# Composing Continuous Services in a CoAP-based IoT

Benjamin Billet, Valerie Issarny, Géraldine Texier

# Composing Continuous Services in a CoAP-based IoT

Benjamin Billet*, Valérie Issarny*†, Géraldine Texier‡†
*Inria, France; first.lastname@inria.fr
†Inria@SiliconValley
‡IRISA/IMT-Atlantique, France; first.lastname@imt-atlantique.fr

***Abstract*—We revisit core Service-Oriented Architecture –SOA– concepts to integrate advanced continuous processing techniques. The resulting architecture allows overcoming the high heterogeneity and the very large data scale that arise in the Internet of Things. Further, we discuss how to leverage the CoAP protocol so as to enable the resource-efficient deployment of the proposed architecture over the large diversity of sensor networks.**

***Index Terms*—IoT; middleware; streaming**

## I. Introduction

The Internet of Things (IoT) vision advocates the blending of the technology into the environment so that human beings interact with the physical world as simply as they do with the virtual world. In practice, the environment and the underlying everyday objects (the *Things*) are massively enriched with sensors and actuators, thanks to the continuous improvements of miniaturization techniques. Using a worldwide machine-to-machine network, the Things can cooperate autonomously and collaborate with users through high-level interfaces. With the IoT, various use cases emerge in several areas, such as transportation, health and energy management. Still, actually leveraging the IoT within applications is challenged by: (i) the software and hardware heterogeneity of sensors and actuators, as well as of their communication networks, and (ii) the high volume of data continuously produced over time by a large number of sensors [1].

The IoT literature has suggested that *Service-Oriented Architectures* (SOA) are promising solutions for alleviating the Things' heterogeneity, thanks to the provided interoperable discovery and access mechanisms [2]. In a nutshell, a service is a well-defined self-contained functionality offered by a service provider, or host. A service consumer (human being or device) can discover which services are available by requesting a dedicated repository that references the hosts and the services they provide. Further, a SOA provides to the service consumer an abstract way to invoke the service without knowing the technical details related to the host and the execution environment [3]. In the IoT context, a SOA abstracts the Things as service providers for sensing and actuation, hiding the actual hardware, software and communication subtleties [2]. A good illustration of a SOA for the IoT is the *Web of Things* (WoT) [4] that is based on RESTful services. Through the coupling with the Web, the IoT is abstracted as a large network of resources.

Given that every Thing embeds sensors, the amount of data produced in the IoT is very high in practice. A solution to address this challenge is to leverage the continuous processing techniques introduced by *Data Stream Management Systems* (DSMS). DSMSes play a similar role as *DataBase Management Systems* (DBMS) but process data streams instead of finite sets of data. DSMSes typically support use cases where (i) the amount of data is too high for storage, (ii) the amount of data is ever-growing and requires a single-pass processing, and (iii) the processing system must adapt to strong variations in the processing load and resource availability. Processing each piece of data one after another and forgetting them once processed, as in DSMSes, is a satisfactory approach for various IoT scenarios, where the actual sensor measurements are of limited value. The users are more interested in high-level meaningful information, while the sensors typically produce simple low-level measurements that must be processed (e.g., filtered, aggregated or correlated) in order to be useful. In addition, the value of the information is decreasing over time, i.e., the usefulness of oldest measurements decays compared to the most recent ones (e.g., temperature in a given area). Finally, the sensors produce time-dependent data, typically samples (periodic measurements) or events related to physical resources and phenomena. Data streams are convenient representation for such information.

Following, designing a *Continuous Service-Oriented Architecture* (CSOA) that blends the SOA concepts with continuous processing is beneficial for the IoT. A CSOA features software components that process data streams continuously, while separating their actual execution from the data access interfaces, in the same way SOA separates the service implementation from the service access. At the Thing level, Things become continuous service providers. At the infrastructure level, additional providers support services for storage or computation offloading (e.g., fog computing) [5]. Beyond the IoT, a CSOA is useful for any service-oriented scenario where (i) data are time-dependent and permanently produced and/or (ii) the amount of data to process is too high to be stored. However, bringing continuous processing to SOA is not

trivial and requires revisiting the SOA concepts. A SOA is designed for processing finite sets of data: when invoked, a service acquires all its input data, performs a computation and, finally, produces a result that is sent back to the service consumer. When the inputs are data streams, the service reads one item from the inputs, processes this item and produces a result, each result being part of a result stream. In addition, the service providers must manage the stream behavior over a long period of time, especially when sudden variations of the production rate occur (e.g., a data burst following the detection of an event). In scenarios involving a large amount of data, a CSOA must provide tools for simplifying the dynamic adaption of the processing network, e.g., using delegation, migration and distribution mechanisms. In the context of our research, we have investigated how to adapt SOA concepts to meet the specifics of continuous stream processing within IoT networks.

