https://doi.org/10.1093/comjnl/bxab143

An Analysis of Computational Resources of Event-Driven Streaming Data Flow for Internet of Things: A Case Study

Alonso Tenorio-Trigoso[1], Manuel Castillo-Cara[2,*], Giovanny Mondragón-Ruiz[3], Carmen Carrión[4] and Blanca Caminero[4]

[1]*Escuela de Posgrado, Universidad Peruana Cayetano Heredia, Lima, Peru.*
[2]*Universidad de Lima, Lima, Peru.*
[3]*Center of Information and Communication Technologies, Universidad Nacional de Ingenieria, Lima, Peru.*
[4]*Albacete Research Institute of Informatics, Universidad de Castilla-La Mancha, Albacete, Spain.*
[*]*Corresponding author: jmcastil@ulima.edu.pe*

Information and communication technologies backbone of a smart city is an Internet of Things (IoT) application that combines technologies such as low power IoT networks, device management, analytics or event stream processing. Hence, designing an efficient IoT architecture for real-time IoT applications brings technical challenges that include the integration of application network protocols and data processing. In this context, the system scalability of two architectures has been analysed: the first architecture, named as POST architecture, integrates the hyper text transfer protocol with an Extract-Transform-Load technique, and is used as baseline; the second architecture, named as MQTT-CEP, is based on a publish-subscribe protocol, i.e. message queue telemetry transport, and a complex event processor engine. In this analysis, SAVIA, a smart city citizen security application, has been deployed following both architectural approaches. Results show that the design of the network protocol and the data analytic layer impacts highly in the Quality of Service experimented by the final IoT users. The experiments show that the integrated MQTT-CEP architecture scales properly, keeps energy consumption limited and thereby, promotes the development of a distributed IoT architecture based on constraint resources. The drawback is an increase in latency, mainly caused by the loosely coupled communication pattern of MQTT, but within reasonable levels which stabilize with increasing workloads.

Keywords: smart city; Internet of Things; real-time stream processing; computing performance; data-driven analysis; complex event processing.

Received 21 December 2020; Revised 25 June 2021; Editorial Decision 25 August 2021
Handling editor: Dr Fabrizio Messina

1. INTRODUCTION

Internet of Things (IoT) applications are normally classified considering its utilization domain (i.e. smart services, smart building, smart transport or smart city) but IoT applications, even from the same domain, have different quality of service (QoS) requirements. Hence, based on these requirements the IoT applications map into different IoT implementation technologies [1]. For example, traffic management or citizen security applications within a smart city are characterized by requiring an infrastructure that receives and processes data generated continuously as results of temporary events that follow a continuous temporal flow structure [2]. In this context, one of the fundamental characteristics of IoT is that the set of sensors connected by a wireless network collects information with the least possible human interaction in order to provide a new type of application [3]. These applications usually process the information collected by the sensors to make decisions and analyse past behaviours.

Most of the architectural deployments to enable IoT solutions are based on cloud computing models. Relying on a cloud approach means offering a secure 24-hour service with high availability and leveraging the computing and storage capacity of cloud providers [4–6]. However, using these resources efficiently and composing the best services to achieve the optimal QoS is a challenge, as recent research shows [7].

One of the most interesting aspects of computing infrastructures is to create, manage and integrate services that can improve system scalability and optimize the use of the infrastructure's computational resources [8]. This fact has led to the development, among others, of a wide range of event processing mechanisms, synchronous and asynchronous communication protocols, or virtualized dedicated microservices [9, 10]. For example, there are different categories of IoT streaming applications, such as Extract, Transform and Load (ETL), Stream Machine Learning or Complex Event Processing (CEP), and each of them provides specific QoS [11].

In view of the above, prior to the deployment of an IoT application some challenging decisions must be made related to the standard messaging protocol and the real-data processing layer [12]. The study presented in this paper neither intend to analyse IoT network protocols nor IoT data processing in isolation, but application performance of IoT deployments as a whole [13]. To that end, two alternate implementations of an IoT application have been developed. In the first one, the stream processing framework combines the messaging protocol, Message Queue Telemetry Transport (MQTT), at the network layer and CEP at the data processing layer. More precisely, Apache Flink [14], a framework that provides advanced stream processing capabilities for IoT applications, is used as a CEP engine [15]. This implementation is compared to a traditional, ETL-based implementation based on a Hyper Text Transfer Protocol (HTTP) server and POST requests. Hence, the value of a real-stream processing framework based on Flink is discussed and compared to a traditional approach based on ETL.

The experiments focus on evaluating scalability in terms of latency, resources and energy of a typical IoT application. According to the obtained results, choosing a proper framework at both the network and the data analytic layers highly impacts the QoS experimented by the end users of the IoT application. Moreover, the experiments show that the components of the stream processing framework (MQTT-CEP) work properly together. This can lead to the development of a distributed IoT architecture based on constrained resources. All the above has been outlined in Fig. 1.

Thus, the main contributions of this work can be summarized in the following highlights:

- Implementation of two architectures based on alternative protocols for the same IoT application (HTTP-POST and MQTT-CEP).

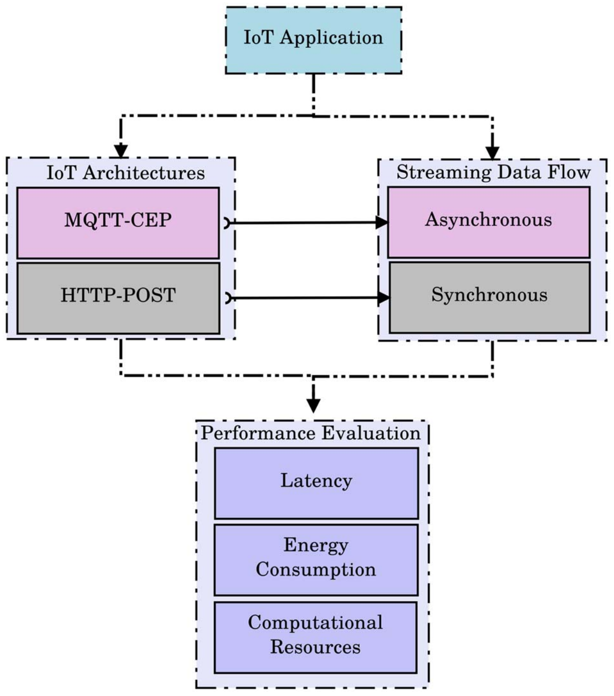


FIGURE 1. Overall evaluation of the case study application.

- Development of IoT event-driven data modelling for the two architectures under evaluation.
- Analysis of contextual transmission of event-driven streaming data flow for synchronous and asynchronous communication.
- Evaluation of resource and energy consumption, as well as latency in global and local terms, for the two architectures under evaluation.
- Identification of relevant criteria for selecting the most appropriate architecture according to the requirements of each application.

The paper is organized as follows. Section 2 presents the related work on IoT architecture solutions and their performance. Then, Section 3 depicts the main characteristics of the use case application. A detailed description of the architectures under study and their stream data flows are provided in Section 4 and Section 5, respectively. Section 6 shows and discusses the obtained results. The final section, Section 7, summarizes some conclusions and points out future work related to this research.

2. RELATED WORK

In this section, some applications based on IoT and their architectures are reviewed, as well as work related to the performance evaluation of IoT environments.

2.1. IoT: architectures and ecosystem

The Information and Communication Technologies backbone of a smart city is composed of one or more IoT [9] applications that combine technologies such as low power IoT networks, device management, analytic or event stream processing [16]. The integration of all the involved technologies makes possible to extract raw data from smart objects and sensors, process data and finally extract insight value to improve citizens lives. IoT designs are usually modelled as a multi-layer architecture [17, 18]. For example, in [17] authors proposed a Smart City Data Analytics Panel 3-layer architecture including a platform layer, security layer and data processing layer. Similarly, IBM [18] adopted a three-layer model including an instrumented layer, an interconnected layer and an intelligent layer.

