Sensor Observation Streams Within Cloud-based IoT Platforms: Challenges and Directions

Antoine Auger†, Ernesto Exposito‡ and Emmanuel Lochin†
†Institut Supérieur de l'Aéronautique et de l'Espace (ISAE-SUPAERO),
Université de Toulouse, 31055 Toulouse Cedex 4, France
{antoine.auger, emmanuel.lochin}@isae.fr
‡Laboratoire Informatique de l'Université de Pau et des Pays de l'Adour (LIUPPA)
ernesto.exposito@univ-pau.fr

***Abstract*—Observation streams can be considered as a special case of data streams produced by sensors. With the growth of the Internet of Things (IoT), more and more connected sensors will produce unbounded observation streams. In order to bridge the gap between sensors and observation consumers, we have witnessed the design and the development of Cloud-based IoT platforms. Such systems raise new research challenges, in particular regarding observation collection, processing and consumption. These new research challenges are related to observation streams and should be addressed from the implementation phase by developers to build platforms able to meet other non-functional requirements later. Unlike existing surveys, this paper is intended for developers that would like to design and implement a Cloud-based IoT platform capable of handling sensor observation streams. It provides a comprehensive way to understand main observation-related challenges, as well as non-functional requirements of IoT platforms such as platform adaptation, scalability and availability. Last but not the least, it gives recommendations and compares some relevant open-source software that can speed up the development process.**

***Index Terms*—Internet of Things; sensors; observations; streams; Quality of Information; Autonomic Computing; Cloud Computing; Software Architecture Patterns**

I. Introduction

In a report issued in 2011, Cisco predicts that 50 billion connected *Things* will be in use worldwide in 2020 [1]. In order to report information, these *Things* will require adequate connectivity and will contribute to the extension of the Internet of Things (IoT) [2], [3]. Among these *Things*, sensors are ubiquitous. Indeed, whether physical or virtual, they represent an opportunity to gather information about our daily lives and our surrounding world.

To ingest and process large data streams coming from sensors, we have witnessed the deployment of many Cloud-based IoT platforms. Their main purpose is generally to provide enhanced services or information to end consumers (either users or applications) by taking advantage of gathered observations (e.g., Smart City services for citizens). However, the design and the implementation of such platforms raise new research challenges. In particular, the collection, processing and consumption of dynamic and heterogeneous unbounded observation streams are challenging.

The past decade has seen the development of many Cloud-based IoT platforms. Some of these solutions are proprietary commercial solutions. Therefore, they are of little interest for developers that would like to design and implement their own platforms with custom features. Other research efforts exist but they mainly focus on specific considerations (protocols for observation collection, sensor selection, in-network aggregation, etc.) and cannot be directly applied by developers.

On the contrary, this paper aims to present main design challenges and some open-source softwares that may help to build Cloud-based IoT platforms able to deal with sensor observation streams. Far from being exhaustive, this paper provides a comprehensive way to understand major observation-related challenges, as well as non-functional requirements of IoT platforms like platform adaptation, scalability and availability. Finally, it also gives recommendations and compares some relevant open-source software that can speed up the development process. The thoughts and "lessons learned" presented in this paper come from software documentation, research papers and from the custom implementation of an integration platform for QoI Assessment as a Service (iQAS) [4]. The iQAS platform is one of our previous contributions, intended for Smart City stakeholders who want to assess, better understand and improve Quality of Information (QoI) in a collaborative way.

The rest of this paper is structured as follows: Section II introduces three observation-related challenges, dealing with observation levels, Quality of Observation and unbounded observation streams. Then, Section III presents technical solutions to perform observation collection, processing and consumption. Section IV focuses more on non-functional requirements for Cloud-based IoT platforms. Finally, Section V presents related work while Section VI concludes and gives some perspectives.

II. Observation-related challenges

A. Observation levels

IoT platforms receive observations from sensors. Each observation may be considered as the representation of a physical-observed phenomenon (the temperature of a place, a person that enters a room, etc.) or a virtual-occurred event (a

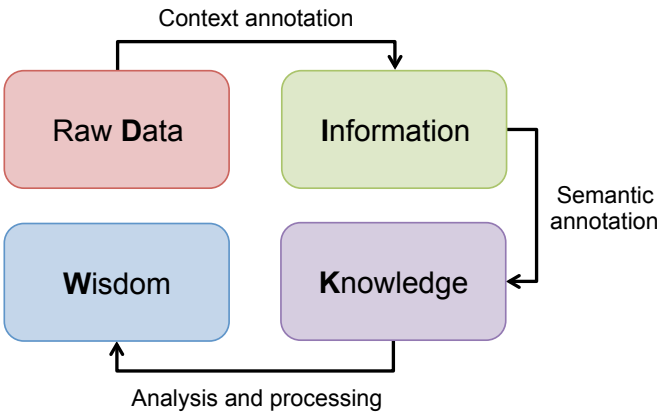


Fig. 1: Example of four observation levels that could be delivered by an IoT platform implementing the "DIKW ladder"

new tweet from someone, an incoming e-mail, the availability of a new software update, etc.).

