



Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks

Sabrine Khriji¹ · Yahia Benbelgacem¹ · Rym Chéour² · Dhouha El Houssaini¹ · Olfa Kanoun¹

Accepted: 17 June 2021 / Published online: 26 July 2021
© The Author(s) 2021

Abstract

The growth of the Internet of Things (IoTs) and the number of connected devices is driven by emerging applications and business models. One common aim is to provide systems able to synchronize these devices, handle the big amount of daily generated data and meet business demands. This paper proposes a cost-effective cloud-based architecture using an event-driven backbone to process many applications' data in real-time, called REDA. It supports the Amazon Web Service (AWS) IoT core, and it opens the door as a free software-based implementation. Measured data from several wireless sensor nodes are transmitted to the cloud running application through the lightweight publisher/subscriber messaging transport protocol, MQTT. The real-time stream processing platform, Apache Kafka, is used as a message broker to receive data from the producer and forward it to the correspondent consumer. Micro-services design patterns, as an event consumer, are implemented with Java spring and managed with Apache Maven to avoid the monolithic applications' problem. The Apache Kafka cluster co-located with Zookeeper is deployed over three availability zones and optimized for high throughput and low latency. To guarantee no message loss and to simulate the system performances, different load tests are carried out. The proposed architecture is reliable in stress cases and can handle records goes to 8000 messages in a second with low latency in a cheap hosted and configured architecture.

Keywords Cloud computing · Event-driven · Micro-services · Kafka · Wireless sensor network · Internet of things · Amazon web service IoT

✉ Sabrine Khriji
sabrinekhriji@ieee.org

Extended author information available on the last page of the article

1 Introduction

The growing popularity of internet and communication technology leads toward a society, where everything is connected. The Internet of Thing (IoT) paradigm has led to a digital revolution in electronics and computing fields, with a low-cost data storage, mobile computing [1], artificial intelligence [2], Software as a Service (SaaS) [3] and cloud computing [4]. The number of IoT devices is currently in its peak of inflated expectations, predicted to increase by 21% between 2016 and 2022 to around 18 billion [5]. IoT devices with cellular connections are projected to reach 1.5 billion in 2022 or around 70% of a wide-area IoT category. In fact, the IoT is supported by an alliance of key assets, including the massive proliferation of smart devices, the confluence of low-cost technologies such as sensors, large amount of data (Big Data), High Performance Computing capabilities (HPC) and Wireless Sensor Networks (WSNs) [6–8]. In the context of WSNs, the cloud architecture is designed to monitor, control and record many wireless sensor nodes [9, 10]. For this reason, it needs to host massive amount of continuously generated data and satisfies numerous competing analytical requests.

Cloud computing is designated essentially to adjust rapidly resources to meet business demand. Indeed, it allows enterprises to easily deploy their applications and run it faster, with improved mechanisms designed to facilitate the maintenance and manageability. Cloud providers use a “pay-as-you-go” model, which leads to unexpected costs if users are not familiarized with pricing models. Designing an enterprise architecture is nowadays a big challenge because system architects require to extract the best formula according firstly to the cost model against the core business, and the choice between the implementation of the own services or working with software as a service for additional costs [3]. The cost saving is a major challenge facing many cloud applications. Nowadays, different solutions are developed to design cloud architectures, which can solve complex problems such as the standardization, resource metering, HPC application, Quality of Service (QoS), interoperability and security [11]. One of these solutions is the Amazon Web Service (AWS) IoT by providing a deep functionality to span the edge to the cloud. Amazon’s servers allow users to rent-out services in order to scale down hardware to avoid over spending. This on-demand infrastructure, for scaling applications or tasks, increases massively the hardware power available to the application requirement [12].

Our motivation for creating a novel cloud-computing architecture is the requirement of high data availability with low cost. The delay introduced among the extensive processing and storage data transferred to the cloud provides irrelevant and redundant information, leading to network bandwidth saturation and compromising the entire application [13]. Reducing the data flow at the local level and only transmitting relevant data to the cloud platform need to be considered. As the information is duplicated, it is possible to support the loss of several servers or even entire data centers, depending on the configuration. Big data approaches relying on traditional data warehouses often pose latency problems, making them unsuitable for new data use cases [14]. It is important to consider the entire flow of incoming events instead of only keeping the results of these

events, at the risk of having data that are not immediately up to date on all the hosting machines. Given the astronomical amount of data potentially exploitable, approaches must first be put in place to manage this wealth of information, to store and prioritize this data in an adhesive way, with a view to their subsequent exploitation. Above all, they must be aggregated in a very short period. A response time is made mandatory with regard to the computer values of the data. Real-time data processing (streaming) is able to consider time constraints and computing, to deliver accurate results with very low latency.

Underpinned by different design patterns such as publish/subscribe systems, event-driven and event-stream processing, the flexibility of integrating new functionalities is the primary feature to be considered during the design of the cloud architecture. Thereby, a Real-time processing Event-Driven cloud-based Architecture (REDA) is designed based on message-driven reactive systems with a high value of resilience and scalability. The implementation of this concept involves several key steps including the choice of the processing mechanism, the way of development and deployment of micro-services and the appropriate monitoring tools. To sum up, the distinctive characteristics of our design are as follows: High-availability, elasticity, a distributed infrastructure, with a reduced latency and high throughput values. In the next section, IoT cloud solutions are investigated. The main contributions of the paper are:

- The proposed architecture (REDA) is designed mainly from the Amazon Web Service IoT core architecture. It opens the door as a free software-based implementation hosted independently of any cloud provider or in a local cluster. It aims to process, store and visualize devices or generated data in real time. The developed architecture is cost-effective since it uses open source technologies. Moreover, it guarantees a high reliability by using the load test ensuring no loss of data. Furthermore, users can add new applications, and many developers can work on different applications in parallel.
- REDA provides important additional features since it is based on the query results history, which are cached regardless of the message size. Thereby, the user can access the results of previous queries without relaunching them. As consequence, both time and computing power are saved.
- An investigation of optimization methods is carried out including topics with more partitions, batching and worker threads for the producer to provide high availability, high reliability, high throughput and low latency.
- REDA is evaluated in terms of availability, reliability, high throughput and low latency. They are carefully validated and demonstrated during the implementation and after the deployment. The cloud-connected WSN is implemented in order to monitor the soil moisture.

The paper is organized as follows. Section 2 presents relevant previous works related to high-level description of IoT cloud solutions. The design of the proposed cloud architecture is provided in Sect. 3. Section 4 gives insights about the implementation. Section 5 deeps into the back-end cloud architecture and components deployment. Section 7 concludes the paper with discussion and directions for future works.

2 High-level IoT cloud solutions

The cloud computing paradigm is emerging as a reliable service delivering autonomous and smart behavior from virtualized storage technology-based data centres [15]. The cloud architecture enables to ensure the data security as well as inter-application flows. Wherever the data are located, the cloud architecture provides end-users with visibility on the location, or status of data transfer. Despite this pervasive development, critical challenges affect the reliability of this ongoing concept. Authors in [16] present an user-centric cloud-based model of an end-to-end interaction between various stakeholders in the cloud centric IoT framework. The cloud platform is based on the Manjrasoft Aneka and Microsoft Azure platforms to highlight the integration of storage, computing and visualization paradigms across both public and private clouds. In fact, Aneka Worker Containers are deployed as instances of Azure Worker Role. The developed model aims to reduce the time and cost involved in engineering IoT applications.

A cloud-based architecture of IT platforms for ubiquitous environments is presented in [17]. This architecture includes three layers, namely, cloud service, Machine to Machine (M2M) service and ubiquitous service. The interoperability between objects is ensured via semantic description models of a cross-layer, where the application management layer is correlated with the lower layers to gather information related to the specification context. Although this framework is a high-performance in vehicular monitoring, the reliability metric is not considered.

In [18], an integrated message-oriented architecture based on the Message Queuing Telemetry Transport (MQTT) protocol for smart home is proposed. Owing to the limited network bandwidth, restricted access to networks and devices, storage, computing, unreliability and energy resources, message-based models and MQTT are considered for intelligent objects for the IoT scenario. The presented architecture is centered on the publishing/subscribing messages. Some smart object models are developed on Raspberry Pi, and the associated applications are implemented in Python. Only this work remains applicable for limited network bandwidth, storage, computing and energy resources.

A cloud system architecture to monitor processes and analyse network performance data for industrial wireless sensor network (IWSN) is introduced in [19]. A central office is in charge of all interactions between the IWSN coordinating node and the outside world. This solution provides reliable and resilient systems for the data acquisition, processing and storage.

Many cloud providers such as Google and Amazon offer different IoT solutions. AWS IoT and Google Cloud IoT include several managed and integrated services enabling secure link, flexible manageability as well ensuring real-time data analysis and visualization. They offer a secure way to connect devices, store and analyse data such as sensors and microcontrollers to the cloud via tablets or phones [20]. The adoption and dissemination of cloud computing architectures are constrained by emerging concepts and the heterogeneity of existing solutions. The difficulty of such adoption is manifested by the absence of standards, the

heterogeneity of architectures and Application Programming Interface (APIs). Several cloud services offer free cloud storage for consumers, while others offer a kind of subscription-based tax.

