Dioptase: a distributed data streaming middleware for the future web of things

Benjamin Billet, Valérie Issarny

RESEARCH

Dioptase: A Distributed Data Streaming Middleware for the Future Web of Things

Benjamin Billet*† and Valérie Issarny^

*Correspondence:
benjamin.billet@inria.fr
ARLES Project-Team, Inria
Paris-Rocquencourt, France
Full list of author information is available at the end of the article
†benjaminbillet@inria.fr
^valerie.issarny@inria.fr

Abstract

The Internet of Things (IoT) is a promising concept toward pervasive computing as it may radically change the way people interact with the physical world, by connecting sensors to the Internet and, at a higher level, to the Web, thereby enacting a Web of Things (WoT). One of the challenges raised by the WoT is the in-network continuous processing of data streams presented by Things, which must be investigated urgently because it affects the future data models of the IoT, and is critical regarding the scalability and the sustainability required by the IoT. This cross-cutting concern has been previously studied in the context of Wireless Sensor Networks (WSN) given the focus on the acquisition and in-network processing of sensed data. However, proposed solutions feature various proprietary and highly specialized technologies that are difficult to integrate and complex to use, which represents a hurdle to their wide deployment. At the other end of the spectrum, cloud-based solutions introduce a too high energy cost for the envisioned IoT scale, considering the energy cost of communication over computation. There is thus a need for a distributed middleware solution for data stream management that leverages existing WSN work, while integrating it with today's Web technologies in order to support the required flexibility and the interoperability of the IoT. Toward that goal, this paper introduces *Dioptase*, a lightweight Data Stream Management System for the WoT, which aims to integrate the Things and their streams into today's Web by presenting sensors and actuators as Web services. The middleware specifically provides a way to describe complex fully-distributed stream-based mashups and to deploy them dynamically, at any time, as task graphs, over available Things of the network, including resource-constrained ones.

Keywords: Data Stream Management System; Internet of Things; streaming; middleware

1 Introduction

The Internet of Things (IoT) is a promising concept toward pervasive computing and one of the major paradigm shifts that the computing era is facing today [1]. In the IoT, everyday objects, the "Things", get networked so that they can cooperate autonomously, and allow humans to interact with the physical world as simply as they do with the virtual world [2, 3]. However, the IoT paradigm raises tremendous challenges, including the ability to perform continuous processing of data streams presented by Things. Data stream management is indeed a cross-cutting concern for the IoT [4], which must be studied urgently because it affects the future data models of the IoT. Specifically, applications aimed at the IoT have to manage data acquired from the physical world and thus deal with the consumption of continuous

data that evolve over time, as opposed to consuming data of the traditional Internet that are primarily discrete. Hence, in the IoT, data become volatile since they are useful only when they are produced and processed, while requests become persistent since they are permanently executed.

The continuous processing of sensed data has been extensively studied in the context of Wireless Sensor and Actuator Networks (WSAN) given the focus on the acquisition of data from the physical world. This has resulted in the introduction of dedicated Data Stream Management Systems (DSMS), which are, in the case of WSANs, tools to manage and process streams across a sensor network [5]. Historically, DSMSs were part of relational database research, as extension of Data Base Management Systems (DBMS), establishing a theoretical background for data stream management. In contrast to DBMS, WSAN research focuses on very low-power devices and emphasizes in-network processing in order to save energy and increase the lifetime of the networks [4], as one exchanged bit is sometimes equivalent to 1000 CPU cycles [6]. WSAN-based DSMSs thus adapt formal algebras and data models of DBMSs [4, 7], while featuring custom operations for continuous stream processing as well as probabilistic operators [8, 9] that are designed to reduce the device's processing load (CPU, memory and energy) and correct the errors that occur within mobile and distributed sensing environments (transient errors). Still, WSAN-based DSMSs are facing major challenges which prevent them from being used directly in the IoT:

1 They are characterized by various levels of in-network processing, with the use of fully or partially centralized approaches based on a single or many collection points. The systematic use of proxies in WSANs to solve resource constraints is indeed a bottleneck and a threat to the scaling up aim, a mandatory criterion of the IoT.
2 They introduce many proprietary technologies (from both network and development perspectives) which can be used only in specific sensor networks and are difficult to use for developers who are not expert in the domain [2, 4]. As a solution, given that today's Web connects smoothly a huge number of highly heterogeneous devices [10], Web-based DSMSs for the IoT promote interoperability, standardization and openness by using Web-based techniques and methods that enable stream management [11], making the IoT part of the greater Future Internet as a Web of Things (WoT). However, existing approaches do not suit well the energy- and resource-efficiency requirements of resource-constrained Things, because of the overhead associated with Web technologies that makes them working only on the most powerful Things [3], or "smart Things" (typically smartphones or plug computers).
3 Due to the limited resources of the devices, WSAN-based DSMSs are dedicated to specific tasks composed from a fixed set of operations (e.g., relational operators). As a result, it is either not possible, or at least very difficult, for developers to apply new operators once the network has been deployed. The developer can only compose a fixed set of existing operators provided by the DSMS. This is not appropriate for dynamic and large networks like the IoT, which is expected to run various contextual tasks that are not predefined.

Most of the above problems are related to the resource constraints of the existing sensor technologies. However, moderately powerful Things, or "average Things",

are emerging, pioneered by sensor technologies like Imote2[1] and Sun SPOT[2]. These devices are likely to expand drastically in the near future while their cost will decrease, as more and more IoT appliances are expected to be released by the industrial world[3]. Typically, this class of devices can accommodate Web technologies provided the technologies are adequately revisited, knowing that only a subset of these technologies are useful for implementing Web services in order to achieve the Web of Things (WoT) vision. Hence, we argue that WSAN- and Web-based techniques need to be integrated within a fully-distributed streaming middleware that is able to run directly onto every type of average and smart Things. As a benefit, the IoT will be more interoperable, each Thing will be more autonomous and the need of proxies will be mitigated.

Toward this end, this paper introduces a customizable distributed DSMS middleware, called *Dioptase*, whose contributions are as follows:

- *Dioptase* adds flexibility to state of the art DSMS solutions for resource-constrained devices, by introducing a high-level application model that can map any IoT/WoT application onto the entities of the network (sensors, actuators, users, services, databases, etc.). In this model, each Thing is abstracted as a generic device that can be dynamically assigned communication, storage and computation tasks according to its available resources, enabling the applications to be directly executed in the network without any proxy (in-network processing).
- *Dioptase* features a customizable middleware architecture that is versatile enough to be deployed on a large class of Things that vary significantly in terms of resource availability (e.g., sensors, smartphones or plug computers), provided these Things are able to communicate directly through the Internet infrastructure (typically the average Things that use 6LoWPAN) [12]. Unlike WSAN-based DSMSs that target specific sensor networks, *Dioptase* enable developers to use the same middleware on moderately powerful sensors (e.g. Sun SPOT), smartphones, personal computers, servers and the cloud.
- From a technical perspective, the flexibility of *Dioptase* is based on a lightweight domain-specific language (DSL) designed to express continuous processing tasks. The DSL syntax is specifically optimized to be interpreted on the huge number of average Things that are more powerful than small sensors but very limited compared to smart Things. This mechanism enables the dynamic deployment of tasks in isolated sandboxes which are naturally safer than arbitrary binary-code deployment [13, 14]. As a benefit, developers can build applications composed of tasks deployed in the network at any time, using standard Web services. To achieve this, *Dioptase* features relevant optimizations of Web technologies (small Web server, subset of protocols, compression, etc.) and leverages advanced stream management techniques (in-network processing, approximation and dynamic reconfiguration).

As detailed in the following, *Dioptase* makes it possible: (i) to integrate the Things with today's Web by exposing sensors and actuators as Web services, (ii) to manage

[1] `http://www.xbow.jp/Imote2.Builder_kit.pdf` (last access: 10-14-2014).
[2] `http://www.sunspotworld.com` (last access: 10-14-2014).
[3] `http://www.technologyreview.com/news/527356/business-adapts-to-a-new-style-of-computer` (last access: 10-14-2014).

physical data as streams, and (iii) to use any Thing as a generic pool of resources that can process streams by running tasks that are provided by developers over time. The rest of this paper is organized as follows: Section 2 first discusses the role of proxies in WSANs and reviews related work in the area of streaming solutions for WSANs and IoT/WoT, highlighting required capabilities for data streaming middleware in the future IoT/WoT context. Section 3 then presents the *Dioptase* application model for the WoT, which allows the design of mashups[4] that compose the streams flowing in the WoT. Following, Section 4 describes the architectural design of the *Dioptase* middleware together with its implementation, while Section 5 provides an evaluation of *Dioptase* for both average and smart Things. Finally, Section 6 draws some conclusions and sketches our perspectives for future work.

2 Background

Our work is motivated by the two following main goals:

- We want to make Things able to execute complex tasks that are not predefined at the Things' deployment time so as to enable developers to use the WoT as a pool of generic resources, without unneeded intermediaries (proxies, gateways, base stations, etc.). The role of such intermediaries is specifically discussed in Section 2.1.
- We want to integrate the work done on data streaming for wireless sensor networks with the Web in order to actually achieve the Web of Things (WoT) vision [15], which has led us to base our research on the work on data streaming as part of the Web and of WSANs. Existing DSMSs for WSANs are presented in Section 2.2 with their related advantages and drawbacks.

Our solution specifically lies in enabling stream-oriented mashups that may be dynamically deployed and reconfigured, which suits well the real-world use cases that are commonly presented in the IoT/WoT literature and highlights the increased autonomy of Things (e.g., see [2]).