In general, application protocols over transmission control protocol/Internet protocol (TCP/IP) or user datagram protocol/IP are required to collect data from IoT devices. For example, the framework of the operative project HABITAT (Home Assistance Based on the IoT for the Autonomy of Everybody) [19] applies the standard HTTP protocol for IoT communication. HTTP is based on the representational state transfer architecture with large text-based overhead and requires regular polling or posting of updates. Thus, alternate protocols to HTTP have arisen to overcome its drawbacks in IoT contexts [20]. In particular, MQTT is an asynchronous, event-driven, publish/subscribe messaging system which exhibits low bandwidth and high latency [21]. A static comparative analysis of the most well-known message protocols for IoT Systems is presented in [10], including HTTP, Constrained Application Protocol, advanced message queuing protocol and MQTT. But it did not consider dynamic network conditions and overheads incurred in the retransmission of packets, which may produce different results.

Regardless of the number of layers, one characteristic of smart city applications involves the acquisition of data in real-time and making decisions. However, it is often difficult to perform real-time analysis on a large amount of heterogeneous data and sensory information that are provided by various sources [3]. So, as the classical batch processing ETL technique is not tailored to real-time processing, modern stream processing architectures such as Apache Storm [22] or Apache Flink [14] are used to process the streams. These technologies are mostly implemented in a cloud-based environment (AWS EC2, Microsoft Azure). Thus, there are some cloud-based commercial ready-made IoT solutions, like FIWARE [23, 24], AWS IoT [25] or IBM Watson IoT platform [26] available to developers to this end.

2.2. IoT: performance evaluation

Cloud and distributed systems approaches are the basis for big data solutions provided for smart cities, just as IoT solutions are the basis for collecting data from sensors and devices in the city [9] or a health specific application such as Dengue Infection [27]. In an IoT context, having a single data store and common processor can be helpful for data analytic as it enables a global view of the overall system and is a familiar architecture. However, when millions of IoT devices such as sensors or smartphones send stream data to a common server, the communication network and also the server can speedily become a bottleneck [28, 29].

The analysis of computational resource consumption and latency in different architectures for IoT, e.g. fog and cloud computing [8], has attracted a great deal of interest from the scientific community. In this context, research such as [8] depicts a comparative study of fog and cloud computing in CEP-based real-time IoT applications. Specifically, the authors show that latency in fog computing platforms is reduced by 35% compared to a cloud-based approach without affecting the QoS. Furthermore, the paper shows how the deployment of an efficient fog computing ecosystem saves up to 69% of energy consumption compared to cloud computing, that is, green energy for IoT.

Moreover, recent research in the literature establishes mechanisms for efficient distribution of computational resources in IoT application development. In [7] an integrated Hidden Markov and Ant Colony (HMAC) mechanism is proposed obtaining a QoS improvement over other methods compared in the same research. The HMAC proposal proves to be a mechanism with the following benefits: low response time, low cost, high reliability and low energy consumption.

On the other hand, as is well known, IoT suffers from certain limitations, such as power consumption, computational limits, etc., which need to be analysed and optimized. Hence, Heidari *et al*. [5] develop an in-depth literature study on the findings of the whole procedure on IoT offloading. The analysis is very extensive and focuses on the metric quality of certain parameters in different architectures (cloud / fog / edge computing). More specifically, the QoS parameters analysed are response time, latency, flexibility, complexity and performance. In addition, the authors analyse and compare these parameters with different ideas put forward by other researches in the literature.

From the aforementioned above, extensive analysis of various QoS parameters within different architectures and protocols for IoT applications has been carried out, which highlights the interest in this topic.

3. USE CASE: THE SAVIA APPLICATION

The suitability of the IoT technologies analysed will be performed considering the SAVIA application as a use case. SAVIA is an IoT application to avoid gender violence within the scope of a smart city [30]. It should be noted that the technical requirements of the network protocol and the data analytic layer are cross-cutting and can be directly extrapolated to other IoT application domains such as smart parking, smart farming or smart shopping applications [31, 32].

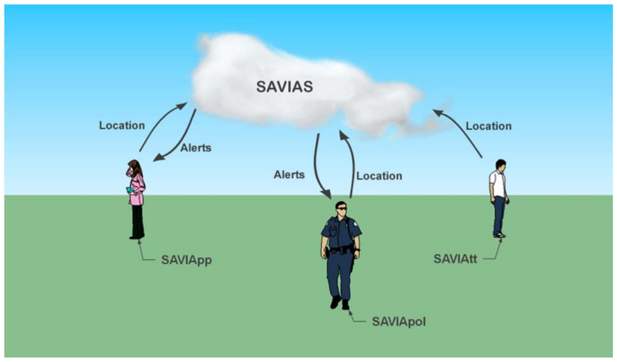


FIGURE 2. Overall schema of SAVIA system.

SAVIA's application context is the local environment of a city. A desirable design goal is to be scalable and with the necessary QoS to detect possible problems of gender violence in time. More specifically, the SAVIA system monitors at all times the distance between a victim of gender violence and her aggressor. Failure to comply with the departure distance will generate a real-time alert in SAVIA. The functional details of SAVIA are described below.

SAVIA comprises a set of applications that address the problem of citizen security relative to gender violence. The roles involved in SAVIA are victims (from now on, *SAVIApp*), aggressors (from now on, *SAVIAtt*), and the police (from now on, *SAVIApol*).

The SAVIA system consists of three smartphone applications (one for each role). *SAVIApp*, *SAVIAtt* and *SAVIApol* applications send position information to the SAVIA server application, i.e. *SAVIAs*. The main feature of *SAVIAs* is real-time monitoring, via global positioning system location information, of *SAVIApp* and *SAVIAtt* position information to uphold restraining orders and identify threatening situations. If the distance between a *SAVIApp* and *SAVIAtt* is close to that defined by a restraining order, the *SAVIAs* sends a message to both the *SAVIApp* and *SAVIAtt* (see Fig. 2).

SAVIAs stores the position of each victim and aggressor because this information can be used in a trial or to study the user's behaviour and location. In addition, historical position data can be used to determine if an aggressor stays relatively close to a victim but does not violate the restraining order. Such behavioural profiles can be used to justify further preventive actions. *SAVIAs* sends messages to *SAVIApol* and *SAVIApp* when it detects anomalous situations. Similarly to comparable applications, *SAVIApp* has a panic button for emergencies. Therefore, there are different roles and communications between them that will be optimized as explained later.

3.1. Ecosystem

In the SAVIA system model, different applications have been developed, namely, *SAVIApp*, *SAVIAtt* and *SAVIApol* (victim, aggressor and police mobile applications, respectively), which constantly communicate with the server, *SAVIAs*. *SAVIAs* will be the core element of the global system since it will process all information for all roles. *SAVIAtt*, *SAVIApp* and *SAVIApol* will be only the auxiliary devices/applications that will feed all the information to *SAVIAs* to be processed. The applications/devices will process only their own information, i.e. they do not have direct communication among them.

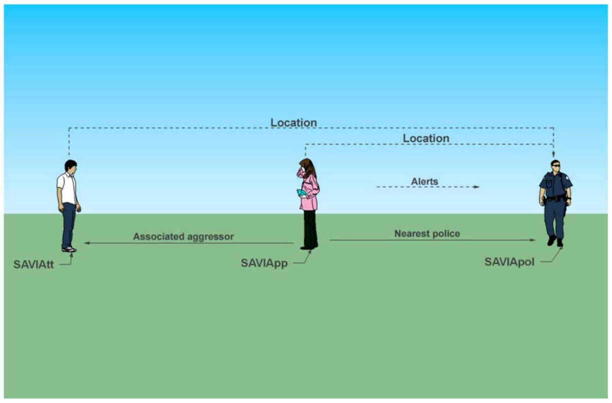


FIGURE 3. Overall schema of SAVIA relationship.