Notwithstanding the relevant overview of cloud-based architectures for WSNs provided by the aforementioned works, showed several limitations that inspire us to propose the back-end cloud architecture. Given the astronomical amount of data potentially exploitable, approaches must first be put in place to manage this wealth of information, to store and prioritize this data in an adhesive way, with a view to their subsequent exploitation. Above all, they must be aggregated in a very short period. Real-time data processing (streaming) is able to consider time constraints and computing, to deliver accurate results with very low latency.

Most of the related works focused on the design and implementation of domain-specific monitoring systems based on WSNs, rather than on data-intensive services or on real-time data processing. Big data approaches relying on traditional data warehouses often pose latency problems, making them unsuitable for new data use cases like in [21]. It is important to consider the entire flow of incoming events instead of only keeping the results of these events at the risk of having data that are not immediately up to date on all the hosting machines. Works on [13] and [19] suffer from the bandwidth saturation and needs to reduce or to process the data flow. Thus, some cloud-based architectures are simulation-based such as [17] and [18]. Finally, a response time is made mandatory with regard to the computer values of the data which is a limitation of the work performed in [24].

Considering the limitations of existing solutions and according to the comparative analysis performed in Table. 1, a new cloud-based architecture is designed. To sum up, the distinctive characteristics of the proposed design are as follows: high-availability, elasticity, a distributed infrastructure, with a reduced latency and high throughput values as described in the next section.

3 REDA: Proposed back-end cloud-based architecture for real-time processing

The main purpose of this paper is to create a cloud-based architecture with high availability, high throughput and low latency using open source frameworks. To this end, the internal AWS architecture is used, which offers a secure way to connect things such as sensors and microcontrollers to the cloud. Moreover, this service enables users to store and analyse their data and remotely control devices via tablets or phones. The AWS IoT internal structure includes, mainly, the message broker, which presents the interface between things and AWS IoT. The message broker service is responsible for the synchronization of IoT devices and the transmission/reception of messages. Devices report their state by publishing packets to a specific topic, and the message broker forwards received messages to existed subscribed client(s) or AWS services. The AWS message broker is similar to the MQTT message broker, but it supports in addition HTTPS or MQTT over the WebSocket protocols to publish messages. When the message arrives to the AWS IoT, the user can apply same

Table 1 Comparative study of existing cloud-based architecture systems

	Features	Communication Protocol	Cloud	Limitations
[13]	Layer of computing based on Fog Computing	ZigBee	Microsoft Azure	Bandwidth saturation Reducing data flow Needs for data processing
[17]	3-layers architecture: Cloud, M2M and ubiquitous services	OIDM2M	LTE modem	The reliability metric
[18]	Integrated message-oriented architecture based on MQTT	MQTT	Not mentioned	Simualtion-based Limited network bandwidth, storage computing capabilities and energy resources
[19]	Implemented RESTful services at a coordinator node level	IPv6	RESTful	Simulation-based
[21]	OEC framework for building, managing, and slicing the resources pool	Not Mentioned	AWS	Latency problem Abscence of query results history No synchronization schemes
[24]	Event-driven ETL pipeline	SQS	AWS	SQS data throughput bottleneck Execution time

high level tasks such as filtering messages, inserting or updating data in the database or same low level tasks such as data processing or data mining [22].

In this level, the system requirements need to be defined. Since components such as message broker and database are very important in terms of system reliability and availability, those units can cause the entire system crash, in case of fail. To solve such problems, it is important to upload a replica of components, which are synchronized together to form one cluster. This cluster of nodes must be accessible from devices as one node to avoid system connection complexity. In the other side, the number of messages generated from devices is continually increased. The conception of architecture that offers high throughput, and low latency is highly required. The size of the cluster of nodes for these requirements depends on the application and processing goals. Other problems such as disk read/write throughput, network bandwidth and nodes resources configuration must be investigated during the cluster deployment [23].

3.1 Event-driven architecture

An event-driven architecture is commonly used in advanced micro-services-based applications by relying on events to communicate between decoupled services. It is push-oriented, where all events are executed on demand [25, 26]. As a result, there is no need to pay for continuous polling to verify an event. So, the network

bandwidth consumption, the CPU usage and Secure Sockets Layer (SSL)/Transport Layer Security (TLS) handshakes are decreased significantly. By dividing the services into micro-services and decoupling them, they still interoperable, but when one service fails, the others are continued operating.

Generally, the internal event-driven architecture includes the event producer (publisher), event manager and event consumer (subscriber) [27]. The publisher is the creator of events. The subscriber is the entity related to generate events, which can be either affected by the events or integrated to process events. The event manager, called also event channel, is a middle-ware application, which receives events from the producer and forwards it to correspondent subscribers. Moreover, it is used to forward messages from event sources to event consumers. This task presents a challenge in a real-time environment by avoiding the direct communication between the publisher and the consumer and adding additional internal sub-tasks.

A detailed description of each component of the proposed event-driven backbone for the back-end cloud architecture, seen in Fig. 1, is presented in the following subsections.

3.1.1 Event producer-based MQTT message broker

To send data to the remote subscriber (receiver), the choice of the appropriate messaging protocol among considerable protocols such as MQTT, Constrained Application Protocol (CoAP), Advanced Message Queuing Protocol (AMQP) and HTTP [28] is highly recommended. The MQTT protocol is distinguished by short message transmission capability and low bandwidth usage, which makes it suitable for M2M communications of the connected object type [29]. In addition, it features more different types of messages than other communication protocols. MQTT's features make it a good option for transmitting high volumes of sensor messages to cloud systems [37].

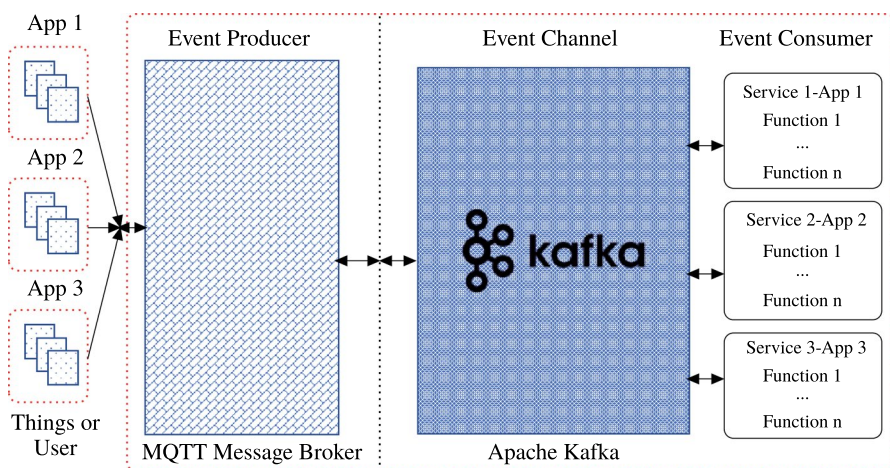


Fig. 1 Proposed event-driven backbone for the back-end cloud architecture

The MQTT network includes publisher nodes, topics, subscriber nodes and a broker. It allows devices to concretely send information on a given subject to a server that works as a message broker. The broker pushes this information to previously subscribed customers. With MQTT, all customer devices that need to communicate with each other have to inter-operate with the same broker. The latter stores the messages received from the sending entities (publishers) and relays them to one or more receivers (subscribers). Messages are sent via a given information channel (or topic). As a result, when the broker receives a published message, it broadcasts it to all subscribers but only those who have subscribed to the given information channel can receive it.

3.1.2 Event channel-based Apache Kafka

The rapid evolution of Internet traffic (expansion, complexity, speed, etc.) has multiplied the problems concerning data traffic, and in particular the processing of data flows in real time, such as messages, logs or videos. Many publish/subscribe systems, known as messages brokers, are introduced on the market, which are fast, reliable and scalable [30]. Apache Kafka and RabbitMQ are two popular open-source pub/sub systems widely adopted in companies. The results of benchmarking Apache Kafka and RabbitMQ (AMQP) in [31, 32] show that Apache Kafka has higher throughput than RabbitMQ. Indeed, it outperforms RabbitMQ producer by sending messages a rate six times faster.