Previous studies have proposed taxonomies to denote the different observation levels that an IoT platform may provide. Indeed, the same phenomenon or event can be reported in several ways, including more or less details about the unit of the measure, sensor type, location, etc. As a matter of fact, these taxonomies use ladder representations to denote these different observation levels. For instance, Sheth proposed the "Data, Information, Knowledge, and Wisdom (DIKW) ladder" for the IoT [5]. Such taxonomies are useful to estimate the level of complexity required to process and "understand" the observations by consumers. In the case of IoT platforms, these consumers may either be applications or users. Figure 1 shows different observation levels that an IoT platform may deliver to its consumers. Please note that definitions of observation levels may vary according to authors and use cases.

Raw Data generally refers to unprocessed observations directly coming from sensors. Raw observations can either refer to phenomena (for physical sensors) or events (for virtual sensors). **Information** denotes richer observations that have been processed or annotated with Context information [6]. For instance, sensor provenance, spatio-temporal information and sensor confidence level are some example of concrete Context attributes. The use of semantic-based representations (in general using ontologies) allows consumers to consume machine-understandable **Knowledge**. We kindly remind the reader that ontologies allow to model sensor-related thematic fields (such as the weather or oceanography for instance) with the definition of concepts, relationships, as well as other representation details (units, ranges values, etc.). Therefore, ontologies also allow high-level inference and reasoning from sensor observations. The most popular ontology for sensors and observations is certainly the Semantic Sensor Network Ontology (SSN) [7] and have been developed by the W3C. Finally, the analysis and processing of incoming Knowledge is denoted as **Wisdom**. To deliver this last level of sensor observations, IoT platforms must often perform Complex Event Processing (CEP) or other advanced processing techniques (see also Section III-B).

B. Quality of Observation (QoO)

For developers, the design and implementation of information-centric platforms come with new research challenges closely linked to information quality [8]. Considered for a while, common Quality of Service (QoS) metrics have shown limitations to characterize and evaluate information quality [9]. In practice, QoS mostly refers to network QoS (i.e., mainly to network packet transportation). Besides, network QoS is no longer suitable to characterize information required by a given consumer within a specific context.

Quality of Information (QoI) has been introduced to extend the commonly-used QoS metrics (bandwidth, delay, jitter and losses), which were too restrictive. In [9], Bisdikian et al. defined QoI as "*the body of tangible evidence available (i.e., the innate information properties) that can be used to make judgments about the fitness-of-use and utility of information products*". Others quality dimensions have been extensively studied, such as Context information [6], in particular in the domain of Context-aware systems [10]. In the following, we use the term "**Quality of Observation (QoO)**" to denote QoI applied to sensor observations in general.

Within IoT platforms, Observations are the new Information. Most of the time, these platforms provide services by assuming that observations received from sensors are reliable and of better quality than that required by consumers. Unfortunately, since this is not always the case, IoT platforms should be able to characterize Quality of Observation to take appropriate decisions if needed. Depending on sensors, applications and use cases, it may be relevant to use several quality dimensions (e.g., network QoS, QoI and Context information together) to improve this characterization process. For instance, using both network QoS and QoI, an observation consumer can better understand if some outdated observations are the result of poor network performances or due to a sensor sampling rate too low.

C. Unbounded observation streams

Compared to common information-centric systems that rely on traditional databases, IoT platforms have to deal with unbounded observation streams. To cope with their inherent challenges, non-blocking operators and sliding windows are some techniques that are almost always considered to process data streams [11]. These techniques allow the design and the implementation of **adaptation mechanisms** such as Aggregation, Fusion or Filtering for instance. Within IoT platforms, such adaptation mechanisms are required given that underlying sensors may produce observations of diverse quality (i.e., diverse spatio-temporal granularities) due to their different capabilities. To handle this observation heterogeneity, the platform may enable or disable adaptation mechanisms on the fly to meet consumer needs.

Figure 2 shows an example of three unbounded observation streams. The first one (marked as "Incoming stream") will be used to produce the two others thanks to a sliding

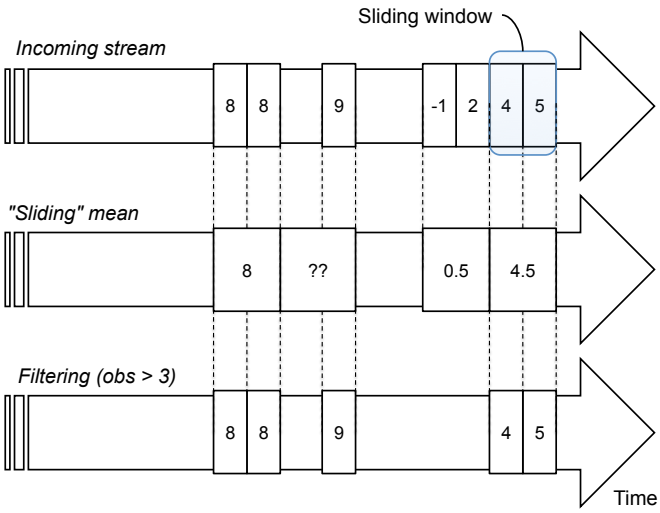


Fig. 2: An example of two non-blocking operators applied to an original unbounded observation stream