2.1 Intermediaries in WSANs

Usually, a WSAN is composed of (i) several motes equipped with one or more sensors and a wireless interface, and (ii) more powerful devices, typically fixed and continuously-powered, that embed actuators [4]. In addition, a WSAN leverages proxies, gateways or base stations for carrying out collection and computation tasks, as well as communication with other networks, such as the Internet. Nowadays, the above intermediaries are not anymore required for communication between motes and the Internet, thanks to the standardized stack composed of IEEE802.15.4 and 6LoWPAN, which is intended to replace proprietary communication proxies (application level) by standardized IP routers (network level) [12]. As a benefit, motes have an IPv6 address, or an equivalent made of the network identifier and a small address, and can communicate directly with the Internet.

Regarding data collection, proxies are still needed in order to enhance the sensor network capabilities, e.g., for implementing heavy computation (offloading), centralized management and task deployment, caching and security/privacy (access

[4] In web development, a mashup is an application that composes data and services from many sources, using open programming interfaces. Some examples can be seen on `http://www.programmableweb.com/mashups` (last access: 10-14-2014).

control, key management, etc.). However, offloading data collection and processing to proxies is energy-consuming due to the wireless communication, which holds for any wireless device, including smartphones [6, 16]. Similarly, cloud-based stream processing is quite popular today, and there are some attempts to use it with sensor networks and IoT: cloud of sensors, cloud-based IoT, cloud-assisted remote sensing, etc [17, 18]. However, the same problems arise regarding communication costs, availability (specifically for mobile Things with sparse connectivity), latency and privacy.

As a solution to the above problems, it has been proposed to let the sensor network performs as much in-network processing as possible before sending anything to a proxy or the cloud, in order to: (i) reduce the amount of transferred data and (ii) make use of the motes at their full potential. For example, structural health monitoring is a case where a huge amount of measurements is produced quickly because of the vibration sensors. These types of sensors are very sensitive and detect a lot of 3-axis accelerations, saturating the network and exhausting the sensors' batteries. In such a case, pre-aggregation, pre-filtering and compression can be performed within the motes instead of the base station [19].

Consequently, in our opinion, centralized intermediaries (proxies, surrogates, cloudlets and the cloud) should be leveraged primarily for heavy computation, while in-network processing should be favored for common and simple tasks (filtering, merging, etc.) as well as for complex tasks when powerful/specialized enough Things are available. To this end, *Dioptase* is intended to avoid reliance on those intermediaries whenever possible, by running on devices that support 6LoWPAN or IPv6 and communicate directly with the Internet. Nevertheless, in cases where intermediaries are needed, *Dioptase* can be deployed on them and run as a middleware layer for deploying tasks dynamically and managing data streams.

2.2 DSMSs for WSANs

The work most related to ours may then be classified into three major families of DSMSs for WSANs, which are respectively based on: (i) the relational model, (ii) macro-programming and (iii) Web services.

We also identify related work on supporting the construction of mashups in the WoT although focused on the exchange of discrete data like Actinium [20], COMPOSE [21], Eywa [22] and the Thin Server architecture [23]. However, these solutions consider Things as passive data providers and shift the computation logic into powerful servers or into the cloud. As we said before, in our opinion, centralization is not suitable for the WoT from a scaling up perspective, even in the cloud, as it weakens the entire network and increases the overall energy consumption.

Relational DSMSs extend the relational model by adding concepts that are necessary to handle data streams and persistent queries, together with the stream-oriented version of the relational operators (e.g., selection or union). The sensor network is then managed as a large database that can be queried using a SQL-like language, with some specific operations. The database may further be distributed (each node runs a part of the query), centralized (a powerful node collects all the data and applies queries) or partially centralized (with many powerful nodes) [24]. From a practical perspective, queries are translated into query plans that are distributed in the network. State of the art DSMSs primarily differ with respect to: the

expressiveness of the query language, the associated algebra, and assumptions made about the underlying networking architecture. A well-known DSMS is *TinyDB* [25], which exposes the sensed data as a relation (i.e., table) on which it is possible to apply queries over the sensed values as well as the metadata associated with the sensors. During the handling of queries, all the nodes execute the queries that are distributed in the network and the results of each query get aggregated as they traverse the routing tree maintained by the system. In the same vein, *Cougar* [26] acts as a database of sensors where the query plans are provided to proxies that take care of activating the relevant sensors and applying the operations on the collected data. *MaD-WiSe* [27] offers a runtime system for queries that is fully distributed, and each sensor may directly execute part of a query plan and then deal with sensor-specific tasks. *Borealis* [28], previously *Aurora*, uses data stream diagrams, which express the combination of relational operators over the streams received by the system. From a theoretical perspective, various systems propose custom extensions to the relational model as well as custom implementations of the relational operators. For instance, *STREAM* [29] distinguishes streams from relations, where the latter can be handled by classical relational operators. New operators then deal with translation from stream to relations (typically using windows), and vice versa (using streamers). *EQL* [30] moves a step forward, by enabling the developers to express composite queries in a very concise way, in order to detect and track complex events which involves various types of sensors (e.g., gas leak). Other proposals [7, 8, 9, 31] deal with issues as diverse as blocking and non-blocking operators, windows, stream approximation, and various optimizations.

State-of-the-art WSAN-based DSMSs suffer from proprietary protocols and technologies specifically designed to handle the characteristics of resource-constrained devices. As a consequence, proxies are often used to collect, process and present sensed data on the Internet, creating (i) an unwanted bottleneck, (ii) a single point of failure and (iii) an increased energy consumption if no proper in-network processing technique is used. To alleviate such effects, a DSMS for the WoT should include a middleware layer designed to run directly on Things without any intermediary (except for conversions at physical and link levels), given that modern device classes are emerging and allows more flexible data stream management based on the use of Web technologies. In addition, such middleware must reuse and extends the rich theoretical background of relational DSMSs, especially the data models proposed to describe streams and the non-blocking operators initially designed for WSANs.

Macroprogramming-based DSMSs enable users to express tasks over the WSAN using a DSL instead of a query language. The resulting tasks, or macroprograms, are compiled into microprograms to be run on the networked nodes, hence easing the developer's work who no longer has to bother with the decomposition and further distribution of the macroprograms. Macroprogramming-based DSMSs are overall similar to classical macroprogramming approaches aimed at WSAN. However, they feature additional primitives and mechanisms oriented toward stream management. For instance, Regiment [32] introduces a functional language that enables programming the WSAN and manipulating the streams that flow in the network. As for Semantic Streams [33], it defines a declarative language based on Prolog, which features data structures to handle streams, together with mechanisms

to reason about the semantics of sensors. For instance, the system is able to compose or adapt data according to the available sensors and the given request.

As outlined above, existing macroprogramming-based DSMSs follow a static approach where the macroprograms are compiled into microprograms that are deployed once for all. Specific techniques can be used to dynamically update the network: (i) dynamic reconfiguration and (ii) dynamic deployment. However, the former techniques usually assume that the tasks are already implemented on the devices [34], while the latter techniques usually support binary deployment (e.g., Deluge [13]). Instead, a DSMS for the WoT must provide a high-level of dynamicity by making possible to change both the global and the local behaviors of the network at any time. To this end, the developers should be provided a way to represent WoT applications as abstract programs that are distributed dynamically in the actual network. In addition, sandboxes should be used to increase the overall reliability, as an attacker can benefit from arbitrary binary deployment to deploy malicious code on any open device.

Service-oriented DSMSs aim to integrate with classical service-oriented architectures, thereby taking advantages of the existing infrastructure (interaction and discovery protocols, registries, service composition based on orchestration or choreography, etc.). Similarly to database-oriented relational DSMSs, the simplest service-oriented DSMSs are centralized with a unique point of data collection [11, 35, 36], or semi-distributed based on a set of data collection points [37, 38]. However, these DSMSs focus mainly on the problem of presenting streams as services, without reusing the existing and valuable theoretical work from WSANs. In practice, these approaches are based on well-known Web service technologies. For RESTful services, some studies use specific mechanisms of the HTTP protocol, like *Web hooks*, *long polling* and *HTTP streaming* [11]. As for SOAP services, some work extends the SOAP architecture by adding new *message exchange patterns* (MEP) designed for stream communication (e.g., the capability for a service to receive multiple requests and produce multiple responses in parallel when invoked) [39]. Usually, sensors are presented as Web resources, identified by URIs [11, 35, 38]. The paradigms used to broadcast streams vary from one solution to another. *Stream Feeds* [35] uses pull requests to gather historical data and push requests to receive new data issued by the sensors. *RMS* [11] goes a step further by building upon a topic-based pub/sub infrastructure, while *WebPlug* [38] uses an infrastructure based on pollers that periodically check the state of resources.

Integrating data stream management into service-oriented architectures is a logical evolution of sensor networks, as Web technologies provide a greater flexibility, ease of use and interoperability compared to existing WSNs technologies. The proposed solutions, in particular, enable Things to communicate through the Internet and expose their resources as standardized Web services. As simple as the present Web, these services can be used to build mashups that interact with the physical world. However, existing solutions are limited by their scope. Indeed, much research is focusing on how to present streams as Web services, and neglects many complex aspects like continuous processing of streams (merging, filtering, adaptation, approximation, etc.). Reusing theoretical and practical foundations that were established by the two other families of DSMSs is a crucial step to enable the IoT

to take advantage of WSAN capabilities together with the flexibility, the reliability and the interoperability of the Web, which guided the design of the *Dioptase* application model and supporting middleware toward the WoT vision.

3 The Dioptase Application Model for the WoT

The *Dioptase* application model for the WoT allows developers to easily build mashups able to manage, process and compose streams produced within networks of Things. This model is oriented toward the high-level description and distribution of stream-based mashups as components over the network, enabling the dynamic deployment of these components over resource-constrained Things.

