A Stream Processing Architecture for Heterogeneous Data Sources in the Internet of Things

David Corral-Plaza*, Inmaculada Medina-Bulo, Guadalupe Ortiz, Juan Boubeta-Puig

UCASE Software Engineering Research Group

Department of Computer Science and Engineering, University of Cadiz,

Avda. de la Universidad de Cádiz 10, 11519 Puerto Real, Cádiz, Spain

{david.corral,inmaculada.medina,guadalupe.ortiz,juan.boubeta}@uca.es

***Corresponding Author:**
David Corral-Plaza
Address: School of Engineering, Avda. de la Universidad de Cádiz 10, 11519 Puerto Real, Cádiz, Spain
Telephone number: +34 956 015138
E-mail: david.corral@uca.es

Abstract

The number of Internet of Things (IoT) and smart devices capable of producing, consuming and exchanging information is constantly increasing. It is estimated there will be around 30 billion of them in 2020. In most cases, the structures of the information produced by such devices are completely different, thus providing heterogeneous information. This is becoming a challenge for researchers working on IoT, who need to perform homogenisation and pre-processing tasks before using the IoT data. This paper aims to provide an architecture for processing and analysing data from heterogeneous sources with different structures in IoT scopes, allowing researchers to focus on data analysis, without having to worry about the structure of the data sources. This architecture combines the real-time stream processing paradigm for information processing and transforming, together with the complex event processing for information analysis. This provides us with capability of processing, transforming and analysing large amounts of information in real time. The results obtained from the evaluation of a real-world case study about water supply network management show that the architecture can be applied to an IoT water management scenario to analyse the information in real time. Additionally, the stress tests successfully conducted for this architecture highlight that a large incoming rate of input events could be processed without latency, resulting in efficient performance of the proposed architecture. This novel software architecture is adequate for automatically detecting situations of interest in the IoT through the processing, transformation and analysis of large amounts of heterogeneous information in real time.

Keywords: Heterogeneous Data, Stream Processing, Internet of Things, Complex Event Processing, Software Architecture.

1. Introduction

In recent years, the number of devices able to produce information in a smart way has grown exponentially, leading to the Internet of Things (IoT). The IoT is the concept of connecting such devices to the Internet and to other devices, originating a large network of data producers collecting information from real-world entities.

An IoT device can be any intelligent device, such as a smartphone, sensor, tablet or wearable which is able to send or produce new information. These IoT devices can be part of a more sophisticated system that uses this information to process and analyse it in order to get knowledge from or feedback to any other IoT device. The number of these devices is increasing daily and it is estimated that by 2020 the number of IoT devices in the whole world will be around 30 billion [1]. Moreover, in the last few years, the number of means to produce, consume, and exchange information has grown significantly and the huge amount of data exchanged has led to the coining of the term Big Data. In addition to this term, the three V's (Volume, Velocity and Variety) of Big Data have emerged [2]. The first V refers to the large amounts of information generated by these systems. The second refers to the speed at which these data are generated by the sources, and the third V refers to the heterogeneity of these data.

One of the big challenges that developers and researchers find when working with such amounts of data from the IoT is the data format or structures. Although a certain consensus is being achieved regarding data protocols [3], such as MQTT or AMQP, and common semantics, such as NGSI, JSON, YAML or XML, a huge number of IoT systems still produce information in an unstructured and completely heterogeneous format, and this number keeps growing.

The challenge of homogenising data structure leads us to highlight two major issues: data processing and data analytics. Concerning data processing, it is mandatory to provide users with the necessary infrastructure in order to transmit and process such heterogeneous data. This infrastructure has to process and prepare such data for further analysis. On the other hand, concerning data analytics, once the data have been properly processed, data analyses have to be performed in order to detect situations of interest and perform the required actions when necessary. Both issues have to be solved in real time, necessitating the ability to process and analyse heterogeneous data to detect situations that will be promptly notified to the affected users.

Let us illustrate the problem in the smart water management domain. These organisations need to detect certain situations of interest such as leaks in pipes, frauds in water bills, anomalies within water networks, etc. In order to detect such situations, these companies need to analyse the data from their smart water meters. In this kind of scenario, organisations usually have several areas composed of a large number of IoT sensors, which can read values such as the total volume of the water transmitted, the time supplying water, the time without providing water, etc. These networks are commonly built up of different IoT sensors from diverse manufactures, thus producing information with a different structure or even different data, in other words, producing heterogeneous data. Given this scope, these organisations need a system or platform that enables them to work with the heterogeneous data coming from their different IoT sensors, in order to analyse these data so the previously mentioned situations can be detected.

Hence, the required system has to consume or receive this heterogeneous data from the smart water meters and perform a homogenisation to make the data ready to work with. Then, once the data are suitable to use, it is necessary to perform an analysis in order to infer and detect situations of interest for the company. Performing the homogenisation manually or individually for each incoming amount of data can be very costly. Furthermore, all these steps should be performed in real time, so organisations can react as quickly as possible to the alerts detected by this architecture.

Therefore, most systems that process heterogeneous data sources [4–6] are forced to perform a pre-processing or normalisation task to be able to analyse such data. The data normalisation stage usually requires many processing tasks, consuming resources and time. Moreover, it is common to store this information in order to reuse it later, so storage resources are also needed.

Some approaches have attempted to perform the homogenisation on the sources [7], which is not efficient enough when considering large IoT networks with huge amounts of devices as sources. Others [8] have opted for periodically processing batches of this heterogeneous information in order to normalise it. However, this means information is not being processed in real-time, so situations of interest may go unnoticed and not be detected ahead of time.

To tackle these problems, there are certain key features that any architecture designed for processing and analysing heterogeneous data has to implement in order to properly provide such functionalities [9,10]. These features are as follows: scalability, Stream Processing (SP), data analytics, data storage, multiple event types, predictive tools, real time and performance evaluation. These features are presented and described in Section 4.1.

The main contribution of this paper is thus an architecture which permits the homogenising, processing and analysing of heterogeneous and unstructured data in order to detect situations of interest in real time. The data this architecture is able to process are obtained from IoT sensors observing the context and providing structured or unstructured data about it, i.e., air quality sensors, water management, smart parking, etc. The data may come in structured (JSON/XML) or unstructured (CSV as raw data) formats. The architecture is based on emerging technologies such as SP, Data Serialization Systems (DSSs) and Complex Event Processing (CEP). SP platforms allow us to build streaming applications that transform or react to data streams. They are scalable and several machines can be used to obtain higher performance in our application. Using them, we are able to develop applications that consume heterogeneous information, perform a series of transformations, and make this information ready to work with in real time. DSSs allow us to transform information into a compact binary format. The information size decreases and, for that reason, the overall performance of the solution improves. DSSs are well-embedded with several programing languages such as Java and are very useful when working with data in several formats, since they enable all this information to be homogenised in order to process it. CEP technology allows us to analyse information and react to it in real time. In CEP, streams of information, as simple events, are received and compared with previously defined patterns. These patterns contain certain conditions that the simple events stream has to satisfy in order to generate more meaningful events in real time.

In addition, we tested the performance and the behaviour of the proposed solution in a real case scenario related to water treatment and management. We integrated multiple sources of heterogeneous data, pre-processed them to achieve the homogenisation of this data, and analysed it to detect situations of interest using the CEP engine. Moreover, stress tests were carried out to evaluate the architecture performance and scalability.

To sum up, the main aim of this study was to provide an architecture able to combine the capabilities of a CEP engine with the performance of an SP platform for consuming, processing, and transforming messages with DSS. As a result, we get a complete scalable system able to consume and transform huge quantities of heterogeneous data sources from the IoT and detect situations of interest in real time.

Finally, we aim to answer the following research questions:

- (RQ1) How can heterogeneous data be structured in an IoT domain?
- (RQ2) What processes have to be conducted to homogenise heterogeneous data coming from IoT sensors?
- (RQ3) Is the proposed architecture (a combination of Stream Processing, Data Serialization Systems and Complex Event Processing) able to process, homogenise and analyse heterogeneous information?
- (RQ4) Is the proposed architecture suitable for IoT domains?
- (RQ5) Is the proposed architecture more effective and efficient than other existing architectures when required to handle, process and analyse large amounts of heterogeneous data in real time?

The rest of the paper is organised as follows. Section 2 describes the related work. Section 3 describes the technologies used in this work. Section 4 presents the requirements and our architecture for heterogeneous data processing. Section 5 describes the case study in which the proposed solution has been tested and its implementation. Section 6 explains the experimental validation conducted, the results obtained, the comparison of our proposal with others and the answers to the research questions. Finally, Section 7 highlights the conclusions of the proposal and the future work lines.

2. Related Work

In this section, we describe proposals in the literature about integrating heterogeneous data. In Section 5, the domain of application is Smart Water Supply Management, and thus studies in this particular domain are discussed. Additionally, we will describe proposals that deal with IoT heterogeneous data processing through the use of Apache Kafka and other technologies.

2.1 Water Supply Management

Concerning systems specifically implemented for water management, three important approaches and a European Project can be mentioned.

Mohammed Shahanas et al. [11] proposed a smart water management system to generate alerts when water levels are below certain limits. They deploy water level sensors alongside an Arduino board to perform the data collection task. Then, with Echo and Push modules, these data are sent to the central server. The central server is a Raspberry Pi, where the data analysis is performed. Thus, when the water level goes below a certain threshold, an email or SMS is triggered. In addition, the data are sent to a cloud platform to be visualised, and are stored in a local database.