window. Please note that a sliding window can either be time-based (e.g., 2 seconds) or count-based (e.g., 2 observations). In this example, we use a 2-observation sliding window. The second stream is computed by doing the average of observation values contained within the sliding window. Finally, the last stream only contains observations that report a value greater than 3. To simplify the figure, we intentionally neglected the processing time required to obtain the last two streams.

Despite the fact that challenges of data streams have been identified and extensively studied in the literature, the implementation of IoT platforms capable of correctly handling unbounded observation streams is still a challenging issue. Late 2013, this statement has motivated the creation of the **Reactive Streams Initiative**[1]. The main goal of this ongoing initiative is to provide a standard for "asynchronous stream processing with non-blocking back pressure". Within this project, several working groups have been formed. They address various aspects from runtime environments to network protocols. According to this initiative, Reactive Streams have to be responsive, resilient, elastic and message-driven. The interested reader can read the *Reactive Manifesto*[2] that describes these main requirements. As for developers, this initiative has already produced Java and JavaScript Application Programming Interfaces (APIs) that may be reused to develop new IoT platforms.

III. Implementation considerations

In the following, we assume that developers want to implement a Cloud-based IoT platform following either the *Lambda architecture* [12] or the *Kappa architecture*[3]. Given the Related Work (see Section V) and from our own developer experience, we found this assumption realistic enough.

[1]http://www.reactive-streams.org
[2]http://www.reactivemanifesto.org
[3]http://kappa-architecture.com

Figure 3 shows a high-level comparison of these two software architecture patterns. The Kappa architecture can be seen as a simplification of the Lambda one [13]. Instead of maintaining two different codes for real-time and batch layers, developers may now focus on a single and unified *Stream Processing layer* to process observations. In this case, the system needs to retain the full log of observations worthy of interest. Then, a new processing job may take these historical observations and output different results according to code processing changes and consumer needs. Although the implementation of these two architectures is different, the philosophy of having two distinct serving models (real-time and offline) remains valid.

In this section, we highlight recent software solutions that we have found particularly appropriate to address observation collection (see Section III-A), observation processing (see Section III-B) and observation consumption (see Section III-C).

A. Observation collection

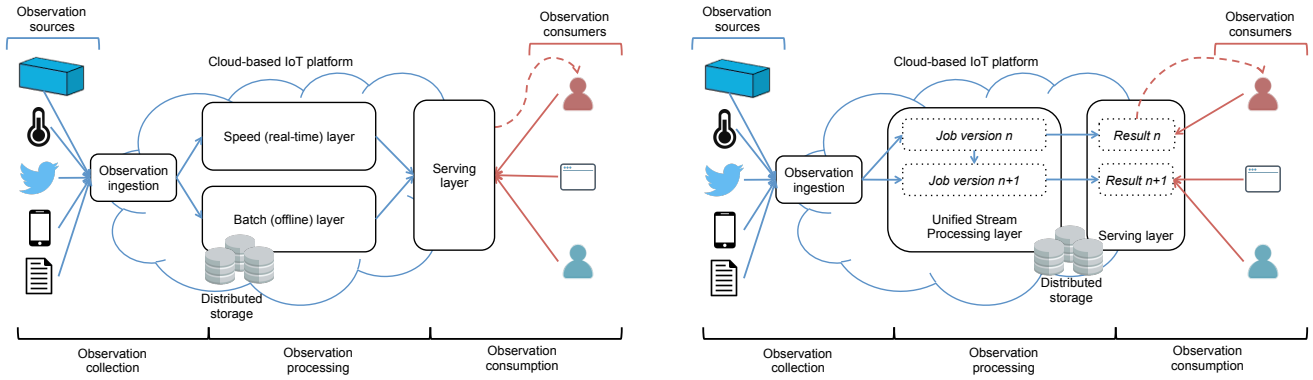
Collection corresponds to the ingestion of observations from sensors into the platform, as well as the pre-processing of these observations (by sensors or the platform, when applicable). Sensors may directly send their observations in case of direct connectivity with platform (using IoT protocols such as CoAP for instance). If required, middlewares and other gateways can also be involved at this collection phase. However, the study of these means is out of the scope of this article. Instead, this section focuses more on virtual sensors and platform-side software.

a) Virtual sensors: when developing IoT platforms, developers may want a convenient way to test the good behavior of the whole system. In particular, when assessing platform scalability (see Section IV-B), developers may not have sufficient time or resources to perform a real deployment. During the development of the iQAS platform, we addressed this issue by proposing a custom **Virtual Sensor Container (VSC)** packaged into a Docker[4] container. Using Docker virtualization, developers can quickly set up hundreds of virtual sensors that generate observations to their IoT platform.