Given that *SAVIAs* will be able to exchange all regularized information among the applications under some specific rules, we could say an indirect flow of information exists between them, represented by dot points in Fig. 3. We can see there the three applications that make up the SAVIA system and the indirect relationship between them.

3.2. SAVIAtt - SAVIApp

It is shown how *SAVIApp* must be associated to *SAVIAtt*. This association between them, which must not involve information exchange under any circumstances, is made through *SAVIAs*. That is, the two roles are defined in *SAVIAs*. Then, with identifiers of the application, device and the users' information, the users are linked. Hence, once both roles have been associated, *SAVIAs* will check the *SAVIApp* user safety, i.e. checking if a restraining order issued by a court is being fulfilled.

3.3. SAVIApol–SAVIAtt

Another indirect interaction that exists in the system occurs between *SAVIApol* and *SAVIAtt*. As mentioned before, there is no direct information exchange between them, but all information will be exchanged through *SAVIAs*. *SAVIApol* will get the coordinates of *SAVIAtt*, in case of an emergency alert when *SAVIAtt* could be a potential danger to *SAVIApp* integrity, i.e. when *SAVIAtt* breaks the rules of a restraining order. In these cases, *SAVIApol* can see *SAVIAtt* on the map of its mobile device because the location of *SAVIAtt* will be sent by *SAVIAs*.

SAVIApol cannot display the location of *SAVIAtt* if the events listed above are not given.

3.4. SAVIApp - SAVIApol

SAVIApp continuosly sends its location to *SAVIAs* and *SAVIAs* looks for the *SAVIApol* closest to the *SAVIApp*, in case an alarm is detected. Such alarm will be triggered automatically by *SAVIAs* when a violation of the restraining order by *SAVIAtt* is detected. In addition, *SAVIApp* can also send manual alerts, by pressing the panic button. In any case, *SAVIAs* looks for the *SAVIApol* closest to the *SAVIApp*. This communication of sending/receiving the localization through to *SAVIAs* will be constant until the alert generated is closed by the *SAVIAs* system administrator.

4. IOT ARCHITECTURES UNDER STUDY FOR SAVIA

As noted before, SAVIA must offer the necessary QoS to detect possible problems of gender violence in time. However, a critical aspect in the SAVIA system along with the response time is its scalability or ability to control the concurrency given by multiple requests in the underlying architecture of the application generated for the different roles, that is, *SAVIAtt*, *SAVIApp* and *SAVIApol*. Also, the restraining order and automatic alerts need to be redefined because many queries are sent to the system, where the correct distance between these two roles is verified.

In this context, when developing the architecture and data model, there was a problem associated with high concurrency in access to the database. Thus, two different architectures are proposed and subsequently their evaluation results and performance will be compared in terms of latency and consumption of energy and resources.

4.1. IoT event-driven data modelling

The first implementation, called POST architecture, consists on a client-server design with an application server based on the HTTP. For the data analysis, a classic ETL processing will be implemented. The data model is preset so that the data is stored in the database, and subsequently extracted and analysed. Then, they are processed generating higher order events if they meet the alarm condition.

In a second implementation, called MQTT-CEP architecture, data streaming of the data collected by the *SAVIAtt* and *SAVIApp* applications is processed and events are used to establish the communication between the different components. The main result of this task is the ability to notify affected parties of alerts arising from lower-level events. For this, Apache Flink has been used as a CEP engine for general events and a Broker in MQTT to send alerts to the different profiles.

MQTT protocol is an easy-to-implement and lightweight application layer protocol designed for resource-constrained devices. It uses a topic-based publish-subscribe (or client-broker) model. So, when a client publishes a message M to a particular topic T, all the clients subscribed to the topic T will receive the message M. Like HTTP, MQTT relies on TCP and IP as its underlying layers. Nevertheless, MQTT was designed to have lower protocol overhead than HTTP.

4.2. HTTP-POST architecture

A POST architecture is obtained directly through a direct conversation between the different applications and the server, i.e. there is a synchronous communication. This communication is established through HTTP. In itself, a direct connection is made from the application to the server for the exchange of information, cutting off communication when one of the two users request it.

The HTTP server works in a request-response manner through TCP connections with client applications. Once connections are established, the HTTP server listens to certain ports for requests from *SAVIApp* and *SAVIAtt* and starts to process the received information according to the gender security algorithm.

For this, a POST communication has been established, that is, a request method supported by HTTP used by the World Wide Web. By design, the POST request method requests that a web server accepts the data enclosed in the body of the request message, most likely for storing it. It is often used when uploading a file or when submitting a completed web form.

Figure 4 shows the communication flow procedure between the different roles for a POST Architecture (from now on, POST). As can be seen, the three different roles are played by sending their position information, i.e. latitude/longitude to the central server, *SAVIAs*. It must be highlighted that *SAVIAs* have a database that stores these values. In order to be able to analyse the distances between *SAVIAtt* and *SAVIApp*, these values are constantly extracted and analysed. In case the distance between these two is breached, an alert is generated and this is sent to the *SAVIApol* and *SAVIApp* roles through *SAVIAs*.

As can be seen, POST has a very high concurrency in two critical sections on any server: (i) constant concurrency in the insertion, extraction and analysis database in each new value of the roles; and (ii) the high concurrency given by the constant synchronous communication between the different roles of receiving alerts with *SAVIAs*. Note that in this type of communication there are continuous polls for alerts, even if none has been generated and does not really exist (process that will be detailed in Section 5.1).

4.3. MQTT-CEP architecture

In contrast to POST, we have an MQTT-CEP architecture that establishes asynchronous communication. This type of

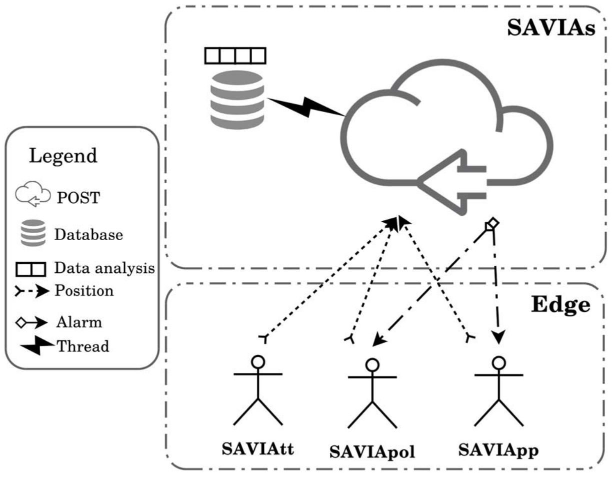


FIGURE 4. Overall schema of synchronous implementation.

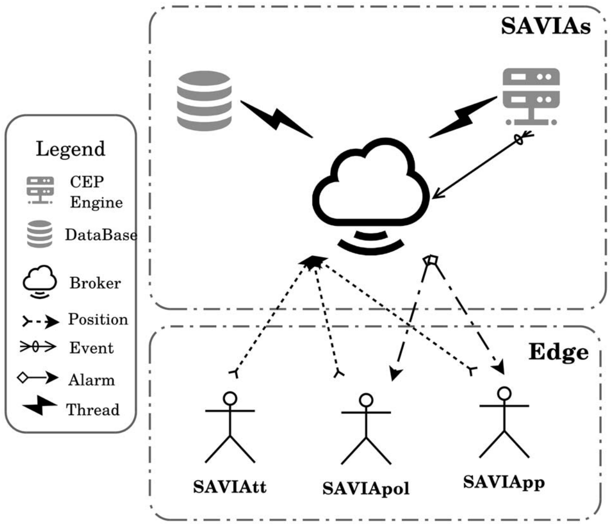


FIGURE 5. Overall schema of asynchronous implementation.

communication is characterized mainly since a connection is only established when there is a notification, i.e. an alert is generated. Therefore, the main objective of implementing MQTT-CEP is to eliminate high concurrency at the connection and database level, i.e. iterative queries and extractions (process that will be detailed in 5.2).