3.1 Dioptase Component Model

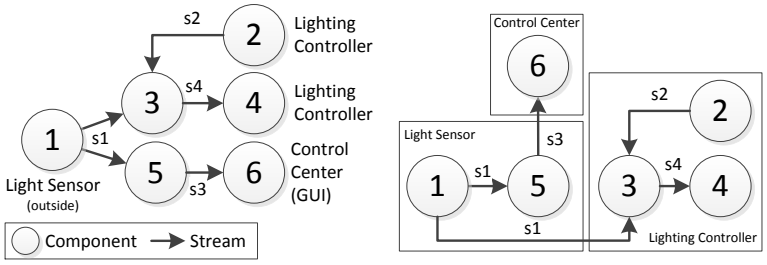
As illustrated by the WSAN work, we identify four high-level roles that each Thing may play, usually in combination, depending on its resources: (i) A *production* role where the Thing presents sensor data as streams, (ii) a *processing* role where the Thing continuously processes streams, (iii) a *consumption* role where the Thing acquires streams and drives actuators, and (iv) a *storage* role where the Thing saves data extracted from streams (in its memory, or persistently).

A *Dioptase mashup* is thus composed of distributed components, called *atomic components*, derived from the above roles: *producer*, *processor*, *consumer* and *storage*. These components interact (are connected) by continuously exchanging data as *streams*. The mashup can then be easily described as an acyclic directed graph (VL, EL) where the nodes $vl_i \in VL$ are producers (sources), processors, consumers (sinks) and storages, and the edges of the graph, $el_j \in EL$, are streams that link components together.

The mashup graph is equivalent to the query plan that can be found in DSMSs that present sensor networks as databases. However, query plans are strongly coupled to the query language capabilities that are limited w.r.t. the set of operations that can be executed. In contrast, the high-level nature of the *Dioptase* components makes it possible to easily represent any element of a WoT application as components that produce and consume streams. 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. In addition, processors may implement any type of continuous computation, or *task*. This flexibility allows the representation of mashups that can describe complex tasks for a wide variety of entities (sensors, actuators, servers, users, services, etc.).

As an illustration, Figure 1a presents an example of a simple mashup that analyzes outdoor light in order to control an indoor lighting system. In this mashup, a producer ① reads the light value and another producer ② monitors the lighting system state. These data are acquired by a processor ③ that produces an event stream for the lighting system ④. At the same time, the light measurements are saved by a storage ⑤ and are consumed by the lighting control application ⑥ that presents historical values to the administrator.

We call this graph a *logical mashup graph* because it describes the tasks that the network has to perform. This graph is provided by the developer either directly



(a) Logical mashup graph. (b) Physical mashup graph.

①	Producer	Reads the light sensor every x seconds.
②	Producer	Produces *on* if the light is turned on, or *off* if the light is turned off.
③	Processor	If ② produces *on*, produces *switch-off* if the light reading $> l_{max}$. If ② produces *off*, produces *switch-on* if the light reading $< l_{min}$.
④	Consumer	Switch on/off the light according to the events received from ③.
⑤	Storage	Stores each light reading.
⑥	Consumer	Asks for data stored in ④ and presents it to the application.

Figure 1: Logical and physical mashup graphs for lighting control.

or expressed as a query that is translated into a mashup graph. Using information provided by a discovery system (e.g., registry or distributed protocol [40]) that is aware of Things' locations and available resources, the logical mashup graph is automatically converted into a *physical mashup graph* (VP, EP), were each $vp_i \in VP$ is a pair (vl, n) that maps a component vl onto a host device n, as depicted in Figure 1b. In particular, depending on its capabilities, a Thing can be assigned either a single component or an entire subgraph. The problem of computing the physical mashup graph from the logical mashup graph is a variation of the *task mapping problem*, where a set of communicating tasks with several properties (constraints, requirements, resource consumption, etc.) have to be mapped to a set of connected nodes given their characteristics (location, hardware capabilities, etc.). Task mapping within *Dioptase* is beyond the scope of this paper, and the interested reader is referred to [41] for relevant baseline together with [42] for a specific *Dioptase* solution.

In our component model, each component defines some *input ports* for the consumption of streams, depending on the component type, and at most one *output port* where new stream items are produced. Provided the data types specified for the input and output streams match, any output port can be connected to any input port through a one-to-one connection. Theoretically, stream communication between components can be achieved in three ways: (i) *pull*, where a consumer requests a producer to send the data stream, (ii) *push*, where a producer requests a consumer to process its data, and (iii) *hybrid*, which allows the two previous modes. The choice of either mode is not important from a functional perspective and defines only which component should initiate the transmission. In our work, we consider that a consumer must be autonomous and does not have to process an unwanted stream. As a consequence, the data exchange between two components is pull-based, as a component always decides how to connect its input ports.

3.2 Data Stream

According to the literature [5, 29, 43], a *stream* is a sequence of discrete items that are linked by some properties (e.g., same source, same type, time coupling, etc.). The size of this sequence is theoretically infinite and it is not possible to know its end *a priori*. In *Dioptase*, each *stream item* is a tuple associated with a *timestamp* that can be explicit, if generated with the tuple, or implicit, if defined when the tuple is received [5, 29, 43]. Then, as for relations in relational databases, a *Dioptase* stream adheres to a *schema* that defines the *attributes* of each tuple. In addition, the *Dioptase* schema is intended to take into account semantic aspects of the sensed data and the characteristics of the data source. In practice, the schema is composed of:

- The *Semantic concept* of the attribute (e.g., temperature or pressure), which helps Things to reason about the produced data in order to, e.g., compose them automatically (e.g., kinetic energy $= \frac{1}{2} \times \text{mass} \times \text{speed}^2$) or select the most relevant algorithms for approximation, prediction or interpolation.
- The *Concrete type* of the attribute, i.e., the data type. The most simple types are integer, real or boolean, but more complex types can be considered, like image or audio/video sequence.
- *Metadata* that are specific to the semantic concept, and make the system more adaptable. For example, the *unit of measurement* can be used to adapt automatically to requests that involve different units for the same semantic concept (e.g., kelvin, celsius and farenheit for temperature).

The properties defined in the schema can be defined using a standard vocabulary in order to reason automatically about these data, according to external knowledge provided by the developers. For example, the unit and semantic type can refer to ontologies of physical concepts and related models (prediction, interpolation or error models) [44].

The connection between a component's output port to the input port of another component is established through a *connector*, i.e., a software component that manages the transport, adaptation and presentation of the data as streams, between two components. We introduce two types of connectors for stream transportation in *Dioptase*: *local connector* and *remote connector*. The former manages connections between two components that are running on the same Thing and optimizes communication accordingly, while the latter acquires data from a component that is running on another Thing.

Various specializations of the remote connector may be envisioned, notably for interfacing the *Dioptase* middleware with other data stream management systems, sensor networks (e.g., a CoAP connector) [12] or existing services (e.g., a meteorological database). This remains an area for further extension of the *Dioptase* middleware, while our current middleware implementation supports HTTP-based streaming (polling, hooks and websockets [45, 46]).

3.3 Stream Processing

Stream-based communication requires dedicated support for data processing. Indeed, as streams are unbounded, it is not feasible to store the entire stream before applying any operation. Although some operations are naturally non-blocking, i.e.,

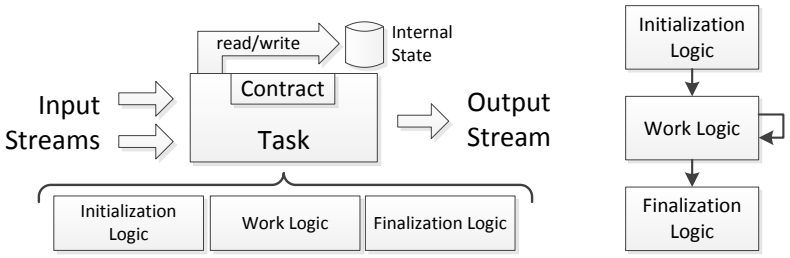


Figure 2: Processor architecture (left) and lifecycle (right).

able to produce tuples without detecting the end of input streams (e.g., set intersection), some other operations are unable to produce any item before the acquisition of the entire streams (e.g., set difference) [47]. The current *Dioptase* middleware handles blocking operations using windows, although this is not detailed in the paper; the interested reader may refer to [43] for classical windowing techniques. Concerning non-blocking operations, traditional WSAN-based DSMSs fix the set of operations that can be applied (typically relational operators). However, this is too restrictive, especially in light of the increasing capabilities of Things. Instead, the developers should be provided means to dynamically specify complex tasks for execution by Things. Hence, *Dioptase* introduces the processor components, which perform non-blocking processing.

Thanks to processor components, Things are able to perform any computation over data streams that is not necessarily defined at the time the Things are deployed. Specifically, a processor executes a given task, i.e., a sequence of operations, over one or more streams, where the task may be provided at any time. A task can be either *compiled* (directly implemented on the Thing by the developer, using the platform's native language) or *interpreted*, i.e., described in a lightweight DSL, which is directly interpreted by the middleware. While the *Dioptase* DSL, called *DiSPL* (Dioptase Stream Processing Language), supports generic-purpose structures (control flow statements), specific primitives are provided to manipulate data streams (e.g., read/write into streams or build new stream items) and atomic components (e.g., create new storages or migrate a processor). As a benefit, *DiSPL* enables the developer to describe a wide range of complex tasks and dynamically send them to any known Thing, at any time. Technical details about *DiSPL* to describe interpreted tasks are provided in the next Section.

Compiled tasks are less flexible than interpreted ones, but they are more efficient (native code) and are useful to implement the library of common operations (e.g., compute an average value, count the number of items) or *operators*. These operators are often used in practice by developers and it is better to express them as compiled tasks in order to improve the efficiency of WoT applications. In addition, *Dioptase* includes various packages of operators dedicated to approximation (e.g., linear prediction [48], sampling), correction and compression of sensed data.