Narendran et al. [12] presented a system designed for sustainable water management, organised in four layers. The first layer consists of monitoring and management modules, which are sensors that collect information about water sources. The second one consists of a networking layer that collects the data from the previous layer and sends them to the next level. The third layer is a middleware, which is composed of several functionalities, such as monitoring modules associated with resources or distribution and storage. The fourth layer hosts an alert system for notifying alerts to the local community regarding the water level.

Additionally, Salvi et al. [13] proposed an architecture for monitoring and analysing a smart multi-level irrigation system. This architecture is composed of several nodes, which collect information from sensors. These sensors send the information to a second layer where master nodes are in charge of performing communication between single nodes and the cloud server. ThingSpeak is used as the cloud server, because it provides functionalities to analyse and visualise the collected data.

All of these works on water management present a weak analysis layer, since they do not use any technology that supports real-time analysis of the information collected from the sources. Moreover, they collect information from request/response platforms such as ThingSpeak, rather than using a SP or messaging system, which permits efficient data collection.

Finally, we would like to highlight Smart Water 4 EU (SW4EU) [14], one of the first large projects to address smart water management in Europe. In this project, over 21 participants, including companies and universities, worked together to establish the foundations of smart water networks in Europe, focusing on water quality management, leak management, energy optimization and customer interaction. Their aim was to provide these features in real time, through modelling for decision and control support and automation strategies. The SW4EU approach and results were presented worldwide at different conferences and symposia.

2.2 Heterogeneous Data Processing

Among the approaches that use Kafka to integrate several data sources or even as a messaging system in an architecture, the following can be highlighted.

Carcillo et al. [15] proposed an architecture to detect credit card fraud, using streaming analysis with Kafka, among other technologies. In this case, Kafka is used for fault-tolerant transaction collection to send the data streams to their Spark module for fraud detection. SCARFF gets an incoming rate of 240 transactions/s, probably motivated by the high cost of executing machine-learning algorithms.

Stripelis et al. [4] presented an architecture to prevent and predict asthma attacks integrating several data sources through Kafka, using SparkR and Spark MLlib to build prediction models for asthma attacks. They perform data homogenisation using a Mediation layer, which maps the schemas from the heterogeneous data sources into a common harmonised schema for the analytics components. Moreover, there is no evaluation of the solution’s performance.

The study by Zeydan et al. [5] also deserves special attention. These authors presented an architecture to predict upcoming alarms based on the analysis of current streams of alarm information. They perform the homogenisation of the heterogeneous sources in a first module. This information is then sent to a Kafka Cluster, which it is used along with Apache Storm to detect and predict these alarms. In addition, these predictions are sent to a Web client.

Amini et al. [16] proposed a comprehensive and flexible architecture for real-time traffic control. Among other software, their architecture uses Kafka for input data management and communication between modules. The data are received in well-formed JSON, so the pre-processing of the data for use is unnecessary. They carry out a simulation, but do not present any numerical results related to the performance of the given architecture.

D’Silva et al. [17] also proposed an architecture to process real-time and historic streams of IoT data. Any kind of IoT device is used as a data source that sends information to a Kafka Cluster. Then, Kafka streams these data to Spark, which performs the analysis. The results are presented in a dashboard. They present an evaluation in which they test their architecture simulation sending 1 000 simple events to the Kafka Cluster.

Finally, Jung et al. [18] developed UTOPIA, an architecture for processing real-time streams of information in the cloud. In this case, Kafka is used to receive the real-time transmission of a large-capacity data stream. Then, the evaluation of this information is performed using Storm. In addition, they present an evaluation in which the maximum incoming rate of the platform is 10 000 stream data per second.

None of these approaches benefits from the advantages of using Kafka as a SP platform (as we do in our proposal); it is used purely as a messaging system. In fact, most of the transformations they perform can be done more efficiently using Kafka Streams API rather than creating different modules to carry out those tasks.

Furthermore, some additional approaches provide alternative ways to process heterogeneous data. For instance, Hu et al. [19] proposed an architecture to process heterogeneous streams of data, distributed in three layers. The first layer is composed of sensors as data sources. The second layer is a middleware in which a Global Sensor Network (GSN) is used. The last layer is used for the analysis, which is carried out using Apache Storm. The data sources are integrated using already existing wrappers in the GSN middleware.

Montori et al. [20] presented an architecture able to integrate data from different sources, such as public cloud platforms like ThingSpeak, to open data provided by governments. In their proposal, they have an orchestrator, which is able to return a service record or a data stream, given a set of parameters determined by the user’s choice. In addition, other users are able to publish their own services as an endpoint for everyone.

Montori et al. [6] also proposed SenSquare, a data platform in which they run the experiments and algorithms presented. In SenSquare the data sources can be reliable, those that come from official or government sources, or unreliable, those that come from open data platforms. Moreover, end-users can be producers of this information using a coded smartphone application. The homogenisation task is mandatory for unreliable sources, so they designed a classification algorithm that uses certain mandatory parameters in order to perform the classification of such data with an accuracy of 80%.

In addition, Oteafy [21] presented HetSense-BSD, a framework to consume data from heterogeneous sources (Wireless Sensor Networks, IoT devices, etc.). Using a local BSD mediator with some other modules, they are able to match requests from users in order to get data from the sources. The mediator is able to receive the request, retrieve the information from the heterogeneous sources through their protocols, and provide it to Information services.

Additionally, Santos et al. [22] proposed PortoLivingLab, an urban-scale, multisource sensing infrastructure deployed in the city of Porto, Portugal. They use SenseMyCity (a mobile crowdsensing research tool to gather sensor data from participants’ smartphones), UrbanSense (a platform for monitoring environment values in a city) and BusNet (an infrastructure that enables vehicle-to-vehicle and vehicle-to-infrastructure communication). All the information is stored and processed in the services they run in a backend.

Finally, Estévez-Ayres et al. [23] proposed an architecture that analyses educational data from heterogeneous data sources in real-time. The students produce events when they use educational resources thanks to monitoring agents. These events are sent to the analysis infrastructure and stored for historical records. In the analysis layer, they run services to monitor how students work in groups and to monitor how students work in specific assignments. Moreover, they perform an evaluation of the proposed solution that generated 384 702 events in two academic years, all of them processed in real time.

Most of these proposals, which do not use Kafka, do not benefit from the multiple advantages offered by SP platforms to be able to work with large amounts of information. In addition, not all of them consume and process these data in real time, but store and process them in batches, which consumes additional resources and makes it difficult to instantly react to situations of interest.

3. Background

This section describes the technologies used in our proposed architecture.

3.1 Stream-Processing Platforms

For many years now, well-known messaging systems have allowed us to send information synchronously or asynchronously, but the concept of SP platform [24] has recently emerged. SP is the processing of data in motion, or in other words, computing data directly as it is produced or received.

SP has been proposed to satisfy the requirements of processing and transforming huge amounts of information in real time. Situations in which SP fits perfectly include IoT sensor events processing, financial trades, smart cities or, website traffic. SP can be applied to any kind of scenario where the volume, velocity, and variety of the data are huge.

The advantages [25] of SP platforms include the following:

- SP platforms can process unlimited amounts of data; performance will depend on the solution and the transformations required.
- SP is event-based; these platforms react to input data and functionally perform the transformations over the data.
- SP platforms compute the data directly, without storing it, so storage resources are not critical in these applications.
- SP platforms can easily scale in horizontal adding nodes for processing more data, without affecting performance.

Among the different platforms that currently exist, we highlight Apache Kafka [26], Apache Spark [27] and Apache Flink [28]. Kafka has a specific Application Programming Interface (API) designed for SP: Kafka Streams API, which allows us to create Streaming Applications that consume data from Kafka input topics, process and transform them, and publish the results in a Kafka output topic. Apache Spark also presents a Streaming API to build Streaming Applications. In Spark, we can consume information from different sources like Kafka topics or Hadoop, perform the required transformations and publish the processed data in the same platforms we used as sources. Apache Flink also provides SP features combined with data analytics. In Flink, the information will be retrieved from several sources such as Kafka, Hadoop or Twitter, the transformations and analytics required are performed, and the results can be sent to specific endpoints, quite similar to the ones used as input sources.

In Kafka Streams, the information is processed at the same time as it is consumed from the Kafka input topic (very low latency), while in Spark or Flink the information is processed in micro-batches from the input data streams sources. Spark and Flint present not only SP capabilities, but also analytics capabilities at the same time as the information is being processed. Finally, choosing one or another will depend on the context and application, with Apache Kafka being more suitable for processing small messages but at a high input range (requiring low latency), i.e. in a IoT scenario, while Spark or Flink are more focused on processing bigger messages with greater complexity [29].

3.2 Data Serialization System

When huge amounts of information have to be processed, an operation which includes the serialization and deserialization of these data is required. In computer science, serialization is the process of converting data structures, objects or information into a format that is easy to store or transmit. Once the data are transmitted, the deserialization process is performed, in order to reconstruct these data and use them.

Therefore, a DSS is a technology allowing us to serialize and deserialize messages. Such a system should process these messages as quickly as possible. Some DSSs use schemas, which are a representation of the data structure being transmitted, and are highly useful for the homogenisation process. A DSS is used in several scenarios, including database storage, processing streams of data or communication in messaging systems. The choice of a DSS for an application depends on factors such as readability, data complexity or speed necessities.

Various alternatives for use as a DSS currently exist, the most widely used being Apache Avro [30], Apache Thrift [31] and Google's Protocol Buffers [32]. All of these three DSS are platform independent, which means they can be used in several programming languages.

Apache Avro is a library for DSSs that relies on schemas that represent the structure of the data. Avro provides a serialization and deserialization process, which is able to transform the data to be transmitted or stored quite quickly. Apache Thrift is a more complex DSS than Avro, providing more features than just serialization or deserialization. It is also based on schemas to represent the data to be computed. Finally, Google Protocol Buffers is a DSS based on XML, in which you represent the data structure in a simplified XML syntax. These files can be reused in order to transmit, in a light manner, the data through your architecture.