Apache Kafka, as a distributed streaming platform, is designed to play the role of a central data pipeline. It is a topic-based message broker, which is presented as a stream of records, and it is divided into several partitions and distributed on different servers as shown in Fig. 2. A partition gives the possibility to transfer messages in parallel. The first step of message replication (noted as [1] in Fig. 2) is that the producer has to flush-out all the bytes to the network. The broker leader picks-out the message entry timestamp when the producer request reaches the message broker to calculate the message committing time (noted as [2] and [3] in Fig. 2). After that,

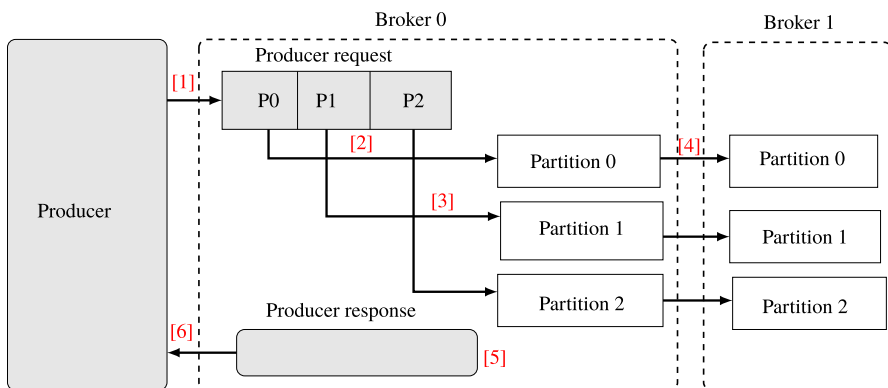


Fig. 2 Apache Kafka internal structure

the message broker pulls the producer request and appends it to the local log (noted as [4] in Fig. 2). Depending on the acknowledgement type, the message broker replicates messages flow on one or all In-Sync broker replicas (noted as [5] in Fig. 2). After the message replication, the topic leader puts the producer response into the response queue and flushes out the response bytes to the producer (noted as [6] in Fig. 2). Apache Kafka offers three types of acknowledgement:

- Acks = 0: The producer does not lose time waiting of acknowledgement from the broker.
- Acks = 1: The topic leader commits firstly the record and answers the client.
- Acks = -1: The topic leader needs to wait all In-Sync replicas to acknowledge one request.

3.1.3 Event consumer-based micro-services

The evolution of the IoT and M2M requires a new structuring of the application modules. Based on this observation, analysts recommend switching to micro-services to break down the system into partitioned subsystems, but communicating according to well-defined protocols [33]. Micro-services are a modularization concept, which serve to divide a larger software system in loosely decoupled services and influence the organization and software development processes. Micro-services can be deployed independently of each other, and a change on one service can be brought into the production regardless of changes to other micro-services. The implementation of micro-services can be managed with agile systems. This offers also the opportunity of continuous delivery/deployment of complex application possible functionality. Thus, the reactive capabilities integrated into the services can avoid the inevitable spread of failures, and performance problems and the need to use numerous tools in application protocols such as HTTP and Representational State Transfer (REST).

As seen in Fig. 3, each module has to deal with only one task that is part of the overall workflow. According to the dedicated task, each micro-service is written in a specific language. Thus, a large application can include micro-services written in Node.js, Ruby on Rails, Python and Java.

3.2 Messages serialization/deserialization with Protobuf

A challenge in distributed systems is the choice of communication techniques, and data serialization/deserialization libraries between the client and the server. Protobuf, as a solution provided by google, is a popular serialization/ deserialization solution [34]. It offers flexible, efficient and high performance way to solve problem related to client server communication such as multiprogramming languages, message expansion and processing time.

Many researches aim to compare such technologies to facilitate the choice and to get the best performance. The most appropriate way to do this, is to benchmark the work of those technologies in a specific context. The benchmark in [34, 35] presents

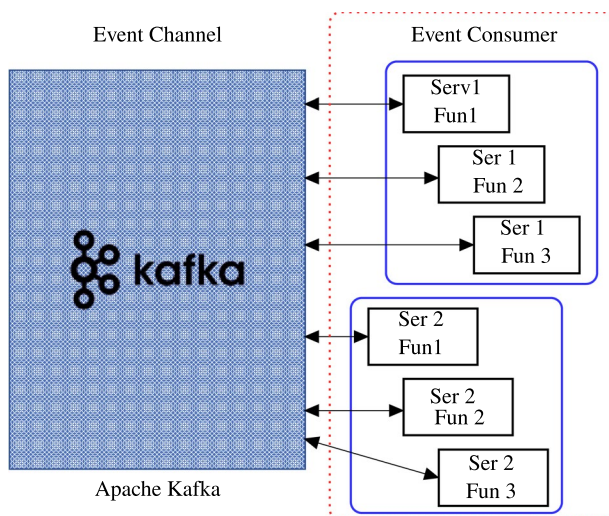


Fig. 3 Micro-service architecture

the benefits of using protobuf against other serialization libraries such as JSON and Binary JSON (BSON). Protocol buffers offer messages with reduced size up to six-fold. This benchmark compares obtained message sizes after decoding for more than 50,000 records, and four message types Command, Response, Request and acknowledgement are investigated. Table 2 gives an overview of binary serialization results and the benefits of using protobuf against other serialization library such as JSON and BSON. Another benchmark aims to examine the performance in terms of serialization and deserialization time for fixed structured data and message size.

Tables 2 and 3 demonstrate that Protobuf is the best choice against XML and JSON from the viewpoint of serialization/deserialization time. To guarantee a high performance communication between devices and cloud applications, Protobuf is selected as messages serializer/deserializer. Thereby, protobuf is used in the proposed architecture.

To conclude, this section describes in details the back-end core architecture. Basically, the proposed architecture is integrated to take-up and act on MQTT broker

Table 2 Comparison of different serialization libraries: Obtained message sizes after serialization [34]

Message	Message count	Original message size [Byte]	Protobuf size [Byte]	BSON size [Byte]	JSON size [Byte]
CMD	2773	86290	116995	885222	914192
REPS	24918	2486123	6551425	31048951	37116788
REQ	241	2764	5159	47838	48950
ACK	23758	308015	502539	4310316	4377701
TOTAL	51690	2883192	7176118	36292327	42457631

Table 3 Comparison of different serialization libraries: Serialization/deserialization time [35]

Technology	Serialization time [ms]	Deserialization time [ms]
XML (XStream)	2.869	5.128
Json (Jackson)	0.148	0.209
Google Protobuf	0.047	0.105

forwarded and directed events (records) by offering many benefits. Running one system with complex functionalities in only one application is known as monolithic application, which causes a good deal of software crisis during the development process. This is exactly the reason why the micro-service design pattern are used as a set of loosely decoupled services managed with agile development process with continuous delivery/deployment of the complex application. One central component in micro-service design is the event-manager. It is responsible for routing events from the producer to the consumer, and it is characterized with availability, durability, throughput and latency. Apache Kafka as stream processing platform outperforms RabbitMq in terms of handling massive data input. It is also suitable for the event processing, offering an expansion of event-driven architecture known as event-stream processing architecture.

4 Implementation of cloud-connected wireless network

In this section, the implementation of the proposed cloud-connected wireless network to monitor the soil moisture is presented as depicted in Fig. 4. The first component contains a number of wireless nodes integrating a soil moisture sensor. Nodes are communicating with the gateway using the Simple Wireless Abstract Protocol (SWAP) [36]. The gateway consists of a Raspberry Pi, which is used as a central node in the system to run the data processing unit. For reliable and fast data transmission between the sensor network and back-end (cloud data hosting and processing architecture), google protocol buffers are used. After forwarding records to the specific micro-service, which represents a Java running application, collected data need to be stored on the database. In order to design and implement the whole system, certain requirements at each level of the design need to be considered.

4.1 Sensing unit

A real-time IoT-based sensor node to monitor the level of water in a plant is developed by using a soil moisture sensor. This application is needed for designing an automated irrigation system. The control of irrigation is highly recommended to reduce the over-use of water and the manpower. To control the level of water on the soil, this unit compromises the soil moisture sensor, namely, VH400 [37] and the wireless node, called panStamp [38]. Indeed, panStamp is a small low-power wireless sensor mote programmable from the Arduino IDE. It uses a MSP430 core

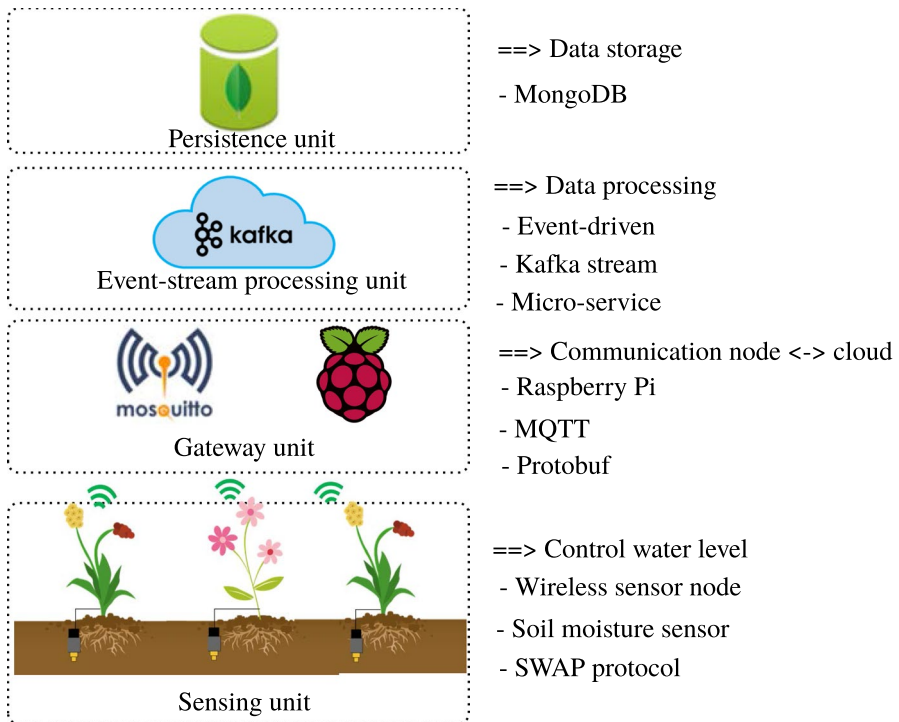


Fig. 4 Proposed cloud-connected wireless network architecture

embedded with a CC1101 RF transceiver, which forms CC430F5137 SoC. It provides data transfer speeds at a rate up to 600 kbps. It can offer a data range of communication up to 200 meters in outdoor applications. The SWAP protocol offers a firmware behind the scenes. It is used to provide an M2M interoperability between simple wireless devices. Each endpoint has its own representation which serves as a way to read value from sensors.

