The COMPOSE API for the Internet of Things

Juan Luis Pérez, Álvaro Villalba and David Carrera
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya - BarcelonaTech (UPC)
{juan.perez,alvaro.villalba,david.carrera}@bsc.es

Iker Larizgoitia and Vlad Trifa
Evrythng
{iker,vlad}@evrythng.com

ABSTRACT

The COMPOSE project aims to provide an open Marketplace for the Internet of Things as well as the necessary platform to support it. A necessary component of COMPOSE is an API that allows *things*, COMPOSE users and the platform to communicate. The COMPOSE API allows for *things* to push data to the platform, the platform to initiate asynchronous actions on the *things*, and COMPOSE users to retrieve and process data from the *things*. In this paper we present the design and implementation of the COMPOSE API, as well as a detailed description of the main key requirements that the API must satisfy. The API documentation and the source code for the platform are available at [9].

1. INTRODUCTION

The Internet of Things (IoT) is composed of objects, either connected to the Internet or not. The COMPOSE project aims to provide a technological platform for easily creating services based on the Internet of Things (IoT), thus unleashing the full potential of an Internet of Services (IoS) based on the IoT. As a consequence the simplification of the ingestion (get data produced by a sensor stored and processed on real time), advertisement (allow for discovery of objects), location (transparently reaching devices) and composition (aggregate data of multiple devices) of Internet-connected objects lies in the forefront of the COMPOSE requirements. The COMPOSE project brings a number of components, such as a GUI, a semantic registry, a cloud runtime, and some communication libraries among others. In this paper we focus on the COMPOSE data plane [9], which is responsible for managing data storage and processing.

The main focus of the COMPOSE data plane is to provide a rich set of features to store and process data through a simple REST API, allowing objects, services and humans to access the information produced by the devices connected to COMPOSE. The platform described on this paper allows for a real time processing of device-generated data, and enables for simple creation of data transformation pipelines using user generated logic. Unlike traditional service composition approaches, usually focused on addressing the problems of functional composition of existing services, one of the goals of the COMPOSE data plane is to focus on data processing scalability. Other components within the COMPOSE project provide added capabilities to automatically create compositions of high-level services using existing tools [18].

WWW'14 Companion, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04. http://dx.doi.org/10.1145/2567948.2579226.

The approach to the IoT taken in COMPOSE is the one known as the Web of Things (WoT), where objects are able not only to communicate among themselves, but also they do so utilizing standard web-enabled protocols. Therefore, to communicate with the COMPOSE platform, objects need to be web-enabled (either directly or by using a proxy), and they have to implement the COMPOSE specific protocol to be integrated in the platform.

The Web of Things is a viable solution to build more scalable, open, and flexible IoT applications for various reasons: First, native integration of devices (as opposed to only integrate their data or using a Web page to control them) allows treating devices and their services just like any other Web resource; Second, native integration diminishes the costs to network heterogeneous devices as the Web infrastructure is already in place: HTTP is a highly versatile and omnipresent protocol thanks to its simplicity, powerful and scalable Web servers are freely available as open-source projects, HTTP clients and libraries exist for virtually any programming language and platform; And finally, Web applications are often simpler and faster to develop than classic desktop software applications. Current software for real-world integration and business applications are tailored for specific use cases, thus are often too rigid and closed to be customized by end-users easily.

Any Object which is web-enabled and implements the COMPOSE communication protocol is known as a Web Object (WO). All WOs will hold a virtual identity in COMPOSE, named a Service Objects (SO). SOs are standard internal COMPOSE representations of physical objects. COMPOSE specifies a SO API to communicate with WOs through the platform. The SO API is also exposed internally towards the rest of the components within the COMPOSE platform.

Service Objects can be used as mere data endpoints for WOs, but at the same time they can be used to aggregate and combine data coming from different SOs, creating dataflows that are exposed to the users. For instance, a Service Object could be created and deployed to represent aggregated operations across other existing SOs (e.g. calculating the maximum temperature across a large number of existing sensors registered in the platform).