3.3 Complex Event Processing

CEP is an emerging technology that allows us to process, analyse and correlate large amounts of information in the form of events —a simple event is a representation of a change of state— with the aim of detecting situations of interest (complex events) in real time [33].

The purpose of CEP is to analyse information as it arrives at our system, so we do not have to store it in a persistence system if not required. CEP is focused on analysing, processing and reacting to streams of events, which makes it a perfect tool to be combined with the previously mentioned SP platforms as will be seen in the evaluation in Section 6. We define the situations to be detected through the use of complex event patterns whose syntax depends on the CEP solution to be used. The information, as events, is compared to these previously defined patterns and triggers complex events (a more meaningful event which represents a situation of interest) when the conditions specified on the patterns are satisfied.

Regarding CEP solutions, there are several options, such as Esper [34] or Apache Flink. Esper is a well-known CEP engine solution that can perform data-stream analytics over information, which is represented as simple events, and can detect specific alerts or situations that are defined using their syntax, all in real time. Apache Flink, as mentioned, presents SP features combined with certain CEP capabilities. Flink allows us to define some transformations over streams of data, and then performs data analytics with these processed data. In addition, other alternatives, such as Apache Spark or Apache Storm [35], may be able to perform CEP analytics using specific modules, embedded in these solutions, but their main goal is not to perform CEP with the data they process.

In Esper, the information is sent as simple events using its API, which means that the information is directly processed, while in Flink, Spark or Storm, we have to connect our application to a data source from which the information is consumed in micro-batches. In all these solutions, complex data analytics can be performed using several operators, such as sliding and batch windows, aggregation functions or custom operators. Finally, an important advantage of Esper is that it does not need to be stopped in order to add new configurations at runtime, while in Flink, Spark or Storm it is necessary to stop the application in order to add or modify current configurations.

4. Architecture for Processing Heterogeneous Data in the Internet of Things

In this section, we first describe the required features an architecture should have in order to consume, process, and detect situations of interest from heterogeneous data in the IoT. The proposed architecture is then presented.

4.1. Architecture Requirements

In order to provide the required functionalities, an architecture for consuming, processing, transforming and detecting situations of interest from heterogeneous data sources in the IoT requires certain features. These will determine whether the proposed architecture can efficiently and effectively accomplish the desired functionalities. Below, we enumerate and describe key features required in a heterogeneous IoT data processing solution:

- (F1) Scalability: the system must be scalable, allowing the management of larger amounts of information. In most cases, more sources of information may be added to the existing architecture, and thus it should be able to properly handle such an increase in IoT data volume [9].
- (F2) Stream Processing: the system should use SP in order to process the information as it occurs. SP technologies allow us to perform transformations of the data as we receive them and to analyse them. Thus, SP is key in providing the required functionalities in real time [9].
- (F3) Data Analytics: the system must provide at least one mechanism to analyse these heterogeneous data. One of the most important advantages in the scope of IoT is the ability to analyse the data we receive in order to detect situations of interest and notify them to interested agents [10].
- (F4) Data Storage: the system should provide a data storage layer in order to save important information. This information can be useful in other processes or for historic analytics. Moreover, due to the volume of data in the IoT, these storage services should be able to handle large amounts of data [10].
- (F5) Multiple Event Types: the system should accept multiple event types as input sources, in structured (JSON, XML, Java Map…) and unstructured formats (CSV, Strings, raw data…). This feature will provide the architecture with the ability to process and integrate heterogeneous data sources, being suitable therefore for real scenarios [9].

- (F6) Predictive Tools: the system may also provide the developers with Machine Learning (ML) or Predictive technologies in order to improve the quality of the service and results provided [9].
- (F7) Real Time: the system should process the received data in real time or near real-time. If we can perform all the provided functionalities immediately, we will be able to react to the situations of interest as soon as possible in the most efficient way [9,10].
- (F8) Performance Evaluation: an evaluation of the proposal should be available in order to prove its correctness and effectiveness. Performance is key in many IoT scenarios; therefore, this should have been evaluated so as to provide organizations with enough information to judge the quality of the proposed solution and its suitability to the scenario in question [9].

Taking these features into consideration, we propose the architecture described in the following subsection.

4.2. Proposed Architecture

Our specific goal is a data stream architecture structured in three layers (see Figure 1): data sources, data processing and data consumers.

Data sources produce the heterogeneous information that will be used in the data processing layer. This processing layer makes use of the previously described technologies in charge of heterogeneously processing and analysing data sources. Then, data consumers receive the processed information, which provides them with valuable knowledge.

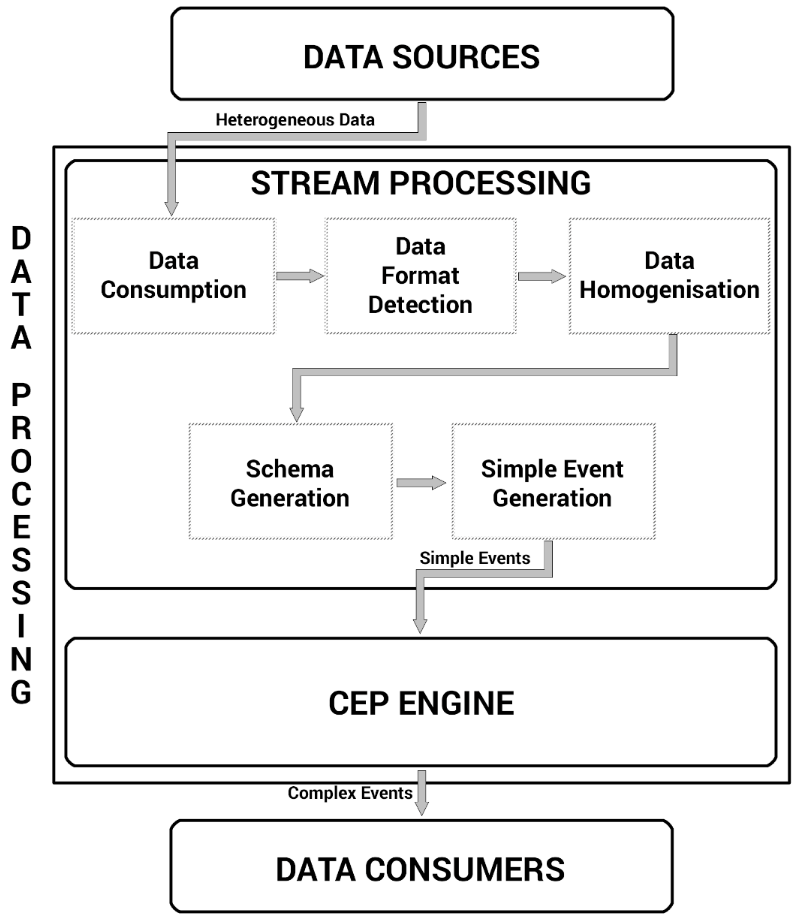

Fig. 1 – Proposed architecture

4.2.1 Data producers

This layer is composed of devices that can send heterogeneous information to a specific Stream Platform. Here, we could have devices related to IoT, social networks APIs, remote sensors, SQL and NoSQL databases, etc. There are no restrictions about the data structure, because the data we receive and process in our architecture are heterogeneous. These heterogeneous data are sent or consumed by the SP platform that will process them.

4.2.2 Data processing

This layer is in charge of transforming and processing the data received from the sources. We can distinguish two components, as explained in the following subsections.

4.2.2.1. Stream Processing

This is the main component in our data processing layer. It is in charge of receiving the heterogeneous data from several input data sources and readying it to be used in the analytics.

This platform must be scalable, so it can process large amounts of input data without affecting the processing performance. In addition, this platform should be able to work in real time, i.e. process the heterogeneous data as they occur.

The homogenisation is achieved thanks to the use of the DSS. Combining SP with a DSS, we can consume heterogeneous messages, process and transmit them quickly and easily [36]. This whole process is fully automatic; human intervention is not necessary at any time. The homogenisation process comprises the following tasks:

- **Data Consumption.** The heterogeneous data are received or consumed from the data sources. How and when depends on the solution. A data collector layer might be implemented in order to gather these heterogeneous data.
- **Data Format Detection.** The data are processed in order to infer the format in which are represented, such as JSON, XML, YAML or raw data.
- **Data Homogenisation.** The required transformations are performed using SP capabilities along with DSS features in order to homogenise the data.
- **Schema Generation.** Together with the previous task, the schema of the data transmitted is generated, this schema will be used in the next step.
- **Simple Event Generation.** Once the data are homogenised and the schema has been retrieved, the heterogeneous information previously received are transformed into simple events to be used for the analytics.

Let us illustrate this process taking the same example that we used in the introduction, the smart water management domain. The heterogeneous data produced by these smart water meters are sent or consumed in the stream processing application. Here, we analyse these data in order to detect the data format in which they come (CSV, JSON, XML, raw data, etc.). Once the data format is detected, the data are homogenised using SP and DSS features. Meanwhile, the schema of these smart water meter readings are created. Finally, once the smart water data are homogenised, they are used to generate the simple event. This simple event, which represents a reading from a smart water meter, is sent to the CEP engine in order to be used as input in this data-stream analytic technology.

4.2.2.2. CEP Engine

The CEP engine is the main analytic module in our architecture. It is in charge of detecting situations of interest. It receives the already homogenised data from the SP platform as simple events and evaluates them against the patterns. These patterns define the situations of interest to be detected. If the conditions of a pattern are satisfied, a new complex event is generated and notified to the data consumers. This evaluation is performed in real time in order to act as quickly as possible when the situation defined in the pattern is detected by the CEP engine.

