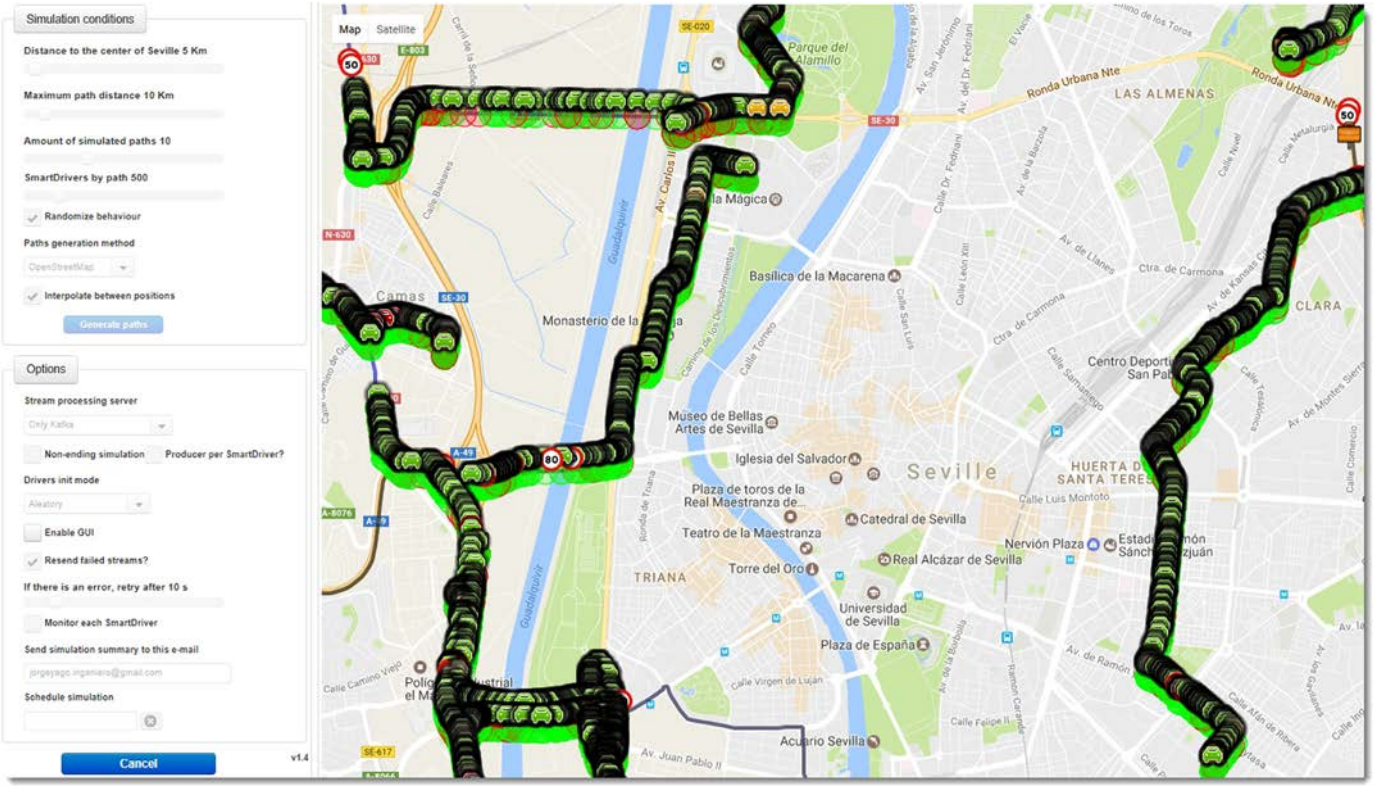
This simulator was designed to generate tracks from the outskirts of a city to the center itself, because this is one of the most common cases of travels nowadays, but it could also generate tracks inside the center and even only on the outskirts. Fig. 4 shows the simulator executing a user defined configuration. In our simulator, the influence among drivers is graphically represented changing the color of the circle that represents their proximity area. As a reference, a video with a running simulation can be seen at: https://goo.gl/V3rQMb.

By using the control panel, it is possible to configure a wide range of scenarios, to modify the average path lengths and the simulation speed, and to increase or reduce the amount of simulated SmartDrivers driving through each generated path. Additionally, the service that generates simulated paths is configurable, allowing the use of the Google Maps API or an internal service built on top of OpenStreetMap cartography and provided by the HER-

[9] https://play.google.com/store/apps/details?id=uc3m.enti.smartDriver 2017-03-15).

[10] https://www.google.es/maps/ (Visited 2017-01-08).





MES platform. Because these services offer the minimum set of points that define the path, the simulator is configured to interpolate points in the path and thus achieve enough resolution along the journey. This is necessary for computing the second-by-second position of simulated drivers. All the generated paths consist of a series of sections, each one with its own speed limits (Fig. 5).

All simulated drivers will have their own random factor that alters their speed through each section. This random factor varies the speed from 50% slower to 50% faster, but always with a minimum speed of 10 km/h in order to ensure that the simulation is completed within a reasonable period of time. This way, if a section is limited to 50 km/h, the speed of simulated drivers will

Table 2
Traffic and inhabitants in Seville at rush hour.

Year	Rush hour vehicles	Inhabitants	Ratio
2015	66,634	693,878	10.41
2014	59,516	696,676	11.71
2013	56,934	700,169	12.30
2012	57,770	702,355	12.16
2011	56,614	703,021	12.42
2010	51,933	704,198	13.56
2009	71,758	703,206	9.80
2008	78,628	699,759	8.90
2007	86,485	699,145	8.08
2006	90,710	704,414	7.77
2005	83,359	704,154	8.45

range from 25 km/h to 75 km/h. The heart rate information is also simulated for each driver with a value that starts at 70 beats per minute and is influenced by a random factor that ranges from 10% lower to 10% higher.