Figure 1 provides a visual description of the different entities involved in the COMPOSE platform, where *Things*, end users and developers create an IoT ecosystem based on the Internet of Services. In this paper we will focus on the Service Objects abstraction, describing the API that gives access to them as well as the design and implementation of the back-end that sits behind the API front-end.

The API documentation and the source code for the platform are available at [9].

The rest of the paper is structured as follows: Section 2 describes de core virtual identity in COMPOSE, which are the Service Objects; Section 3 elaborates on the design of the COMPOSE platform;

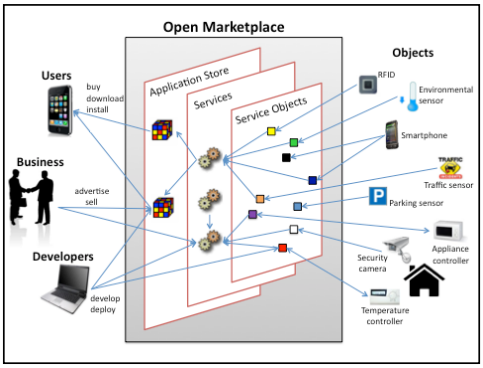

Figure 1: COMPOSE platform

Section 4 introduces the Service Objects API; Section 5 describes the processing pipeline that resides at the core of Service Objects; Section 6 discusses the implementation of the API operations; and finally Section 7 provides a brief discussion on existing related work, and Section 8 provides some conclusions on the presented work.

2. SERVICE OBJECTS

COMPOSE assumes that each physical object (e.g. a smartphone) has different sensors (temperature, GPS, accelerometer...), and each sensor produces data that has one to several dimensions (e.g. wind speed has two dimensions: direction and strenght). Service Objects are the virtual representation of a physical object, and they exist in the COMPOSE platform. Every time one of the sensors of the physical object produces a new reading, it results in a *sensor update* (SU) being sent to the COMPOSE platform, and it is digested and processed by the Service Object associated to the physical object. In this virtual representation, each sensor is mapped to a sensor *stream*, and each sensor update contains a set of dimensions, known as *channels* in COMPOSE. So a Service Object can be seen as a virtual entity which has different streams (as many as sensors) and each stream produces updates composed of a tuple of channels.

Each SO deployed in the COMPOSE platform is internally stored and described as a JSON document which contains all the necessary information to provide the data processing logic encapsulated in the SO. The processing logic is expressed using basic logical, string and arithmetic operators. The sources of data for the data processing logic can be Sensor Updates generated by a WO or SO, as well as the result of queries for data stored in the back-end. The communication between the different SOs in the data pipeline is driven by events, and the connections between them to create data paths is built through the use of subscriptions. SOs are deployed into the COMPOSE platform and later on accessed using a RESTful API.

Figure 2 illustrates an example in which one Web Object connected to the COMPOSE platform pushes data to be stored and processed. The WO is a SmartPhone with GeoLocation capabilities (e.g. GPS enabled). The WO sends a Sensor Update (SU) to the platform everytime that wants to get its position reported. The SU is received in the platform through its virutal counter-part: the Service Object (SO). The COMPOSE web-based protocol is used through the SO API to push the SU into the platform. At that point, the platform determines that there is another SO which is subscribed to the data produced by the SmartPhone. As such, it is subscribed to the data ingested by the first SO. The platform takes care of forwarding the SU to the second SO in the pipeline. The functionality of the second SO is to provide GeoFencing (e.g. determining whether a device is inside of a virtual fence defined by four different geographic

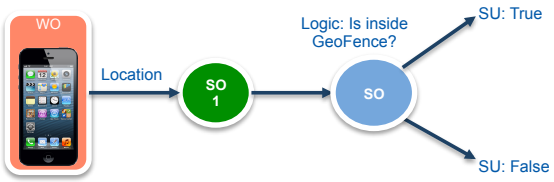

Figure 2: Service Objects example

coordinates). When the second SO gets the SU forwarded, it runs the SU information through its processing logic pipeline and emits another SU, this time containing a boolean value that takes value *true* when the SmartPhone position is located within the GeoFence. Note that this second SO provides data processing logic within the COMPOSE data plane, and that any external users, services or entities can decide to subscribe the second SO to obtain GeoFencing information about the SmartPhone.