Figure 5 shows the different components present in it. To establish MQTT-CEP, two main mechanisms have been implemented: (i) Flink-CEP as a complex event processing engine and; (ii) MQTT as a communication protocol between the different actors.

In this case, it can be seen how in the edge part the elements of the system have to perform the same tasks as in the POST architecture, i.e. the roles send the message with the information to *SAVIAs*; however, a subscription/publication-based philosophy is used for its implementation through the MQTT Broker. Once the message reaches the cloud, *SAVIAs*, the processing of simple events is carried out through two threads of execution in parallel, namely: (i) a first thread of execution stores the information received in the database; and (ii) the second thread of execution that forwards the message to the CEP engine. More precisely, the Apache Flink framework has been used in SAVIA implementation. Note that in [11] the authors recommend the use of this framework for CEP applications.

Thus, as the messages arrive at the CEP engine, they are queued in its stack in order to analyse the different data that, in case of complying with the specified behaviour pattern, in this case analysing the distances, generates an event complex. In the event of a CEP event, it is sent to the MQTT Broker which, through a subscription-based protocol, notifies the specified roles in the form of an alert.

As a summary, it is worth mentioning that in both cases the objective is the same, i.e. generate an alert under a controlled behaviour pattern and notify users. Likewise, the procedure and mechanisms used differ widely, each one having its own pros and cons, which are detailed in Section 6. Before discussing the results of the performance analysis on the two possible types of cases, it is important to detail what the process of communication between the different roles and the cloud infrastructure.

5. CONTEXTUAL-STREAMING DATA FLOW

As detailed above, our SAVIA application has been implemented following both architectural approaches: the traditional POST architecture and the MQTT-CEP architecture. In order to evaluate the performance of these two architectures, 'In-situ Events' [30] has been chosen as a CEP case study. These types of events are characterized by the generation of complex events from the analysis of simple events in the same fraction of time, i.e. real-time analysis. Therefore, the objective for both architectures, MQTT-CEP and POST, is the same, i.e. evaluate in real-time the distance between *SAVIAtt* and *SAVIApp*, and, if the distance restraining is broken, send an alert to the nearest *SAVIApol*.

It should be noted that, for this experiment, only the alert generation mechanism has been changed, summarized as POST or MQTT-CEP, being all the architecture and applications that underlie the SAVIA ecosystem exactly the same.

5.1. Synchronous communication

As it has been observed, synchronous communication has the main component of opening an ordered communication thread between a transmitter and a receiver, which exchange information between them. The implementation in our system can be

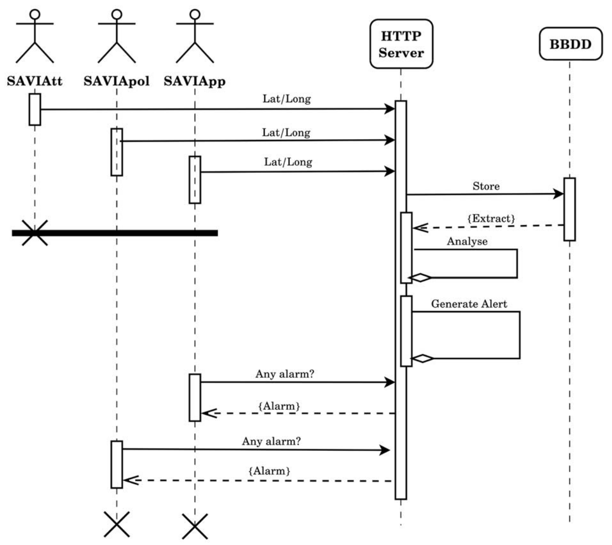


FIGURE 6. Message flow schema with POST.

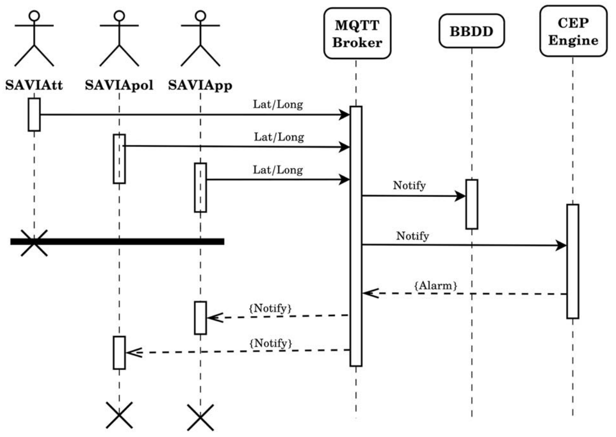


FIGURE 7. Message flow schema with MQTT-CEP.

seen in Fig. 6. More precisely, the communication process is established as follows:

1. All three applications send their position in latitude/longitude to the server. To do this, they open a synchronous communication using the HTTP protocol. In the case of *SAVIAtt*, once you send your position, you do not have any more interaction with the system.
2. The position sent by the different roles is stored in the database (BBDD).
3. The positions are extracted from the database and are stored in main memory.
4. Having the last positions in main memory, these are analysed verifying our condition of generation of alerts, that is, the distance between *SAVIAtt* and *SAVIApp*.
5. In the event that there is a breach in the distance established between *SAVIAtt* and *SAVIApp*, an alert is generated.
6. If the alert is generated, it is sent to the roles established for it, namely, *SAVIApol* and *SAVIApp*, through synchronous communication through HTTP.

Thus, in this type of communication it can be observed that there is constant communication between *SAVIApp* and *SAVIAtt*, i.e. these roles are constantly making requests to the server to check if there is any notification for them. This means that communication is being established regardless of whether there is an alert or not, generating constant unnecessary communication if there is no alert.

Therefore, in this type of communication, it is possible to notice a high concurrence that exists both in the communication phase of alert notifications and also in the constant requests to the database to extract and analyse the information. Faced with this large concurrence, an asynchronous system is proposed in which these two mechanisms can be created as independent services (microservices), decongesting the high concurrency generated in synchronous communication.

5.2. Asynchronous communication

Asynchronous communication differs from the synchronous one in the fact that data are analysed as soon as they reach the cloud through CEP, leaving the database as a storage service only. In addition, this type of communication creates two additional microservices to be able to analyse and send the information, namely, the CEP engine for analysing the information, and the MQTT Broker to send complex events to the corresponding roles in the form of an alert. This communication procedure between users and the cloud through these services is depicted in Fig. 7. In detail, the communication process is established as follows:

1. All three applications report their position in latitude/-longitude to the server through the MQTT Broker. In the case of *SAVIAtt*, once you send your position, you do not have any more interaction with the system.
2. The Broker opens two communication threads in parallel: (i) one for storage in the database; and (ii) another that sends the CEP engine for analysis.
3. The CEP engine receives information from users as simple events and analyses it according to established patterns.
4. In the event of a breach in the distance established between *SAVIAtt* and *SAVIApp*, CEP generates a complex event in the form of an alert. This alert is sent to the MQTT Broker.
5. If the alert is generated, it is notified to the roles established for it, i.e. *SAVIApol* and *SAVIApp* through the MQTT Broker.