This paper summarizes the concepts of the resulting CSOA in relation with service management: interface descriptions, instantiation, deployment, and composition (Section II). We then focus on the challenge of deploying CSOA over today's and even next generation IoT that builds upon heterogeneous networking technologies. We more specifically present how CSOA may conveniently leverage the IETF CoAP protocol for accommodating highly diverse and resource-constrained Things (Section III). We then illustrate CSOA in action, in the context of a mobile app that connects with Things to provide runners with environment-aware guidance for the sake of well-being (Section IV). Finally, we draw some conclusions with area for future work (Section V).

## II. From Discrete to Continuous Services

The fundamental concept of SOA is the service [3], to which we refer as *discrete service*. A discrete service is a software component that provides a set of *discrete operations*, i.e., functions for processing finite sets of data. When invoked, a discrete operation processes the finite sets received as inputs and produces new finite sets as results. The service consumer initiates the invocation by sending a message to the service provider, which performs the processing and sends back a message containing the results. In practice, this exchange is managed by a messaging protocol (e.g., SOAP).

The fundamental concept of a CSOA is the *continuous service*. A continuous service is a software component that consumes a set of *data streams* as inputs and produces a set of data streams as outputs. For that purpose, a continuous service provides a set of *input ports* and a set of *output ports*. An input port allows connecting streams to the continuous service; each stream is continuously processed by the service to produce new results. An output port enables service consumers to access these results while they are produced, as a stream. When a piece of data, or *stream item*, is available on an input port, one or more *continuous operations* are triggered to process the item. These operations produce new items (results) which are available to the service consumers through the relevant output ports. In practice, a *streaming protocol* transfers the items, hence managing the flow of items from service providers to service consumers over time.

### A. Representing service interfaces

The operations provided by a discrete service define the service's *interface*. In order to reason about services, a *contract* formally defines the interface, specifying the provided operations, the inputs-outputs of each operation and the associated properties (e.g., types, QoS).

Following, the ports provided by a continuous service define the service's *interface*. The *contract* specifies the *schemas* of: (i) the streams readable by each input port and (ii) the streams produced by each output port. Given a contract, the service consumer can reason about the operations of the continuous service and the streams that each operation processes and produces. In addition, the contract enables checking that streams conform to the required schemas when they are connected to input ports. Concretely, the *stream schema* describes the *structure* and the *behavior* of the stream.

We represent the structure of streams as in DSMSes that tend to describe streams as sequences of tuples. The schema then defines the number of attributes of the tuples and a set of constraints for each attribute (type, domain, etc.). These constraints strongly depend on the target scenarios. For example, in our IoT research, attributes are constrained by: (i) a semantic type (e.g., temperature), which is used to reason about the physical phenomena, (ii) a unit (e.g., kelvin), for automated conversion, and (iii) a concrete type (e.g., integer).

Various factors influence the stream behavior, that is, the production and the transport of the data. Therefore, the schema stores: (i) design-time properties of the stream (e.g., sample-oriented/event-oriented, sampling frequency), (ii) dynamic properties measured on the stream (e.g., production rate, sensing errors) and (iii) QoS requirements that must be satisfied in order to carry the stream (e.g., ordered/unordered stream, percentage of items lost, latency).

### B. Service instantiation

*Instantiation* refers to the deployment of a service on a service provider. In a SOA, a discrete service instance is associated to a unique address (e.g., URI) that the consumers use to invoke the service's operations. In practice, service providers can instantiate the same service more than once, each instance being identified by a different address. Several consumers can invoke a service instance any number of times: each execution is isolated, but some services maintain states (stateful services) that can have an impact on the next results.

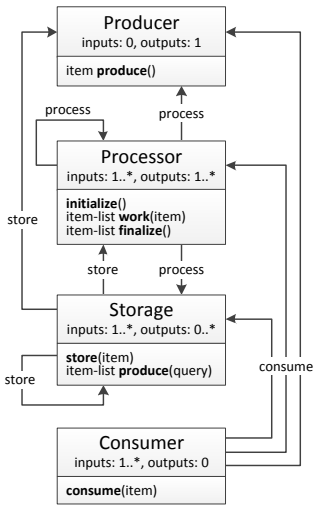



Figure 1: The four-roles model for continuous services

The instantiation of a continuous service also leads to the assignment of an address. However, unlike a discrete service where the instantiation and the execution are two well-separated actions, a continuous service starts processing its inputs right after the instantiation. Consequently, the continuous service is given the set of streams to connect to its input ports at instantiation time. Given the unbounded nature of streams, the continuous service instance is strongly tied to this set of streams. However, a continuous service can be instantiated any number of times with different input streams, provided that each instance has its own address.