The description of the SO includes a processing pipeline composed of different stages that need completed everytime that a Sensor Update is received by the SO. They include input and output data filtering, queries to the back-end and data transformation among others. The COMPOSE platform is responsible to parse the SO description and turn it into a computing entity within the data ingestion pipeline. Internally, the SO logic can access the data contained in the SUs through JSONPath [13] expressions.

When a new SO is deployed it will is connected to the other SOs to which it is subscribed to build new data paths, and depending on its definition, a background process will be initiated to compute the expected SO outputs for any historical data that is available in the platform for the SOs to which it is connected. This background process will generate the data following a lazy approach and always according to the data processing modifiers found in the SO description. Such modifiers can be leveraged by the COMPOSE platform administrators according to their business models, as well as by privacy observers to regulate and enforce data storage policies.

3. DESIGN PRINCIPLES

Service Objects can subscribe to data streams from other SOs and perform some computation on them, following the data processing logic provided by COMPOSE developers. Their output is also a set of data streams that can be accessed by other Service Objects as well. The streams generated from a SO can be seen as a set of variables whose values are calculated in function of other SO streams. If a SO stream is a source of data for another SO stream, whenever the source stream emits a new Sensor Update (SU) hich actually is a JSON document according to the Web Streams Protocol defined hen the SO streams will update their value and potentially emit a new SU. Internally in the COMPOSE platform a SO is a description document (JSON) that defines the behaviour of the SO as a response to new inputs. The SU produced by a SO is derived from its inputs by using algebraic, boolean, array and string operations. The COMPOSE GUI and SDK will provide means to create SO description documents within the COMPOSE platform using visual tools. The composition of different data sources will take place through the GUI and will directly translate into the SO elements that will be later on parsed and processed in real-time by the platform.

The design of the SO is intended to be extremely scalable. As such, it follows three key design principles: it is event-driven, lock-free and stream-oriented. To design a system that is event-driven and stream-oriented, each SO is responsible to produce one output (a SU) every time that one input (another SU) is received.

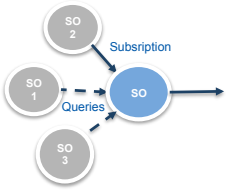

Figure 3: SO data flow design model

From the point of view of the COMPOSE platform, the SUs that any SO generate are spontaneous, as they are triggered by events generated from outside the platform by WOs that are not under control by COMPOSE. This fact poses a design challenge, as multiple SUs (inputs) are required by one SO to produce another SU (output), while not all inputs will reach the SO synchronously because the WOs are not synced in any way. Even more, the SOs that will trigger the execution of multiple SOs may not even know the existence of other SOs needed by the SOs found deeper in the pipeline. See for instance the case presented in Figure 3, in which one SO produces outputs that are based on the information generated by three different SOs. The SO is subscribed to the three SOs, so whenever one of the three SOs produces a new SU (as a result of a WO sending data to the platform), the SU will be forwarded to the SO that in turn will have to produce another SU, generated as a result of the customized data processing logic embedded in the SO. Any of the SOs can be triggering the action (SO1, SO2 or SO3), but data from all of them will be needed to produce the output. As it is desired to keep the system lock-free, the data flow model proposed in this document assumes that the SO will have two different sources of data: SUs originated somewhere within the COMPOSE platform and forwarded to the SO, and data being requested by the SO to other components of the COMPOSE platform. The former means that one SO is subscribed to other SOs, which in turn generate SUs that are forwarded to the subscribed SO. The latter means that the SO can pull data stored for other COMPOSE SOs that will be used to produce the output. Both models are needed to produce an output. As an example, Figure 5 illustrates the case of a SO that needs to access data from three different SOs to produce an output. In particular, in the case that a SU from SO2 is received, the SO will query data associated to SO1 and SO3 to produce a new SU.