5.3. Discussion about asynchronous vs. synchronous communications

This section will compare the operation of the two architectures previously described. In the asynchronous communication mechanism, in addition to eliminating high concurrency through database queries, optimization can be seen in the communication procedure. This optimization consists of the advantage provided by the MQTT Broker, which, by means of the subscription procedure, when a complex event is generated by the CEP engine in the form of an alert, publishes it by notifying those who subscribe to it. This eliminates the constant question/answer process that occurs with POST.

Now, this improved asynchronous communication process implies the use of specific services that must be evaluated. First, two additional services have been integrated, such as the MQTT Broker and the CEP engine. Then, and in relation to the integration of these services, according to their own specifications, they make use of a data buffer to store the events. For example, the CEP engine queues the simple events that it receives from users to be analysed and, once the complex event is analysed and generated, it is sent to the MQTT Broker. Thus, the MQTT Broker also uses a buffer that stores these complex events sent by the CEP engine in order to be notified to users. It is important to highlight that the MQTT Broker uses a subscription/publication process that does not immediately notify the user as in POST.

Thus, these additional services must be evaluated, not only in the computational consumption they need, but also to identify the latency involved in using these services that make use of their own internal buffer.

6. PERFORMANCE EVALUATION

In this section, the main outcomes of the evaluation analysis are presented considering the two architectures described in the previous sections. The analysis is based on a real deployment of both POST and MQTT-CEP architectures. Besides, the behaviour of the authorized users of the system has been modelled (see details in next section) to better analyse the scalability of the system with the number of generated alarms. The performance evaluation covers both a cost and performance analysis including the use of the computational resources, latency and energy consumption, see Table 1 [7].

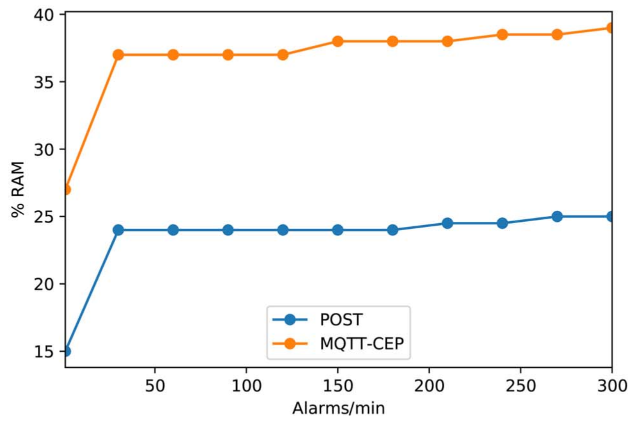
6.1. Testbed description

A real testbed has been designed so that a performance analysis of both architectures can be carried out, focusing on the application selected as use case, namely, SAVIA. To that end, the behaviour of the application end-points (*SAVIApp*, *SAVIAtt* y *SAVIApol*) has been modelled with two entities: SAVIA-Source and SAVIA-Receiver. The model allows to be able to stress-test the system, i.e. by issuing a large number of alarms. Besides, the application back-end is implemented in the SAVIA-Node entity. Next, the main features of these entities are detailed:

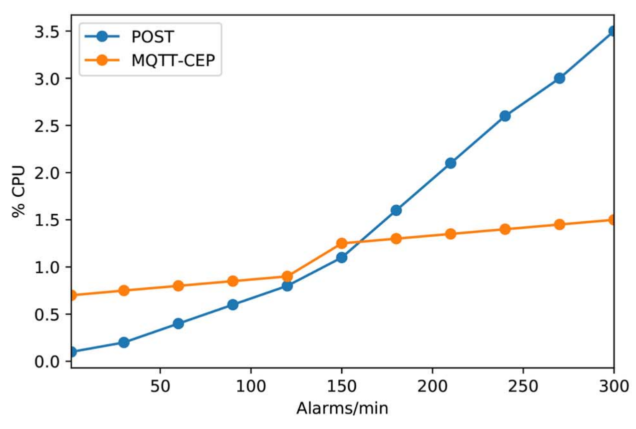
TABLE 1. Definition and representation of the performance parameters considered.

Parameter	Data type	Unit of measure	Sample values
CPU Usage	Quatitative	Percentage	(1, 5, etc)
RAM Usage	Quatitative	Percentage	(15, 30, etc)
CPU Energy	Quatitative	Joules	(0.21, 0.4, etc)
RAM Energy	Quatitative	Joules	(0.1, 0.4, etc)
Latency	Quatitative	Seconds	(0.5, 2, etc)

- *SAVIA-Source*: This entity is in charge of sending the information collected by the sensors to SAVIA-Node. So, it actually will play part of the roles of *SAVIApp*, *SAVIAtt* and *SAVIApol*. For the tests, Source has been written as a script in Python that allows us to modify some parameters such as the number of authorized users, the load of the system or the number of alerts that should be generated per unit of time. More precisely, SAVIA has a total of 450 authenticated users that provide sensor data to the system. So, in every experiment, data flow from these users are sent, and a controlled number of alerts is generated, depending on the particular case. This entity has been implemented in a computer located in the same network as SAVIA-Node. The entire alert package is fully controlled knowing exactly the numbers of alerts generated. Thus, in the course of these tests, a stress test has been carried out taking as a case study the generation of 1, 30, 60, 90, 120, 150, 180, 210, 240, 270 and 300 alerts evenly for a minute. For example, when 450 data records are sent, 30 alerts are generated every minute for a total time of 15 minutes. This allows to obtain the average performance in those 15 minutes. Once the test with 30 alerts is completed, all the services are restarted and the same procedure is carried out for the other cases, although the number of alerts is changed. Therefore, the number of alerts generated is controlled in advance.
- *SAVIA-Node*: This entity will execute all the components of *SAVIAs*, that were described in detail in Section 3. Moreover, it will also be instrumented with the monitoring tools required for the performance analysis. This includes resource usage and energy consumption. More precisely, SAVIA-Node has been deployed in a computer with Intel i7 3.60 GHz $\times$ 8 processor and 8 GB RAM. The use of resources has been measured thanks to the PERF tool [33]. It should be noted that the MQTT-CEP implementation, based on Apache Flink, has two critical processes that should be fine-tuned in order to optimize the performance of the CEP engine, namely, the *JobManager* and the *TaskManager* [14]. On one



(a) RAM Percentage (%).



(b) CPU Percentage (%).

FIGURE 8. Graphs regarding the generation of alerts/min based on the percentage of consumption in RAM and CPU for POST and MQTT-CEP.

hand, the JobManager coordinates job's distributed execution, task scheduling, failure management, etc. On the other hand, the TaskManager executes the tasks and subtasks on the data flow, as scheduled by the JobManager. The latency-optimized values for the SAVIA application are a buffer size of 512 MB with 175 total threads.

- *SAVIA-Receiver*: This entity will receive all the generated alarms and will send back a message to SAVIA-Node including its departure time that will be used to compute the latency. The Receiver has been implemented in a computer located in the same network as SAVIA-Node.

6.2. Computational resources

The use of computational resources has been analysed, both in terms of RAM memory and CPU usage percentage, with different number of generated alarms. The results obtained for the different implementations carried out are shown in Figs 8a and 8b respectively. First, there are clear differences in the results obtained for the architectures under analysis that we will analyse in detail below.

On the one hand, in the consumption of RAM it can be seen that the use of this is constant for both types of architecture, i.e. they do not vary with respect to the number of alerts generated. This is due to the fact that the RAM consumption is established with the service as such and not with the number of alerts that are generated. However, we can see the great difference that exists in terms of POST, which is minor, compared to MQTT-CEP. This makes sense since the CEP engine and the MQTT Broker each have an internal buffer so they need RAM memory. In this way, the CEP engine can store both simple and complex events; as well as the MQTT Broker can store the alerts generated to be notified to the recipient users. On the other hand, in terms of CPU consumption, we can see that it increases with the number of alarms generated. However, in the case of MQTT-CEP we can see the variation is minimal. This is because the CEP engine is optimized for the analysis of simple events and generation of complex events. However, for POST, we can see that from 150 alerts generated the CPU consumption increases drastically. This is because the calculations made for the generation of alarms are based on distance comparisons that are executed directly in the CPU.