Since drivers are simulated, there is no way to measure their actual stress, so certain custom conditions have been added to reproduce stressful situations. Thus, sudden increases or decreases of speed, sharp turns, or the proximity of stressed drivers cause a weighted degree of stress that is accumulated, leading to an increase of heart rate, a decrease of R-R intervals and different levels of stress. Similarly, relaxing situations like straight roads, continuous speed, or the absence of other drivers around, cause a gradual decrease of heart rate, incrementing R-R intervals and lower levels of stress.

Regarding the rest of options, it is possible to send the events to two different versions of the streaming server component explained in Section 5. Moreover it has the possibility to start to simulate all the SmartDrivers at once, randomly or following a lineal progression that starts a 10% of the drivers every 10 s, which means that 100 s are needed to have all the simulated drivers running. This is the approach used on the comparative tests carried out in this paper in order to study the performance of each system under an increasing load of clients.

In order to be as close as possible to the original mobile application, the simulator has been implemented in Java 8 and creates independent threads with the same logic the SmartDriver application implements, particularly with respect to the communications with the streaming server. Since the simulator creates thousands of threads, enough CPU processing power as well as a big quantity of free RAM is needed. Therefore, the simulator had to be distributed across multiple machines to achieve enough number of simulated SmartDrivers. As a reference, an i5 desktop computer with 8GB RAM could run up to 20,000 simultaneous SmartDrivers.

Even though this simulator is specific for the SmartDriver scenario, its model can be easily expanded to send additional information regarding each driver and to request different information from the server-side. Furthermore, implementing the receiving component of the server-side that stores the event information is quite simple. Thus, the simulator can be used as a benchmarking tool to perform stress tests on a smart city infrastructure. Our particular challenge was to estimate the hardware infrastructure needed to support a medium-sized city such as Seville (Spain). Seville city is close to 700,000 inhabitants[11] and in 2006 there were more than 90,000 drivers commuting at rush hour (from 14:00 to 15:00)[12]. Table 2 shows these data during the last years, where the effects of the last economic recession can also be noticed.

[11] http://www.ine.es/jaxiT3/Datos.htm?t=2895 Visited (2017-03-15). [12] http://trafico.sevilla.org/imd.html Visited (2017-03-15).

5. Streaming server

Considering that the infrastructure was intended to be used on a smart city, it was necessary to implement a publish-subscribe message platform that could serve as many simultaneous users as possible and almost in real-time. Besides, it also had to be scalable and interconnectable with other smart cities. Although there are several well-known alternatives that fit the requirements, like ActiveMQ, [13] RabbitMQ [14] or Kestrel, [15] Apache Kafka has emerged in the last few years as a powerful and capable platform for building real-time streaming applications [33]. It is horizontally scalable, fault-tolerant, fast, and nowadays runs in production in important companies such as LinkedIn, Twitter, Netflix or Spotify among oth- ers. [16] We therefore consider that Apache Kafka is the most ade- quate alternative for our case study described in Section 3.
In order to compare the efficiency of different infrastructures, two options were implemented and explored: a solution based on the Ztreamy framework [34] and another based on Apache Kafka Streams [35].
In both cases, the same data format was used to send the information from SmartDriver and to receive it from the Streaming Server. As detailed in Section 3, each SmartDriver will send two types of information: vehicle locations and data sections, and will receive data about surrounding vehicles.

5.1. Ztreamy-based streaming server

By the time the architecture of the HERMES project was designed, there was no data streaming solution that fitted its needs. Therefore, an ad-hoc solution built on top of the Ztreamy HTTP-based publish-subscribe system was developed. This solution consisted in several stream processing entities as well as REST services whose operations were based on a short-term geographical data base. Since the system needed to handle a high rate of HTTP requests from the SmartDriver mobile application, the NGINX[17] open source web server was deployed to act as a load balancer. NGINX shows superiority in handling concurrent connections, response time and use of resources compared to the Apache HTTP server [18] or Lighttpd [19], and it is considered to be the most efficient
and lightweight web server today [36]. Fig. 6 shows the architec- ture of the Ztreamy-based stream server from the point of view of
the SmartDriver application. The different components communicate through HTTP. They may run on a single server machine or they may be distributed on several server machines.

The stream server consists of the following main components:

- Load balancer: In order to increase the number of SmartDrivers that Ztreamy is able to handle, an NGINX server applies HTTP load balancing and distributes the requests among the collectors.
- Collectors: Ztreamy servers to which the SmartDriver application posts data. These servers validate the received data items and orchestrate the interactions with other services needed to handle them. They are also responsible of responding with feedback data.
- Main stream: Data items received by the collectors are then aggregated into the main stream, which is managed by a separate Ztreamy server.

[13] http://activemq.apache.org/ (Visited 2017-02-15).
[14] https://www.rabbitmq.com/ (Visited 2017-02-15).
[15] https://github.com/twitter-archive/kestrel (Visited 2017-02-15).
[16] https://cwiki.apache.org/confluence/display/KAFKA/Powered+By/
[17] https://www.nginx.com/ (Visited 2017-03-01).
[18] https://httpd.apache.org/ (Visited 2017-03-08).
[19] https://www.lighttpd.net/ (Visited 2017-03-08).