4.2 Gateway unit

In this unit, the panStamp receiver node collects the sensed data from remote nodes and forwards them to the embedded board Raspberry Pi. To ensure the collection of data from multiple sensor nodes and running algorithms to exchange messages with the cloud in a short time, the MQTT protocol is implemented. In fact, it is designed to be a lightweight publisher/subscriber messaging transport protocol. The MQTT client integration needs to guarantee no losses of messages and dynamically handling of distributed system problems. The SWAPdmt running on the Raspberry Pi board has to be connected to the cloud for data exchanging. The listing 1 presents the protobuf file chosen for messages exchanging between the cloud and the

application running on Raspberry Pi. Here, the Protobuf file includes two messages structures: MainMessage and payload.

```
syntax= "proto2 ";
package MainMessage;
message MainMessage {
  required string application ID = 1 ;
  required string direction = 2 ;
  required string ID = 3 ;
  required string timeStamp = 4 ;
  required bytes payload =5;
}
message payload{
  required string modemID=1;
  required string panstampID = 2 ;
  required string registerNr=3;
  required string value =4;
}
```

Listing 1: Protobuf file to handle input and output messages traffic

Each Raspberry Pi running SWAP-dmt application must have a unique identifier. This identifier is the same as the gateway identifier (implemented as an incremental integer). Each transmitter node within the SWAP network must be also configured with a unique identifier number. With the help of this number, the gateway (Raspberry Pi) can configure the transmitter node remotely. Each transmitted message needs to be enriched with timestamp and is uniquely identified with ID field. The main message contains the payload field including values generated from the wireless node.

4.3 Event-stream processing unit

The event-stream processing presents an expansion of the event-driven architecture in terms of functionality. Generally, it represents the first computing task on event-driven architecture. It contains a set of technologies to assist the processing of incoming data flow. In fact, it offers the ability to sense and identify a meaning of the incoming data to indicate the occurrence of a logic situation. Thus, enables the stream application to fast react to opportunities and threats [39]. The choice of event processing pattern over an event-driven architecture depends on the business requirements and the target architecture.

Kafka streams are introduced as a lightweight library designed for processing and analyzing data stored in Apache Kafka. Apache Kafka streams ensures the fault-tolerant by instances replication. This means running another instances of the same application, which are typically located on different machines. After nodes replication, the stream application is dynamically scalable and elastic during runtime. Kafka streams leverages for this reason the topic partitioning feature of Apache Kafka consumer to add parallel processing of records capability. In this case, each instance is in charge of processing some topic partitions. The results of processing can be either reproduced to Apache Kafka or other external

system such as the database. In addition, it offers the important processing tools such as aggregation of data and the ability to differentiate between many timing concept (event time, processing time, etc.). It is also based on many functionalities already implemented in Apache Kafka broker like scaling by partitioning the topics and can be integrated into any Java application. Kafka streams leverage the transactional sending of messages feature provided by Kafka broker to guarantee that the message will be only processed one time. The difference between this feature to other processing frameworks is that Kafka streams automatically handles commits on the input topic offsets, saves on the state stores, writes the results to output topic and waits for acknowledgements [40]. Kafka streams is a horizontal scalable distributed processing engine. guarantee this option, increasing the number of topic partition is the first step. To manually scale down the processing capability, one or more streaming instance need to be removed.

To sum up, the aim of creating Kafka streams is to introduce a new framework having more performance than Kafka Consumer. In fact, Kafka streams offers many benefits by extending Apache Kafka internal working philosophy and by avoiding additional effort and cost related to design and deploy new Cluster. For these reasons, the stream processing part is implemented with Kafka Streams.

Kafka is actually a bridge between the gateway unit and the processing cluster, providing an access point for external systems to consume semantic data. After sending records to the cloud, Kafka streams are the first step on the processing chain, which serves to implement a complex processor topology.

As represented in Fig. 5, the head of processor topology is the source of records, called data source. It presents the source of data input, which receives and processes all records. The source node forwards the records to the next processor, which is identified with the name entry point, and it is responsible for decoding records and sending them to the appropriate processor. The entry point processor has the functionality to choose in which direction the message should be forwarded. In addition, the cloud destination processor has the functionality of decoding the payload message and testing included values. In this step of processing, the message with unsafe and unacceptable values is forwarded to the alert processor. Then, the alert processor notifies in the first step the appropriate gateway. then, it send it back to Apache Kafka for future processing on micro-services. The micro-service needs in addition to react in real-time to each wrong operation of wireless sensor nodes. An example of message filtering for alerting purposes is the node battery level or high soil moisture. The stream processing application writes all incoming records after processing back to Apache Kafka. The sink processor is also responsible for the serialization and deserialization of records key and value. Micro-services are java running applications implemented with Spring boot and managed with Apache Maven. Each micro-service handles a small task by implementing computational logic and listens to a specific Kafka topic, which represents the data source. By distributing the complex functionalities into small tasks, micro-services are used to reduce the system complexity. To hold the real-time capability, micro-services have to deal with the database, which also offers this feature.

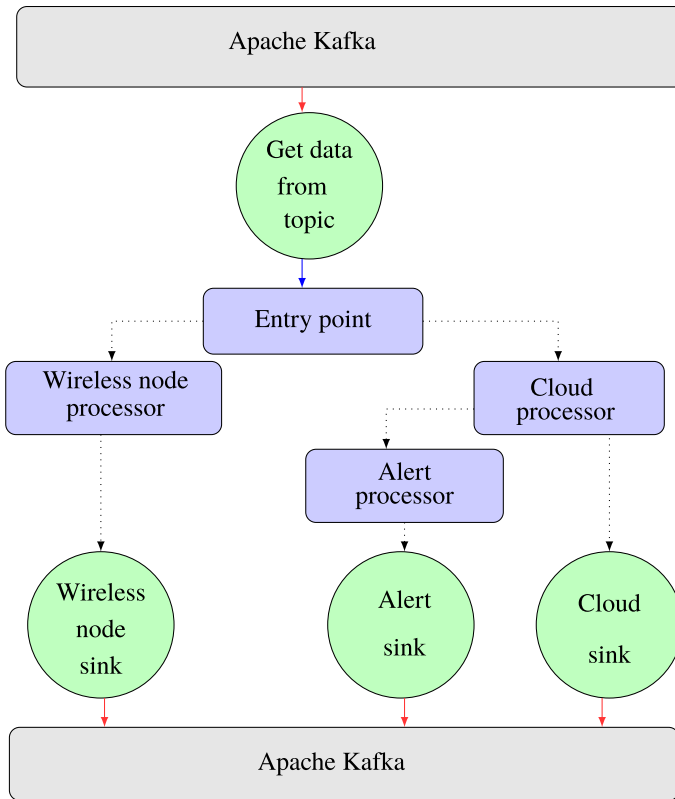


Fig. 5 Stream processing application implementation: Processor topology

4.4 Persistence unit

The persistence unit is responsible for storing data analyzed by the cluster processing layer to allow further analysis. It is also an access point for external systems to retrieve the stored data. Several solutions and products for big data databases are introduced. Each with its own specification, strengths and limits. MongoDB [41] is used as distributed document database, which is designed for web applications and big data storage. In fact, it extends the functionality of SQL and gives more flexibility for the design of data models. Moreover, it provides a powerful query language, index structures and many more functions. MongoDB claims also to provide the functionality of SQL while achieving the high speed and scalability of key value stores.

During this section, the back-end cloud architecture and components deployment of REDA are described. The target architecture is designed to be running independently from the cloud providers, but the choice of the cloud provider is still difficult because the aim of this work is to implement a high available architecture without long-term contracts or up-front commitments. AWS is one of the most known cloud providers. It is designed by Amazon to allow application providers to fast and securely host their application. With AWS it's possible to choice the operation system, database and to create a virtual network, additionally it uses an end to end approach to secure the resources, offers mechanisms to real time monitoring the hardware, include new software metrics.