6.3. Latency analysis

Figure 9 shows the latency for the different architectures analysed with a variable number of alerts generated. In this case, a notable difference can be observed with respect to the values of each architecture. On the one hand, POST has a constant and minimal latency regardless of the number of alerts generated; while the latency for MQTT-CEP grows as the number of processed alerts increases.

It should be remembered that in these tests the number of users of the system is fixed throughout the experiment, so the latency due to the communication established between *SAVIAtt* and *SAVIAs* should not first be an aspect that affects the observed behaviour. However, the increase in the number of alarms generated affects the computational load of the Apps and increases the use of outgoing network bandwidth. For POST the time used to process an alert is minimal since, as mentioned, this architecture makes a synchronous connection for each user so there is no type of latency other than the users' internet connection (although in the tests remember that *Source* and *Receiver* are in the same local network). However, for MQTT-CEP, the high latency can be generated by the MQTT Broker, when notifying the users, or by the CEP engine, when comparing the simple events that arrive and generating the

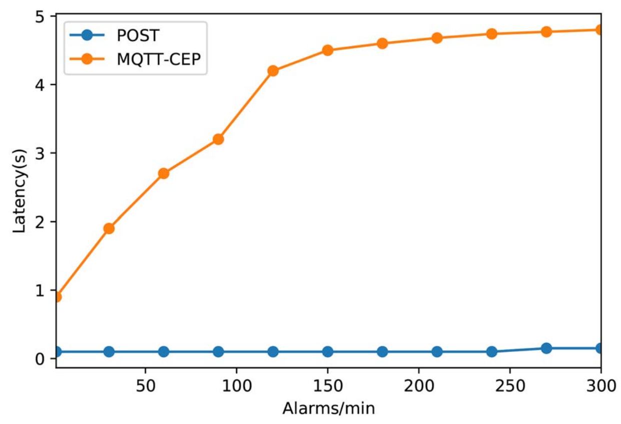


FIGURE 9. Graphs regarding the generation of alerts/min based on the Latency for POST and MQTT-CEP.

complex events or be a combined effect of both. To understand better this result, we need to analyse the time consumed in each part of the process. This will be tackled in the next section.

Anyway, it should be noted that the latency values observed in the MQTT-CEP implementation seem to stabilize under 5 seconds, which is reasonable enough for the SAVIA use case (and other applications alike, that do not have too demanding latency requirements).

6.4. MQTT-CEP latency analysis

In this section, we are going to focus our attention on the latency of CEP-Broker architectures. Therefore, a flow data analysis specific for this context will be performed. The objective is to detect the bottleneck in latency.

So, having into account the system abstraction model and the MQTT-CEP architecture, the CEP-Broker has been instrumented to send back a message coming from Source, as well as a message from Receiver, in order to calculate an estimation of the one-way latency of the messages (see Fig. 10).

We assume here that the upward and backward latency are the same. So, the total time or latency (in seconds), L_{total}, from *SAVIA-Source* to *SAVIA-Receiver* can be defined as the sum of times of three sectors, as shown in the following equation:

$$Latency = L_{total} = L_{source} + L_{CEP} + L_{Broker}$$

where,

- L_{source} will be the time from the generation of a message from Source until it arrives at *SAVIA-Node*. It should be noted that, for the calculation of this value, and due to the fact that the MQTT Broker that receives the packets has its own messaging manager and is unfeasible to know exactly the time in which the alarm is distributed, a confirmation message will be sent. The shipment from SAVIA-Source will be made by subscription to the Broker and carrying its departure time. Then, as the *CEP-Broker* has been instrumented to send back a message to the SAVIA-Source, L_{Source} is measured as half of the round-trip time for the message to travel from Source and back again.
- L_{CEP} will be the time in CEP, that is, the time in which the data reach the CEP engine T_1'', is analysed and the complex event in the form of an alarm, T_2'' is obtained as output.
- L_{Broker} will be measured as the half of the round-trip time since the event leaves CEP, the alarm is published through the Broker and reaches the *SAVIA-receiver* and a confirmation message is back to *SAVIA-Node*.

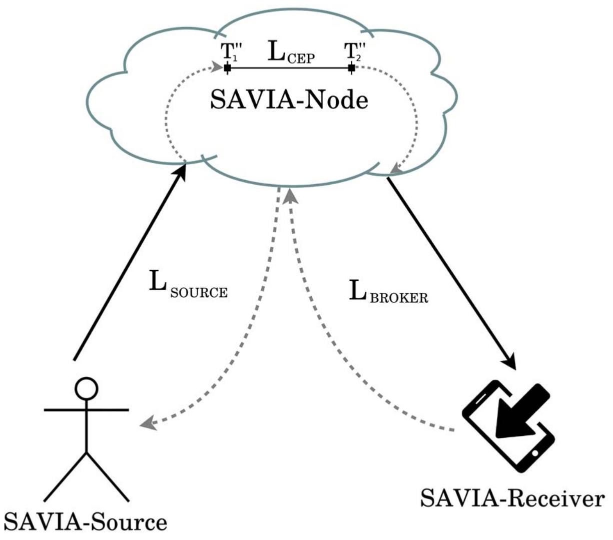


FIGURE 10. SAVIA abstraction model for testing with the interaction of the different entities.

As it can be seen from the latency graph, the time using MQTT-CEP is much higher compared to POST used in this study (see Fig. 9). In this sense, what is shown in this section is how this time difference between one system and another can be justified, and in what part exactly this increase occurs. This study only assesses the behaviour of MQTT-CEP with respect to latency, in order to verify if the point of loss is in: (i) sending/receiving, (ii) CEP engine or (iii) MQTT Broker.

Regarding the sending/receiving of the data, it has not been represented since it is a minimum and constant value regardless of the amount of data sent. This value has been, for all the alarms generated, 10^{-5} seconds. In this sense, the analysis will be carried out in the data analysis sections in the CEP engine and in the MQTT Broker.

The times obtained for L_{CEP} and L_{Broker} can be seen in Fig. 11. It must be taken into account that both the MQTT Broker and CEP Engine have their own Buffer and queue

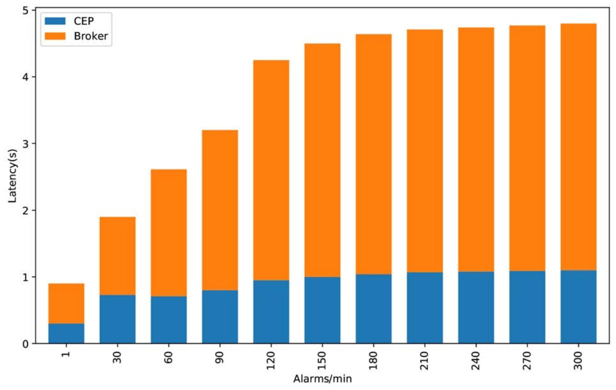


FIGURE 11. Representation of time for the MQTT-CEP architecture, specifically, the time in the Broker and CEP.

manager that distribute the packages according to their own internal management. In some initial tests, the configuration of these managers has been optimized in order to obtain the shortest response times. The values shown in the graphs are the results obtained after optimizing the configuration of the different internal processes of these two services (explained in Section 6.1).