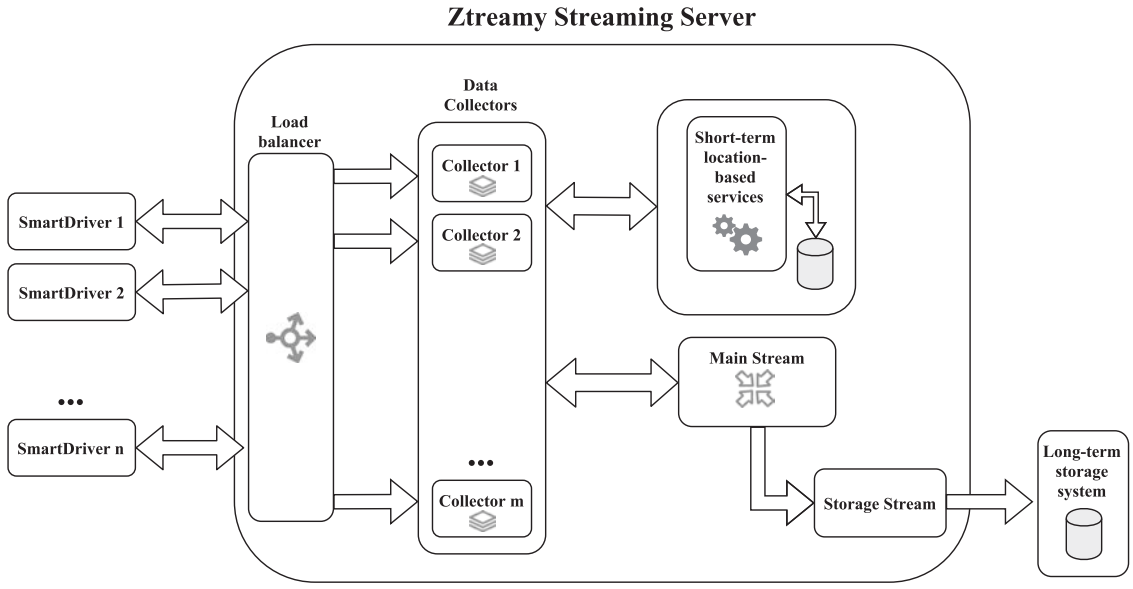


Fig. 6. Partial view of HERMES Ztreamy streaming server infrastructure.

- Storage stream: This stream filters the data items that do not need to be stored out of the main stream. The HERMES servers that manage data persistence consume this stream in order to receive the data they have to store.
- Short-term location-based services: the streaming infrastructure needs to perform some real-time computations and keep some short-term data, like information about surrounding drivers.

5.2. Kafka-based streaming server

Apache Kafka [6] is an open-source distributed streaming platform that was initially developed by LinkedIn in 2010 and is now maintained by the Apache Software Foundation. Written in Scala and Java, it implements a publish-subscribe messaging system that is designed to be fast and scalable. Applications can publish and subscribe to streams of records, with fault tolerance guarantees and the possibility of processing streams as they occur. Since Kafka does not use HTTP for ingestion, it delivers better performance and scale. As other publish-subscribe messaging systems, Kafka stores streams of records in categories called topics, and each record is basically made up of a key-value pair with a timestamp. In our implementation those values are JSON objects with the driving telemetry sent by SmartDriver. A topic is a category for grouping of messages of a similar type, so one topic for each type of information is needed: Vehicle Location, DataSection and Surrounding Vehicles.

Producers in the Kafka architecture publish messages to a topic and consumers subscribe to topics in order to receive those messages. In this scenario, the SmartDriver mobile application undertakes the producer role, publishing all the information about their location as well as driver's heart rate information and stress related data. This information is sent periodically to the VehicleLocation and DataSection Kafka topics. Then, those topics are consumed by a Java application using Kafka Streams. The application, called Data Analyzer, processes the vehicle locations and data sections streams to produce information about surrounding drivers for each user, which is then sent to the SurroundingVehicles topic. SmartDriver application also plays the role of consumers, subscribing to the SurroundingVehicles topic and consuming the published messages by pulling data from the brokers. An overview of the process is shown in Fig. 7.

In order to set an optimal configuration, message size limits have been set in Kafka after an analysis of the messages sent by the SmartDriver application. As commented in Section 3, data sec- tion messages are the largest pieces of information sent by Smart- Driver. They are aggregations of vehicle locations and heart rate information, as well as some statistical calculations, but no single one goes over 1MB. On the other hand, if there is any failure that makes sending any given message at its due time impossible, it is temporary stored and SmartDriver tries to send it later, together
with the next messages. Given this fact, Kafka brokers and produc- ers have been configured to accept messages up to 2MB. A repli- cation factor of one was used in all cases, since performance de- creases as the replication factor increases and because failure tol- erance measurement is out of the scope of this paper.

To evaluate the Apache Kafka solution, we have set up three different Kafka based scenarios:

- Kafka configuration 1: This is the minimum configuration, consisting of a single node with a single broker, as can be seen at Fig. 8. In this situation, all simulated SmartDrivers send their data to a unique node. One single broker can handle thousands of the incoming streams seamlessly. The main limitations are network capacity and server write throughput. However, this is only half of the problem. Our simulated SmartDrivers act also as consumers, and therefore they request information from the broker. Apache Kafka uses partitioning as the unit of parallelism to serve consumers. Each partition is related to a directory in the server file system, so more partitions lead to more open file handlers, which could turn to be a configuration issue.
- Kafka configuration 2: This setting consists of a single node with multiple brokers, as can be seen at Fig. 9. Despite still run- ning on a single node, a multiple-broker architecture is a first step towards distributing the system. Zookeeper is responsible for managing the load over the nodes, distributing the brokers among them. This architecture is able to handle more produc- ers. However, this is still a basic configuration and, since Kafka is distributed in nature, a cluster typically consists of multiple nodes with several brokers. The results shown in this paper are based on a two-brokers architecture in order to reveal the first step up in scale in relation with the one-broker solution.