The lifecycle of a processor is divided in three steps: (i) deployment of the processor and initialization of the required resources (global variables, parameters, libraries, etc.), (ii) processing of each new stream item, and (iii) termination of the component that frees all the resources previously initialized. As shown in Figure 2,

```
"operator":"dioptase.count",
"inputs":{
    "main":{
        "type":"any",
        "scope":"tuple"
    }
},
"output":{
    "count":{
        "semantic":"none",
        "unit":"none",
        "concrete":"integer"
    }
}
```

```
"operator":"dioptase.join",
"inputs":{
    "input1":{
        "type":"any",
        "scope":"tuple"
    },
    "input2":{
        "type":"any",
        "scope":"tuple"
    }
},
"output":"dynamic",
"params":{
    "attribute":{
        "type":"string"
    }
}
```

Figure 3: Examples of contracts for *COUNT* (left) and *JOIN* (right) operators, in JSON.

these three steps are described by corresponding sections in the task: *initialization logic*, *work logic* and *finalization logic*. Each step is allowed to read and write data into the *internal state* maintained by the processor, which is a structure that can be serialized and moved into another Thing if necessary. In addition, the task is characterized by a *contract* that defines the schemas of the input and output streams that are compatible with the task operations. This contract is used by the processor at deployment time to instantiate its ports and the related schemas. Figure 3 presents the contract of two operators (in JSON): (i) an operator that counts the tuples (any type) of a single input stream and produces one output stream composed of singletons (attribute name is *count*) that do not have semantic type and unit, and (ii) an operator that performs an inner join on two input streams, given an attribute name as a string parameter (called *attribute*), and produces an output stream with a schema built dynamically from the concrete input streams when the operator is deployed.

4 Dioptase Architecture and Design

Figure 4 depicts, from a high-level perspective, the *Dioptase* middleware architecture supporting the dynamic deployment of distributed mashups within the WoT. First, to manage the specifics of different classes of Things and platforms, Thing-specific low-level functionalities are separated into *Drivers*. These drivers are loaded when the middleware starts and are used by other modules. Drivers have to be implemented for each class of Things and provide, in particular, the communication routines, the access to the Thing's sensors and actuators and the storage management functions.

At run-time, the *Component Manager* runs the components that are deployed on the Thing. These components produce and consume streams, locally and remotely, through the connectors that manage data transport. The processors run the tasks that are either provided by developers or obtained from a standard operator library (e.g., selection, join, sort). Non-predefined tasks are deployed at run-time and are described using the *DiSPL* DSL that is run by the embedded *Interpreter*.

Network communication is carried out through Web services that expose the resources of the Thing (access to streams and metadata, manage components, settings,

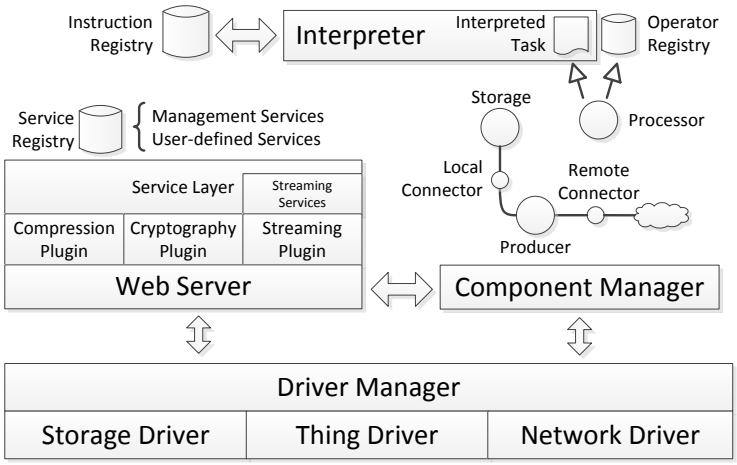


Figure 4: Middleware architecture.

etc.). For this purpose, the middleware embeds a lightweight all-in-one *Web client/server* optimized to run with few resources. The services are written in native code and are directly compiled with the middleware. Their implementations are well-decoupled from the Web server and, instead of the costly TCP transport protocol, Web services protocols for resource-constrained devices can be used, such as CoAP or HTTPU (HTTP over UDP) [49].

Precisely, only a subset of HTTP is useful to implement a Web service [49] and, consequently, our small HTTP implementation supports only a limited set of requests (GET/POST only), headers, MIME types, encodings (UTF-8 only), languages and mechanisms. Similarly, lightweight formats are used whenever possible (binary serialization or JSON) for describing services parameters and responses' content. Basically, only simple requests/responses are supported, with the smallest set of mandatory HTTP headers. For the Thing with higher capabilities, additional HTTP standard functions, like *Compression* (e.g., gzip, deflate) or *Cryptography* (e.g., SSL, TLS), are provided as *Plugins* that can be enabled or disabled according to the Thing's resources. Compression is particularly interesting, as it can reduce drastically the amount of exchanged data and the energy consumption [19].

All the components presented in Figure 4 are intended to be deployed directly on the Thing. However, in order to run on a large number of Things and to handle the hardware heterogeneity of Things (heterogeneous resources, specific capabilities, etc.), *Dioptase* is highly modular and can be adapted to the resources of the Things. Concretely, the middleware deployment consists of two steps: *customization* and *deployment*. Customization of the middleware consists into removing irrelevant modules (e.g., compression/cryptography plugins or the interpreter component) and adding or implementing new modules based on the specific capabilities of the Thing (e.g., hardware video decoding). For example, during this phase, the Thing's owner may implement new operators and register them in the standard library, for future usage. Similarly, the *DiSPL* DSL can be extended by defining additional packages of instructions (e.g., a wrapper for a library deployed onto the Thing). Ultimately, the customized middleware is deployed onto the Thing and connected to the network.

In fact, customizing the middleware is rather straightforward, as the modules are clearly identified. Nevertheless, even if it has to be done only once, this operation can be time-consuming. Fortunately, a great deal of it can be simplified, by providing pre-packaged and preconfigured versions of *Dioptase* built for specific classes of Things (depending on their hardware resources). Regarding the development of Thing-specific components (e.g., supporting a video decoding chip), widely used libraries can be shared between developers or, in the future, provided by the vendors.

4.1 Middleware Services

Dioptase is a service-oriented middleware that exposes the Thing's resources (sensors, actuators, components and streams) as services, and more specifically RESTful services because of performance constraints [49]. The main middleware services are the *streaming services* that enable access to streams, and the *management services* which are used to manage and control the Thing and the middleware modules.

Streaming Services are implemented using two different techniques supported by the web server's streaming plugin: (i) *HTTP streaming*, where the connection is never closed and each item is sent as chunks in the HTTP response, and (ii) *Web hooks*, which establish a callback service in the client in order to enable the server to send new items as HTTP requests. We use both techniques because of their respective advantages and drawbacks. On the one hand, HTTP streaming implies maintaining a TCP connection and Web hooks lead to a large overhead (request headers) [11]. As a consequence, if the stream's data rate (i.e., stream items per second) is high, HTTP streaming is more efficient as it introduces a constant overhead (the TCP connection) independently of the number of stream items. On the other hand, if the data rate is low, Web hooks are more suitable because they avoid the use of an infinite connection.

Access to a stream is done in two steps: *access request* and *streaming*. The first step consists in calling the service stream as a regular RESTful service with the desired streaming method (HTTP streaming or Web hooks) as a parameter. The second step is different according to the method: in the case of HTTP streaming, the data are embedded in the response and, in the case of Web hooks, the callback service is invoked for each new stream item. Figure 5 presents an example of a simple stream of light values, accessed over HTTP streaming and Web hooks.

This behavior is abstracted by using the remote connector, which manages these low-level aspects by opening or closing callback services transparently. However, if it is not possible to directly access a Thing through the network (e.g., because of NAT), using a proxy is mandatory and Web hooks communication is disabled. This problem, which is related to some networks (e.g., LAN, 3G), will be alleviated in the future because of the use of IPv6 that solves the addressing problem. As a benefit, NAT mechanisms will disappear [50], enabling each Thing to be accessed directly through a public address.

Management Services enable developers and other Things to control the components that are running on the middleware and to deploy new ones, as shown in Table 1 that summarizes the usual services and their parameters. For example, a new processor can be deployed by providing a task and a set of streams to use as inputs. These streams are identified by a specific URI

```
1   GET /streams?id=light-stream&mode=stream HTTP/1.1
2   ...
3   HTTP/1.1 200 OK
4   Transfer-Encoding: chunked
5   Content-Type: application/x-www-form-urlencoded
6   ...
7   17
8   t=566175600&light=226.3
9   18
10  t=566177500&light=201.08
11  ...
```

This is a network dump of the HTTP request (lines 1-2) sent to a producer by a consumer to acquire a stream called "light-stream", and the resulting HTTP response (lines 3-11) where the stream items are written as they are produced.
Specifically, the request contains a parameter *mode=stream*, indicating that the consumer wants to receive the stream items using the HTTP streaming technique. Accordingly, the HTTP response is then configured to use chunks (line 4) and the streams items are written as chunks in the response while they are produced: lines 7-8 and 9-10 are two stream items (t is the timestamp and *light* is the attribute name), written as HTTP chunks. The HTTP response is not closed by the server until the stream reaches its end.