On the one hand, it can be observed that the time taken to notify users of an alarm, L_{Broker}, increases as the system has to manage a greater number of alarms. This is because, as mentioned above, the broker sends an asynchronous notification to users. In addition, the MQTT Broker's own buffer distributes complex events as it queues them, so that with a greater number of events it can be observed that the distribution slows down.

However, L_{CEP} experiences a slight increase as there are more simple events to process in the buffer, presenting an almost constant behaviour, which indicates that the Apache Flink CEP engine is capable of adequately handling received alarms and is not the neck of bottle.

6.5. Energy consumption

Finally, results on energy consumption are reported in Fig. 12. Regarding the consumption of RAM (see Fig. 12a) and CPU (see Fig. 12b) it can be seen how MQTT-CEP remains almost constant while POST can be seen to have a point where the increase occurs abruptly from 150 alerts. This increase occurs since POST, when using a synchronous architecture, must open the different connections not only for receiving data, but also for sending data (or simple communication establishment).

On the other hand, it must be taken into account that to generate an alert in POST the CPU is constantly verifying distances as data arrives and in memory writing and reading both data. Therefore, the process of opening many connections in a small-time frame produces that there is a greater demand of RAM and CPU with respect to constant reading and writing in RAM and analysis of distances in CPU.

In the case of MQTT, having the CEP engine, which is a device that analyses simple events, we see that it remains constant. This makes sense since on the one hand we have an asynchronous communication given by the MQTT Broker so that multiple synchronized connections are not generated as POST, but as a complex event is generated, i.e. an alert, it is published in the Broker to send it to the recipient by subscribing to this.

6.6. Discussions

Having into account the results of the previous section, and in order to provide meaningful information that might be extrapolated to similar applications, Table 2 summarizes the main findings of the evaluation that has been carried out. Details will be discussed next.

Evaluation shows that when SAVIA is implemented as a MQTT-CEP architecture 3,2GB of RAM are needed, for this particular software deployment. So, this is a minimal compulsory requirement for the IoT nodes of SAVIA. Note that Apache Flink framework is based on stateful stream processing and SAVIA data are located in memory to yield good access time. This means that this can be a limiting factor for other similar applications, specially when run on devices with scarce resources.

From the aforementioned, results show that MQTT-CEP has a low computational overhead in the system. Moreover, this overhead remains at low levels even when the number of alerts increases; MQTT-CEP architecture is a good solution to be implemented in fog architectures where computational resources are limited. On the other hand, CPU usage in the POST architecture exhibits an increasing trend. However, the maximum CPU usage is around 3,5% for the range of tested alarm rates, which is not critical.

Regarding latency, SAVIA human-scale real-time requirements are a critical parameter in the MQTT-CEP architecture. Recall that HTTP works as a client-server model while MQTT uses a model of publication-subscription to a topic through a MQTT Broker. This MQTT Broker is an agent that manages the messages received and, according to the tests carried out, its performance significantly degrades as the number of messages published in a topic increases. This is in our case the number of alerts generated. Nevertheless, maximum latency values stabilize under 5 seconds, which is reasonable for the SAVIA use case. If SAVIA latency requirements were tighter (i.e. in control systems applications) these results could made the MQTT-CEP architecture unfeasible to meet the stringent timing requirements. In that case, the POST implementation yields lower and more predictable latency values.

Considering energy consumption, and since it may be a critical design aspect in fog architectures, it is observed that the MQTT-CEP architecture presents better performance. This architecture keeps energy consumption limited when faced with a high number of alarms. Hence, POST energy consump-

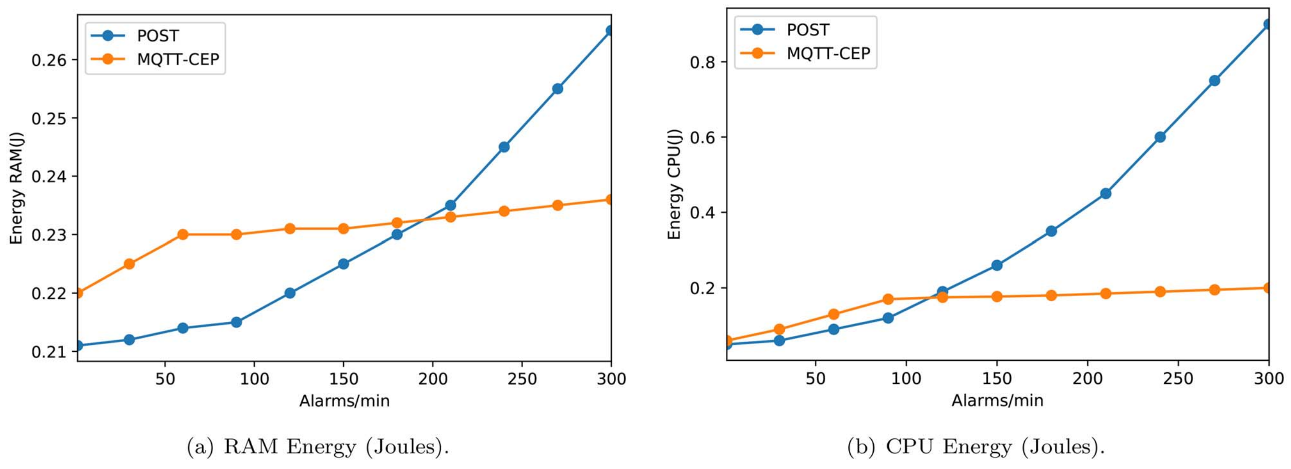


FIGURE 12. Graphs regarding the generation of alerts/min as a function of the energy in RAM and CPU for POST and MQTT-CEP.

TABLE 2. Framework comparison: POST vs. MQTT-CEP.

Parameter	MQTT-CEP	POST
RAM usage	Stable	Stable
CPU usage	Not critical	Increasing trend (but not critical)
Latency	Medium	Low
Energy	Low	Increasing trend
Alarm rate	Not limited with reasonable performance	QoS degradation beyond 300 req/min

tion increases with the alarm rate, and this is directly related to the increase in CPU usage. This might be a limiting factor if battery-operated devices are used.

Finally, it is important to note that the maximum alarm rate achieved in the experiments is limited by the maximum performance supported by the HTTP-POST architecture. Because of the large number of simultaneous database read/write accesses and their analysis to detect an alarm case, when 300 alarms/min were issued the system was not able to detect all of them; thus limiting the QoS this architecture is able to provide. The MQTT-CEP architecture does not exhibit such limitation, because data are processed on the fly by two different threads: one writes the data into the database and the other runs the CEP engine to perform the analysis. Therefore, the predicted application alarm rate can condition the architecture choice.

To sum up, and generally speaking, in applications similar to SAVIA, if latency is a critical element and predictable values are required, the HTTP-POST architecture would be more advisable, as it offers lower and more stable latency values. However, due to the organization of its components and the multiple queries on the database, the maximum admissible request rate is lower than MQTT, which can be a limiting factor. Note that indeed the assignment of these technologies to IoT applications is not always one-sided and may be different for a particular application depending on its specific requirements.

7. CONCLUSIONS AND FUTURE WORK

This work has developed two alternative solutions for deploying real-time applications for smart cities, namely, HTTP-POST and MQTT-CEP. A thorough analysis has been carried out comparing both approaches in terms of feasibility, scalability, latency and energy consumption. The CEP framework based on Apache Flink has been successfully integrated with MQTT, which has allowed us to obtain an event-driven implementation for the application selected as a use case, SAVIA. The experiments show that both architectures have strengths and weaknesses. A closer look to the application requirements in terms of latency and expected alarm rate should be considered in order to select the most adequate architecture.

As future work, a fog computing implementation of the MQTT-CEP architecture will be carried out, in order to make latency more adequate to more stringent real-time requirements of IoT applications while retaining the energy savings.