### C. Specializing continuous services for the IoT

SOA does not make any assumption about the type of operations implemented by a service. However, in the context of the IoT, one can observe that Things can assume a recurring set of high-level roles. For example, a continuous service may read a sensor and present the readings as a stream, while another may simply store and retransmit streams.

In order to facilitate the development of IoT-based applications, we introduce four types of continuous services, which come along with specific interfaces, life cycles and interaction patterns (see Figure 1):

- A *producer* exposes data sources (e.g., sensors, databases, Web services) as streams, acting as an entry point for external data that are made available to other continuous services.
- A *processor* consumes and processes streams in order to produce new streams.
- A *storage* reads streams and stores their items – or a subset of them– temporarily or permanently. Other continuous services can query the storage to retransmit the stored items as new streams.
- A *consumer* acquires streams in order to drive actuators or fill user interfaces. It acts as exit point for the data produced by other continuous services.

According to Figure 1, each of the above service types specifies a minimal set of operations for the service contract. The developers must implement these operations in order to create continuous services of a given type.

The four service types together support complex applications involving a wide variety of entities (sensors, actuators, servers, users, services, etc.). For example, end-users, GUI and actuators can be abstracted as consumers while sensors, databases, crowd-sensors and any other type of data source (e.g., a Web service that gives information about the weather) can be abstracted as producers. Similarly, memory, filesystems and cloud storages can be abstracted as storages, making them able to store and forward stream items on demand. Finally, processors may implement any type of continuous computation.

### D. Dynamic deployment of continuous services

Service deployment in a SOA can be static or dynamic. In the *static* case, services –qualified as *built-in services*– are deployed *a priori* on the service providers; new services cannot be deployed *a posteriori*. In the *dynamic* case, new services –qualified as *dynamic services*– can be deployed at any time according to the application needs. To do so, service providers embed deployment services for uploading: (i) the executable code (interpreted or binary) of new operations and (ii) the corresponding service contract.

Dynamic services are useful for stream processing, especially for the dynamic allocation of resources when the number of streams and their data rate dramatically increase (i.e., services can be dynamically moved or replicated across the available devices). In the IoT context, dynamic services make the Things reusable for various use cases, making unanticipated scenarios easier to implement in the future. In addition, dynamic services can serve the purpose of autonomous networks, where a Thing can participate to distributed computations initiated by other Things (delegation of processing *aka* surrogate computing).

DSMSes often introduce dedicated *Stream Processing Languages* (SPL). A SPL allows expressing requests that combine continuous processing operators (declarative language), and implementing new operators. A SPL provides the developer with primitive types, program structures, definitions and tools that are tailored for stream processing. Interestingly, DSMSes already embed interpretors or compilers for the SPLs they support, making these SPLs a relevant foundation for implementing and executing dynamic continuous services.

However, in the IoT context, hardware resources can be strongly limited. We thus have developed a new lightweight SPL, called *DiSPL* [6]. DiSPL enables efficient parsing and interpretation even on resource-constrained

devices. For that purpose, a DiSPL program is compiled into a restricted bytecode, which is executed by a lightweight virtual machine installed on the Things. As a benefit, any Thing (i.e., small devices, smartphones, computers, etc.) can host dynamic services and thus participate to various scenarios.

Practically, DiSPL is a general-purpose language with stream processing capabilities. DiSPL is inspired by the Scheme language [7], enabling the developers to describe various complex operations, while providing data types and programming primitives for stream processing specific to our CSOA. The capabilities of the DiSPL language are illustrated through the use case presented in Section IV.

### E. Composing continuous services

A major feature of a SOA is supporting the composition of the operations of existing services in order to create *composite services* [3]. Contracts describe the inputs and the outputs of service operations and enable the specification of a workflow of service invocations that represents the logic of a new more complex operation. In practice, the execution of composite services can be either centralized or distributed. The former approach –called *service orchestration*– introduces an orchestrator to manage the invocations of the underlying services. The latter approach –called *service choreography*– leverages negotiation and routing mechanisms to let the service providers manage the composite services autonomously [8].

Continuous services may be composed in the same way as discrete services to create more complex tasks and applications. A *continuous composite service* is then a set of continuous operations applied to a set of streams: the data to process are injected in the composition by producers and flow through processors and storages until reaching the consumers. Formally, the flow of data between the continuous services naturally forms an acyclic directed graph, called a *logical graph*, with vertices being continuous services and edges being streams.