4.2.3 Data consumers

This layer represents the end-user devices or services, which consume the processed information, or, in other words, the complex events detected that represent situations of interest.

In this layer, we can locate persistent store systems such as SQL and NoSQL databases; smartphones that receive push notifications about situations related to their owners' contexts, or other services that could use the information already processed as data sources, etc.

5. Case Study Description and Implementation

In this section, we present the case study, in which the proposed architecture was tested. In addition, the selection and the implementation of the technologies for the architecture are described.

5.1. Case study description

In order to demonstrate the feasibility of our proposal, we tested our architecture in the domain of water supply networks. Ostfeld's definition of water supply networks in [37] is well-known: "*A water distribution system is a complex assembly of hydraulic control elements connected together to convey quantities of water from sources to consumers*". Thus, the main purpose of these supply networks is to deliver water from sources to consumers with appropriate quality, quantity and pressure.

Indeed, water is a vital socioeconomic resource. The growing demand for water, for both domestic and industrial purposes, threatens the sustainability of groundwater, and affects agriculture, industry and drinking water. This situation is further aggravated by hidden leaks, which cannot be detected until the meter is read, causing excessive water consumption and customer billing [38]. Furthermore, fraud in the water supply networks is currently another problem to take into account. Therefore, strategic, safe, efficient and sustainable resource management is now a key challenge. Such a challenge is currently being tackled by smart water supply networks; water distribution systems with sensors deployed in their network. These sensors can collect information on water volume, time periods supplying water, time periods with lower consumption, etc. This information is then sent to the main central processing server, which will transform, process and analyse the information.

Recently, the local company in charge of the water management supply in the city of Puerto Real (Spain), *Grupo Energético de Puerto Real S.A.* (GEN) [39] contacted us with the aim of finding technical support to make profitable use of their emerging smart water supply network. Their water supply network is composed of an increasing number of smart water meters and they require an appropriate software architecture to take advantage of the information they obtain from them. Therefore, the main goal is to manage such a network more efficiently, and in real time, in order to meet the objective of optimising water management and consumption.

This goal can be met by real-time detection of situations of interest for the water supply enterprise, such as leaks in the supply network and in indoor facilities, poorly dimensioned water meters, anomalies in consumption patterns and prospective frauds, among others.

The network information the enterprise provided us with comprises 111 smart water meters, which compute several values throughout the day. From each water meter reading, we can obtain the following values:

- **serialNumber**: the serial number of the water meter.
- **dateTime**: date and time of the water meter reading.
- **volumeM3**: the volume of water detected by the water meter in cubic meters.
- **volumeL**: the volume of water detected by the water meter in litres.
- **type**: the use of the water meter (domestic, industrial, etc.).
- **starts**: number of times that the water meter has detected water passing through.
- **batteryLevel**: battery level of the water meter as Integer.
- **batteryLevelStr**: battery level of the water meter as String.
- **sleepingTime**: time in seconds without consumption.
- **leakingTime**: time in seconds from which the water meter detects an unusual or low consumption.
- **normalTime**: time in seconds that the water meter works without problems.

Listings 1, 2 and 3 show how these events of smart water meters can be structured. The information produced by the smart water meters will be homogenised using SP and DSS. This information is then sent to the CEP engine as simple events in order to detect situations of interest in real time.

As an initial approach, the situations GEN requested to be detected are the following:

- **Reading errors**: happen when one or more of the sensors of a specific water meter is not working properly, i.e., negative values in water consumption.
- **Water leaks**: happen when the meter detects no start but there is consumption.
- **Unusual consumption**: happens when the consumption value is not the usual, i.e., consumption equal to zero over a period of time.

Listing 1 – Input event as JSON

```
{
    "eventTypeName": "WaterMeasurement",
    "serialNumber": "A87SAA77188",
    "dateTime": "2018-07-06T05:00:00.000Z",
    "volumeM3": 5.4366397,
    "volumeL": 5436.6397,
    "type": "INDUSTRIAL",
    "starts": 1048,
    "batteryLevel": 3,
    "sleepingTime": 6739610,
    "leakingTime": 45816,
    "normalTime": 47613,
    "batteryLevelStr": "ALTO"
}
```

Listing 2 – Input event as XML

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <eventTypeName>WaterMeasurement</eventTypeName>
    <serialNumber>A87SAA77188</serialNumber>
    <dateTime>2018-07-06T05:00:00.000Z </dateTime>
    <volumeM3>5.4366397</volumeM3>
    <volumeL>5436.6397</volumeL>
    <type>INDUSTRIAL</type>
    <starts>1048</starts>
    <batteryLevel>3</batteryLevel>
    <sleepingTime>6739610</sleepingTime>
    <leakingTime>45816</leakingTime>
    <normalTime>47613</normalTime>
    <batteryLevelStr>ALTO</batteryLevelStr>
</root>
```

Listing 3 – Input event as raw data