The back-end cloud architecture and components deployment are described besides of the placement of the Virtual Private Cloud (VPC) for resources hosting as illustrated in Fig. 6. The VPC allows the definition of the own network architectures, which corresponds largely to conventional networks. In addition, the VPC gives system architects complete control over their virtual network

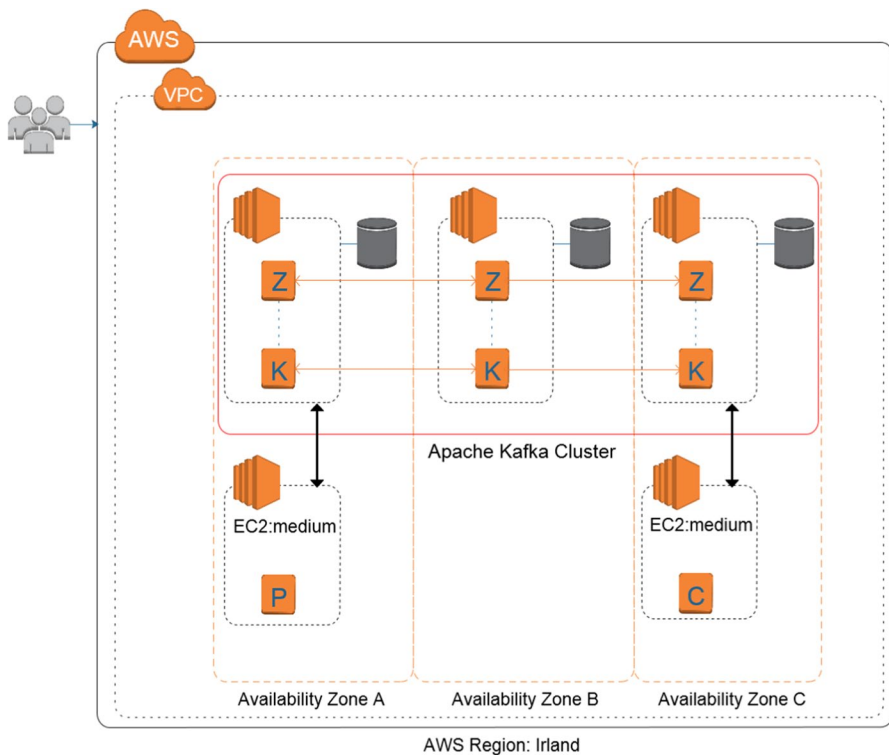


Fig. 6 AWS Architecture for load test

environment, including choosing an IP address range, creating subnets and configuring route tables and network gateways.

The AWS hosts a physical machine, namely Amazon Elastic Compute Cloud (EC2) instances, which provides a flexible provision of computer resources as a primary service. The Amazon EC2 can perform the business requirement with an almost unlimited amount of virtual machines. During the creation of EC2 instances, it is necessary to provide the Availability Zone, or simply the isolated location within the AWS region. As seen in Fig. 6, the AWS offers in each region three Availability Zones, and this is the key of high availability (Table 4).

The instance model, data persistence strategy and network bandwidth are the key of success for the cloud architecture in terms of processing capability, simplicity of monitoring and costs. The deployment of MQTT message broker replicas guarantees a high availability in terms of devices synchronization. After this step, Apache Kafka cluster co-located with Zookeeper is deployed over three availability zones and optimized for high throughput and low latency. The final step of architecture deployment is always the creation and execution of load tests or simply the simulation of high input/output traffic.

5.1 MQTT message broker setup

To ensure a dynamic fault-tolerant system and prevent network fails, a MQTT broker cluster is required, which presents the entry point (synchronization level) in the target architecture. Mosquitto is a high available MQTT broker, designed to handle a big number of simultaneous MQTT clients with minimal RAM usage [27]. Moreover, Mosquitto enables the transmission of records automatically between a duplicated number of brokers [42]. To take advantage of this feature, it is important to split topics across many brokers, and each broker forwards messages to the others.

5.2 Zookeeper Quorum architecture

The first step of setting a Zookeeper Quorum is to create replicas across several predefined hosts. Each replica runs in a separate AWS availability zones. The Zookeeper cluster expands this features by integrating each node in a separated zone. This solution continues working in the worst case, even when two instances fail. After the creation of three EC2 instances in the same VPC and assigning a static IP

Table 4 AWS T2 instances configuration and costs

Model	CPU	Mem [GB]	Cost [Dollar/Hour]
t2.nano	1	0.5	0.0058
t2.micro	1	1	0.0116
t2.small	1	2	0.023
t2.medium	2	4	0.0464
.	.	.	.
t2.2xlarge	8	32	0.3712

address for each instance, Zookeeper servers are settled up. Each server is contained in a single Java JAR file, which is related to a specific configuration file. Each Zookeeper broker saves the data in dataDir repository. Then, it listens by default to port 2181 to connect with Kafka cluster and sends periodically (tickTime) heartbeats to all brokers in the quorum as depicted in listing 2.

```
tickTime=2000,
dataDir=/var / zookeeper/,
clientPort =2181,
initLimit=5,
syncLimit=2,
server.1=zoo1:2888:3888,
server.2=zoo2:2888:3888,
server.3=zoo3:2888:3888
```

Listing 2: Simple Zookeeper configuration file on one server

5.3 Kafka cluster setup

Similar to Zookeeper Quorum, Apache Kafka is deployed across many instances to create a fully distributed multi-broker clusters. Zookeeper and Kafka brokers are co-located on the same instance to save additional costs. Apache Kafka provides more than 150 configuration parameters for scaling the desired architecture. It is often used to store critical data, therefore it is one of the important components of the project data infrastructure. To start a simple message broker, it is not necessary to manually control all those parameters. A basic configuration file is presented in the listing 3.

```
broker.id—Id of the broker,
log.dirs — Directory where Kafka must commit message,
port — Port on which Kafka will accept connections from
producers and consumers,
zookeeper.connect — Comma separate list of ZooKeeper nodes.E.g.
hostname1: port1 , hostname2: port2 ...
```

Listing 3: Simple Kafka configuration file on one instance

The first benchmark is designed to test different acknowledgement types with various messages sizes. For a fixed message size and a fixed amount of data (8 GB), both throughput and latency will be measured. As presented in Fig. 7a and b, the message acknowledgement has a deep impact on the throughput as well as on the latency.

Records with small sizes are usually the biggest problem for message brokers. Benchmarks presented in Fig. 8a and b demonstrate this by measuring the throughput (MB/s) and latency (ms) for many records sizes. Almost, Apache Kafka with basic configuration weakens with handling the incoming data flow with small sizes. Messages exchange in the SWAP network have usually a very small size between 256 and 512 bytes. Therefore, records transmitted from python application listened

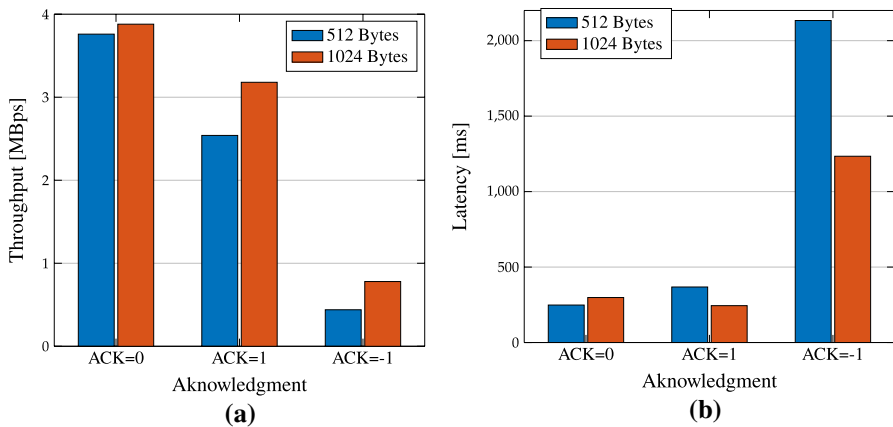


Fig. 7 Apache Kafka latency for many message acknowledgement: **a** Throughput, **b** latency

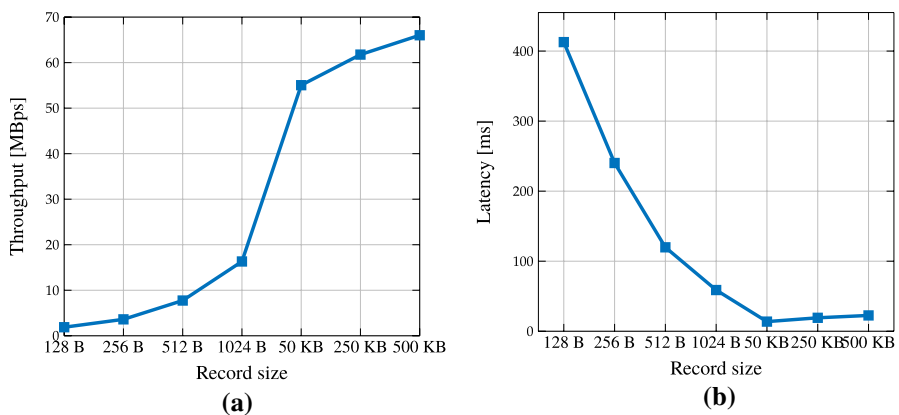


Fig. 8 Benchmark for many message sizes: **a** Throughput, **b** latency

to the gateway node in the serial port are serialized in the Protobuf format. The serialization with Protobuf will in addition decrease the record size. Apache Kafka as demonstrated in the last benchmark weakens with handling small size records. To avoid such problem, three optimization solutions are introduced in this paper. The first one consists on the creation of topics with more partitions. The second approach is to batch records with the producer, and the third solution is to add more worker threads to improve the Kafka producer.