The execution of a logical graph consists in: (i) instantiating the continuous services and (ii) connecting their ports according to the graph edges. To this end, the service providers executing the continuous services are selected according to a set of scenario-dependent properties to minimize/maximize (e.g., throughput, energy consumption). This problem is a variation of the *task mapping problem*, where a set of communicating tasks with several properties (constraints, resource consumption, etc.) must be mapped to a set of connected nodes given their characteristics (location, hardware capabilities, etc.). Its solutions span centralized and distributed approaches. The centralized approach computes and executes an allocation plan on a single machine (similar to service orchestration). The distributed approach lets the nodes compute parts of the allocation plan based on the knowledge they have about their peers (similar to service choreography). In our work, we studied this problem for stream processing in the IoT, leading to introduce a dedicated centralized solver in [9].

## III. Mapping CSOA on the IoT Network

Abstracting Things as providers of continuous services that are dynamically deployable and easy to compose, as promoted by CSOA, paves the way for advanced applications in various domains (e.g., smart cities, industry 4.0, precision medicine, citizen science), thanks to the increasing diversity of IoT networks willing to share data.

Existing IoT networks can be classified regarding the following criteria: (i) traffic pattern (high or lower data rate), (ii) communication range, from centimeters (short-range) to several kilometers (long-range), and (iii) power supply, from continuously powered to battery-powered. As an example, a smartphone typically embeds a set of sensors that can transmit data using long-range communication (e.g., at countryside scale) technologies, such as 3G, LTE, or even 2G. However, these solutions are not appropriate for monitoring areas that are out of mobile coverage, or for deploying lightweight sensing infrastructures where sensors with limited energy or short-range communication are sufficient (e.g., WBAN - Wireless Body Area Networks). Short-range protocols are specifically designed for battery-powered sensors, such as Zigbee, Bluetooth Low Energy (BLE), or the recent improvements of the WiFi protocol (HaLow). In addition, Low-Power Wide Area Networks (LPWAN) [10] technologies (e.g., SIGFOX, LoRa, WI-SUN and NB-IOT) provide solutions for long-range scenarios, as illustrated in [11].

As a consequence, sensors are deployed for different purposes, by different actors, using various technologies. In practice, implementing the CSOA over such a rich networking environment requires to tame the underlying heterogeneity. In that direction, we developed Dioptase, a CSOA-based stream processing middleware for the IoT [6], that provides an implementation of the DiSPL virtual machine. Dioptase enables Things with heterogeneous capabilities (computation resources, energy, etc.) to host dynamic continuous services and to participate to large-scale continuous processing tasks. Dioptase can be used to develop discrete and continuous services, making the developers able to deal with discrete data (e.g., access to service contracts and various metadata) and/or data streams, depending on the use case. For this purpose, Dioptase provides various messaging and streaming protocols as connectors between services. In practice, developers implement services in a protocol-agnostic manner and then select a set of relevant connectors at deployment-time, depending on the targeted Things. For example, a developer can choose to use Web hooks [4] connector on a smartphone or a CoAP connector for resource-constrained devices [12] (see Figure 2).

### A. CoAP meets resource-constrained continuous services

In 2010, the Internet Engineering Task Force (IETF) created the Constrained RESTful Environments (CoRE)

working group to design the Constrained Application Protocol (CoAP) [12] to implement communication with Things in a RESTful way. CoAP has been designed for constrained environments and for equipments having strong limitation in energy and/or capacity. Very similar to HTTP, CoAP implements client/server interactions so that a CoAP client collects data from the CoAP server (e.g., a sensor) by simply sending a GET request and receiving a response from the sensor containing its current state.

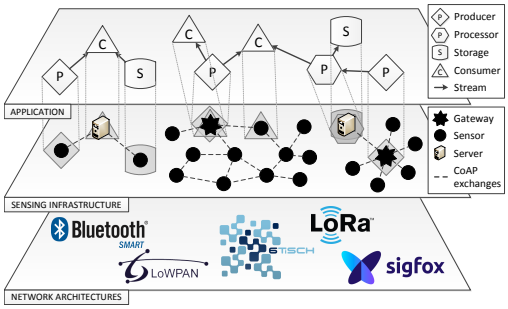



Figure 2: CSOA over IoT networks using CoAP

The CoAP protocol was first designed to be used in a pull mode, enabling a client to periodically request the state of the sensor. Therefore, getting the sensor's state means that the sensor must process a request message and reply with a response message, which is not efficient regarding computation and energy consumption (e.g., if the state has not changed between two polls, the interaction is useless). As an alternative to the periodic reception of the request message, CoAP has been enriched with a resource observation mode. In such a mode, the CoAP client is able to register with a sensor as an observer and will be notified when the sensor's state is updated. More precisely, the CoAP observer mode follows the observer design pattern [13], as specified in RFC 7641 [14]. Practically, the CoAP client registers as an observer with a CoAP server by sending a GET command including an *Observe option*. Once registered, the client will be notified of when the resource state is updated, until it de-registers.