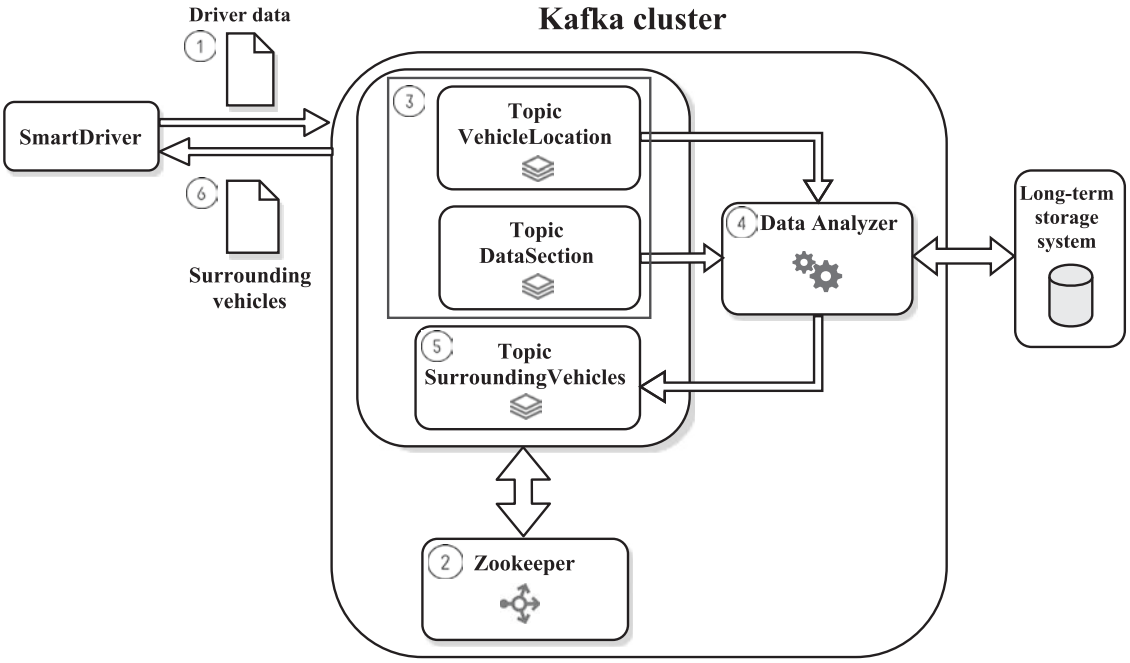


Fig. 7. Overview of information flow using Kafka.

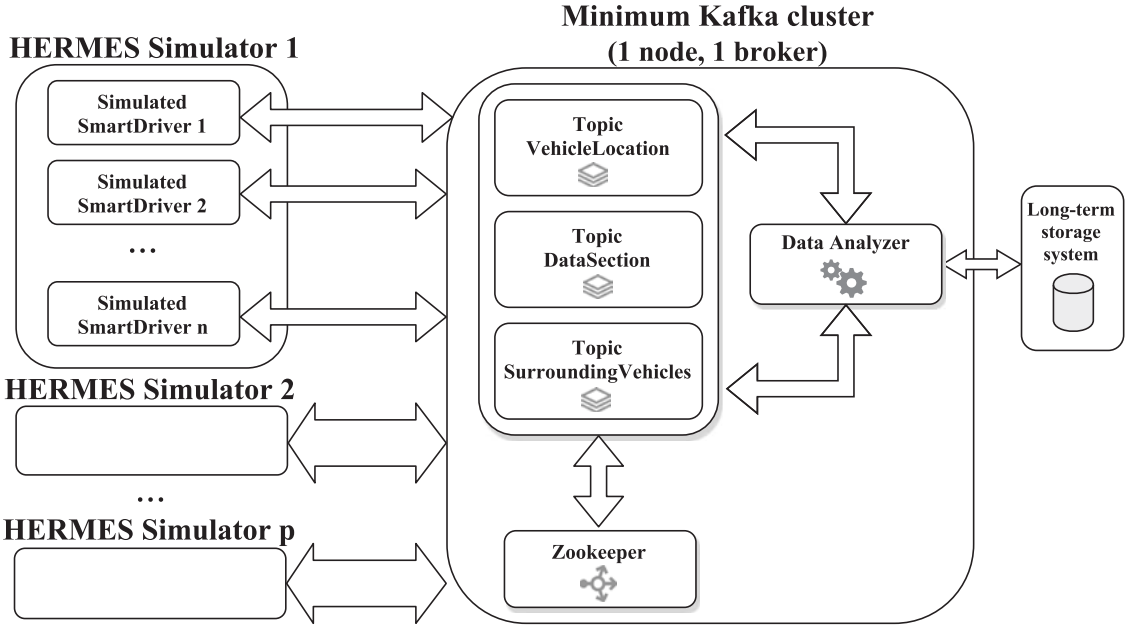


Fig. 8. Kafka configuration 1: Minimum Kafka cluster consisting of only one node with one single broker.

- Kafka configuration 3: This is an intermediate model that consists of multiple nodes with multiple brokers, managed by a replicated Zookeeper inside each node.
- Kafka configuration 4: This architecture consists of multiple nodes with multiple brokers with Zookeeper as an independent unit, as can be seen at Fig. 10. Although it is possible to setup a highly-distributed solution, our experiments are based on a 5 server configuration, being 3 servers for the Kafka distributed network. This configuration is enough to support our study case, yet scalable. Moreover, it is easy to manage and monitorize to prevent possible overloading of any node and to take corrective actions.