Our VSC proposal is composed of three main components, namely a REST web server, the sensor application and an observation file (see Figure 4). For now, a VSC publishes the observations contained in the observation file (one record per line) to an URL through HTTP protocol with JSON body. VSCs are fully customizable and developers can specify sensing rate, URL to publish, etc. VSCs also expose APIs to modify their individual behavior while they are running. This may be useful in the case where the platform has some control over sensors. Wireless Sensor and Actuators Networks (WSAN) [14] are an example of such systems.

b) Platform-side software: In order to complete observation collection, it is important that developers do not reinvent the wheel and use existing solutions like Apache NiFi[5] for instance.

[4]https://www.docker.com
[5]https://nifi.apache.org



(a) Lambda-style architecture

(b) Kappa-style architecture

Fig. 3: High-level overview of two Cloud-based IoT platforms following the Lambda and Kappa architectures, respectively

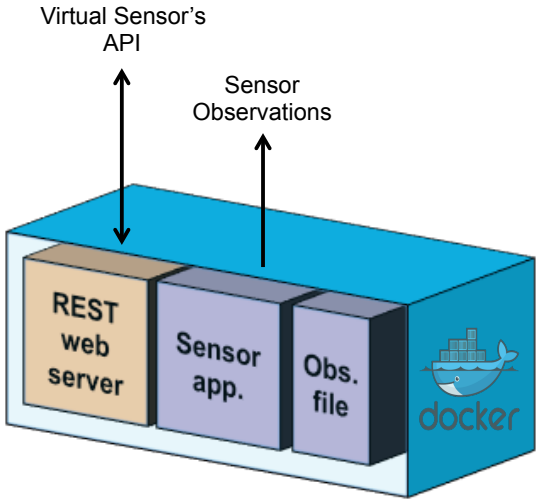


Fig. 4: High-level composition of a Virtual Sensor Container (VSC) packaged into a Docker container

NiFi is an open-source solution specifically designed to perform data ingestion at platform-side. First created by the National Security Agency (NSA), it became an open-source project in 2014 and a top-level project within the Apache foundation in 2015. This software allows developers to create real-time information *Flows* by linking *Processors* through a web-based graphical user interface. The main advantages of NiFi are the number of available processors (that allow to connect with message brokers, databases, to perform basic information processing, routing, etc.), its distributed architecture (deployment in local or with Zookeeper cluster) and the possibility to add new custom processors. However, even if Apache NiFi allows basic processing, a more powerful solution is often needed to extract value from Raw Data and provide higher observation levels (e.g., Knowledge or Wisdom) to consumers.

	Storm	Spark streaming	Samza	Flink	Kafka Streams
First release	2011	2013	2014	2015	2016
Windowing	time-based or count-based	time-based	time-based	time-based or count-based	time-based
Back-Pressure	yes	yes	yes	yes	N/A
Auto-scaling	no	yes	no	no	yes

TABLE I: Some examples of popular Apache softwares that enable Complex Event Processing (CEP)

B. *Observation processing*

A challenging feature of IoT platforms is that they may analyze, process and transform received observations according to different consumer needs. Among these needs, some of them may relate to pre-processed observations. For instance, an alert monitoring service may want to be notified only when there is a gas leak (Smart City use case). Of course, observation processing must be performed in real-time, in order not to introduce additional latency.

Most of the time, the processing of observation streams is achieved with **Complex Event Processing (CEP)** [15]. CEP help to track and analyze observation streams in order to infer occurred events. These events are then used by the platform to provide services to consumers (alerts, reporting, statistics, etc.). In practice, the term "CEP" is also used to denote Event Stream Processing (ESP).

In this section we perform a short and non-exhaustive comparison of popular CEP softwares of the Apache foundation. Table I lists five Apache solutions. In particular, it surveys the following features:

a) First release: the date of the first known version of the given software.

b) Windowing: the type of supported windowing (time-based and/or count-based).

c) Back-Pressure: this mechanism corresponds to the ability of a system to be resilient under load. A CEP that supports back-pressure is able to warn upstream components (in case of stress for instance) and it is able to adapt its production rate to the consumption rate of downstream components.

d) Auto-scaling: ability for a CEP to handle more requests, jobs or tasks. A CEP that provides this feature can scale up during peaks and scale down afterwards. This is often achieved with dynamic resources allocation or dynamic work re-balancing.

As a continuously and quickly evolving field, few CEP comparisons are publicly available. The interested reader can found a comparison of Spark, Flink and Storm in [16]. This comparison has been performed by Yahoo Storm Team in 2015 and confirms that the choice of a particular CEP software is subject to many factors such as performance but also security, integration, etc. and that there is no "clear winner at this point". From our own experience, Flink and Kafka Streams are good solutions for a developer who wants to quickly add CEP feature to its platform. They both support real-time event processing and can be either used for speed or batch processing. They are also both compliant with the Reactive Streams philosophy: Flink natively supports Back-Pressure while Kafka Streams relies on a Kafka cluster to handle processing load among instances. Lastly, these two CEP solutions provide Java clients that can take advantage of Java 8 Lambda Expressions.