5.3.1 Creation of topic with more partitions

The first message transmission optimization solution is to increase the number of partitions for each topic. The topic partitions are the logical parallelism unit offered by Kafka broker. The producer can send messages to different partitions in parallel.

The message broker handles those messages from its side also in parallel. Finally, the Kafka consumer is able to poll/consume those messages in parallel.

The idea of the next benchmark, as illustrated in Fig. 9a and b, is varying the number of partitions for the same record size (in byte) transmitted to topic. For each measurement, a fixed amount of data (8 GB) is transmitted.

As presented in Fig. 9a and b, increasing the number of partitions loads generates better throughput. In fact, to get better performance, it is important to assign a big number of partitions for topics. Starting from a specific number of partitions, both throughput and latency of the message broker are constants. Due to resource limits, increasing the number of partition leads to decrease the message broker performance. The point of recession is not the optimal number of partitions because this benchmark focuses only on the producer performance, and it is running dependably on the consumers processes performance.

5.3.2 Batching record by producer

Another optimization behavior is to simulate the case of average message size with batching producer out-going messages. The producer can batch outgoing messages transmitted to the same partition, i.e., it collects multiple messages to send together in a single request. Therefore, a big size message can be processed and handling incoming records effort using the message broker is decreased. In real-time, routing and processing environment batching can be seen as a latency killer, because the Kafka producer postpones transmitted records until the batch size is reached. The next benchmark, as seen in Fig. 10a and b, demonstrates the impact of batching on improving the throughput and latency by increasing the batch size. Larger batch size results in better throughput. This is a result of reducing requests number to the broker and decreasing requests load on producers as well brokers side. As presented in Fig. 10b, increasing the batch size do not have practically a bad effect on the latency.

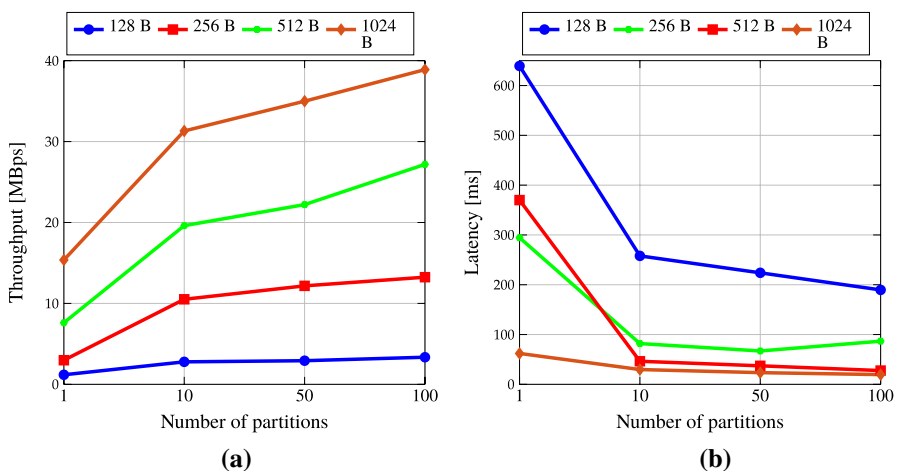


Fig. 9 Optimization by increasing number of partitions: **a** Throughput, **b** latency

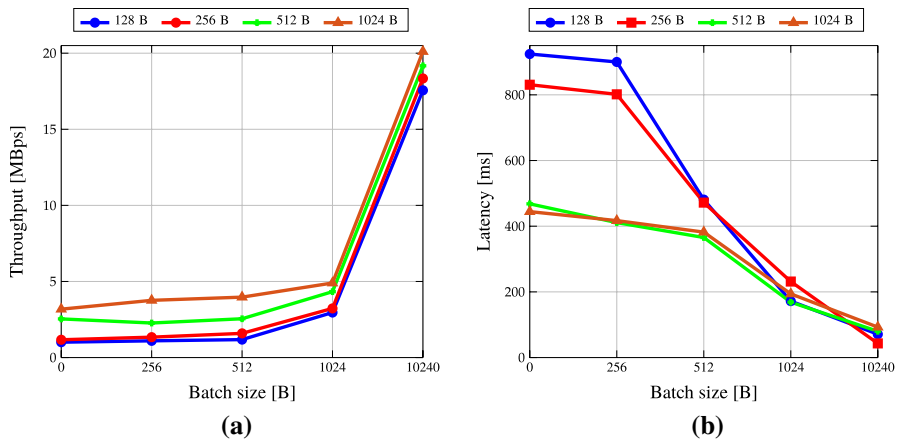


Fig. 10 Apache Kafka producer improving with batching: **a** Throughput, **b** latency

The value of latency is reduced noticeably by increasing the batch size from 0 to 10240 bytes. This is related to the internal structure of the Kafka producer, which creates a logical transmission channel with each topic partition leader. This channel presents a queue, which follows the FIFO principle, and each record transmitted via this queue has a specific time called the queue time. The network bandwidth has an impact on records queuing time. So, configuring the cluster with a high network throughput can decrease the queuing time of records. Saving records with batching from piling up in the transmission queue are the reason of decreasing the latency. Otherwise, sending a batch of records in one request needs only one acknowledgement from the server, which can reduce from its side the network overhead.

5.3.3 Improving Kafka consumer by adding more partitions

Another important part in the optimization process is to setup an efficient Kafka consumer, which presents an instance of micro-service in the architecture to support a big amount of data stored in Apache Kafka. As seen in Fig. 11, similar to the Kafka producer, increasing the partitions number of topic and batching the out-going records dive to more performance. Batching can cause additional problems by records consuming in case of failure. So, when the Kafka producer fails, it needs to repeat records reading from the last record committed. This requires either an advanced committing strategy or handling records redundancy in the micro-service application. Consumers assign to one consumer group to distribute records processing of one or more topics. To ensure the parallel processing of data on the micro-service, the developer has to implement a thread save behavior, which instantiates many consumer instances with the same group-name.

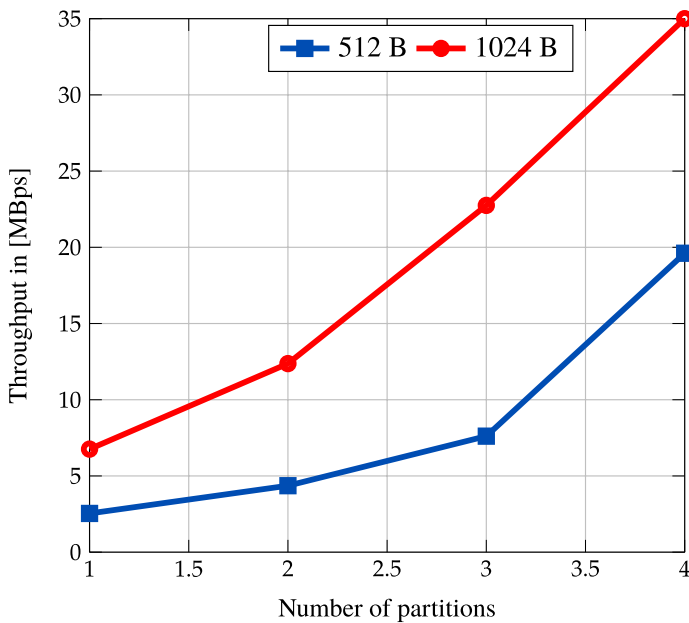


Fig. 11 Increasing consumer throughput by adding more topic partition

5.4 Load test setup

The load test, also known as system benchmarking, is an important step for distributed systems. It aims to simulate high traffic of incoming/outcoming data. The architecture used to generate the load test and measure the performance is depicted in Fig. 6. The message producer and consumer are located each in a new instance, that makes them separate from the Kafka cluster and gives a transparent and real benchmark results. After the deployment of Apache Kafka, the load test starts. The goal of such tests is to ensure that the message broker does not crash in many conditions. The first load test case is to simulate high traffic of incoming data generated from devices. After a high load generation, the behavior of systems needs to be checked in many sides such as brokers status or computational resources. The topic added to the Kafka broker is created with 10 partitions. This number of partitions can guarantee the desired throughput and latency without batching of records. The results from Kafka-Perf are represented in the listing 4:


```
4069000 records sent , 8391.21 records/sec (2.05 MB/sec) ,
430.23 ms avg latency , 416 ms 50th , 480 ms 95th , 636 ms 99th ,
2283 ms 99.9th .
```

Listing 4: Apache Kafka producer performance results

The load test is started at the time 11:50 and takes nearly 15 minutes execution time. As described in listing 4, the producer continues sending records with throughput 2 MB/s (see Fig. 12). When the records reach the topic leader, they must be firstly committed and saved in the disk and then replicated in the other two brokers. Figure 14 illustrates the read/write disk throughput. Figure 13 presents the CPU consumption during the run of load test on one message broker (leader).