Each SO deployed in the COMPOSE platform is internally stored and described as a JSON document which contains all the necessary information to provide the data processing logic encapsulated in the SO. The processing logic is expressed using basic logical, string and arithmetic operators. The sources of data for the data processing logic, as it was discussed before, can be Sensor Updates generated by a different SO, as well as the result of queries for data stored in the back-end. The communication between the different SOs in the data pipeline is driven by events, and the connections between them to create data paths is built through the use of subscriptions.

The description of the SO includes a processing pipeline composed of different stages that need completed everytime that a Sensor Update is received by the SO. They include input and output data filtering, queries to the back-end and data transformation among others that will be described in more detail later in this document. The COMPOSE platform is responsible to parse the SO description and turn it into a computing entity within the data ingestion pipeline. Internally, the SO logic can access the data contained in the SUs through JSONPath [13] expressions.

When a new SO is deployed it will be connected to the SOs to which it is subscribed to build new data paths, and depending on its definition, a background process will be initiated to compute the expected SO outputs for any historical data that is available in the platform for the SOs to which it is connected. This background process will generate the data following a lazy approach and always according to the data processing modifiers found in the SO description. Such modifiers can be leveraged by the COMPOSE platform administrators according to their business models, as well as by privacy observers to regulate and enforce data storage policies.

In the process of data transformation, the COMPOSE platform will track the data manipulation actions that take place and will enrich the data being ingested with provenance information that will later be leveraged to enforce security and privacy rules.

operation	***Target URI***	***Role***
Create	POST /	Create a new SO posting a JSON document.
Retrieve	GET /	Retrieve the list of all the SOs created.
Retrieve	GET /<soId>	Retrieve attributes from the <SO_Id> Service Object.
Update	PUT /<soId>	Modify the <SO_Id> ServiceObject.
Delete	DEL /<soId>	Delete the <SO_Id> Service Object.
Retrieve	GET /<soId> /streams	Retrieve the list of all the SO streams.
Create	POST /<soId> /streams/ <streamId> /subscriptions	Subscribe the Service Object <soId> to a service posting a subscription JSON document.
Update	PUT /<soId> /streams/ <streamId> /store.data	Store <soId> data puting a JSON document.
Retrieve	GET /<soId> /streams/ <streamId>	Retrieve the list of all the data of <soId> Service Object.

Table 1: API operations

4. API

Service Objects are exposed through a RESTful API that uses HTTP as a transport and that acts as the SO front-end. This basically implies that SOs can be identified unambiguously using uniques URIs. The API provides resource actuations through the four main HTTP operations: GET (retrieve), POST (create), PUT (update) and DELETE. Table 1 summarizes the COMPOSE Service Objects API. In the table, soId represents the unique ID associated to each Service Object registered in the platform.

SOs are created by POSTing a JSON document to the / resource. The document is a basic description of the main properties of the Service Object about to be created. The following example illustrates the case of a SmartPhone object enabled with three different sensors (GPS location, Microphone and Temperature Sensor), each one becoming a stream of data in the SO abstraction. The device is also presenting the capability to be activated through the platform: when the *vibrate action* is invoked on it, the device will vibrate to notify something to the user carrying it. Note that some other fields exist in the complete version of the Service Object description, but it have been shortened in this example for the sake of clarity.

```
POST /
Accept: application/json
Content-type: application/json
```

```
{ "name":"user21",
  "description":"Smartphone of user21 in deployment",
  "streams": [
      { "name":"location",
        "channels": [ {"name":"lat"}, {"name":"lon"} ],
        "description":"Outdoor location of the smartphone"
      },
      { "name":"microphone",
        "channels": [ {"name":"noise"} ],
        "description":"Quantity of noise"
      },
      { "name":"temperature",
        "channels": [ {"name":"temp"} ],
        "description":"Phone temperature"
      }
    ],
  "actions":["vibrate"]
}
```