(a) HTTP streaming.

```
1   GET /streams?id=light-stream&mode=hook&hook=hook1 HTTP/1.1
2   ...
3   HTTP/1.1 200 OK
4   ...
5
6   GET /hooks/hook-name?t=566175600&light=226.3 HTTP/1.1
7   ...
8   HTTP/1.1 200 OK
9   ...
10
11  GET /hooks/hook-name?t=566177500&light=201.08 HTTP/1.1
12  ...
13  HTTP/1.1 200 OK
14  ...
```

This is a network dump of the HTTP request (lines 1-2) sent to a producer by a consumer to acquire a stream called "light-stream", and the resulting HTTP response (lines 3-4). While the stream items are produced, they are pushed by the producer to the consumer as HTTP request-response (lines 6-8 and 11-13).
Specifically, the first request contains a parameter *mode=hook*, indicating that the consumer wants a stream using the Web hooks technique HTTP streaming. This request indicates to the producer that the Web hook that should be used to send back the stream items is called "hook1". Then, for each stream item produced into the stream, the producer sends an HTTP request to the consumer, using the hook name (*/hooks/hook1* is the callback URI of the consumer): lines 6-8 and 11-13 are two stream items, encoded in the URI query string (t is the timestamp and *light* is the attribute name).

(b) Web hooks.

Figure 5: Example of streams (access request and streaming) over HTTP.

that describes local and remote streams (e.g., *dioptase://localhost/stream-name*, *dioptase://server:port/stream-name*). Figure 6 presents an example of a deployment request over HTTP for an interpreted processor that consumes two streams and executes a given *DiSPL* program. Then, the middleware deploys the processor, instantiates each connector according to the given stream URI and starts the execution in accordance with the lifecycle presented earlier. In addition, at deployment time, a processor or a producer can be asked to save a history of their output streams that can be queried later. Once deployed, a processor can be stopped and removed, as well as any other component.

Deploying a storage component is a similar operation, provided the storage type is supported by the Thing. At present, the *Dioptase* prototype supports three types of

```
1  POST /processors/new HTTP/1.1
2  Content-Type: multipart/form-data; boundary=fyrdm2
3  ...
4
5  --fyrdm2
6  Content-Disposition: form-data; name="id"
7
8  processor-name
9  --fyrdm2
10 Content-Disposition: form-data; name="code"
11
12 <some DiSPL code>
13 --fyrdm2
14 Content-Disposition: form-data; name="inputs"
15
16 dioptase://localhost/aLocalStream
17 dioptase://173.194.34.24:9000/anotherStream
18 --fyrdm2--
19
20 HTTP/1.1 200 OK
21 ...
```

This is a network dump of the HTTP request (lines 1-20) sent to a *Dioptase* instance by a developer to deploy an interpreted processor called "processor-name" with a given piece of *DiSPL* code that consumes two streams, and the resulting HTTP response (lines 22-23).
Specifically, the parameters are encoded in the *multipart/form-data* MIME format (line 2), which is the common format for high-length parameters [51]. The processor name is defined at lines 5-8, the *DiSPL* code at lines 9-13 and the URIs of the input streams that must be consumed by the processor are defined at lines 14-18. Given these URIs, the operation will specifically consume a local stream, produced by an operation already deployed on the Thing, and a remote stream currently produced by another Thing (173.194.34.24).

Figure 6: Example of deployment of an interpreted processor through HTTP services.

storage: (i) memory storage (fixed or extensible), (ii) file storage, and (iii) database storage (for embedded databases). Unlike producer and processor components, storages have a memory of past states that can be queried *a posteriori*. A storage component can produce a stream only when it receives a query that expresses some constraints that can be temporal (items between two timestamps, items older than x, etc.), volumetric (the x last items) or a combination of them. The complying results are presented as a new stream that ends when the last item is sent. Each storage type supports these constraints, but some storages can accept specific parameters (e.g., the database storage can handle a SQL query directly).

Similarly, actuators are presented as Web services and are based on the information provided by the *Thing Driver* about the physical actions that the Thing is able to perform. Each action can receive specific typed parameters that compose an *actuation contract* which defines the name and the type of each parameter, and the type of the returned result if any.

Other services can be used to access the Thing's metadata about the embedded sensors and actuators, the Thing's capabilities (e.g., hardware, location, load, energy level, operator library), and the components that are currently deployed (e.g., input/output schemas and load).

4.2 Dioptase Stream Processing Language (DiSPL)

As already mentioned, non-blocking operations are executed by processors, which are components dedicated to the execution of (i) *compiled tasks* that are linked to the middleware during the customization phase, and (ii) *interpreted tasks* that are

Service Path	Description and Parameters
/streams	Access to a stream. *id=<stream id>, mode=stream|hook, *hook=<hook name> (if mode=hook)*
/thing/properties	Get the properties of the Thing: sensors, actuators, supported storage types and metadata. **sensors=true|false, *actuators=true|false, *storages=true|false, *metadata=true|false*
/thing/sensors	Get identifiers, units, semantic/concrete types and metadata of one or all the sensors. **id=<sensor id>*
/thing/actuators	Get identifiers and parameters (names, types, metadata) of one or all the actuators. **id=<actuator id>*
/thing/operators	Get identifiers and contracts (name/type of inputs, number of inputs) of one or all the compiled operators. **id=<operator id>*
/thing/actuate	Executes one of the actuation service. *id=<actuator id>, service=<service name>, actuator-specific parameters (key/value)*
/components/running	Get the name and the input/output stream URIs of deployed components (producer, processor, storage).
/components/remove	Stop and destroy a component. *id=<component id>*
/processors	Get identifiers and input/output streams schemas of one or all the processors, possibly in extended form (logic, state, schemas). *id*=<processor id>, *extended=true|false*
/processors/state	Get the content of the internal state of a processor. *id=<processor id>*
/processors/new	Deploy a new processor and starts it. *id=<processor id>, code=<source code> or operator=<operator id>, inputs=<URIs>, *state=<start state>, *history=true|false, *h-size=<history size>, operator-specific parameters (key/value)*
/processors/history	Get a processor history as a stream. *id=<processor id>, *t-start=<timestamp>, *t-end=<timestamp>|now, *nb=<nb of items>*
/processors/migrate	Migrate a processor to another device. *id=<processor id>, to=<URI>, *forget-state=true|false*
/producers	Get identifiers, sensor names and output streams schemas of one or all the producers. *id*=<processor id>*
/producers/new	Deploy a new producer. *id=<producer id>, sensor=<sensor id>, *sampling=<sampling rate>*
/storages	Get identifiers, types and input/output schemas of one or all the storages. *id*=<storage id>*
/storages/new	Deploy a new storage. *id=<storage id>, type=<storage type>, storage-specific parameters (key/value)*

**xxx*: optional parameter *<yyy>*: any value *a|b|c*: parameter value can be *a*, *b* or *c*

Table 1: Common *Dioptase* services and their parameters.

described using the *DiSPL* DSL and deployed during the execution. This makes it possible to build logical and physical mashup graphs that use both compiled and interpreted tasks. Using the management services presented in the previous section, the developer is able to ask any known Thing to create a processor that executes either (i) a compiled task by providing its identifier, or (ii) an interpreted task by providing the *DiSPL* source code of the task.

The literature in stream processing already features languages like IBM SPL [52] but, in our case, the programs are intended to be interpreted directly onto the Things, as opposed to resource-rich servers. As a consequence, we introduce a new stream processing language, designed to be parsed efficiently by resource-constrained devices. Our language is based on the properties and the syntax of

```
1  init:                                    ;initialization section
2      (define count 0)                     ;creates a global variable for counting
3  work:                                    ;work section
4      (define diff (getNewItems "inStream"))  ;gets the set of new items
5      (if (> (size diff) 0)
6          ((set count (+ count (size diff)))  ;computes the new total
7          (write (item (now) "count" count))) ;writes the total into the output stream
8      )
```

Figure 7: COUNT operator expressed using DiSPL.

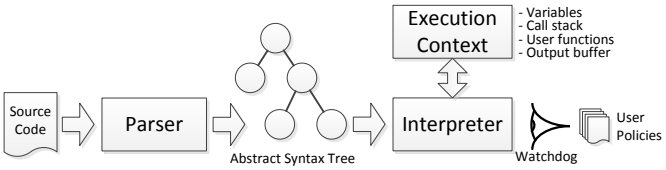


Figure 8: Interpreter architecture.

the functional language Scheme [53], which we chose for its simplicity and flexibility; S-Expressions have a very small grammar. The core of the language remains the same (variable definition, conditions, arithmetic and boolean expressions, etc.) but without λ-calculus support, which is not essential to describe continuous processing tasks, and increases the resources consumption of the interpreter. The general-purpose nature of the language makes feasible the description of a wide range of complex customized tasks, enhanced by various primitives dedicated to stream management. Figure 7 shows an example of a simple *COUNT* program that uses instructions for reading the new incoming stream items (*getNewItems*), build new stream items (*item*), and write data into the output stream (*write*). A larger example is given in Appendix A, which consists in the implementation of a Bloom Filter [54] using *DiSPL*. Other instructions are related to the Thing management and includes the ability to create and deploy new components, connect components' ports and monitor the Things' resources (memory, CPU load, battery).

As shown in Figure 8, interpreted tasks rely on a dedicated *parser*, which converts the source code into an *abstract syntax tree* (AST). Then, the processor sends the AST to the *interpreter* which builds an *execution context* for the given task. This context is used to store information like local variables or the call stack. Driven by the processor, the interpreter runs each section of the task and stores the global variables into the internal state of the component (i.e., the set of variables that are required to restore a component). Finally, the interpreter is monitored by a watchdog that collects information about the running task (execution time, consumed memory and CPU, etc.). This watchdog can kill any processor when resources are low, according to some policies provided by the user or the administrator of the Thing.

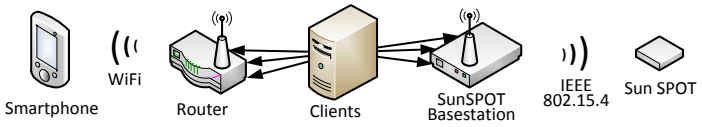


Figure 9: Experimental test bed.