The registration of the observer prevents the server from receiving future polling messages (they were traditionally sent by the client to receive the CoAP server's state). This alleviates the performance issues arising for resource-constrained servers, but introduces a new problem: the protocol must now provide an ordering mechanism for the notification messages. To do so, the server adds a strictly increasing sequence number in the notification of its state. The overhead of this mechanism involves the cost of adding the observe option in the CoAP messages (maximum size of 4 bytes) and the storage and maintenance of a list of observers by the server.

Complementary to RFC 7641, the IETF draft in [15] defines a protocol to enable dynamic resource linking for constrained RESTful environments. The draft details the exchanges between the CoAP client (the producer) and the CoAP server (the sensor) to dynamically specify the observation parameters such as the minimum, maximum period or data changes that trigger the observation. After the registration, the producer will be notified of the sensor's state and will produce a new stream item on the corresponding output port. This implies that the sensor should have enough battery, storage and processing capabilities to be able to implement a CoAP server. If the sensor is too constrained, a proxy is required to play the role of a CoAP server. Depending on the network architecture used to gather data from the sensors, the proxy can be, for example, a server storing sensed data in the cloud but also an architectural element having enhanced capacities such as a gateway for a wireless sensor network. In such cases, the proxy caches the sensor's state, acting as the origin CoAP server. Using a proxy not only relieves the sensor from forwarding multiple times its state, it enables also to answer when the sensor is sleeping or to control the access to the sensor. Although implementing a proxy serving several clients observing a constrained resource is not a trivial problem (as shown by Tanganelli *et al.* [16]), it addresses particularly well the case of a sensor network managed by a third-party operator which delivers a sensing service to customers (e.g., see Figure 2).

### *B. Secure access*

The access to sensing infrastructures raises security considerations, especially when dealing with constrained Things. Transport Layer Security (TLS) is the protocol traditionally used to secure communication with authentication and ciphering, but it is implemented over TCP contrary to CoAP that is implemented over UDP. The security of CoAP exchanges is addressed by the Datagram Transport Layer Security (DTLS) protocol, i.e., the adaptation of TLS that deals with UDP's message losses or desequenced delivery. The RFC 6347 [17] defining DTLS standardizes four levels of security (including the absence of security) for CoAP. The RFC also describes the exchanges between the CoAP Client (also the DTLS client) and the CoAP server (DTLS Server) to initiate a session with security parameters exchange and handshake before issuing CoAP communication. Bootstrapping with a third-party operated infrastructure or with a proxy is a different challenge when security parameters or credentials need to be gathered. Garcia *et al.* [18] provide a detailed view of the current proposals to deliver *Authentication* and *Authorization* mechanisms for the IoT and present a CoAP-based bootstrapping solution using the Extensible Authentication Protocol (EAP). Nevertheless, securing the IoT and the CoAP protocol is still an ongoing topic. Interesting work is currently done in the CoRE working group to reconcile the need of security with the constrained resources of IoT devices. In particular, the group is defining an Object Security of CoAP (OSCOAP) [19] compatible with the observation mode to address the

protection of CoAP exchanges when intermediate nodes (e.g., proxies) are involved in the communication.

## IV. CSOA in Action

We illustrate CSOA and the related added value of leveraging today's IoT networks using the example of a mobile application for runners. A study published in [20], performed on marathon runners of all levels, argues that air temperature is the most important factor affecting runners performance. They show that the variation of the air temperature leads to reduce the running speed and that pollution has an impact, mainly when linked with air temperature. We then present a mobile application that assists runners in their journeys by providing them with advices about the running route and their pace based on information about the pollution and the weather along the route as well as about physiological data.

Examples of advices delivered to runners about their intended journey include: Whether the weather expected during the run will be good enough; Whether the atmospheric pollution is low enough to run safely or slowing the pace should be considered and/or considering an alternate route to avoid some unsafe area; Whether the weather requires to slow down as, e.g., the temperature may affect the runner's physical condition; etc.