C. Observation consumption

In the considered architectures, observation consumption is performed through the serving layer by consumers (either applications or users) that can ask for different kind of observations (Raw Data, Information, Knowledge or Wisdom). Incoming requests may involve real-time and/or offline processing and may change over time. Therefore, developers need to reconcile both observation production and observation consumption. Indeed, these two processes are correlated and highly dynamic.

To comply with the Reactive Streams vision, we advice developers to use **message brokers** that implement the Publish-Subscribe pattern [17]. A message broker allows developers to define several message queues (or topics) that can serve as buffers between key components of the IoT platform. Since most of message brokers are distributed (see Table II), they offer a reliable, high-throughput and low-latency observation distribution. With a message broker, sensors can asynchronously publish their observations without waiting for a consumer. When an observation consumer is interested by a given topic, it subscribes to it and start listening synchronously for messages directly from the message broker.

Table II presents three popular message brokers. In the following, we highlight some notable differences between them:

	RabbitMQ	Apache ActiveMQ	Apache Kafka
First release	2007	2012	2014
Solution based on	AMQP	JMS	N/A
Distributed	yes (cluster)	yes (Zookeeper cluster)	yes (Zookeeper cluster)
Exchange types	Queues, Topics	Queues, Topics	Topics
Routing support	yes	yes	no
Written in	Erlang	Java	Scala
Producer performance (messages/sec.)	25000	2000	50000
Consumer performance (messages/sec.)	4800	5000	22500

TABLE II: Comparison of three popular message brokers

a) Implemented protocol: RabbitMQ and ActiveMQ implements Advanced Message Queuing Protocol (AMQP) and Java Message Service (JMS), respectively. Differently, Kafka relies on the "log" data structure abstraction and does not keep track of what messages clients have consumed. Instead, within Kafka, all messages are retained during a specified time. Finally, Kafka writes/reads messages directly from disk, leveraging kernel-level input/output.

b) Distributed: all presented message brokers can be distributed to improve their scalability. For Apache brokers, it is first required to set up a Zookeeper cluster that coordinates the whole cluster.

c) Exchange types: within RabbitMQ and ActiveMQ, sensors can either publish their observations to a given message queue or to a given topic. Within Apache Kafka, there is no queue abstraction since it is assumed that sensors and producers express their interests through topics (*what* observations are about) rather than message queues (*where* observations are stored).

d) Routing support: some message brokers allow the routing of messages between their queues/topics based on a routing key. Since it does not provide queue abstraction, Apache Kafka does not provide routing either.

e) Producer and consumer performances: it is difficult to evaluate performances of a message broker since results may vary according to deployment, configuration and message benchmarks themselves. The figures presented in this paper are taken from a LinkedIn technical report dated from 2011 [18]. We estimate these figures from two comparative graphs presented in this report to give an order of magnitude of the supported loads. Again, we kindly warn the reader that these experimental results are highly dependent on software versions, configuration and used benchmarks.

On the one hand, Apache Kafka [19] seems to be a particularly reliable and scalable solution, with high-throughput delivery rate and low latency in the case of observation

streams. It is successfully used at LinkedIn to handles more than 10 billion message writes with peaks of 172000 messages per second and to deliver more than 55 billion messages [20]. On the other hand, both RabbitMQ and ActiveMQ have more features than Kafka (such as message queues and routing support for instance). In fact, there is no message broker suitable for all implementations and use cases. Besides, it is important to bear in mind that observation consumers are often considered as the bottleneck when it comes to observation consumption. Therefore, we argue that the features and the APIs provided should primarily be considered when choosing a message broker solution.

IV. Non-functional requirements

So far, this paper has shown that sensor observation streams bring new observation-centric issues that translate into concrete implementation challenges. In Section III, we made the assumption that a Cloud-based architecture was a suitable solution to deploy an IoT platform without elaborating more. In this section, we highlight three non-functional requirements for IoT platforms and we explain how Cloud Computing paradigm [21] can help to address them.

A. Platform adaptation

Cloud Computing has promoted the Everything as a Service (XaaS) model [22]. As a consequence, we have recently witnessed the birth of Sensing as a Service [23]. This model consists in taking advantage of certain features of Cloud-based platforms (pay as you go, elasticity, multi-tenancy, Service Level Agreements, etc.) while considering distinct entities and stakeholders that maintain, manage and take advantage of sensor-based platforms. Cloud-based IoT platforms fall into the Sensing as a Service model, acting as middlewares between sensors and sensor data consumers. Indeed, they are required to bridge the gap between sensor capabilities from one hand and consumer needs from another hand. In order to bridge this gap and guarantee Service Level Agreements (SLAs), IoT platforms should adapt their own behavior. This adaptation must take into account consumers' SLAs (that may include QoO aspects) but also available resources and adaptation mechanisms at platform-side.