5 Experimental Results

In order to evaluate our system, we implemented a prototype[5] of *Dioptase* in Java and deployed it onto devices with heterogeneous capabilities. The choice of Java is motivated by (i) the advances in porting the Java Virtual Machine to small sensors [55], (ii) the existence of all-in-one Java sensors, such as Sun SPOT, and (iii) the huge number of operating systems that supports Java, enabling us to work directly with a wide range of devices (computers, smartphones, embedded systems, etc.).

The experiments presented in this section have two goals. First, we want to show that the customization phase enables the use of the *Dioptase* middleware on heterogeneous Things in order to serve HTTP streams with suitable performances relative to available resources. Second, we aim to analyze the overhead due to the code interpretation mechanism, by comparing the consumption of resources by compiled and interpreted tasks, respectively.

During our experiments, we focused on two Things: a *Galaxy Nexus* and an *Oracle Sun SPOT*. The Galaxy Nexus is a smartphone that we consider representative of today's smart Things, i.e., a very powerful and mobile Thing [3]. The device embeds a dual-core 1.2GHz CPU (ARM Cortex-A9), one gigabyte of memory, and it runs with the Android 4.2.2 "Jelly Bean" operating system. Sun SPOTs are wireless motes developed by Sun Microsystems (today Oracle) that embed a small Java Micro Edition virtual machine called Squawk. The Sun SPOT v6 integrates a 400MHz CPU (AT91SAM9G20) and one megabyte of memory. These motes are a perfect example of averagely powerful Things (or *average Things* for short) that, from our perspective, will compose the future IoT/WoT (average power, but modern execution environment) and that are targeted by our middleware. The same customized middleware (~209 KB) is deployed on both the Spot and the phone and embeds all the modules, except the compression and cryptography plugins that are not used during the experiments.

5.1 Stream Serving Experiment

Our first experiment analyzes the ability of the *Dioptase* middleware to efficiently serve streams. Toward that end, a producer is deployed on the Thing and acquires data from the embedded light sensor. Every 500 milliseconds, the producer performs a new measurement and sends it (~100 B/s) to each consumer connected to the component. Each consumer is deployed on a standard computer and, because the Spot and the phone have very different capabilities, the number of clients is different between the experiments. All the experiments generate raw data directly

[5] We are finalizing the prototype for release. The current version is made available to reviewers at `http://www.rocq.inria.fr/arles/index.php/component/content/article/248` (last access: 10-14-2014). The source code is password-protected: *dioptase_inria_rev*.

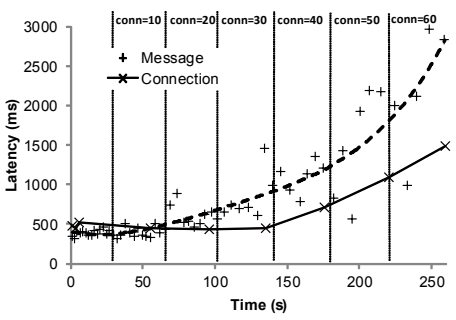


Figure 10: Latency (Spot).

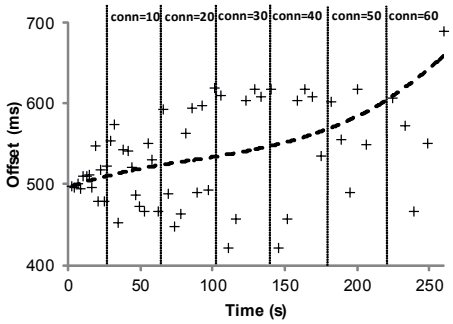


Figure 11: Jitter (Spot).

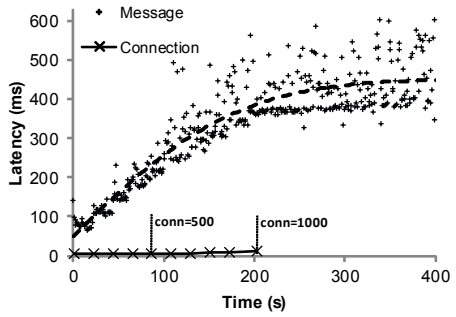


Figure 12: Latency (phone).

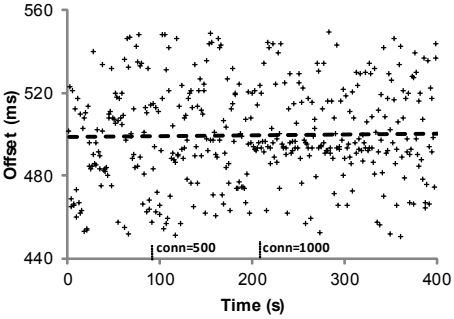


Figure 13: Jitter (phone).

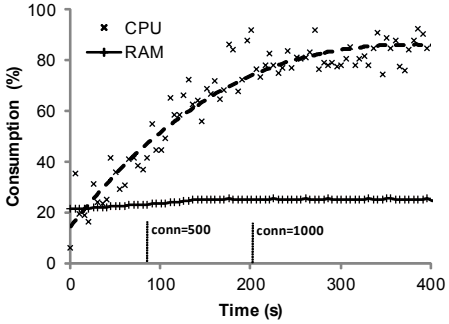


Figure 14: CPU/RAM consumption (phone).

into the devices' storages (the phone and the Spot embed two flash storages of respectively 16 GB and 4 MB). These data are retrieved and processed *a posteriori*. Before the beginning of the experiment, time informations are broadcasted (UDP) to synchronize the internal clock of each device; time error is less than ten milliseconds.

As depicted in Figure 9, communication between clients and the Spot is done through a base station that is used as a router between the Ethernet network and the radio IEEE 802.15.4 network. The experiment is run in two phases: (i) every 2 seconds, the client opens a new connection to the Spot, with a limit of 10 connections, then (ii) every 40 seconds, the client opens 10 new connections in order to stress the device. The connection's opening time, the time interval between

two messages (jitter), and the time between the production and reception of a light measurement (including the transmission time and the middleware processing time) are collected. Figure 10 presents the average time used to open a connection and the latency between the production and the consumption of a stream item. Figure 11 shows the latency between two stream items. Ideally, this time should stay close to the production interval time (i.e., 500 ms).

The phone experiment is done through a direct WiFi 802.11g connection (access point). The same data as in the previous experiment are collected. However, the connection's opening phases are slightly different to take into account the higher capability of the phone. The experiment starts with 100 established connections and, every 20 seconds, 100 new connections are opened with a limit of 1000. Figures 12 and 13 show the same information as the previous Spot experiment. Unlike with Spots, it is possible to read data about CPU and memory consumption, using the system files */proc/stat* and */proc/meminfo*. Figure 14 presents these measures, acquired every 5 seconds (this long duration was chosen in order to avoid influencing the other readings).

As expected, the devices resources decrease as the number of connections increases, up to a critical threshold that is clearly visible in Figure 10. After around 40 connections, the latency increases significantly (packet loss and resent many times) and, as a consequence of the Thing's overload, the jitter grows quickly (Figure 11). For smart Things, we can see that even with 1000 connections, network and resource usage stay stable, as shown in Figure 12. These results on smart Things are very encouraging, with regard to Web-based DSMSs' performances [11, 35], which makes *Dioptase* a good solution for data streaming, with the benefit of advanced stream processing capabilities.

Assessing the performances of our middleware against other DSMSs is actually extremely difficult as the classes of Things and the criteria considered in other work are very different. *Dioptase* is designed to run on average Things, provides an in-network interpretation mechanism, and presents sensed data as embedded streaming Web services. These features are unique and cannot be compared to existing DSMSs. WSAN-based DSMSs typically focus on energy consumption for tiny Things but not on the ability to handle many heterogeneous parallel tasks. In contrast, Web-based DSMSs focus on smart Things, powerful servers, desktop computers or even the cloud. As a consequence, average Things provide inferior performances, in terms of simultaneous connections and processing speed.

Still, it is worth highlighting that the capability for an average Thing to serve around 30 streams of two measurements per second with a limited latency (< 500 ms) is, in absolute terms, suitable for most of the envisioned IoT/WoT scenarios [2]. For example, let us consider the scenario of the SmartPark project[6], where informations about parking space availability are collected in order to synchronize and guide the drivers toward free parking spots. Specifically, each vehicle is equipped with a wireless communication device and exchanges informations with the Things (presence sensors) that are deployed at each parking spot. In this case, if each of these Things handles 30 streams, as shown in our performance experiments, it enables the entire parking network to manage and process thousands of streams

[6] `http://smartpark.epfl.ch` (last access: 10-14-2014)

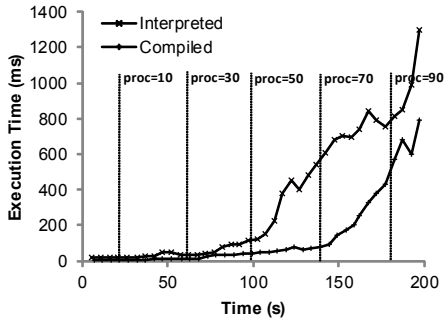


Figure 15: Processing time (Spot).

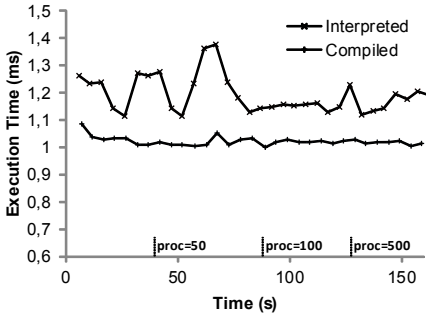


Figure 16: Processing time (phone).

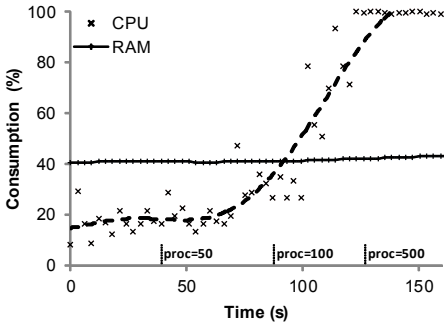


Figure 17: Interpreted joins (phone).