The corresponding response message contains the field "id" that is the unique identifier of the SO within the COMPOSE platform (the <soId> specified in Table 1). The response also contains the list of the <streamsId> in the field "streams".

```
{ "id": "13740600949717e0426e5fdf34e6998aa00b865",
  "name": "user21",
  "createdAt": 1379402251644,
  "updatedAt": 1379402251644,
  "description": "Smartphone of user21 in deployment",
  "streams": ["location", "microphone", "temperature"],
  "actions": ["vibrate"]
}
```

Requesting the list of all the <soId> streams would result in the following JSON document as a response for the particular example of the SmartPhone.

```
{ "streams": [
    { "name": "location",
      "channels": ["lat", "lon"],
      "description": "The location of the samartphone."
    },
    { "name": "microphone",
      "channels": ["noise"],
      "description": "Quantity of noise"
    },
    { "name": "temperature",
      "channels": ["temp"],
      "description": "Phone temperature",
    } ]
}
```

As discussed in Section 2 a SO update is actually a JSON document containing, among other information, a tuple of values that correspond to the channels of a device sensor, what is represented as a *stream* in the SO representation.

An example of pushing data from the SmartPhone to its corresponding SO counter-part is achieved by submitting the following JSON data in the body request for the /<soId>/streams/<streamId>/store.data url. The <soId> would be obtained from the previous creation of the SO. streamId should be picked from the list of streams existing in the SO description. In this example, temperature data is being pushed to the platform. As it can be observed in the following JSON document, the information for the *temp* channel, which is associated to the temperature stream, includes the actual temperature value as well as other information such as units, update time and a series of custom fields that the WO can decide to add when generating the SU for future reference.

```
{ "channels": [
    { "name": "temp",
      "current-value": 22.58,
      "type": "numeric",
      "unit": "m/s2"
    } ],
    "name": "temperature",
    "lastUpdate": 194896800,
    "customFields": {
        "covered-period": "24h",
        "averageLastHour": 32,
        "risk": "low",
        "averageLastDay": 42
    }
}
```

To retrieve and delete a SO, the corresponding operation can be invoked for each <soId> URL. For the former a response JSON document describing the success of the operation is generated, as well as a 200 HTTP status code. For the latter, a new SO description must be associated to the PUT HTTP operation on the <soId> URL, resulting in the SO description being updated and a response generated analogously to the SO creation case described above.

Finally, retrieving data associated to a SO stream results in a JSON document containing an array of all the tuples stored for this stream.

The creation of subscriptions is at the core of the COMPOSE platform and allows for the definition of data processing paths that are followed by SUs being generated by external WOs and afterwards ingested by the platform. Subscriptions can be internal (when one SO wants to get SUs generated by other SOs to be forwarded to it), or external, when entities outside of the COMPOSE platform wants to be notified about any SUs produced by one particular SO. The following example illustrates the case of an external URL (http://external.eu/process) that wants to be used to forward SUs generated by the SO with ID <soId>. Note how the subscription document includes the URL and the HTTP method to be used to forward the data to the external entity.

```
POST /<soId>/streams/<streamId>/subscriptions HTTP/1.0
Accept: application/json
Content-type: application/json

{
"type":"http.callback",
"callbackUrl":"http://external.eu/process"
"method":"POST"
}
```

5. PROCESSING PIPELINE

One of the features the SO is the capability to transform (aggregate, merge, filter, join among other possibilities) SUs generated by one or several sources, and generate new ones as a result of the transformation. For that purpose, the declaration of the SO transformation logic is similar to the structure of a SU. It contains streams with channels, and each channel contains a so-called current-value field that represents the output value that the SO will emit after ingesting a new SU, assuming that the output is not filtered. In a SO document, the content of the current-value field is a JavaScript variable assignment using any mix of basic operator and functions from the Math object, String object, Array object, as well as shorthand conditional expressions (a = b ? true : false). The result of the assignment to current-value will always be numeric, a Boolean, a string or an array of the previous types.