To achieve dynamic adaptation at runtime, loop-based adaptation frameworks like the **Autonomic Computing** paradigm [24] are commonly envisioned, especially within Cloud-based systems [25]. Regarding the Autonomic Computing paradigm, it has been defined by IBM as the ability for systems to "*manage themselves given high-level objectives from administrators*" [24]. More generally, autonomic systems are a set of *Autonomic Elements*. Each of these elements is composed of one or many *Managed Elements* controlled by a single *Autonomic Manager*. The latter continuously monitors the internal state of its different *Managed Elements*; then analyzes this information; and finally takes appropriate decisions based on both its knowledge base and high-level objectives. In the end, these decisions are converted into actions and transmitted to appropriate *Managed Elements* for execution. These different steps form the MAPE-K adaptation control loop (Monitor, Analyze, Plan, Execute, Knowledge base).

In [24], IBM has identified four *self-** fundamental adaptation properties for autonomic systems (self-configuration, self-optimization, self-healing and self-protection). In the case of IoT platforms, self-optimization is critical to deliver high-quality observations to each consumer and may involve differentiated processing of observation streams (e.g., Fusion, Aggregation, Filtering) or other adaptation mechanisms (Machine Learning, Caching, etc.). Self-healing and self-protection features can also be implemented to provide better platform scalability and platform availability. We explain in more details these two requirements in following sections.

B. Platform scalability

According to the NIST definition, Cloud-based platforms are characterized by on-demand self-service, resource pooling and rapid elasticity [21]. While **scalability** denotes the capacity of a system to grow in order to accommodate a more important amount of work, **elasticity** feature refers to the ability for a scalable system to release unused resources when the workload decreases.

Many strategies exist in order to build scalable Cloud-based systems. The use of "shock absorbing" technologies (message brokers, Reactive Streams, distributed databases, load balancers, etc.) is generally sufficient to handle a small number of observation streams and build a first prototype of an IoT platform. However, when there are too many observation producers or consumers, the platform may be unable to process and deliver observations to its consumers according to the contracted SLAs. To avoid such a scenario, Cloud-based platforms may be configured to automatically provide horizontal scalability (by deploying additional virtual instances) or vertical scalability (by increasing the allocated resources per virtual instance).

The scalability of an IoT platform can be evaluated with two main scenarios. The first one is a scenario where the platform must answer to an increasing number of observation requests per second. To emulate this scenario, developers can use dedicated stress tools like the open-source load-testing framework Gatling[6] for instance. The second scenario is the one where the platform has to ingest more observations. This situation arises either when more sensors are connected to the platform or when the sensing rate of some sensors increases. This scenario may be easily emulated if developers use our Virtual Sensor Containers approach (see the description of a VSC in Section III). By taking advantage of Docker virtualization, developers may quickly configure and deploy hundreds (nay thousands) of VSCs depending on the capabilities of the Docker machine.

C. Platform availability

The availability of a system is generally represented as a fraction of time during which the system has successfully

[6] http://gatling.io

answered to consumer requests. For Cloud-based platforms, availability is commonly expressed with "Monthly Uptime Percentage". Commercial Cloud-based IoT solutions such as IBM Watson IoT platform [26] or AWS IoT [27] guarantee a Monthly Uptime Percentage greater than 99%, offering some service credit when this SLA clause is not met. However, commercial platforms only take into account the availability of the Cloud infrastructure and not the availability of sensors themselves (considered to be developer's responsibility).

Differently, within a custom-made IoT platform, we argue that platform availability is affected by both the underlying Cloud infrastructure and sensors themselves. When developing their own IoT platforms, developers should consider a broader notion of availability and implement appropriate mechanisms to handle sensor failures or sensor unavailability. For instance, one may imagine a preliminary sensor selection process based on sensor battery lifetime.

Another challenge that may impact platform availability is the **CAP theorem** [28]. This theorem states that a shared data system could only offer two of the three following features at a given time: Consistency, Availability and tolerance to network Partitions. Yet, a great majority of IoT platforms rely on distributed NoSQL databases [29] and message brokers, which are two shared data systems. Regarding distributed message brokers, most of them focus on Availability and Consistency, assuming a reliable network to synchronize and manage the whole cluster. According to the CAP theorem, one would say that these softwares sacrifice the tolerance to network Partitions feature but that it is not as simple.

In fact, since its original formulation, there have been discussions around the CAP theorem to also encompass latency [30] and precise the three different features that shared data systems should provide. First, CAP-availability is often impossible to achieve but High-Availability (HA) may be sufficient enough for many SLA-based platforms in practice. In this case, a short unavailability is acceptable but systematic high-latency responses to requests may be considered as platform unavailability. Second, CAP-consistency refers to linearizable consistency, which is very costly and most of the time not even needed. Last but not the least, no network is 100% reliable and network Partitions happen more than expected in reality, even in the context of a Cloud-based IoT platform. Even worse, network Partitions may be caused by a multitude of factors like hardware problems, wrong configurations, software bugs, etc. Even if the CAP theorem describes a very specific read-write use case and relies on very strong definitions that may not be applicable in reality, developers must be aware of the implications of using shared data systems since they may impact the availability of the whole IoT platform.