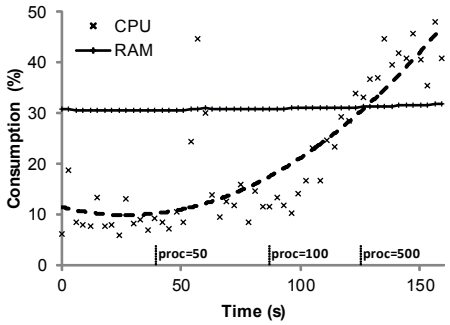


Figure 18: Compiled joins (phone).

(which is clearly more than necessary for this scenario). In addition, as in WSAN work, limiting the amount of data exchanged between Things is a goal of the IoT due to the energy constraints. In-network processing, compression and approximation are therefore used to ensure that only strictly useful data are exchanged by Things, alleviating the need for many simultaneous data streams.

5.2 Stream Processing Experiment

Our second experiment assesses the capability of *Dioptase* to support dynamic deployment of tasks, by evaluating the resource consumption of processors for compiled and interpreted tasks. The chosen task is a hash-based pipelined inner join [56], which is applied many times in parallel on two light streams produced by two different sensors: the light sensor local to the Thing, and a light sensor available from another Spot. As in the first set of experiments, the producer reads the light sensors every 500 ms.

The pipelined inner join requires a memory space that grows proportionally to the size of the input streams. The operator is implemented using one hash table per stream. When a new item x is received from an input stream, the operator checks if it is present in the tables of the other streams. If it is, the item is written in the output stream and stored in the related table.

The Spot experiment is run in two steps, both for compiled and interpreted joins: (i) every 5 seconds a new processor is deployed, with a limit of 5 processors, then (ii) 10 new processors are deployed every 40 seconds. As we said before, we can not acquire the memory and CPU consumption on Spots and, as a consequence,

we measure only the time spent by each processor to run its work section. This time is an image of the real resource consumption, as it increases if the memory and the CPU are overloaded. Figure 15 shows the average execution time, and the experiment is stopped when the Thing load becomes too high (after around one hundred processors).

The phone experiment starts with 10 processors and, every 10 seconds, 10 new processors are deployed. When the Thing reaches 100 processors, 100 new processors are deployed every 10 seconds. Like the previous one, this experiment is run for compiled and interpreted joins. In addition to the execution time presented in Figure 16, we get information about resource consumption shown in Figures 17 (interpreted) and 18 (compiled).

Interpreted joins are of course more expensive than compiled ones, because of the depth-first search of the AST. The figures show that the interpreted join consumes approximately twice as much CPU as the compiled join. However, the execution on the phone is very efficient, with a pretty low difference between the two operators (approximately forty microseconds in the worst case, where some peaks are a consequence of the garbage collector). On the Spot, the Thing is overloaded with 60 interpreted joins and 90 compiled joins. These results are not a CPU problem, which is oversized for these types of operations, but a problem of memory, which is quickly full (especially because of the AST that requires more space than the hash tables).

The results obtained are satisfying, but are also difficult to compare to other DSMSs as, to the best of our knowledge, other DSMSs for constrained devices do not manage fully-dynamic tasks. The pipelined inner join is an expensive operation that consumes CPU and memory continuously, far more than other operations like counting or filtering that are computed in constant time and space. Relatively to the scenarios presented in [2], the *Dioptase* ability to run around sixty complex interpreted operations (respectively ninety compiled ones) in parallel on a single resource-constrained Thing is perfectly compatible with the needs of the IoT/WoT.

6 Conclusion

The IoT and related WoT are expected to become significant enablers of pervasive computing given the interaction with the physical world that they promote. However, numerous obstacles must be overcome by judiciously combining the knowledge acquired from the various visions involved rather than trying to reinvent the wheel.

In this paper, we presented *Dioptase*, a middleware that aims at simplifying building complex mashups based on the multiple data sources of the WoT. *Dioptase* makes it possible to integrate Things, even averagely powerful ones, with the Web and enables them to produce, process and store data streams dynamically. Each Thing, and by extension the entire network, is then seen as a consistent entity, dedicated to the (complex) processing of sensed data, and able to dynamically run tasks written in a DSL called *DiSPL*. This language aims to be simple, but flexible enough to describe such advanced operations and, for interoperability concerns, we plan to write converters from state-of-the-art stream processing languages like SPL [52] or C-SPARQL [57]. We demonstrated that the *Dioptase* middleware can avoid the systematic use of centralized or partially centralized infrastructures,

which are commonly used in WSAN-based DSMSs. In addition, we have shown that *Dioptase* is efficient enough to be deployed on average Things w.r.t. the IoT/WoT needs and use cases, and enables these Things to be integrated in the Web despite the additional complexity of data streaming communication.

Dioptase remains a work in progress. Our work can be first improved in a technical way, especially by enhancing the efficiency of the interpreter or integrating other continuous operators. However, we are more interested in dealing with many other IoT/WoT research problems. First, making each Thing an entity of generic processing is a first step toward simplifying the deployment and the distribution of applications within the WoT networks. The next step is to study how to manage security in this context of dynamic deployment, to avoid making the WoT a wide area of chaos. Access control, encryption, identification/authentication and trust management are the security aspects that must be studied in the future, reusing the existing state of the art technologies for security and privacy [1, 2]. The problems of integrating the semantic Web and enabling Things to collaborate and use public and shared knowledge (ontologies, knowledge base) are still active areas of research, as well as adaptation to unknown cases (overloaded network, breakdowns, transient errors, etc.). We plan, for example, to enable Things to automatically delegate, adapt and split their own tasks according to their environment, their load, their available resources and their capabilities. Finally, small Things must not be ignored in the IoT, of which they are a significant part. Since a lot of Things are mobile, average and smart Things can act opportunistically as gateways and proxies for very resource-limited Things (e.g., RFID chips or small embedded sensors). By presenting small Things as resources of average and smart Things, we want to enable developers to transparently query resource-limited Things in a similar way they query smart Things. In addition, we are working on a prototype of *Dioptase* for the Contiki[7] operating system, in order to integrate more devices to our research.

Competing interests
VI is member of the JISA editorial board. In addition, we may have conflicts of interests with the following members of the board: Gordon Blair, Fabio Kon, Serge Fdida, Gang Huang, Michel Hurfin, Wouter Joosen, Tiziana T Margaria-Steffen.

Author's contributions
In the context of his PhD, BB conducted the research, developed the prototype and performed the experiments. VI provided a continuous scientific feedback, was involved in the revision process and participated in the design of the experiments. All authors read and approved the final manuscript.

Acknowledgements
VI and BB are employed by Inria, the french national institute for research in computer science.

References
[1] Gubbi J, Buyya R, Marusic S and Palaniswami M (2013) Internet of Things (IoT): A vision, architectural elements, and future directions. Future Generation Computer Systems 29(7).

[2] Atzori L, Iera A and Morabito G (2010) The Internet of Things: A survey. Computer Networks 54(15).

[3] Teixeira T, Hachem S, Issarny V and Georgantas N (2011) Service oriented middleware for the Internet of Things: A perspective. In: ServiceWave '11. Proc. of the 4th European conference on Towards a service-based internet.

[4] Mottola L and Picco G. P (2011) Programming wireless sensor networks: Fundamental concepts and state of the art. ACM Computing Survey 43(3).

[7] `http://www.contiki-os.org` (last access: 10-14-2014).

[5] Garofalakis M, Gehrke J and Rastogi R (2007) Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications). Springer, New York.

[6] Silva Neves P. A. C and Rodrigues J. J. P. C (2010) Internet Protocol over Wireless Sensor Networks, from myth to reality. Journal of Communications 5(3).

[7] Golab L and Özsu M. T (2010) Data Stream Management. Synthesis Lectures on Data Management 2(1).

[8] Dezfuli M. G and Haghjoo M. S (2012) Probabilistic Querying over Uncertain Data Streams. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 20(05).

[9] Dezfuli M and Haghjoo M (2012) Xtream: A System for Continuous Querying over Uncertain Data Streams. In: Scalable Uncertainty Management. Springer, Berlin.

[10] Guinard D and Trifa V (2009) Towards the Web of Things: Web Mashups for Embedded Devices. In: WWW '09. Proc. of the 18th International World Wide Web Conferences.

[11] Trifa V, Guinard D, Davidovski V, Kamilaris A and Delchev I (2010) Web Messaging for Open and Scalable Distributed Sensing Applications. In: Proc. of the 10th international conference on Web engineering. Springer, Berlin.

[12] Ishaq I, Carels D, Teklemariam G. K, Hoebeke J, Abeele F. V. d, Poorter E. D, Moerman I and Demeester P (2013) IETF Standardization in the Field of the Internet of Things (IoT): A Survey. Journal of Sensor and Actuator Networks 2(2).

[13] Hui J. W and Culler D (2004) The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In: SenSys '04. Proc. of the 2nd International Conference on Embedded Networked Sensor Systems.

[14] Leontiadis I, Efstratiou C, Mascolo C and Crowcroft J (2012) SenShare: Transforming Sensor Networks into Multi-application Sensing Infrastructures. In: Wireless Sensor Networks. Springer, Berlin.

[15] Corredor Pérez I and Bernardos Barbolla A. M (2014) Exploring Major Architectural Aspects of the Web of Things. In: Internet of Things. Springer, Berlin.

[16] Carroll A and Heiser G (2010) An Analysis of Power Consumption in a Smartphone. In: USENIX '10. USENIX annual technical conference.