6 Limitations and future works

This research paper can be extended in different ways. One of the primary future work is to overcome the limitations of the designed architecture by considering the security aspect in different levels, implementing alternatives to manage the limited data size of MangoDB. Another future approach in our work is to optimize more the developed architecture by adding more availability and scalability features including the creation of n nodes MQTT broker cluster, the data parallel processing and data replication and sharding. Regarding the architecture design evaluation, some parameters need to be considered such as cost, QoS and data reliability. In addition, we plan to use the fog computing at the edge of the network, which is an emerging

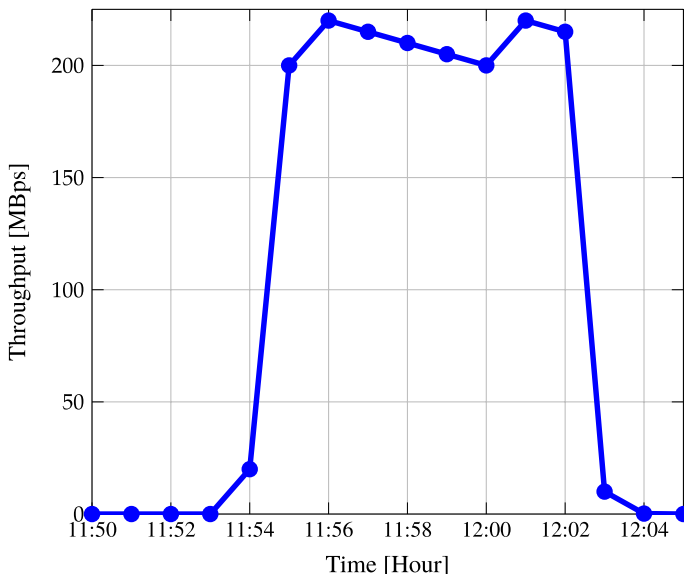


Fig. 12 Producer average outgoing data flow

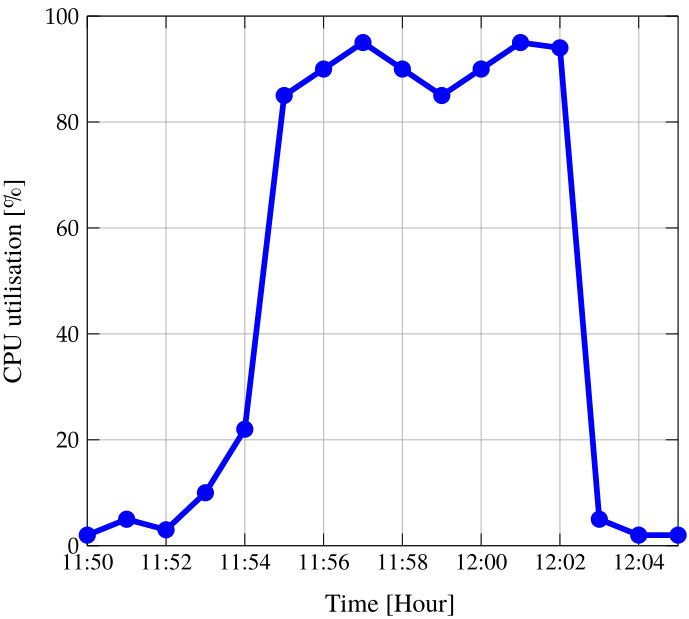


Fig. 13 CPU consumption during benchmark

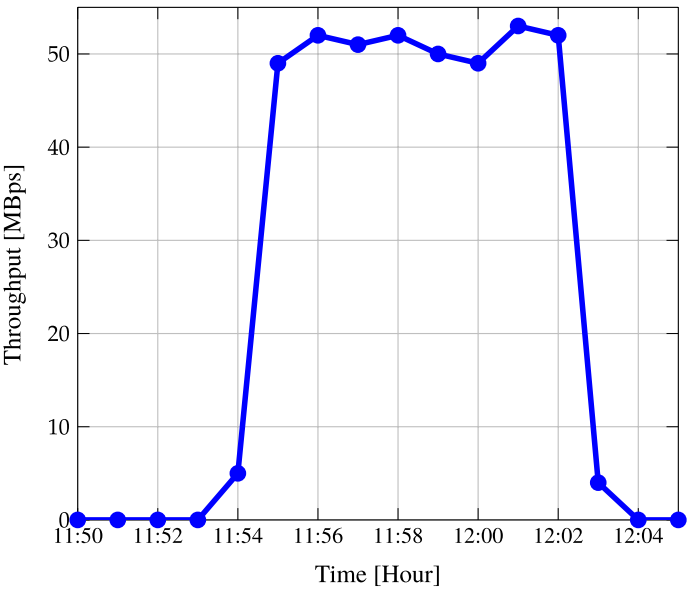


Fig. 14 Disk storage throughput

technology that provides intelligent connection networks between IoT devices and cloud computing platforms. This layer allows local aggregation of data and deletes irrelevant data from the network. Implementing the architecture for application with high demand of data processing such as healthcare applications are presented as on of the main further works. In our solution, the MongoDB is used, which was scalable enough to support some hundreds of requests in one second. There is further analysis addressing this point by expecting a big size storage database as well high memory usage. One possible solution will be there MongoDB Atlas located on the cloud or cloud provider MongoDB implementation. Last but not least, we are planning to extend our evaluation to include recent architectures built on similar or different cloud platforms, such as AWS, IBM and Azure. This will lead to highlight more the novelty and the strengths of our architecture.

7 Conclusion

To process a massive flux of data and to react in real-time, a cost-effective cloud architecture is developed. REDA is composed mainly of four units. A sensing unit contains low-power wireless sensor nodes integrating soil moisture sensors to monitor the water level on the soil. Sensed data are forwarded to the gateway unit (Raspberry Pi), presented as a central node in the system. To collect data from multiple sensor nodes in a fast way and to send/receive messages from/to the cloud running application, the lightweight publisher/subscriber messaging transport protocol, MQTT, is used. The event-stream processing unit is based on an event-driven backend core architecture. After records sent to the cloud, Kafka streams are the first step on the processing chain, a lightweight horizontal scalable processing framework, which serves to implement complex processor topology. To avoid the problem of complexity during the system development, micro-services design patterns are used. In fact, they are Java running applications implemented with Spring boot and managed with Apache Maven. Each micro-service handles a small task by implementing a computational logic and listens to the specific Kafka topic, which presents the data source. To hold the real-time capability, micro-services must deal with database (persistence unit). The MongoDB as distributed document database, is selected due to the internal structure, high availability and scaling capabilities. To deploy our solution, Apache Kafka cluster co-located with Zookeeper is deployed over three availability zones and optimized for high throughput and low latency. To guarantee no message loss and to simulate the system performances, different load tests are launched. Therefore, it is demonstrated, that the proposed architecture guarantees a high data availability by handling about 8000 messages in a second with very low latency in a very cheap hosted and configured architecture.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as