```
WaterMeasurement, A87SAA77188, 2018-07-06T05:00:00.000Z, 5.4366397,
5436.6397, INDUSTRIAL, 1048, 3, 6739610, 45816, 47613, ALTO
```

5.2. Technologies

This subsection describes the selection of the technologies solutions. In order to implement the architecture defined in Figure 1, we needed a Stream Processing platform, a Data Serialization System and a Complex Event Processing engine. In this case, we chose Apache Kafka as our SP platform, Apache Avro as our DSS, and Esper as our CEP engine.

5.2.1 Apache Kafka

Among the different options that currently exist for SP, we chose Kafka [26], which is emerging as one of the most commonly used SP platforms. Kafka has some unique advantages, i.e., message persistence in disk and capacity of sending messages at a tremendous rate, asynchronously. These characteristics are helpful in implementing high throughput in our applications [40]. There exist other alternatives to Apache Kafka, such as Apache Spark or Flink, but they mainly focus on data analytics rather than on SP features, which is the main purpose in Apache Kafka. In Section 5.2.3, we describe and justify the technology chosen for data analytics.

Although Kafka can be used as a messaging system, the Kafka ecosystem is composed of several APIs, one of which is especially designed for SP, Kafka Streams API [41]. Kafka Streams are designed to build SP applications in order to read data from a Kafka input topic, then perform a series of transformations on these data and, finally, publish the results in another topic or service in the system. The main advantages of Kafka Streams are the following: (1) Kafka has no external dependencies on systems other than Apache Kafka itself; (2) Kafka supports exactly-once processing semantics to guarantee that each record will be processed once and only once, even when there is a failure; and (3) Kafka is easy to integrate with other services thanks to its API.

In a Kafka SP topology, we can identify three main elements:

- **Source**: a Kafka input topic, where all the information to be processed is sent. It is used as the start point of an application. This input topic can have as many partitions as necessary, and for each partition there will be a processor instance.
- **Processor**: a processing node of the Kafka Stream Application, which consumes the information from a specific partition of a topic. Then, it performs the necessary

transformations in order to prepare the data for being analysed. In this component, the homogenisation task is performed.

- **Sink**: an endpoint of the Kafka Stream Application, which defines the output topic where the information, already processed, is stored.

5.2.2 Apache Avro

Among the different options that currently exist, Avro is regarded as one of the best due to its simplicity of use, its performance [42] and its integration with almost any programming language or technology [36].

Other systems include Java Serialization, such as Thrift and Google's Protocol Buffers, which only work with compile time code generation. However, this is not a requirement with Avro, and thus it can provide a more optimized runtime performance.

Apache Avro is part of the Apache project so its integration with other Apache tools and Java language is very simple. It is used in our proposal for the data homogenisation task. Avro relies on schemas, which are a way of representing the structure of the data to be processed. These schemas are represented in JSON, which makes it even simpler to work with them.

Once a Kafka Streams processor detects a new message in a Kafka input topic, Apache Avro is used in the homogenisation task, so the schema of these data can be obtained regardless of the format in which data are received. Therefore, Apache Avro is essential for processing heterogeneous data efficiently.

5.2.3 Esper

We chose Esper as our CEP engine solution for several reasons. First, its performance is good, being able to process up to 500 000 events/s on a dual 2GHz CPU. Second, it enables rapid development of applications that process large volumes of incoming messages or events, regardless of whether incoming messages are historical or real-time in nature. In addition, one of the main advantages of Esper over other alternatives like Flink, Spark or Storm is the possibility of adding new configurations at runtime, making it unnecessary to stop the whole system when it is necessary to define and detect new event types or situations of interest. This feature is essential when building real-time systems, because stopping them for just a few seconds can mean loss of essential data that can be used in the detection of situations of interest.

Esper input data are simple events represented by using Plain-Old Java Object (POJO), Java Maps, Object-array, Apache Avro, or XML. Moreover, Esper provides an Event Processing Language (EPL), a SQL-standard language with extensions, which is used to define conditional statements. EPL offers SELECT, FROM, WHERE, GROUP BY, HAVING and ORDER BY clauses. In addition, EPL statements can specify event windows, based on conditions such as length and time. When a pattern is satisfied, it automatically generates a new complex event. These complex events can be used to feedback to the CEP engine in order to detect new complex events based on those already detected.

5.3. Implementation

In this subsection, the implementation of the architecture for the Smart water management domain is presented. The result of applying Kafka Streams, Avro and Esper in the proposed architecture for a Smart water management domain is shown in Figure 2.

In this case, our proposed architecture has water meters as data sources. In the processing stage, we have two modules: (1) A Kafka Cluster where the Kafka topics are created to store and transmit the heterogeneous data; and (2) a Kafka Stream Application which has two processors, which will homogenise the data, and a CEP engine embedded that will analyse these data. Finally, a web application is used as data consumers to show the situations detected by the CEP engine.

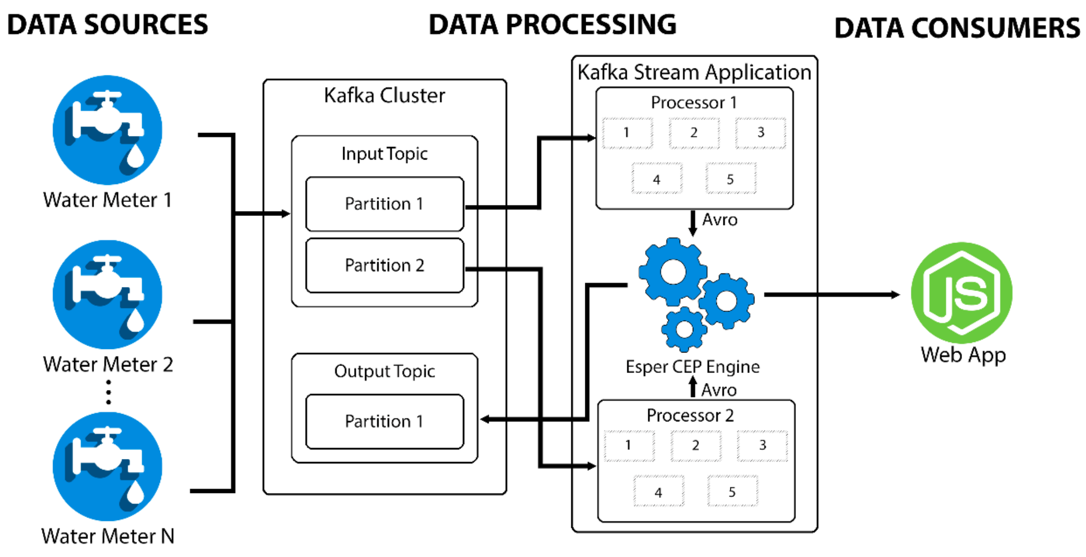

Fig. 2 - Proposed architecture adapted to the water management scenario

5.3.1 Data Sources

The data sources are heterogeneous data from several sources. In this case, the data sources are the 111 smart water meters comprising the smart water networks. Although the architecture can process heterogeneous data, such data must meet minimum requirements. The data produced by these sensors can be sent to the Kafka input topic using the following formats:

- **Structured formats**, such as JSON (see Listing 1) or XML (see Listing 2), in which each value has a key or a label (e.g. *eventTypeName*) to identify the value represented.
- **Unstructured format** (raw data), which could be, for example, a CSV row (Listing 3), where there is no information identification related to the values, only a string with values separated with a special character.

All this information should be published in a Kafka input topic, the one used by the processors as data source in the following stage. The responsibility of how and when these data are sent to the Kafka input topics relies on the sources. In order to avoid this coupling, a data collector layer might be implemented between the data sources and data processing layers. Its functionality, which at the moment is not required, would be to gather the information across these sources and send them periodically to the Kafka input topic.

5.3.2 Data Processing

In the data processing layer, we highlight three components: The Kafka Cluster, which is used to receive and store the heterogeneous data coming from the data sources; The Kafka Streaming App, which is used to process, homogenise and transmit the heterogeneous data; and the Esper

CEP engine, which is used to analyse and detect situations of interest with the homogenised data.

5.3.2.1. Kafka Cluster

The Kafka Cluster contains the Kafka topics where our input data are received. This Kafka Cluster may be executed on a different machine from the one we use for our Kafka Stream Application. It may also be replicated in several machines in order to achieve scalability. Depending on the machine and the circumstances, we can adopt one configuration or another.

In our solution, we use two Kafka topics: one topic for the input data to be transformed in the processors and analysed in the CEP Engine, and the other topic used as output topic, where all the situations of interest (complex events) detected by the CEP engine will be published.

In general, the input sources can be grouped to be sent to different partitions according to the scalability required. In Section 6, we test the input topic with one, two, four and eight partitions, with the aim of proving how the architecture may scale thanks to the use of additional partitions. Kafka Streams API requires having one processor for each partition [41]. In this case, we have two partitions for the input topic, so we have two processors. Thanks to this, we have a concurrent consumption from the partitions of the input topic and we can achieve a better performance of the solution [43]. The output topic only has one partition because interaction is not continuous, so there is for it to be partitioned. Bear in mind that we could scale the number of partitions for the input and output topics as necessary, according to the domain needs.

5.3.2.2. Kafka Stream Application

This is the main processing component and most significant contribution in our architecture. In addition to the processors, we make use of an Esper CEP engine instance.

As previously mentioned, when we use Kafka Streams API, a processor will be instantiated for each partition existing in the input topic we use as a source in our Kafka Stream Application. Thanks to this technique, we are able to provide scalability in our solution because each processor consumes messages from one partition to perform the homogenisation task. This task consists of the following actions, numbered from 1 to 5, as can be seen in the processors in Figure 2:

- ***(1) Consumption of the data from the Kafka input topics***. Each processor can consume and process information from one partition. For this reason, two partitions in the input topic would generate two processors, four partitions would generate four processors, and so on. The messages are consumed by the processors from their associated partitions and the homogenisation task begins.
- ***(2) Data format detection***. Once the data reach the processors, the system detects whether they are structured (JSON or XML) or unstructured (raw data).
- ***(3) Data Homogenisation***. If the data are in an unstructured format (raw data), the system transforms them into a generic JSON. If not, it continues with the next step in the homogenisation task.
- ***(4) Schema generation***. Once the data are in a homogenised JSON format, the system generates the schema for such data using Apache Avro.
- ***(5) Generation of Avro event***. Finally, with the schema created, the system generates the Avro event. This event is sent to the Esper CEP engine to be analysed.

Bear in mind that when processing events with Esper or with any other processing engines, we need to automatically identify what type of events in a particular domain the engine is receiving. This is one of the main difficulties in processing heterogeneous data in real time, since the Esper CEP engine identifies such an event type through the event type name. The problem is that until now the sensor manufacturer or programmer is forced to correctly identify such an attribute in the message sent. If not, the message cannot be taken into account in the CEP engine. Due to the lack of standards, there is no agreement on the way to do this and each manufacturer or sensor owner acts differently.

In order to solve this issue, we propose three alternatives, as detailed below. We will choose one or another depending on whether the information to be processed comes in a structured or unstructured format:

1. If a structured (JSON or XML) event format is received, the label name of the attribute, which contains the event type name value, is configured by default as "*eventTypeName*". Please note that such a label name can be changed in a configuration file as necessary. Therefore, if the message to be processed contains an attribute with the name specified in the configuration file, this attribute is the event type name to be used when creating the schema. For example, if we consume the structured JSON event that we have in Listing 1, the event type name would be "*WaterMeasurement*".
2. If an unstructured (CSV raw data) event format is received, the event type name attribute in the raw data we are processing can be identified by a previously specified position. For example, if we consume the unstructured event in Listing 3, where the first value contains the event type name ("*WaterMeasurement*"), in the configuration file, we would have established in advance that the name attribute is in position "*0*", that is, the first attribute in the received data.
3. If none of the previous scenarios is met, i.e. neither the label name of the attribute or the position for the event type name value are specified, a unique generic event type name is assigned to the schema created. For example, if we consume the unstructured event we have in Listing 3 and we do not specify the position of the event type name attribute, the event type name would be "*GenericEventX*", where "*X*" would be a unique number assigned by the architecture to identify this kind of event.

It is worth underlining that these three methods can be combined in a particular domain, so, using these restrictions, we are able to consume and process unstructured and structured information at the same time.

5.3.2.3. Esper CEP Engine

Finally, once the information is ready to be sent, an Esper CEP engine instance is executed to receive all the previously homogenised events as Avro Events. When a situation of interest is detected, the complex event generated by the CEP engine is published to the Kafka output topic. Note that since Esper has the capability to add new patterns and configurations in runtime, we have also provided the option of sending such new patterns to Kafka input topics. Then, the architecture can automatically detect that the new message is a pattern instead of an event, and deploy it immediately with no need to stop the system. In addition, we can also implement actions to carry out when a complex event is detected, i.e., sending an email, storing the event in a database, sending the information to a web service, etc.

As explained, the situations to be detected using Esper as the CEP engine must be defined using the EPL syntax. The EPL implementation for the three situations described —reading errors, water leaks and unusual consumption— are available at http://dx.doi.org/10.17632/yk8rkgwcz3.1. Note that more complex event patterns will be defined in further collaboration with GEN.

In Listing 4, we can see the implementation of the EPL pattern for the first situation. Firstly, we set the pattern name with the keyword *'@NAME'*. Secondly, we define the event type name (*'WaterMeterAnomaly'*) to be created with *'INSERT INTO'*. Thirdly, we define the list of attributes to be included in the complex event we are creating, using *'SELECT'* followed by the attributes. Fourthly, we specify the simple event type to be used as source, in this case *'WaterMeasurement'*, with the keyword *'FROM'*. Finally, *'WHERE'* is used to specify the conditions that the previously defined simple event has to satisfy in order to generate this complex event.

Listing 4 – EPL implementation for anomaly detection