[17] Rao B, Saluia P, Sharma N, Mittal A and Sharma S (2012) Cloud computing for Internet of Things amp; sensing based applications. In: ICST '12. Proc. of the 6th International Conference on Sensing Technology.

[18] Mohapatra S, Majhi B and Patnaik S (2014) Sensor Cloud: The Scalable Architecture for Future Generation Computing. In: Intelligent Computing, Networking, and Informatics. Springer India.

[19] Xu N, Rangwala S, Chintalapudi K. K, Ganesan D, Broad A, Govindan R and Estrin D (2004) A Wireless Sensor Network For Structural Monitoring. In: SenSys '04. Proc. of the 2nd International Conference on Embedded Networked Sensor Systems.

[20] Kovatsch M, Lanter M and Duquennoy S (2012) Actinium: A RESTful runtime container for scriptable Internet of Things applications. In: IOT '12. Proc. of the 3rd International Conference on the Internet of Things.

[21] Pérez J. L, Villalba A, Carrera D, Larizgoitia I and Trifa V (2014) The COMPOSE API for the Internet of Things. In: WWW Companion '14. Proc. of the Companion Publication of the 23rd International Conference on World Wide Web Companion.

[22] Demirbas M, Yilmaz Y and Bulut M (2013) Eywa: Crowdsourced and cloudsourced omniscience. In: PerCom '13. Proc. of the 11th International Conference on Pervasive Computing and Communications Workshops.

[23] Kovatsch M, Mayer S and Ostermaier B (2012) Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things. In: IMIS '12. Proc. of the 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing.

[24] Hadim S and Mohamed N (2006) Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks. Distributed Systems Online 7(3).

[25] Madden S. R, Franklin M. J, Hellerstein J. M and Hong W (2005) TinyDB: an acquisitional query processing system for sensor networks. ACM Transactions on Database Systems 30(1).

[26] Yao Y and Gehrke J (2002) The cougar approach to in-network query processing in sensor networks. ACM SIGMOD Record 31(3).

[27] Amato G, Chessa S and Vairo C (2010) MaD-WiSe: a distributed stream management system for wireless sensor networks. Software: Practice and Experience 40(5).

[28] Abadi D. J, Ahmad Y, Balazinska M, Cetintemel U, Cherniack M, Hwang J.-H, Lindner W, Maskey A. S, Rasin A, Ryvkina E, Tatbul N, Xing Y and Zdonik S (2005) The design of the Borealis stream processing engine. In: CIDR '05. Proc. of the Conference on Innovative Data Systems Research.

[29] Arasu A, Babcock B, Babu S, Cieslewicz J, Datar M, Ito K, Motwani R, Srivastava U and Widom J (2004) STREAM: The Stanford data stream management system. .

[30] Amato G, Chessa S, Gennaro C and Vairo C (2014) Querying moving events in wireless sensor networks. Pervasive and Mobile Computing .

[31] Le-Phuoc D, Xavier Parreira J and Hauswirth M (2012) Linked Stream Data Processing. In: Reasoning Web. Semantic Technologies for Advanced Query Answering. Springer, Berlin.

[32] Newton R, Morrisett G and Welsh M (2007) The regiment macroprogramming system. In: IPSN '07. Proc. of the 6th international conference on Information processing in sensor networks.

[33] Whitehouse K, Zhao F and Liu J (2006) Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data. In: Proc. of the 3rd European conference on Wireless Sensor Networks. Springer, Berlin.

[34] Szczodrak M, Gnawali O and Carloni L (2013) Dynamic Reconfiguration of Wireless Sensor Networks to Support Heterogeneous Applications. In: DCOSS '13. Proc. of the 9th International Conference on Distributed Computing in Sensor Systems.

[35] Dickerson R, Lu J, Lu J and Whitehouse K (2008) Stream Feeds: An Abstraction for the World Wide Sensor Web. In: The Internet of Things. Springer, Berlin.

[36] Grosky W, Kansal A, Nath S, Liu J and Zhao F (2007) SenseWeb: An infrastructure for shared sensing. IEEE Multimedia 14(4).

[37] Le-Phuoc D, Nguyen-Mau H. Q, Parreira J. X and Hauswirth M (2012) A middleware framework for scalable management of linked streams. Web Semantics: Science, Services and Agents on the World Wide Web 16.

[38] Ostermaier B, Schlup F and Römer K (2010) WebPlug: A framework for the web of things. In: PERCOM '10. Proc. of the 8th International Conference on Pervasive Computing and Communications Workshops.

[39] Lam G and Rossiter D (2012) A Web Service Framework Supporting Multimedia Streaming. IEEE Transactions on Services Computing PrePrints(99).

[40] Hachem S, Pathak A and Issarny V (2013) Probabilistic Registration for Large-Scale Mobile Participatory Sensing. In: PERCOM '13. Proc. of the 13th International Conference on Pervasive Computing and Communications.

[41] Sahu P. K and Chattopadhyay S (2013) A survey on application mapping strategies for Network-on-Chip design. Journal of Systems Architecture 59(1).

[42] Billet B and Issarny V (2014) From Task Graphs to Concrete Actions: A New Task Mapping Algorithm for the Future Internet of Things. In: MASS '14. Proc. of the the 11th IEEE International Conference on Mobile Ad hoc and Sensor Systems.

[43] Golab L and Özsu M. T (2003) Issues in data stream management. ACM SIGMOD Record 32(2).

[44] Hachem S, Teixeira T and Issarny V (2011) Ontologies for the Internet of Things. In: Middleware '11. Proc. of the 8th Middleware Doctoral Symposium.

[45] Loreto S, Saint-Andre P, Salsano S and Wilkins G (2011) RFC 6202 - Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. http://tools.ietf.org/html/rfc6202 (last access: 10-14-2014).

[46] Fette I and Melnikov A (2011) RFC 6455 - The WebSocket Protocol. http://tools.ietf.org/html/rfc6455 (last access: 10-14-2014).

[47] Law Y.-N, Wang H and Zaniolo C (2004) Query languages and data models for database sequences and data streams. In: VLDB '04. Proc. of the 13th international conference on Very Large Data Bases.

[48] Raza U, Camerra A, Murphy A, Palpanas T and Picco G (2012) What does model-driven data acquisition really achieve in wireless sensor networks ?. In: PerCom '12. Proc. of the International Conference on Pervasive Computing and Communications.

[49] Duquennoy S, Grimaud G and Vandewalle J.-J (2009) The Web of Things: Interconnecting Devices with High Usability and Performance. In: ICESS '09. Proc. of the International Conference on Embedded Software and Systems.

[50] Mitzel D (2000) RFC 3002: Overview of 2000 IAB wireless internetworking workshop. http://tools.ietf.org/html/rfc3002 (last access: 10-14-2014).

[51] Masinter L (1998) RFC 2388: Returning Values from Forms: multipart/form-data. http://tools.ietf.org/html/rfc2388 (last access: 10-14-2014).

[52] (2012) IBM Streams Processing Language Specification. http://pic.dhe.ibm.com/infocenter/streams/v2r0/topic/com.ibm.swg.im.infosphere.streams.product.doc/doc/IBMInfoSphereStreams-SPLLanguageSpecification.pdf (last access: 10-14-2014).

[53] Sperber M, Dybvig R. K, Flatt M, Van Straaten A, Findler R and Matthews J (2009) Revised Report on the Algorithmic Language Scheme. Journal of Functional Programming 19.

[54] Kirsch A and Mitzenmacher M (2008) Less Hashing, Same Performance: Building a Better Bloom Filter. Random Struct. Algorithms 33(2).

[55] Maye O and Maaser M (2013) Comparing Java Virtual Machines for Sensor Nodes. In: Grid and Pervasive Computing. Springer, Berlin.

[56] Wilschut A and Apers P. M. G (1990) Pipelining in query execution. In: PARBASE '90. Proc. of the International Conference on Databases, Parallel Architectures and Their Applications.

[57] Barbieri D. F, Braga D, Ceri S, Valle E. D and Grossniklaus M (2010) C-SPARQL: A continous query language for RDF data streams. International Journal of Semantic Computing 4(1).

Appendix A: A Bloom Filter implementation using DiSPL.

```
init:                                                                                                   1
    (define bitsetSize 32)                                                                              2
    (define bitset (bitword bitsetSize false))                                                          3
    (define nbBuckets 8)                                                                                4
    (define nbItems 0)                                                                                  5
work:                                                                                                   6
    ; first, update the Bloom filter with the next pending item                                         7
    (define item (getNextItem "itemStream"))                                                            8
    (if (notnull item)                                                                                  9
        ((define hash (murmur3 128 2 item))                                                             10
        (for 0 to nbBuckets                                                                             11
            ((define index (% (abs (+ (get hash 0) (* i (get hash 1)))) bitsetSize))                    12
            (set bitset index true))                                                                    13
        )                                                                                               14
        (increment nbItems 1)                                                                           15
                                                                                                        16
        ; write the probability of false positive into the corresponding output                        17
        (define p (pow (- 1 (exp (* (- nbBuckets) (/ nbItems bitsetSize)))) nbBuckets))                 18
        (write "probaStream" (item (now) "proba" p)))                                                   19
    )                                                                                                   20
                                                                                                        21
    ; second, perform the presence test if a new item has to be checked                                22
    (define request (getNextItem "requestStream"))                                                      23
    (if (notnull request)                                                                               24
        ((define hash (murmur3 128 2 request))                                                          25
        (define present true)                                                                           26
        (for 0 to nbBuckets                                                                             27
            ((define index (% (abs (+ (get hash 0) (* i (get hash 1)))) bitsetSize))                    28
            (if (not (get bitset index))                                                                29
                ((set present false)                                                                    30
                (break))                                                                                31
            ))                                                                                          32
        )                                                                                               33
        (if present                                                                                     34
            (write request)                                                                             35
        ))                                                                                              36
    )                                                                                                   37
```