Once a SU reaches a SO, it goes through a number of stages in order to transform it into a new output SU. This process of ingesting

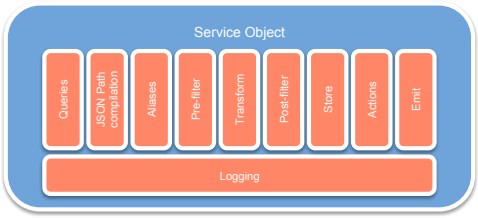

Figure 4: Service Object Processing Pipeline

a SU and processing it until a new SU is produced is the processing pipeline of the SO, which is illustrated in Figure 4.

First of all, when the SO receives a SU from a source to which it is subscribed, it may need access to the data stored for other data sources involved in the data transformation, such as other SOs. This is the querying stage, in which the SO queries its sources and makes data available for the rest of the stages, altogether with the original SU. While this stage is the first one, it may happen that queries are implicitly present in any of the other stages, as the SO may need to access data from other COMPOSE components at any stage.

The following stage is the compilation of all JSONPath expressions used in the SO description and will replace the JSONPath expressions in the SO definition with the actual data that they are pointing to in the SUs.

Next stage is the aliases replacement. Now that all the JSONPaths in the aliases values are processed, the aliases appearing in the SO definition are replaced by their values.

The definition of the SO, which depends on the incoming data, is now completed. The following stages are related only to the transformation of the incoming data to create new one. For that purpose the SO enters the pre-filtering stage, in which the SUs are discarded if the pre-filter assertion is false, and no further stages would follow. The goal of this filter is to avoid any further transformation in case that the input data is invalid.

Once the SUs have been validated then the SO can start with the data transformation process. Data transformation is performed by taking all the SUs extracted from all the SO sources, and operating on their associated data using JavaScript algebraic operations and its Math object functions, String object operations, Array object operations, and boolean operations, to finally obtain a single value for the new SU.

After transformation, another filtering step follows, the post-filtering stage. The SU resulting from the transformation is evaluated, and in case the SU does not pass the assertions the whole process halts, the data is discarded and there no more processing follows.

Finally, the generated SU gets stored and emitted to the SO subscribers. Additionally, in this final stage, actions to be sent back to SOs are triggered. Such actions will end up being sensor actuations that will be driven through the WOs that embed the actual physical objects. A comprehensive log of SO invocations will be maintained at all times.

In COMPOSE, basic physical object actuation is driven through SOs. When a SO gets an action invoked through the SO actions API, the action is initiated to on the corresponding WO, that will act as a proxy for the physical actuator. If a user needs to be able to manually request the execution of a composite action (involving multiple SOs), it is necessary to create a SO that includes the desired action and references to the individual SOs representing each of the physical objects to be actuated, so that the composite action can be properly triggered.

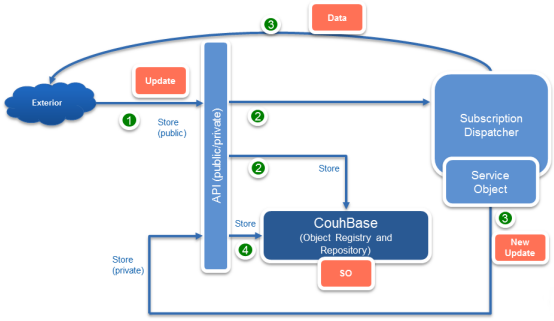

Figure 5: API implementation diagram

The following section of a SO descriptor illustrates the case of a SO that takes as inputs SU that are temperature reads in Fahrenheit degrees and produces outputs that are temperatures in Celsius degrees if and only if the temperature is below 0 °C. Note how the *current-value* of the stream is calculated first by transforming the °F into °C, and afterwards a *post-filter* is used to discard any outputs that would correspond to positive temperatures.

```
"streams":{
 "frozencelsius": {
  "channels": {
   "temp": {
     "current-value": "($.current-value - 32) / 1.8",
     "post-filter": " ({$.current-value} < 0)"
   }
  }
 }
}
```