Implementing the above service for runners requires gathering information from several sources and especially sensors providing observations about: pollution, weather, location, activity, physiology (see Figure 3). Interestingly, the use-case illustrates the heterogeneity of the sensors, their provider and how they are accessed. If some of the observations are produced by sensors embedded in Things belonging to the runners (smartphone, smartwatch or clothing), others are delivered by third-party sensors. For example, temperature, humidity, wind speed and barometric pressure can be measured by individual weather stations in the city or can be collected from weather forecast Web services. Furthermore, ozone, particulate matter, carbon monoxide, sulfur dioxide, and nitrous oxide can be measured by pollution sensors deployed in an infrastructure operated by the city or by an independent provider. Monitoring the exposure to environmental pollution may even benefit from the aggregation of these diverse sources as illustrated by the Ambiciti[1] platform that combines numerical simulations, quantitative observations of fixed and mobile sensors, and qualitative observations provided by citizens.

### A. The CSOA-based runner application

Figure 4 illustrates the CSOA-based implementation of the runner application, although focusing on three relevant sensors, for the sake of conciseness: (i) a particulate matter -$PM_{10}$- sensor monitoring the running area, (ii) a temperature sensor provided by a neighboring weather station and (iii) a wearable pulse sensor.

[1] http://ambiciti.io/

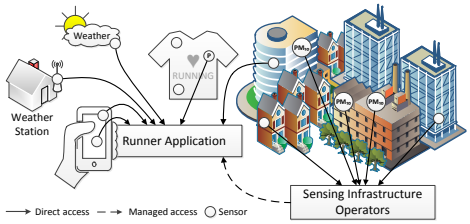



Figure 3: Use-case – the runner application

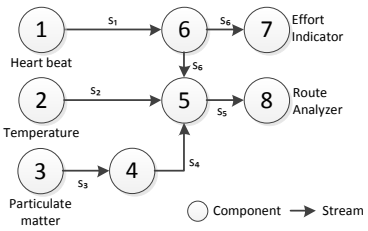
The $PM_{10}$ sensors are CoAP observable resources that are monitored by *producers* (③ on the figure) that also are responsible of computing the average $PM_{10}$ value per one-hour window. The corresponding DiSPL implementation of the $PM_{10}$ producer is given in the following snippet:

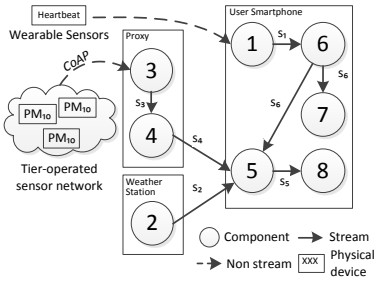
```
; === DiSPL header ===
(archetype "producer")
; contract declaration: one output port for streams of PM10 values
(contract (name "PM10-producer")
  (output-ports
    (port (name "PM10-per-hour")
      (output-rule (type "schema")
        (attribute (name "PM10")
          (semantic-type "particulate-matter")
          (unit "μg/m^3")
          (concrete-type "integer"))))))
; === DiSPL operations ===
(define hour_avg 0) (define start_time 0) (define nb_items 0)
produce:
  ; read the virtual sensor
  (define PM10 (read-sensor "PM10-sensor"))
    (set! hour_avg (/ (+ (* avg nb_items) PM10) (inc nb_items)))
  ; check if the time window is closed
    (when ((- (now) start_time) > 60000)
    ((set! start_time (now))
        (define PM10 (item (now) ("PM10" hour_avg)))
        (write (get-port "PM10") PM10)
    (set! hour_avg 0))
```

The $PM_{10}$ measurements are then collected by a processor (④ on the figure) that is deployed on the runner's smartphone. The processor averages the measurements of several $PM_{10}$ sensors located within the area of interest, as specified by the runner. The processor implementation is as follows:

```
(archetype "processor")
; contract declaration: one input port for streams of PM10 values,
; one output port for streams of aggregated PM10 values
(contract (name "PM10-aggregator")
  (input-ports
    (port (name "PM10") (min 1) (max infinity)))
    ; the output port produces streams following the same
    ; schema as the input port
    (output-ports (port (name "aggregated-PM10")
      (output-rule (type "identity") (port "PM10")))))
; === DiSPL operations ===
(define average 0) (define nb_items 0)
work:
  ; read the items available on each stream
  (define stream_items (next-items "PM10"))
  ; the "PM10" attribute value of each item are averaged
  (define avg 0) (define len (count stream_items))
    (map (lambda (stream_item)
      ((set! pm10 (get "PM10" stream_item))
      (set! avg (/ (+ (* avg nb_items) pm10 (inc nb_items)))))
  (write (stdout) (item (now) ("PM10" avg))))
```