6. Evaluation and results

A series of simulations were carried out in order to validate and compare the alternative architectures. The three different Kafka configurations previously introduced were tested, as well as a predefined Ztreamy setup that was used as reference. To avoid performance fluctuations, server and clients were set to maximum performance and the systems were booted directly into terminal mode to avoid interferences from desktop applications. Moreover, to minimize network problems such as rejected connections, firewalls were disabled for all the machines involved. Furthermore, at least 5 simulations were performed for each setup and the worst case is presented.

The tests were performed using 2 different types of servers:

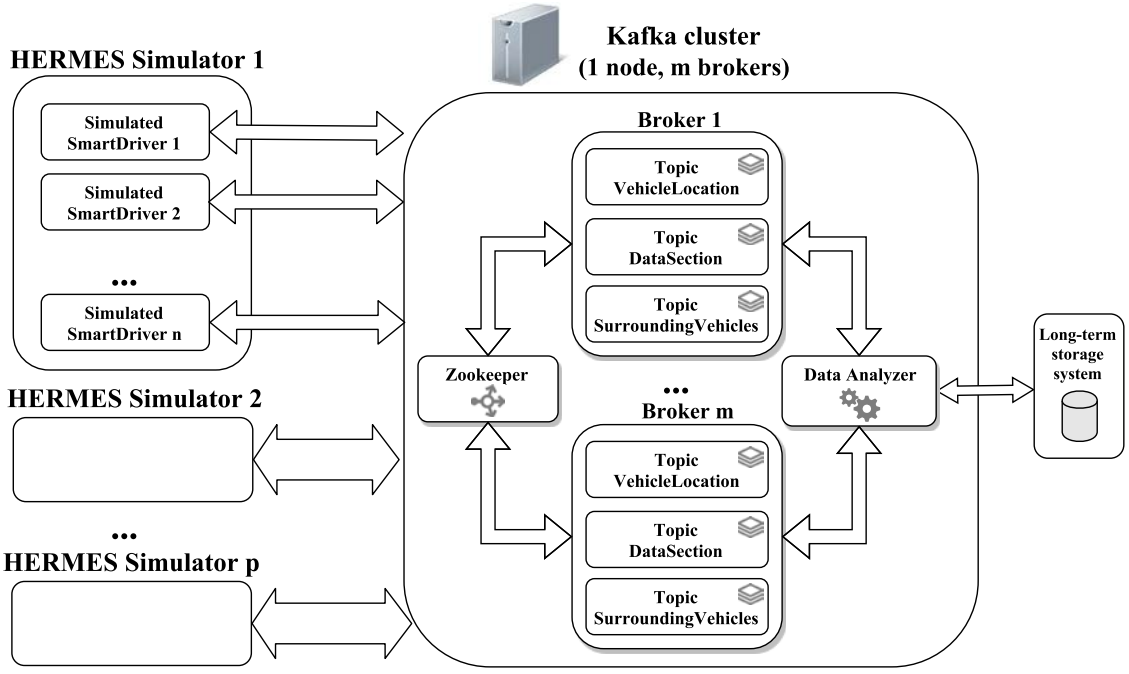


Fig. 9. Kafka configuration 2: Kafka cluster consisting of one node, using different number of brokers.

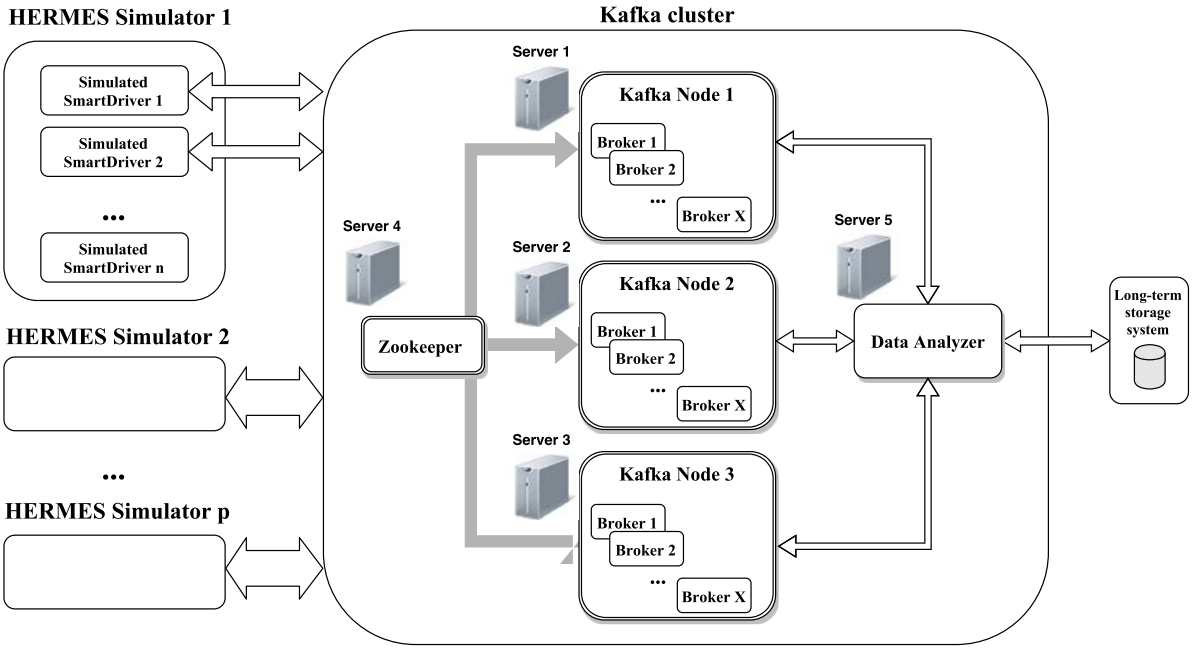


Fig. 10. Kafka configuration 4: Kafka cluster consisting of three nodes deployed on three different machines, using different number of brokers.