```
@NAME('Anomaly-Reading-Errors')
INSERT INTO WaterMeterAnomaly
SELECT a1.serialNumber as serialNumber, a1.dateTime as dateTime,
a1.volumeM3 as volumeM3, a1.volumeL as volumeL, a1.type as type,
a1.starts as starts, a1.batteryLevel as batteryLevel, a1.sleepingTime
as sleepingTime, a1.leakingTime as leakingTime, a1.normalTime as
normalTime, a1.batteryLevelStr as batteryLevelStr,
current_timestamp().toString() as timestamp, 'Water meter with wrong
data' as anomaly
FROM WaterMeasurement a1
WHERE a1.starts < 0 or a1.sleepingTime < 0 or a1.leakingTime < 0 or
a1.normalTime < 0
```

The four patterns are deployed in the Esper CEP engine and the homogenised data are evaluated against them. Once a pattern is triggered by the incoming events, the complex event generated is published to the Kafka output topic and, at the same time, this complex event information is displayed in a web application that enables the company to be aware of the detected alerts.

5.3.3 Data Consumers

Once the complex event detected is published in the Kafka output topic, it is displayed in a web so the company may be aware of these situations in a friendly way.

For testing purposes, the only action currently supported upon complex event detection is sending it to the Kafka output topic. However, other endpoints, such as SQL/NoSQL databases, Smartphone Push notifications, etc., can easily be added in our architecture (this is outside the scope of this paper).

6. Results and evaluation

This section presents an experimental validation in order to show the effectiveness of our platform for real-time processing of heterogeneous data in the IoT, focusing on the domain of the real scenario described in Section 5. To this goal, two tests were conducted.

Firstly, we performed an execution with real-time data from the 111 smart water meters available in the water supply network, in order to test the effectiveness of the architecture in this domain. Secondly, we evaluated the performance of the proposed solution through a stress test using JMeter to simulate a huge number of input events. Both evaluations were carried out using computers with the following features: Windows 10 Enterprise, 64 bits, Intel® Core™ i7-4770 CPU, 12GB RAM, and 115GB HDD.

6.1 Real Scenario Test

In this evaluation, we used the smart water meters as input sources sending the information related to the measurements of their sensors to the architecture. This information is sent as structured JSON (see Listing 1).

For this test, the system processed information in real time for 21 days non-stop. Around 2 310 simple events were processed. We used the three situations of interest to detect the complex events discussed in Section 5.1. A total of 593 complex events were detected. These complex events were published in a Kafka output topic and the topic was used as input data in a web app we implemented so the company's workers could visualise them in a user-friendly way. We checked these complex events detected with GEN and agreed that the architecture had successfully processed their data and detected anomalies in real time. In addition, thanks to this test, the company was able to detect and correct anomalies in some of the readings of their water meters, of which they previously had no knowledge.

6.2 Performance Test

We simulated several rates of incoming events in order to detect the maximum incoming event rate that our platform is able to process deployed in a machine with the features previously described. To do this, we used Apache JMeter as the data producer for our platform since this open source Java application is designed for load testing and performance evaluation. Due to JMeter not having official functionality to publish messages in a Kafka topic, we had to use it in conjunction with Kafkameter, which is an unofficial plugin developed by the community for this purpose. Furthermore, we used Throughput Shaping Timer, which is another plugin that facilitated customization of the number of requests per second we wanted to simulate. Thus, we were able to produce N messages per second (m/s) and send them to a specific partition of a Kafka input topic.

For these tests, we simulated several scenarios with multiple producers sending messages to one or more partitions of the Kafka input topic. We configured these producers to send heterogeneous messages, like those in Listing 1 (JSON message), Listing 2 (XML message) and Listing 3 (raw data message). Please note that when processing raw data, the system automatically performs the necessary transformations, which are not required when data are already structured. It is worth bearing in mind that the system can also accept messages with the structures in Listings 1, 2 and 3 but with a different order and number of attributes according to the domain in question. Then, with the homogenised data, they are sent to the Esper engine in order to count how many simple events we can process in these tests.

We performed four tests: (1) using a single producer sending heterogeneous messages to a Kafka input topic with one partition; (2) using two partitions for the Kafka input topic; (3) using four partitions in the input topic and (4) using eight partitions in our Kafka input topic through the additional use of a second machine with the same features than the first one.

Therefore, we deployed the pattern specified in Listing 5 in the Esper CEP engine. This counts how many events are received and processed in a 10-minutes batch window. Then, we compared the result of this pattern with the number of heterogeneous messages sent by JMeter within 10 minutes, thus being able to check how many events are processed in total by the whole system (Kafka Streams with Avro for the homogenisation of the information and Esper for analysing the simple events) in each test.

Listing 5 – EPL implementation to count the amount of events processed