V. Related work

Data streams [31] have been extensively studied in the literature, including from the perspective of the Internet of Things (IoT) [3]. Instead, this paper envisions sensor observation streams, which can be seen as a particular type of data streams. As a consequence, this restriction also raises specific research challenges linked to the Quality of Information (QoI) [9]. To address this issue and characterize sensor observations, this paper proposes to reuse existing work on QoI and Context information.

Previously, some efforts have been done to standardize either sensor-based platforms or IoT-based architectures in general. For example, OGC SWE standards [32] are intended for Sensor Web systems while the IoT-A project [33] is destined to the IoT. However, the use of such frameworks remains very limited since they require an important learning phase. As a consequence, we witness the development of a great number of custom-made IoT platforms. Regarding implementation considerations, a lot of studies have highlighted the benefits of using Cloud-based architectures to support the IoT [34]. Some of them have even proposed the term "Cloud of Things" [35] to denote this new paradigm. However, most of these studies only focus on technical considerations (by proposing protocols or integration frameworks for instance) and do not take into account observation streams and their inherent challenges. Some Cloud-based commercial IoT solutions like AWS IoT [27] or IBM Watson IoT [26] are also available to developers. Nevertheless, these proprietary solutions are application-oriented and do not provide any QoO insights. Besides, since these platforms are not open-source, developers can only customize them with available proprietary components.

Some platforms have successfully been designed and deployed to handle sensor observation streams [36], providing Cloud-based query processing [37] or QoI characterization for observations [38]. However, these solutions are of little interest for developers since they do not detail how they specifically address challenges (and which ones) related to unbounded observation streams. Therefore, developers cannot learn from these projects and reuse these teachings at design phase.

Finally, closer to our work, Elnahrawy reviewed the different research directions for sensor data streams [39]. Although it presents interesting challenges and solutions, this work focuses on low-level observation collection (sensor gateways, in-network aggregation) and does not consider Cloud-based platforms. On the contrary, our paper aims to present some lessons learned and recommendations from a custom implementation of a concrete Cloud-based IoT platform able to handle sensor streams with QoO support. Different from a simple software review, this paper explains some important concepts inherent to sensor streams (Reactive Streams, Lambda and Kappa Architecture, CAP theorem) and presents some open-source solutions that may help to address them.

VI. Conclusions and perspectives

Far from being exhaustive, this paper aims to help developers in the design and the implementation of Cloud-based IoT platforms capable of handling sensor observation streams. It puts into perspective observation-related challenges with some implementation considerations. Indeed, from collection to consumption through processing, observation streams require suitable softwares or paradigms. By correctly addressing these

challenges from design phase, developers may be able to build IoT platforms that meet important non-functional requirements like platform adaptation, scalability and availability.

With the growth of Internet of Things (IoT) and the emergence of Smart Cities, we believe that observation-related challenges will become more and more critical within these information-centric platforms. Just like network Quality of Service (QoS), we envision that more and more consumers will require guarantees about Quality of Observation (QoO).

We hope that this paper will stimulate the creation of new Cloud-based IoT platforms which collect, extract Knowledge or Wisdom and provide enhanced services to final consumers. We also expect that more IoT-related softwares will be released in the future, providing new tools to developers to deal with sensor observation streams.

ACKNOWLEDGEMENT

This research was supported in part by the French Ministry of Defense through financial support of the Direction Générale de l'Armement (DGA).

REFERENCES

[1] Dave Evans, "The Internet of Things - How the Next Evolution of the Internet Is Changing Everything," Apr. 2011. [Online]. Available: http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf

[2] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.

[3] C. C. Aggarwal, N. Ashish, and A. P. Sheth, *The Internet of Things: A Survey from the Data-Centric Perspective.*, 2013.

[4] A. Auger, E. Exposito, and E. Lochin, "iQAS: An Integration Platform for QoI Assessment as a Service for Smart Cities," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT) (WF-IoT 2016)*, Reston, USA, Dec. 2016, pp. 88–93.

[5] A. Sheth, "Internet of Things to Smart IoT Through Semantic, Cognitive, and Perceptual Computing," *IEEE Intelligent Systems*, vol. 31, no. 2, pp. 108–112, Mar. 2016.

[6] A. K. Dey, "Understanding and using context," *Personal and ubiquitous computing*, vol. 5, no. 1, pp. 4–7, 2001.

[7] M. Compton, P. Barnaghi, L. Bermudez, R. GarcíA-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, and A. Herzog, "The SSN ontology of the W3c semantic sensor network incubator group," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 17, pp. 25–32, 2012.