The aggregated results are further combined with the temperature values and the heart beat measurements (from sensors ② and ① in the figure). According to




| | |
|---|---|
| ① Producer | Reads the remote pulse sensor every $x$ seconds. |
| ② Producer | Reads the temperature sensor every $x$ seconds. |
| ③ Producer | Reads managed $PM_{10}$ sensors every $x$ seconds and averages the one-hour values. |
| ④ Processor | Computes the average of the $PM_{10}$ measurements for each area monitored. |
| ⑤ Processor | Aggregates the $PM_{10}$ maximum values, the temperature values and the heart beat measurements and compare them to past measurements, in order to find the area that should be avoided by the runner. |
| ⑥ Storage | Stores the results of ①. |
| ⑦ Consumer | Displays heart beat statistics to the runner. |
| ⑧ Consumer | Displays notifications based on the results of ⑤. |

Figure 4: Logical (Top-Left) and physical (Top-Right) service composition for the runner application

previous results (available from storage ⑥ and displayed using consumer ⑦ in the figure) and current observations, the application infers how the user is impacted by ambient pollution and thus determines the areas to avoid (processor ⑤ in the figure). If the runner should avoid an area, a notification is issued by the smartphone, which is implemented by the dedicated *consumer* (⑧ in the figure). When the *consumer* receives a notification message, it uses directly the native APIs of the smartphone OS to display the message to the user. The notification *consumer* implementation is as follows:

```
; === DiSPL header ===
(archetype "consumer")
; import native UI functions for the Android OS
(require android_ui.displ)
; contract declaration: one input port for streams of notifications
(contract (name "notification-consumer")
  ; the input port requires one or more input streams
  (input-ports (port (name "notifications")
    ; the input port requires one or more input streams...
    (min 1)
    ; ... that must satisfy a given schema
    (input-rule (type "schema")
      (attribute (name "message")
        (semantic-type "none")
        (unit "none")
        (concrete-type "string"))))))
; === DiSPL operations ===
consume:
  ; get the next available item and decapsulate the message
  (define message (get "message" (next-item "notifications")))
  ; use a native Android OS function for notify the message
  (android-notify message)
```

### B. Communication with the sensors using CoAP

Thanks to the implementation of CSOA over CoAP, the runner application may connect with Things that implement heterogeneous protocol stacks. Figure 5 illustrates relevant protocol stacks for our use-case:

- The $PM_{10}$ sensor is typically part of an urban sensing infrastructure that is managed by a third-party operator. The communication with sensors deployed in an urban environment is subject to strong interferences caused, e.g., by the environment itself (*i.e.*, multipath fading) or by concurrent communications from other equipments. To circumvent the impact of interferences, the network of pollution monitoring sensors can take advantage of the 6TiSCH architecture being currently standardized at IETF [21] [22].
- The temperature sensor may be provided by a neighboring weather station. To offer a direct access to its data, the sensor implements the 6LoWPAN protocol that adapts IPv6 to its restricted capacities.
- Contrary to the above sensors, the pulse sensor is worn by the runner. When part of a Wireless Body Area Network (WBAN), the sensor is accessed through a concentrator, otherwise it is accessed directly. The sensor may implement, e.g., the Bluetooth

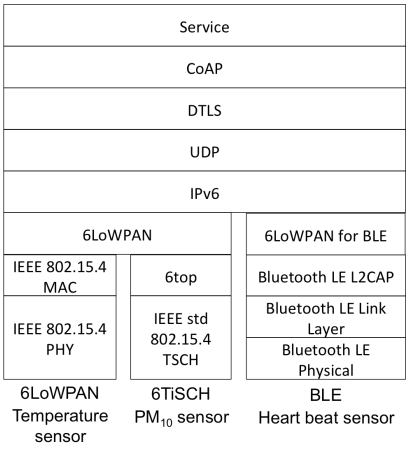



Figure 5: Protocol stacks implemented in sensors

Low Energy (BLE) protocol compatible with IPv6 thanks to a 6LoWPAN adaptation layer.

## V. Conclusion

With CSOA, heterogeneous Things are transformed in providers of continuous services so that applications are able to accommodate the high heterogeneity of, and the large amount of data produced by, the IoT. CSOA features four types of continuous services (i.e., producer, processor, storage and consumer) that implement continuous operations over streams. Thanks to service composition, continuous services may further be assembled to deliver new services. This paper has outlined the main concepts of CSOA, while we introduced a supporting middleware solution in [6].

A critical requirement for the actual deployment of CSOA-based applications within today's and even more next generation IoT is to be able to leverage the rich diversity of Things, regardless of the underlying protocol stacks. In that direction, we have discussed how the latest evolution of the CoAP protocol, including related security propositions, may conveniently be exploited for implementing streaming-based connectors between CSOA continuous services.