- Server type 1: The first one was a mid-range desktop PC consisting of a 4 cores Intel(R) i5-4440 CPU at 3.1 GHz and 8GB of RAM with 64 bits Ubuntu 16.04 LTS.
- Server type 2: The second one used FIWARE[20] virtual machines. Every FIWARE virtual machine instance commented in this paper consisted of 4 cores Intel(R) Xeon E312xx 2.6 GHz and 8GB of RAM with 64 bits Ubuntu 14.04 LTS.

In all cases, the servers were configured with Apache Kafka version 0.10.1.1 and Java Runtime Environment version 8 (update 121), which were the latest versions available at the time of writing this paper. Additionally, the operating system was configured to allow the use of an unlimited number of open file descriptors because the simulator was configured to use persistent connections both in Ztreamy and Apache Kafka. In addition, in the case of Apache Kafka the settings that protect against client connection leaks had to be disabled in order to allow an unlimited number of connections per IP, because each simulator instance creates multiple connections using the same IP. Each test consisted in finding the maximum number of clients that a given streaming server setup can serve.

On the other hand, to monitor the server side during the execution of the simulators, Ztreamy and Apache Kafka were configured

[20] https://account.lab.fiware.org/ (Visited 2017-01-18).

Fig. 11. Ztreamy using server type 1 and errors occurred during the simulation.

to log all the events produced. Additionally, Munin[21] was also used to keep track of server resources and network throughput.

With respect to the machines running the simulators, 5 identical FIWARE virtual machines were created to run an instance of the HERMES simulator each one. Each simulator was configured to create 10 paths of 10 Km each one, increasing progressively the number of clients until a set number, logging the delay produced during the communications. All the simulators were synchronized to start simultaneously in order to achieve the peaking concurrent drivers.

6.1. Ztreamy vs Kafka

In [37], it is explained that the Ztreamy infrastructure was able to handle up to 4000 simultaneous drivers using a single server with 12 Intel Xeon E5-2430 2.5 GHz cores and 64GB of RAM, which represents a rate of approximately 28,000 events per minute. The
authors also show that at larger data rates collectors began to reject some events due to saturation.

As a base case, an initial experiment has been configured to create 4000 simultaneous SmartDrivers with the simulator settings commented in the previous section on the server type 1, which has far less cores and RAM, so the expected results should be worse
than those described in [37]. The chart in Fig. 11 shows the evolution of concurrent SmartDrivers and errors in the Ztreamy infrastructure as a result of timeouts due to server saturation. After analyzing the server and the simulators logs, it was revealed that the server failed to respond in about 100 s in all the simulations. Network problems were discarded because all tests were executed in different days and firewalls were disabled.

Moving to Kafka, we started from the most basic Kafka configuration 1, in which the cluster is composed by only one node, having one broker, managed by a single instance of Zookeeper. The chart in Fig. 12 shows the evolution of concurrent SmartDrivers.
No errors occurred during the simulation because, even with a single node, Kafka is able to deal with thousands of streams. The key to the success lies in its storage system. Apache Kafka uses a log-based system and is extremely efficient using it. Hard disks and RAM have the highest throughput when they are read and written sequentially. Because of that, input streams are attended promptly, freeing up the server for other processes. Equally important is the fact that Kafka lets consumers read messages at their own pace and they are responsible for managing their own offset over the
topic they are reading, releasing the server from the burden of keeping any kind of per-consumer state.

Although there were no errors during the previous test, it was tested the same simulation conditions using a custom distributed Kafka configuration architecture within the server type 1. This custom configuration consisted of 2 nodes and 2 brokers by node. The goal of this setting was to study the load balancing between the nodes and the resources usage of a distributed design within the same server. It resulted on a 34% more RAM and 17% more CPU usage over the initial configuration, so this pseudo-distributed model does not seem effective, at least in low-performance servers. As for the 4000 simultaneous SmartDrivers, neither errors were produced, nor remarkable changes compared to the previous test, as a single node was enough to support the load.

Tables 3 and 4 summarize the average and maximum resources usage during the tests with Ztreamy and Kafka on server type 1.

It can be seen that even with limited resources machines, Apache Kafka supports the amount of simultaneous drivers that Ztreamy was able to tackle with a expensive high-performance server. In a production environment, it would be desirable to use a customized configuration that optimize performance at the lowest cost. Additionally, other factors like availability, scalability or manageability are also decisive in order to choose a solution, so in Section 6.2, different Kafka configuration models are studied to analyze their performance.

6.2. Comparing Kafka configuration models

Apache Kafka is a flexible publish-subscribe messaging system, so the cluster can be transparently expanded without downtime. Different configurations have been tested to have an overview of the possibilities and the amount of users that could be served in the case study. To this end, the first scenario consisted in testing how many SmartDrivers could a single node support on an instance of server type 2, commented in the previous subsection. Latency is considered to be unacceptable if it exceeds 5 s. In this test, the instances of the simulator were used to continuously create new simulated SmartDrivers until delays begun to occur and the server became saturated. It can be seen in Fig. 13 that a single node can support almost 41,000 simultaneous SmartDrivers before exceeding the delay threshold. The amount of RAM memory plays an important role in performance because Kafka stores the disk buffer cache in RAM, which means that enough memory is needed in order to keep messages yet to be consumed by belated consumers. In the simulations, when the number of those belated consumers increased, the number of supported simultaneous Smart-

[21] http://munin-monitoring.org/ (Visited 2017-01-18).