you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References


1. Mukherjee A, Roy DG, De D (2019) Mobility-aware task delegation model in mobile cloud computing. *J Supercomput* 75:314–339. <https://doi.org/10.1007/s11227-018-02729-x>
2. Talib MA, Majzoub S, Nasir Q et al (2021) A systematic literature review on hardware implementation of artificial intelligence algorithms. *J Supercomput* 77:1897–1938. <https://doi.org/10.1007/s11227-020-03325-8>
3. Baranwal G, Vidyarthi DP (2016) Admission control in cloud computing using game theory. *J Supercomput* 72:317–346. <https://doi.org/10.1007/s11227-015-1565-y>
4. Park DS (2018) Future computing with IoT and cloud computing. *J Supercomput* 74:6401–6407. <https://doi.org/10.1007/s11227-018-2652-7>
5. Cerwall P, Jonsson P, Möller R, Bävertoft S, Carson S, Godor, I Ericsson mobility report, On the Pulse of the Net1030 worked Society. Hg. v. Ericsson
6. Chéour R, Jmal MW, Abid M (2018) New combined method for low energy consumption in wireless sensor network applications. *Simulation* 94(10):873–85. <https://doi.org/10.1177/0037549718759432>
7. Khriji S, Houssaini DE, Kammoun I, Kanoun O (2018) Energy-efficient techniques in wireless sensor networks: technology, components and system design. *Energy Harvest Wirel Sensor Netw*. <https://doi.org/10.1515/9783110445053-017>
8. Chéour R, Khriji S, El Houssaini D, Baklouti M, Abid M, Kanoun O (2019) Recent trends of FPGA used for low-power wireless sensor network. *IEEE Aerosp Electron Syst Mag* 34(10):28–38
9. Khriji S, Houssaini DE, Kammoun I, Kanoun O (2018) A fuzzy based energy aware unequal clustering for wireless sensor networks. In: Montavont N, Papadopoulos G (eds) Ad-hoc, mobile, and wireless networks., vol 11104. Springer, Cham (**ADHOC-NOW 2018. Lecture Notes in Computer Science**)
10. Chéour R, Khriji S, Kanoun O (2020) Microcontrollers for IoT: optimizations, computing paradigms, and future directions. In: 2020 IEEE 6th World Forum on Internet of Things (WF-IoT) pp 1-7. IEEE
11. Tabrizchi H, Kuchaki Rafsanjani M (2020) A survey on security challenges in cloud computing: issues, threats, and solutions. *J Supercomput* 76:9493–9532. <https://doi.org/10.1007/s11227-020-03213-1>
12. Li C, Bai J, Luo Y (2020) Efficient resource scaling based on load fluctuation in edge-cloud computing environment. *J Supercomput* 76:6994–7025. <https://doi.org/10.1007/s11227-019-03134-8>
13. Mihai V, Dragana C, Stamatescu G, Popescu D, Ichim L, 2018 Wireless sensor network architecture based on fog computing. In: 2018 5th International Conference on Control, Decision and Information Technologies (CoDIT), Thessaloniki, Greece, 10-13 April . IEEE (2018), pp 743-747. <https://doi.org/10.1109/CoDIT.2018.8394851>
14. Hsu CH, Fox G, Min G et al (2019) Advances in big data programming, system software and HPC convergence. *J Supercomput* 75:489–493. <https://doi.org/10.1007/s11227-018-2706-x>
15. Zhong RY, Xu X, Klotz E, Newman ST (2017) Intelligent manufacturing in the context of industry 4.0: a review. *Engineering* 3(5):616–630. <https://doi.org/10.1016/J.ENG.2017.05.015>
16. Gubbi J, Buyya R, Marusic S, Palaniswami M (2013) Internet of Things (IoT): a vision, architectural elements, and future directions. *Future Gener Comput Syst* 29(7):1645–1660. <https://doi.org/10.1016/j.future.2013.01.010>
17. Seo D, Jeon YB, Lee SH et al (2016) Cloud computing for ubiquitous computing on M2M and IoT environment mobile application. *Cluster Comput* 19:1001–1013. <https://doi.org/10.1007/s10586-016-0573-x>
18. Zhou C, Zhang X (2014) Toward the internet of things application and management: a practical approach. In: 2014 Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks, Sydney, NSW, Australia, 19 June 2014. IEEE , pp 1-6. <https://doi.org/10.1109/WoWMoM.2014.6918928>

19. Chenaru O, Stamatescu G, Stamatescu I, Towards Popescu D, cloud integration for industrial wireless sensor network systems. In: 2015 9th International Symposium on Advanced Topics in Electrical Engineering (ATEE), Bucharest, Romania, 7–9 May 2015. IEEE 2015:917–922. <https://doi.org/10.1109/ATEE.2015.7133933>
20. Kurniawan A (2018) Learning AWS IoT: effectively manage connected devices on the AWS cloud using services such as AWS Greengrass, AWS button, predictive analytics and machine learning. Packt Publishing Ltd
21. Richard Olaniyan, Olamilekan Fadahunsi, Muthucumaru Maheswaran, Faten Zhani Mohamed (2018) Opportunistic edge computing: concepts, opportunities and research challenges. *Future Gener Comput Syst* 89:633–645
22. John O’Loughlin, Lee Gillam (2018) A performance brokerage for heterogeneous clouds. *Future Gener Comput Syst* 87:831–845
23. la Prieta De F, Rodríguez-González S, Chamoso P, Corchado JM, Bajo J (2019) Survey of agent-based cloud computing applications. *Future Gener Comput Syst* 100:223–236
24. Pogiatzis A, Samakovitis G (2021) An event-driven serverless ETL pipeline on AWS. *Appl Sci* 11(1):191
25. Helmer S, Poulouvassilis A, Xhafa F (2011) Introduction to reasoning in event-based distributed systems. In: Helmer S, Poulouvassilis A, Xhafa F (eds) Reasoning in event-based distributed systems, vol 347. Springer, Berlin, Heidelberg (**Studies in Computational Intelligence**)
26. Bruns R, Dunkel J (2010) Event-driven architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse. Springer-Verlag
27. Rieke M, Bigagli L, Herle S, Jirka S, Kotsev A, Liebig T, Malewski C, Paschke T, Stasch C (2018) Geospatial IoT—the need for event-driven architectures in contemporary spatial data Infrastructures. *ISPRS Int J Geo Inf* 7(10):385. <https://doi.org/10.3390/ijgi7100385>
28. Yassein MB, Shatnawi MQ, Aljwarneh S, Internet Al-Hatmi R (2017) Internet of Things: Survey and open issues of MQTT protocol. In: 2017 International Conference on Engineering & MIS (ICEMIS), Monastir, Tunisia, 8–10 May 2017. IEEE 2017:1–6. <https://doi.org/10.1109/ICEMIS.2017.8273112>
29. Soni D, Makwana A (2017) A survey on MQTT: a protocol of internet of things (IoT). In: International Conference on Telecommunication, Power Analysis and Computing Techniques (ICTPACT-2017), Bharath Institute of Higher Education and Research, 173, Agharam Road, Selaiyur, Chennai, India
30. Philippe D, Kyumars SE (2017) Kafka versus RabbitMQ: a comparative study of two industry reference publish/subscribe implementations: Industry Paper. In: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS ’17). Association for Computing Machinery, New York, NY, USA, pp 227–238. <https://doi.org/10.1145/3093742.3093908>
31. John V, Liu X (2017) A survey of distributed message broker queues. arXiv preprint [arXiv:1704.00411](https://arxiv.org/abs/1704.00411)
32. Kreps J, Narkhede N, Rao J (2011) Kafka: a distributed messaging system for log processing. *Proc NetDB* 11:1–7
33. Newman S 2015. Building microservices: designing fine-grained systems. O’Reilly Media, Inc
34. Popić S, Pezer D, Mrazovac B, Teslić N (2016) Performance evaluation of using Protocol Buffers in the Internet of Things communication. In: 2016 International Conference on Smart Systems and Technologies (SST), Osijek, Croatia, 12–14 Oct. 2016. IEEE , pp 261–265. <https://doi.org/10.1109/SST.2016.7765670>
35. Maeda K (2012) Performance evaluation of object serialization libraries in XML, JSON and binary formats. In: 2012 Second International Conference on Digital Information and Communication Technology and its Applications (DICTAP), Bangkok, Thailand, 16–18 May 2012. IEEE , pp 177–182. <https://doi.org/10.1109/DICTAP.2012.6215346>
36. Karagiannis V (2014) Building a Testbed for the Internet of Things. Alexander Technological Educational Institute of Thessaloniki 1–92
37. Khriji S, El Houssaini D, Kammoun I, Kanoun O (2021) Precision irrigation: an IoT-enabled wireless sensor network for smart irrigation systems. In: Hamrita T (ed) Women in precision agriculture. Springer, Cham (**Women in Engineering and Science**)
38. Khriji S, Kallel AY, Reedy S, El Houssaini D, Kammoun I, Kanoun O (2019) Dynamic autonomous energy consumption measurement for a wireless sensor node. In, (2019) IEEE International Symposium on Measurements & Networking (M&N), Catania, Italy. 8–10 July 2019. IEEE 2019:1–5. <https://doi.org/10.1109/IWMN.2019.8805001>
39. Etzion O, Niblett P (2011) Event processing in action. Simon and Schuster, New York City
40. Apache kafka streams documentation, <https://kafka.apache.org/documentation/streams/>, access (January 2021)

41. Chodorow K (2013) MongoDB: the definitive guide: powerful and scalable data storage. O'Reilly Media Inc, Sebastopol
42. Light RA (2017) Mosquitto: server and client implementation of the MQTT protocol. Int J Open Source Softw Process [https:// 2\(13\):265](https://doi.org/10.21105/joss.00265). <https://doi.org/10.21105/joss.00265>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Sabrine Khriji¹  · Yahia Benbelgacem¹ · Rym Chéour² · Dhouha El Houssaini¹ · Olfa Kanoun¹

Yahia Benbelgacem
benbelgacemyahya@gmail.com

Rym Chéour
rym.cheour@enis.tn

Dhouha El Houssaini
dhouha.el-houssaini@etit.tu-chemnitz.de

Olfa Kanoun
olfa.kanoun@etit.tu-chemnitz.de

¹ Chair of Measurement and Sensor Technology, Technische Universität Chemnitz, Chemnitz, Germany

² National School of Engineers of Sfax, University of Sfax, Sfax, Tunisia