```
@Name('Counter')
SELECT count(*)
FROM WaterMeasurement.win:time_batch(10 min)
```

Figure 3 shows the results of these evaluations. In the x-axis, we represent the number of events produced per second (m/s), which starts with 20 000 m/s and increases to 200 000 m/s. In the y-axis, we represent the total number of messages produced during the test, which starts with 11 757 163 messages and ends in 113 394 993 messages. As seen, for the very first test, it was able to handle the workload without problems until 40 000 m/s. If we increase the m/s rate to 50 000 and up, the system suffers a delay when processing the events instantly, requiring more time to complete the processing. For the second test, we managed to process up to 80 000 m/s, and then, as happened before, the system required more time to process the remaining of messages. In the third test, the limit was set at 120 000 m/s without problems. Finally, in the fourth test, we were able to process up to 200 000 m/s.

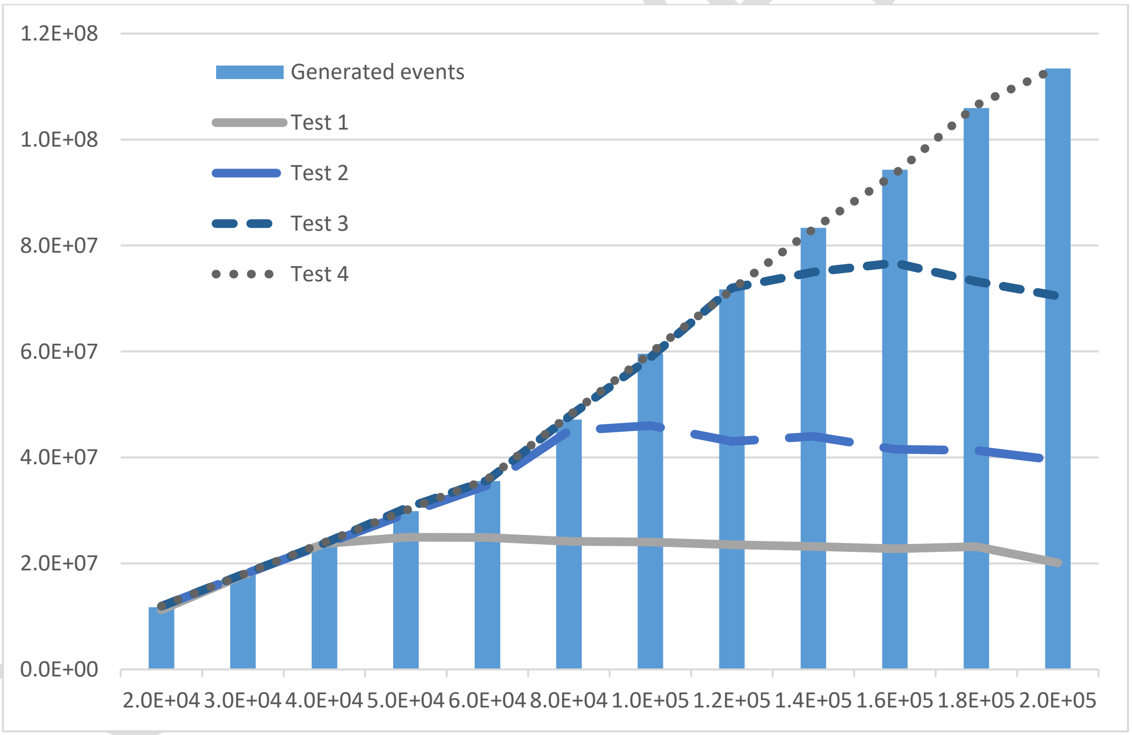

Fig. 3 – Results of the performance tests

In addition, we performed a large-scale simulation of 60 minutes, using 2 partitions receiving events at a rate of 32 000 m/s, in order to verify that our platform does not collapse when executing the simulation for longer execution periods. The simulation was successful; 114 655 932 simple events were processed without problems.

In order to measure the mean time that the architecture takes to process each message, we performed another evaluation using the same set-up as before. We measured the mean time taken by each task in nanoseconds (ns). The results of this evaluation are shown in Table 1. An approximate time of 24 500 ns (0.0245 milliseconds) is required to process each message.

Table 1 – Average time required by each task to process a message

Task name	Mean time (nanoseconds)
Data Homogenisation	7 476 ns
Schema generation	14 347 ns
Creation of the Avro event	2 726 ns

Therefore, we can conclude that our solution is able to satisfy the requirements to process large amounts of information, consuming, homogenising and analysing these data. It can perform long simulations with a high workload perfectly with no latency (such a number of nanoseconds is insignificant when processing thousands of messages per second). It is able to process up to 113 601 799 messages of an average size of 199 B, resulting in a total amount of 22 493 MB of heterogeneous information processed within 10 minutes, deployed in a couple of desktop computers. Moreover, the scalability of the solution was successfully proven, being able to process larger quantities of messages per second when we scale our solution using more computers.

6.3 Proposed architecture vs existing architectures

As we stated in Section 4.1, an architecture for processing heterogeneous data in the IoT should provide a set of key features. The evaluation of such features, motivated and described in Section 4.1, will provide with us a clear vision of the fit of the architecture to the scopes where it is required to handle, process and analyse large amounts of heterogeneous data in real time. Table 2 compares our proposal with the previously described proposals using such key features.

The features evaluated, as previously explained, are: (F1) Scalability, (F2) Stream Processing, (F3) Data Analytics, (F4) Data Storage, (F5) Multiple Event Types, (F6) Predictive Tools, (F7) Real Time, and (F8) Performance Evaluation.

Table 2 – Comparative of heterogeneous data processing proposals

	F1	F2	F3	F4	F5	F6	F7	F8
Our proposal	X	X	X	X	X	-	X	X
Carcillo et al. [15]	X	X	X	X	-	X	-	-
Stripelis et al. [4]	X	X	X	X	-	X	-	-
Zeydan et al. [5]	X	X	X	X	-	X	X	-
Amini et al. [16]	X	X	X	X	-	X	X	-
D'Silva et al. [17]	X	X	X	X	X	-	X	X
Jung et al. [18]	X	-	X	-	-	-	-	X
Montori et al. [6]	-	-	X	X	-	-	-	-
Oteafy [21]	X	-	-	-	-	-	-	-
Santos et al. [22]	-	-	X	X	-	X	X	-
Estévez-Ayres et al. [23]	X	X	X	X	-	-	X	X

With regard to scalability (F1), it can be achieved in most of these proposals thanks to the technologies involved in their architectures, while half of the proposals have SP (F2) or are able to perform it using Kafka, Spark or Storm. Others like [18,21,22] do not mention this paradigm at all. Concerning data analytics (F3) and data storage (F4), most of these proposals perform these key features that any architecture should implement. Dealing with multiple event types (F5) is a special feature that is only present in [17] and in our proposal. Referring to ML (F6), it is implemented in a few of these proposals and is not currently accomplished in our work but we have plans to introduce it (see Section 7, Future Work). On the subject of real time (F7), half of these proposals use technologies to process or analyse the data in batches, which cannot be considered real-time systems. On the contrary, our platform is able to consume, process, analyse and react to heterogeneous data in real time thanks to the combination of an SP platform, a DSS and a CEP engine.

Concerning the performance evaluation (F8), most of these proposals perform an evaluation of their work using a different approach to that we perform in Section 6.2. In [17] they perform a simulation sending only 1 000 simple events to the Kafka Cluster; we expect to have larger number of events in an IoT scenario. Hence, we provided tests with a much higher number of incoming events. Finally, in [18], the proposal performs tests with batch sensor data, proving that their proposed solution can handle around 10 000 m/s as maximum throughput, which is considerably less than the maximum throughput performed by our proposal.

Therefore, these previously analysed proposals fail to accomplish one or more of the required features that are key in any architecture for consuming, processing, transforming and detecting situations of interest from heterogeneous data sources in the IoT. We can affirm that most existing proposals for heterogeneous data processing do not benefit from using Kafka as a SP application, at most to create individual components to pre-process the information. In our proposal, we use Kafka as a communication module and Kafka Streams to perform the transformations, both working as one component. The system thus provides additional advantages. On the one hand, it has the already well-known advantages provided by Kafka itself, such as fault-tolerant storage, communications management and durability, while on the other, it presents the benefits provided by using Kafka Streams API when processing streams of data in real time, such as high scalability, exactly-once processing or high throughput rate. This results in a complete Kafka Stream Application able to consume, transform and process huge amounts of information in real time. While Spark and Storm are commonly used for the analysis, our solution uses Esper, providing us with the advantage of not having to stop the system execution before adding new required analysis of events. In addition, some of these proposals integrate modules for ML and Predictive into their architecture, which we have plans to do as future work, in order to improve and generate better results. Finally, the other proposals mainly focus on processing stored information, while ours can perform real-time processing in streaming. In light of all the above, and as proven in Section 6.2, our proposal's performance stands out from the others.

6.4 Answer to Research Questions

In Section 1, we presented five RQs. The answers to these questions are as follows:

- (RQ1) How can heterogeneous data be structured in an IoT domain?
    - We have determined that the heterogeneous data in an IoT domain may be presented in a structured format like JSON, XML or YAML, or it can be unstructured, also known as raw data, which is information coming as CSV rows, a string with several values separated by a special character. As examples, in Listings 1 and 2, we showed two Water Measurement structured events from a smart water meter. In addition, in Listing 3, the same event is represented using an unstructured format, as raw data. All these three listings represent the same data but in several heterogeneous structures.

- (RQ2) What processes have to be conducted to homogenise heterogeneous data coming from IoT sensors?
    - In order to transform the heterogeneous data into simple events to be used in the CEP engine, a process with 5 steps is carried out:
        - Data Consumption. The heterogeneous data are received or consumed from the data sources.
        - Data Format Detection. The data are processed in order to infer the format in which they are represented (structured or unstructured).
        - Data Homogenisation. The data is transformed into a common format.
        - Schema Generation. The schema of the data is retrieved.
        - Simple Event Generation. Once the data are homogenised, using the schema, the simple events are generated.
- (RQ3) Is the proposed architecture (a combination of Stream Processing, Data Serialization Systems and Complex Event Processing) able to process, homogenise and analyse heterogeneous information?
    - Yes, it is. The proposed architecture is able to process, homogenise and analyse heterogeneous data from several sources. The steps previously mentioned are performed in order to make the data ready to work with. Once the data is ready, a real-time analysis is performed, using a CEP engine which detects situations of interest from these data. In Section 6, we present several tests performed in order to show the efficiency and the effectiveness of our solution.
- (RQ4) Is the proposed architecture suitable for IoT domains?
    - Yes, it is. As shown in Section 5, the proposed architecture was applied to the Smart Water Management Domain. Furthermore, an evaluation performed in Section 6.1 testing the proposed solution with heterogeneous data from smart water meters shows that our solution is suitable for this kind of IoT scenario. As future work, we have plans to test the solution in other IoT domains.
- (RQ5) Is the proposed architecture more effective and efficient than other existing architectures when required to handle, process and analyse large amounts of heterogeneous data in real time?
    - Yes, it is. As described, in Section 6.3, our architecture stands out from others by providing the capability to process huge amounts of information. We are able to process, analyse and homogenise almost 23 Gigabytes of heterogeneous data in real time within 10 minutes, using two desktop computers. Moreover, we successfully tested the proposed solution in a large simulation to make sure it did not collapse.

7. Conclusions and Future Work

In this paper, we have presented a generic software architecture that can be applied to all kinds of scenarios in the IoT. Its main novel advantage is being able to automatically process and analyse heterogeneous data, regardless of their structure. This architecture has benefitted from the integration and use of (1) Kafka Streams API as SP to process streams of heterogeneous data sources as they occur; (2) Apache Avro as DSS in order to perform the homogenisation task effectively and efficiently; and (3) Esper as CEP engine to analyse the homogenised information and to infer situations of interest from these data in real time. Additionally, the performance of the architecture was tested in a real scenario and through stress tests, with promising results.