Fig. 12. Kafka configuration 1 using server type 1.

Table 3
Munin indicators for Ztreamy and Kafka streaming server.

Streaming server		CPU usage	Memory usage	Memory allocated	I/O wait	TCP established connections
Ztreamy	Avg	15.36%	6.40GB	11.19GB	1.42%	$0.61 \cdot 10^3$
	Max	51.39%	7.13GB	11.96GB	2.78%	$3.84 \cdot 10^3$
Kafka	Avg	1.99%	1.24GB	6.55GB	0.80%	$2.91 \cdot 10^3$
	Max	5.55%	1.58GB	7.10GB	1.08%	$7.80 \cdot 10^3$

Table 4
Stream rates for Ztreamy and Kafka streaming server.

Error rate	First try success rate	Error recovering rate	Traffic increase due to errors	
Ztreamy	5.52%	94.48%	99.31%	5.40% ≈ (8.64MB)
Kafka	0%	100%	–	–

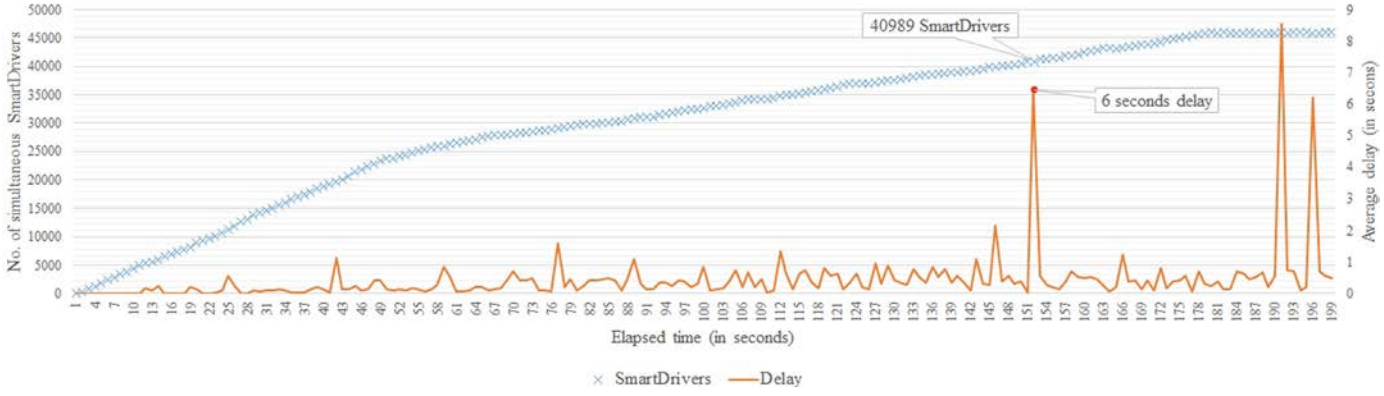


Fig. 13. Kafka configuration 1 using server type 2. Incrementally increasing simultaneous SmartDrivers and delay detected.

Drivers decreased, because of the overload caused by transferring disk data to RAM.

One of the strengths of Kafka is its scalability. Hence, the next step was to test a distributed multi-node configuration. Increasing the number of nodes allows the creation of more partitions and spreading the data to scale the architecture horizontally.

On the other hand, Apache Zookeeper manages the cluster and is responsible of synchronizing the nodes. It is worth mentioning that, for production environments, it is also recommended to configure Zookeeper in replicated mode to warrant availability in case of failures. Zookeeper has a master-slave architecture and is recommended to be run in an odd number of machines to create what is called an ensemble, being one of them the master and the others slaves.

Thus, as there were not enough machines to replicate Zookeeper, it was tested a Kafka distributed configuration using 3 Kafka nodes with 6 brokers, managed by a replicated Zookeeper deployed together with the Kafka nodes, so each machine had an instance of Zookeeper and a Kafka node.

As before, we consider that a given latency value is acceptable if it remains under 5 s, in order to ensure a quick response clients (Fig. 14).

In this case, the cluster is able to support more simultaneous drivers before reaching an excessive delay, but the traffic generated to synchronize contents between the nodes required them to be in a fast network as the number of simulated SmartDrivers increased.

One important improvement was to separate Zookeeper to leave it enough resources, resulting in a cleaner architecture of the