We exemplified CSOA using a mobile application for runners, which combines sensing services provided by sensors carried by runners and by third-party operated sensing infrastructures. The use case illustrates the various ways of leveraging Things within applications from fully decentralized with direct access to Things, to centralized with direct access to a cloud-based platform/infrastructure that gathers observations. The two extreme approaches have their respective advantages and shortcomings regarding the provided quality of service and of experience. We are currently investigating how to enable hybrid approaches that deliver the quality of service and of experience that best suits the applications' requirements.

## References


[1] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad Hoc Networking*, vol. 10, no. 7, 2012.

[2] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, "Interacting with the SOA-based internet of things: Discovery, query, selection, and on-demand provisioning of web services," *Services Computing, IEEE Transactions on*, vol. 3, no. 3, 2010.

[3] M. Papazoglou, "Service-oriented computing: concepts, characteristics and directions," in *International Conference on Web Information Systems Engineering*, 2003.

[4] D. Guinard and V. Trifa, *Building the Web of Things.* Manning, 2016.

[5] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Mobile Cloud Computing Workshop*, 2012.

[6] B. Billet and V. Issarny, "Dioptase: A distributed data streaming middleware for the future web of things," *Journal of Internet Services and Applications*, vol. 5, no. 13, 2014.

[7] M. Sperber, R. K. Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews, "Revised 6 report on the algorithmic language scheme," *Journal of Functional Programming*, vol. 19, no. S1, pp. 1–301, 2009.

[8] S. Cherrier, Y. M. Ghamri-Doudane, S. Lohier, and G. Roussel, "D-lite: Building internet of things choreographies," *arXiv preprint arXiv:1612.05975*, 2016.

[9] B. Billet and V. Issarny, "From task graphs to concrete actions: A new task mapping algorithm for the future internet of things," in *International Conference on Mobile Ad hoc and Sensor Systems*, ser. MASS '14, 2014.

[10] S. Farrell, "LPWAN Overview," Internet Engineering Task Force, Internet-Draft draft-ietf-lpwan-overview-01, Feb. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-lpwan-overview-01

[11] M. Centenaro, L. Vangelista, A. Zanella, and M. Zorzi, "Long-range communications in unlicensed bands: The rising stars in the iot and smart city scenarios," *IEEE Wireless Communications*, vol. 23, no. 5, pp. 60–67, 2016.

[12] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," Internet Requests for Comments, RFC Editor, RFC 7252, June 2014, http://www.rfc-editor.org/rfc/rfc7252.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7252.txt

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: elements of," 1994.

[14] K. Hartke, "Observing resources in the constrained application protocol (coap)," Internet Requests for Comments, RFC Editor, RFC 7641, September 2015.

[15] Z. Shelby, M. Vial, M. Koster, and C. Groves, "Dynamic resource linking for constrained restful environments," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-core-dynlink-02, February 2017, http://www.ietf.org/internet-drafts/draft-ietf-core-dynlink-02.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-core-dynlink-02.txt

[16] G. Tanganelli, C. Vallati, E. Mingozzi, and M. Kovatsch, "Efficient proxying of coap observe with quality of service support," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, Dec 2016, pp. 401–406.

[17] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347, Jan. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6347.txt

[18] D. Garcia-Carrillo and R. Marin-Lopez, "Lightweight coap-based bootstrapping service for the internet of things," *Sensors*, vol. 16, no. 3, p. 358, 2016.

[19] G. Selander, J. Mattsson, F. Palombini, and L. Seitz, "Object Security of CoAP (OSCOAP)," Internet Engineering Task Force, Internet-Draft draft-ietf-core-object-security-02, Mar. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-core-object-security-02

[20] N. El Helou, M. Tafflet, G. Berthelot, J. Tolaini, A. Marc, M. Guillaume, C. Hausswirth, and J.-F. Toussaint, "Impact of environmental parameters on marathon running performance," *PloS one*, vol. 7, no. 5, p. e37407, 2012.

[21] P. Thubert, "An Architecture for IPv6 over the TSCH mode of IEEE 802.15.4," Internet Engineering Task Force, Internet-Draft draft-ietf-6tisch-architecture-11, Jan. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-6tisch-architecture-11

[22] T. Watteyne, M. R. Palattella, and L. A. Grieco, "Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement," RFC 7554, May 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7554.txt


## Acknowledgments


The work is supported by the Inria@SiliconValley Inria International Lab (https://project.inria.fr/siliconvalley/) and the CityLab@Inria Inria Project Lab (http://citylab.inria.fr/) on digital science and technology for smarter cities. The work is partially supported by the European Commission's Horizon 2020 Framework Program, through the H2020 FIESTA-IOT and CHOREVOLUTION projects.