[8] P. Barnaghi, M. Bermudez-Edo, and R. Tönjes, "Challenges for Quality of Data in Smart Cities," *J. Data and Information Quality*, vol. 6, no. 2-3, pp. 6:1–6:4, 2015.

[9] C. Bisdikian, L. M. Kaplan, and M. B. Srivastava, "On the Quality and Value of Information in Sensor Networks," *ACM Trans. Sen. Netw.*, vol. 9, no. 4, pp. 48:1–48:26, 2013.

[10] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context Aware Computing for The Internet of Things: A Survey," *IEEE Communications Surveys Tutorials*, vol. 16, no. 1, pp. 414–454, 2014.

[11] L. Golab and M. T. Özsu, "Issues in Data Stream Management," *SIGMOD Rec.*, vol. 32, no. 2, pp. 5–14, 2003.

[12] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[13] J. Kreps, "Questioning the lambda architecture," *OReilly Online*, Jul. 2014. [Online]. Available: https://www.oreilly.com/ideas/questioning-the-lambda-architecture

[14] R. Verdone, D. Dardari, G. Mazzini, and A. Conti, *Wireless sensor and actuator networks: technologies, analysis and design*. Academic Press, 2010.

[15] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 407–418.

[16] S. Chintapalli, D. Dagit, B. Evans, and others, "Benchmarking Streaming Computation Engines at Yahoo!" Tech. Rep., 2015. [Online]. Available: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

[17] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[18] J. Kreps, N. Narkhede, J. Rao, and others, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[19] N. Garg, *Apache Kafka*. Packt Publishing Ltd, 2013.

[20] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Y. Ye, "Building LinkedIn's Real-time Activity Data Pipeline." *IEEE Data Eng. Bull.*, vol. 35, no. 2, pp. 33–45, 2012.

[21] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[22] P. Banerjee, R. Friedrich, C. Bash, P. Goldsack, B. Huberman, J. Manley, C. Patel, P. Ranganathan, and A. Veitch, "Everything as a service: Powering the new information economy," *Computer*, no. 3, pp. 36–43, 2011.

[23] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Sensing as a Service Model for Smart Cities supported by Internet of Things," *Transactions on Emerging Telecommunications Technologies*, vol. 25, no. 1, pp. 81–93, 2014.

[24] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[25] M. Maurer, I. Brandic, and R. Sakellariou, "Adaptive resource configuration for Cloud infrastructure management," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 472–487, 2013.

[26] IBM, "Watson IoT." [Online]. Available: https://internetofthings.ibmcloud.com

[27] Amazon Web Services, "AWS IoT." [Online]. Available: https://aws.amazon.com/iot

[28] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[29] J. Han, E. Haihong, G. Le, and J. Du, "Survey on NoSQL database," in *Pervasive computing and applications (ICPCA), 2011 6th international conference on*. IEEE, 2011, pp. 363–366.

[30] E. Brewer, "CAP twelve years later: How the" rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[31] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2000, pp. 71–80.

[32] A. Bröring, J. Echterhoff, S. Jirka, I. Simonis, T. Everding, C. Stasch, S. Liang, and R. Lemmens, "New generation sensor web enablement," *Sensors*, vol. 11, no. 3, pp. 2652–2699, 2011.

[33] EU FP7, "IOT-A: Internet of Things Architecture." [Online]. Available: http://www.iot-a.eu/public

[34] A. Botta, W. d. Donato, V. Persico, and A. Pescapé, "On the Integration of Cloud Computing and Internet of Things," in *2014 International Conference on Future Internet of Things and Cloud (FiCloud)*, 2014, pp. 23–30.

[35] S. Distefano, G. Merlino, and A. Puliafito, "Enabling the Cloud of Things," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*. IEEE, 2012, pp. 858–863.

[36] L. Luo, A. Kansal, S. Nath, and F. Zhao, "Sharing and Exploring Sensor Streams over Geocentric Interfaces," in *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. GIS '08. New York, NY, USA: ACM, 2008, pp. 3:1–3:10.

[37] H. N. M. Quoc and D. Le Phuoc, "An Elastic and Scalable Spatiotemporal Query Processing for Linked Sensor Data," in *Proceedings of the 11th International Conference on Semantic Systems*, ser. SEMANTICS '15. New York, NY, USA: ACM, 2015, pp. 17–24.

[38] D. Puiu, P. Barnaghi, R. Tönjes, D. Kümper, M. I. Ali, A. Mileo, J. X. Parreira, M. Fischer, S. Kolozali, N. Farajidavar, F. Gao, T. Iggena, T. L. Pham, C. S. Nechifor, D. Puschmann, and J. Fernandes, "CityPulse: Large Scale Data Analytics Framework for Smart Cities," *IEEE Access*, vol. 4, pp. 1086–1108, 2016.

[39] E. Elnahrawy, "Research directions in sensor data streams: solutions and challenges," *Rutgers University, Tech. Rep. DCIS-TR-527*, vol. 2, p. D3, 2003.