As future work, we are planning to improve the proposed architecture in several ways. Firstly, we will extend the number of endpoints available to perform automatic actions when complex events are detected. Secondly, we will keep the primary focus on the smart water management scenario, performing new analyses with more sensors and we will also test the architecture in additional real scenarios with high loads of processing. Thirdly, we are researching the integration of a module to perform ML analytics and predictions that will feedback to the CEP engine in order to improve the quality of the data analytics.

Finally, we will integrate the proposed architecture with MEdit4CEP [33], a graphical modelling tool that facilitates the definition of event patterns and action for real-time notification. Thus, we will be able to graphically integrate these heterogeneous data and define EPL statements and actions graphically, automatically generating and deploying the corresponding EPL code in the Esper CEP engine.

Acknowledgements

This work was supported in part by the Spanish Ministry of Science and Innovation and the European Union FEDER Funds (No. TIN2015-65845-C3-3-R and No. RTI2018-093608-B-C33) and in part by the pre-doctoral program of the University of Cádiz (2017-020/PU/EPIF-FPI-CT/CP). In addition, we would like to thank GEN Grupo Energético for sharing their data for testing purposes.

Vitae

David Corral-Plaza received the MSc degree in Computer Science from the University of Seville, Spain, in 2017. He is currently working on his PhD at University of Cadiz. His research focuses on the integration of stream processing and complex event processing, event-driven service-oriented architectures, and context awareness in the IoT.

Inmaculada Medina-Bulo received her PhD in Computer Science from the University of Seville (Spain). She has worked at the Department of Computer Science and Engineering of the University of Cádiz since 1995. She is the main researcher of the UCASE Software Engineering Research Group. Her current research interests focus on the integration of CEP in SOA 2.0, IoT, Software Testing and Search Based Software Engineering.

Guadalupe Ortiz completed her PhD in Computer Science at the University of Extremadura (Spain) in 2007, where she worked from 2001 as Assistant Professor. In 2009, she joined the University of Cádiz as Professor in Computer Science and Engineering. Her current research interests focus on the integration of CEP and context-awareness in SOA 2.0.

Juan Boubeta-Puig received the Ph.D. degree in Computer Science from the University of Cádiz, Spain, in 2014. Since 2009, he has been an Assistant Professor with the Department of Computer Science and Engineering, UCA. His research focuses on the integration of CEP in SOA 2.0, IoT, and model-driven development of advanced user interfaces.

References

[1] A. Nordrum, Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated, IEEE Spectrum: Technology, Engineering, and Science News. (2016). https://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated (accessed June 6, 2019).

[2] S. Nadal, V. Herrero, O. Romero, A. Abelló, X. Franch, S. Vansummeren, D. Valerio, A software reference architecture for semantic-aware Big Data systems, Information and Software Technology. 90 (2017) 75–92. doi:10.1016/j.infsof.2017.06.001.

[3] IoT Standards & Protocols Guide | 2018 Comparisons on Network, Wireless Comms, Security, Industrial, Postscapes. (2018). https://www.postscapes.com/internet-of-things-protocols/ (accessed June 6, 2019).

[4] D. Stripelis, J.L. Ambite, Y.Y. Chiang, S.P. Eckel, R. Habre, A Scalable Data Integration and Analysis Architecture for Sensor Data of Pediatric Asthma, in: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 2017: pp. 1407–1408. doi:10.1109/ICDE.2017.198.

[5] E. Zeydan, U. Yabas, S. Sözüer, Ç.Ö. Etemoğlu, Streaming alarm data analytics for mobile service providers, in: NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium, 2016: pp. 1021–1022. doi:10.1109/NOMS.2016.7502953.

[6] F. Montori, L. Bedogni, L. Bononi, A Collaborative Internet of Things Architecture for Smart Cities and Environmental Monitoring, IEEE Internet of Things Journal. 5 (2018) 592–605. doi:10.1109/JIOT.2017.2720855.

[7] D.C.Y. Vargas, C.E.P. Salvador, Smart IoT Gateway For Heterogeneous Devices Interoperability, IEEE Latin America Transactions. 14 (2016) 3900–3906. doi:10.1109/TLA.2016.7786378.

[8] S. Sicari, A. Rizzardi, D. Miorandi, A. Coen-Porisini, REATO: REActing TO Denial of Service attacks in the Internet of Things, Computer Networks. 137 (2018) 37–48. doi:10.1016/j.comnet.2018.03.020.

[9] I. Yaqoob, I.A.T. Hashem, A. Ahmed, S.M.A. Kazmi, C.S. Hong, Internet of things forensics: Recent advances, taxonomy, requirements, and open challenges, Future Generation Computer Systems. 92 (2019) 265–275. doi:10.1016/j.future.2018.09.058.

[10] E. Ahmed, I. Yaqoob, I.A.T. Hashem, I. Khan, A.I.A. Ahmed, M. Imran, A.V. Vasilakos, The role of big data analytics in Internet of Things, Computer Networks. 129 (2017) 459–471. doi:10.1016/j.comnet.2017.06.013.

[11] K. Mohammed Shahanas, P. Bagavathi Sivakumar, Framework for a Smart Water Management System in the Context of Smart City Initiatives in India, Procedia Computer Science. 92 (2016) 142–147. doi:10.1016/j.procs.2016.07.337.

[12] S. Narendran, P. Pradeep, M.V. Ramesh, An Internet of Things (IoT) based sustainable water management, in: 2017 IEEE Global Humanitarian Technology Conference (GHTC), 2017: pp. 1–6. doi:10.1109/GHTC.2017.8239320.

[13] S. Salvi, S.A.F. Jain, H.A. Sanjay, T.K. Harshita, M. Farhana, N. Jain, M.V. Suhas, Cloud based data analysis and monitoring of smart multi-level irrigation system using IoT, in: 2017

International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2017: pp. 752–757. doi:10.1109/I-SMAC.2017.8058279.
[14] Final Report Summary - SMARTWATER4EUROPE (Demonstration of integrated smart water supply solutions at 4 sites across Europe) | Report Summary | FP7-ENVIRONMENT, CORDIS | European Commission. (2018). https://cordis.europa.eu/project/rcn/111476/reporting/en (accessed June 6, 2019).
[15] F. Carcillo, A. Dal Pozzolo, Y.-A. Le Borgne, O. Caelen, Y. Mazzer, G. Bontempi, SCARFF: A scalable framework for streaming credit card fraud detection with spark, Information Fusion. 41 (2018) 182–194. doi:10.1016/j.inffus.2017.09.005.
[16] S. Amini, I. Gerostathopoulos, C. Prehofer, Big data analytics architecture for real-time traffic control, in: 2017 5th IEEE International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS), 2017: pp. 710–715. doi:10.1109/MTITS.2017.8005605.
[17] G.M. D'silva, A. Khan, Gaurav, S. Bari, Real-time processing of IoT events with historic data using Apache Kafka and Apache Spark with dashing framework, in: 2017 2nd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT), 2017: pp. 1804–1809. doi:10.1109/RTEICT.2017.8256910.
[18] H.S. Jung, C.S. Yoon, Y.W. Lee, J.W. Park, C.H. Yun, Cloud computing platform based real-time processing for stream reasoning, in: 2017 Sixth International Conference on Future Generation Communication Technologies (FGCT), 2017: pp. 1–5. doi:10.1109/FGCT.2017.8103400.
[19] L. Hu, R. Sun, F. Wang, X. Fei, K. Zhao, A Stream Processing System for Multisource Heterogeneous Sensor Data, Journal of Sensors. (2016). doi:10.1155/2016/4287834.
[20] F. Montori, L. Bedogni, L. Bononi, On the integration of heterogeneous data sources for the collaborative Internet of Things, in: 2016 IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a Better Tomorrow (RTSI), 2016: pp. 1–6. doi:10.1109/RTSI.2016.7740616.
[21] S.M.A. Oteafy, A Framework for Heterogeneous Sensing in Big Sensed Data, in: 2016 IEEE Global Communications Conference (GLOBECOM), 2016: pp. 1–6. doi:10.1109/GLOCOM.2016.7841499.
[22] P.M. Santos, J.G.P. Rodrigues, S.B. Cruz, T. Lourenço, P.M. d'Orey, Y. Luis, C. Rocha, S. Sousa, S. Crisóstomo, C. Queirós, S. Sargento, A. Aguiar, J. Barros, PortoLivingLab: An IoT-Based Sensing Platform for Smart Cities, IEEE Internet of Things Journal. 5 (2018) 523–532. doi:10.1109/JIOT.2018.2791522.
[23] I. Estévez-Ayres, J. Arias Fisteus, C. Delgado-Kloos, Lostrego: A distributed stream-based infrastructure for the real-time gathering and analysis of heterogeneous educational data, Journal of Network and Computer Applications. 100 (2017) 56–68. doi:10.1016/j.jnca.2017.10.014.
[24] What is Stream Processing?, Data Artisans. (2018). https://data-artisans.com/what-is-stream-processing (accessed June 6, 2019).
[25] V. Gurusamy, School of IT, Madurai Kamaraj University, Madurai, India, S. Kannan, School of IT, Madurai Kamaraj University, Madurai, India, K. Nandhini, Technical Support Engineer, Concentrix India Pvt Ltd, Chennai, India, The Real Time Big Data Processing Framework Advantages and Limitations, International Journal of Computer Sciences and Engineering. 5 (2017) 305–312. doi:10.26438/ijcse/v5i12.305312.
[26] Apache Kafka, Apache Kafka. (2018). https://kafka.apache.org/ (accessed June 6, 2019).
[27] Apache Spark^{TM} - Unified Analytics Engine for Big Data, (2018). https://spark.apache.org/ (accessed June 6, 2019).
[28] Apache Flink: Stateful Computations over Data Streams, (2018). https://flink.apache.org/ (accessed June 6, 2019).

[29] Spark Streams or Kafka streaming, deep dive in a hard choice, Meritis. (2019). https://meritis.fr/bigdata/spark-streaming-or-kafka-streaming-deep-dive-in-a-hard-choice/ (accessed June 6, 2019).
[30] Welcome to Apache Avro!, (2018). https://avro.apache.org/ (accessed June 6, 2019).
[31] Apache Thrift - Home, (2018). https://thrift.apache.org/ (accessed June 6, 2019).
[32] Protocol Buffers, Google Developers. (2018). https://developers.google.com/protocol-buffers/ (accessed June 6, 2019).
[33] J. Boubeta-Puig, G. Ortiz, I. Medina-Bulo, MEdit4CEP: A model-driven solution for real-time decision making in SOA 2.0, Knowledge-Based Systems. 89 (2015) 97–112. doi:10.1016/j.knosys.2015.06.021.
[34] EsperTech, Esper - Complex Event Processing. http://www.espertech.com/esper/ (accessed June 6, 2019).
[35] Apache Storm. http://storm.apache.org/index.html (accessed June 6, 2019).
[36] J. Kreps, Why Avro For Kafka Data?, Confluent. (2015). https://www.confluent.io/blog/avro-kafka-data/ (accessed June 6, 2019).
[37] A. Ostfeld, Water Distribution Networks, in: Intelligent Monitoring, Control, and Security of Critical Infrastructure Systems, Springer, Berlin, Heidelberg, 2015: pp. 101–124. doi:10.1007/978-3-662-44160-2_4.
[38] Toronto Water says $2,500 bill caused by leaking toilet, Scarborough man disagrees, (2018). https://toronto.citynews.ca/2018/05/14/water-bill-leaky-toilet-scarborough/ (accessed June 6, 2019).
[39] Grupo Energético de Puerto Real S.A., GEN Grupo Energético, (2018). http://www.grupoenergetico.es/gen/index.php/servicios/aguas (accessed June 6, 2019).
[40] Z. Wang, W. Dai, F. Wang, H. Deng, S. Wei, X. Zhang, B. Liang, Kafka and Its Using in High-throughput and Reliable Message Distribution, in: 2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS), 2015: pp. 117–120. doi:10.1109/ICINIS.2015.53.
[41] Apache Kafka Streams, Apache Kafka. (2018). https://kafka.apache.org/documentation/streams/ (accessed June 6, 2019).
[42] N. Palmer, E. Miron, R. Kemp, T. Kielmann, H. Bal, Towards Collaborative Editing of Structured Data on Mobile Devices, in: 2011 IEEE 12th International Conference on Mobile Data Management, 2011: pp. 194–199. doi:10.1109/MDM.2011.48.
[43] Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines), (2014). https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines (accessed June 6, 2019).