DATA AVAILABILITY STATEMENT

The data underlying this article will be shared on reasonable request to the corresponding author.

ACKNOWLEDGMENTS

This work has been partially funded by the Spanish Ministry of Science, Innovation and Universities (ref. RTI2018-098156-B-C52), by the Research Plan of the University of Castilla-La Mancha (ref. 2019-GRIN-27060), and by FONDECYT / World Bank (ref. 026-2019 FONDECYT-BM-INC.INV).

REFERENCES

[1] Sethi, P. and Sarangi, S.R. (2017) Internet of Things: Architectures, protocols and applications. *Journal of Electrical and Computer Engineering*, 2017, 1–25.

[2] Mahmud, R., Ramamohanarao, K. and Buyya, R. (2020) Application management in fog computing environments: A taxonomy, review and future directions. *ACM Computing Surveys*, 53, 1–43.

[3] Corral-Plaza, D., Ortiz, G., Medina-Bulo, I. and Boubeta-Puig, J. (2021) MEdit4CEP-SP: A model-driven solution to improve decision-making through user-friendly management and real-time processing of heterogeneous data streams. *Knowledge-Based Systems*, 213, 106682.

[4] Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M. and Ayyash, M. (2015) Internet of Things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17, 2347–2376.

[5] Heidari, A., Jabraeil Jamali, M.A., Jafari Navimipour, N. and Akbarpour, S. (2020) Internet of Things offloading: Ongoing issues, opportunities, and future challenges. *International Journal of Communication Systems*, 33, e4474.

[6] Madumal, M.B.A.P., Atukorale, D.A.S. and Usoof, T.M.H.A. (2016) Adaptive event tree-based hybrid CEP computational model for fog computing architecture. In *2016 Sixteenth International Conference on Advances in ICT for Emerging Regions (ICTer)* 1-3 September, pp. 5–12. IEEE, New Jersey.

[7] Sefati, S.S. and Navimipour, N.J. (2021) A QoS-aware service composition mechanism in the Internet of Things using a hidden Markov model-based optimization algorithm. *IEEE Internet of Things Journal*, 1, 1–8.

[8] Mondragón-Ruiz, G., Tenorio-Trigoso, A., Castillo-Cara, M., Caminero, B. and Carrión, C. (2021) An experimental study of fog and cloud computing in CEP-based real-time IoT applications. *Journal of Cloud Computing*, 10, 1–17.

[9] Mehmood, Y., Ahmad, F., Yaqoob, I., Adnane, A., Imran, M. and Guizani, S. (2017) Internet-of-Things-based Smart Cities: Recent advances and challenges. *IEEE Communications Magazine*, 55, 16–24.

[10] Hou, L., Zhao, S., Xiong, X., Zheng, K., Chatzimisios, P., Hossain, M.S. and Xiang, W. (2016) Internet of Things cloud: Architecture and implementation. *IEEE Communications Magazine*, 54, 32–39.

[11] Nasiri, H., Nasehi, S. and Goudarzi, M. (2019) Evaluation of distributed stream processing frameworks for IoT applications in smart cities. *Journal of Big Data*, 6, 32–39.

[12] Zhang, Z., Wu, J., Chen, L., Jiang, G. and Lam, S.-K. (2019) Collaborative task offloading with computation result reusing for mobile edge computing. *The Computer Journal*, 62, 1450–1462.

[13] Sadrishojaei, M., Navimipour, N.J., Reshadi, M. and Hosseinzadeh, M. (2021) A new preventive routing method based on clustering and location prediction in the mobile internet of things. *IEEE Internet of Things Journal*, 1–1.

[14] Flink, A. *Deployment & Operations – Configuration*. https://ci.apache.org/projects/flink/flink-docs-stable/ops/config.html. [Online; accessed June-2021].

[15] Corral-Plaza, D., Medina-Bulo, I., Ortiz, G., Boubeta-Puig, J.et al. (2020) A stream processing architecture for heterogeneous data sources in the Internet of Things. *Computer Standards & Interfaces*, 70, 103426.

[16] Hammi, B., Khatoun, R., Zeadally, S., Fayad, A. and Khoukhi, L. (2018) IoT technologies for Smart Cities. *IET Networks*, 7, 1–13.

[17] Osman, A.M.S. (2019) A novel big data analytics framework for Smart Cities. *Future Generation Computer Systems*, 91, 620–633.

[18] Harrison, C., Eckman, B., Hamilton, R., Hartswick, P., Kalagnanam, J., Paraszczak, J. and Williams, P. (2010) Foundations for Smarter Cities. *IBM Journal of Research and Development*, 54, 1–16.

[19] Borelli, E.et al. (2019) Habitat: An IoT solution for independent elderly. *Sensors*, 1, 1258.

[20] Naik, N. (2017) Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In *2017 IEEE International Systems Engineering Symposium (ISSE)*, pp. 1–7.

[21] Dizdarević, J., Carpio, F., Jukan, A. and Masip-Bruin, X. (2019) A survey of communication protocols for Internet of Things and related challenges of fog and cloud computing integration. *ACM Computing Surveys (CSUR)*, 51, 1–29.

[22] Chintapalli, S.et al. (2016) Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* Chicago, USA, 23-26 May, pp. 1789–1792. IEEE, New Jersey.

[23] Martínez, R., Pastor, J.A., Álvarez, B. and Iborra, A. (2016) A testbed to evaluate the FIWARE-based IoT platform in the domain of precision agriculture. *Sensors*, 16, 1–22.

[24] Alonso, A., Pozo, A., Choque, J., Bueno, G., Salvachúa, J., Diez, L., Marín, J. and Alonso, P.L.C. (2019) An identity framework for providing access to FIWARE OAuth 2.0-based services according to the eIDAS European regulation. *IEEE Access*, 7, 88435–88449.

[25] Agarwal, P. and Alam, M. (2020) Open service platforms for IoT. In Alam, M., Shakil, K.A., Khan, S. (eds) *Internet of Things (IoT): Concepts and Applications*. Springer International Publishing, New York.

[26] IBM. *IBM Watson IoT platform*. https://internetofthings.ibmcloud.com/. [Online; accessed June-2020].

[27] Sood, S.K., Sood, V., Mahajan, I. and Sahil (2020) Fog-cloud assisted IoT-based hierarchical approach for controlling dengue infection. *The Computer Journal*, 0, 1–13.

[28] Pfandzelter, T. and Bermbach, D. (2019) IoT data processing in the fog: Functions, streams, or batch processing? In *2019 IEEE International Conference on Fog Computing (ICFC)* Prague, Czech Republic, 24-26 June, pp. 201–206. IEEE, New Jersey.

[29] López Peña, M.A. and Muñoz Fernández, I. (2019) SAT-IoT: An architectural model for a high-performance Fog/Edge/Cloud IoT platform. In *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)* Limerick, Ireland, 15-18 April, pp. 633–638. IEEE, New Jersey.

[30] Castillo-Cara, M., Mondragón-Ruíz, G., Huaranga-Junco, E., Arias Antúnez, E. and Orozco-Barbosa, L. (2019) SAVIA: Smart city citizen security application based on fog computing architecture. *IEEE Latin America Transactions*, 17, 1171–1179.

[31] Balaji, S., Nathani, K. and Santhakumar, R. (2019) IoT technology, applications and challenges: A contemporary survey. *Wireless Personal Communications*, 108, 363–388.

[32] Glaroudis, D., Iossifides, A. and Chatzimisios, P. (2020) Survey, comparison and research challenges of IoT application protocols for smart farming. *Computer Networks*, 168, 107037.

[33] perf (2019). *perf: Linux profiling with performance counters*. https://en.wikipedia.org/wiki/Perf_(Linux). [Online; accessed June-2021].