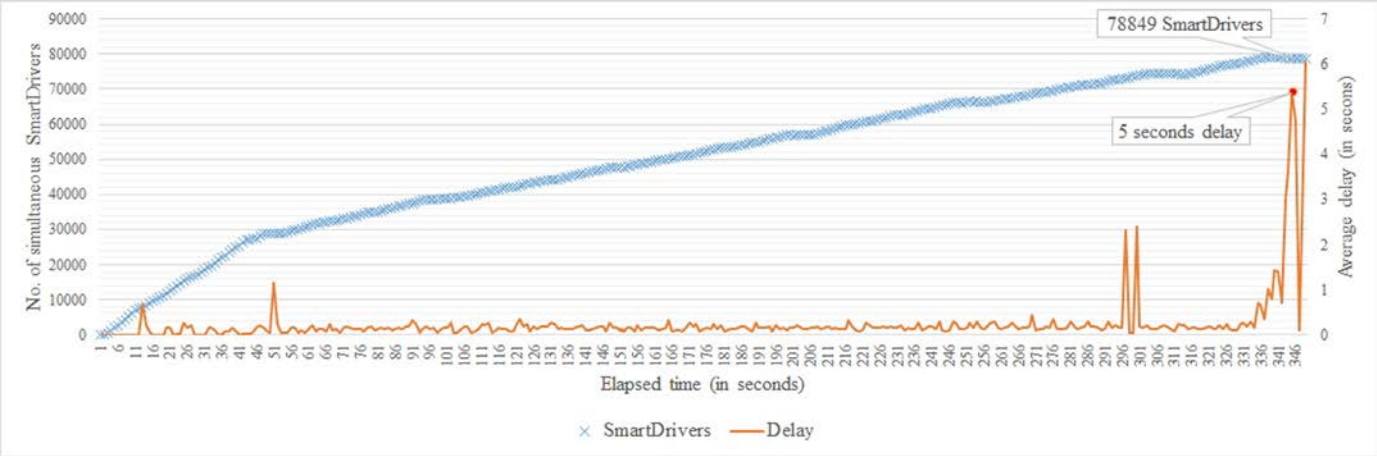


Fig. 14. Kafka configuration 3 using servers type 2. This setting uses 6 brokers per node.

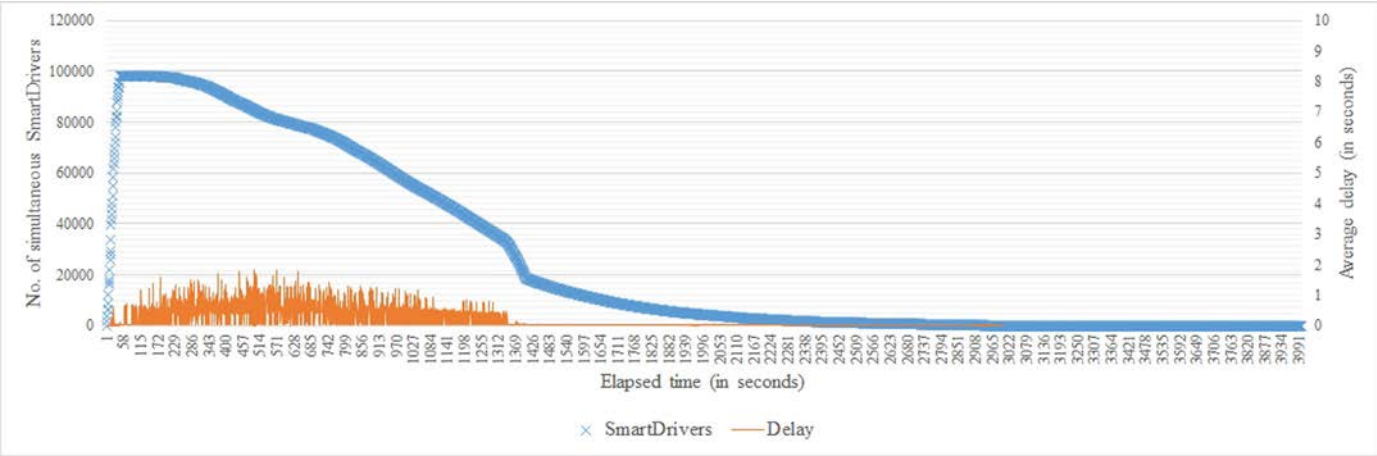


Fig. 15. Kafka configuration 4 using servers type 2. This setting uses 8 brokers per node. Full-size trace of best result achieved covering almost 100,000 simultaneous SmartDrivers.

Table 5
Kafka model configuration comparison.

Kafka config.	Machines	Nodes	Brokers per node	SmartDrivers	Delay (s)
1	1	1	1	40,989	6
3	4	3	6	78,849	5
4	5	3	8	100,000	2.3

whole cluster. Thus, in Kafka configuration 4, we have used only one instance of Zookeeper as the coordinator for our tests, as it was enough for the test cases. Hence, for this distributed configuration, we set up the 3-node cluster using 5 identical trial FIWARE servers with the previously commented features, defining 3 of them as Kafka nodes, one for setting Zookeeper and the last one to deploy the Data Analyzer. The proposed cluster architecture can be seen in Fig. 10.

At this point, it is proved that a distributed solution works better when the number of potential users grows, but adjusting and testing different configurations is advisable to better meet the needs of each particular case.

The final test was carried out using the Kafka configuration 4 and increasing the number of brokers from 6 to 8. With this archi- tecture, there were achieved the best results, as shown in Fig. 15. These tests are summarized in Table 5, where it can be seen how configurations 1 and 3 were not suitable to serve at least 90,0 0 0 simultaneous drivers whereas configuration 4 achieved this goal with some margin.

7. Conclusions and future work

As data sources and the volume of information increase, being able to process quickly vast amounts of information becomes more necessary. Until a few years ago, the best solution was to develop a custom platform to solve the problem, but with the strong rise of Big Data, new initiatives to deal with large volumes of information have emerged. Thus, several options are available when a Smart City scenario has to be solved. After studying a wide range of smart cities infrastructures, it has been found that smart transportation sector has to tackle some of the most demanding requirements, such as real-time services. The publish-subscribe paradigm is a popular solution for handling large volumes of inbound and outbound data flows asynchronously and could be used to manage transport logistics processes. The case shown in this paper demonstrates how was tested the previous streaming server used in HERMES, exposing the SmartDriver real scenario and the simulator implemented to solve the shortage of users in order to test the platform. The open source simulator,