6. IMPLEMENTATION

The COMPOSE platform will leverage different components to implement data paths between SOs. In particular it will consist of a web Front-End (RESTful API) and a data Back-End (CouchBase [3]) in which both the SO data and metadata will be stored, being the former the SO data repository and the latter the SO registry. As the system will be oriented to IoT stream processing, the central data ingestion component will be a scalable stream ingestion topology (STORM [10]) that will process incoming SUs in real time while dispatching subscriptions and queries. For advanced text-based search, the data back-end will be connected with a search engine platform (ElasticSearch [4]). The subscription components in COMPOSE are planned to be replaced by robust solutions already existing such as message dispatching engines such as Apache ActiveMQ [1] for multi-protocol support (e.g. WebSockets [12] or MQTT [5]) or pubsubhubbub [7] for HTTP callbacks. Additionally, there is a planned future integration with the semantic registry [18] using Atom [14] feeds.

This section provides details about the implementation of the API. Figure 5 illustrates the different building blocks that are part of the COMPOSE SO API implementation as well as the data flow across them. The remainder of the section provides a step by step description of the data flow followed by SUs reaching the system.

In a first step, a SU is sent to the API by an external entity through the *update* call of a stream. This call will contain headers to be able to validate the provenance of the SU by the security module.

Once validated, in the second step, the API component will put a tuple in a distributed queue system with the SU, its destination information and an generated unique operation id . Then the data is stored in CouchBase and the operation id is also stored. After that,

the API can return a 200 code to the client. The order of the four stages of the second step is relevant, because the computation of the SU in the next step of the *Service Dispatcher* will only begin when the operation id received can be read from CouchBase. This way we guarantee that the data is stored when the 200 is sent, and at the same time minimize the cases in which the store is performed and the data is not processed if there is a failure in between.

The *Service Dispatcher* (running in Storm) is constantly polling tuples from the distributed queue system (Kestrel). As mentioned before, the first thing done after receiving the tuple is checking the operation id in the database. If the operation id is not there yet, then the tuple is returned to the queue and will be retrieved again latter. This whole process has a timeout assigned to each tuple. Assuming that the retrieved tuple has its operation id written in the database, the *Service Dispatcher* requests the subscriptions to the SU in the tuple, parse them and dispatches the SU to the subscribers. The subscribers can be external entities like HTTP servers, or the SO runtime, as it can be seen in step 3.

The SO runtime follows the pipeline exposed in *Section 6*, by requesting the destination SO document and the needed other last SUs to the API. For each SU generated in the pipeline, a new tuple with an operation id will be sent to the *Service Dispatcher* queue system. Following the same order as in the step two, step four stores the SU and the operation id to the database through a private call to the API.

7. RELATED WORK

In the domain of real-time stream processing, two popular alternatives exist. One of them is the Storm Project [10], a distributed, reliable, and fault-tolerant stream processing system, which was open sourced by Twitter after acquiring BackType. Distributed by the Apache Software Foundation, Apache S4 [2] is the data streaming alternative, supported mainly by Yahoo.

Sentilo [8] is platform developed for similar purposes to the COMPOSE data plane. Although its goals are similar, it is more biased towards the storage of data than on the data processing itself, providing a simple set of interfaces to the creation of data processing agents. Composition of user deployed agents is not clear.

In the scope of service composition, several efforts can be found in the literature [15],[16],[17]. Most of them target the creation of functional workflows by reusing existing services, but usually they don't put the focus on scalability of data flows. The COMPOSE platform targets the creation of an execution environment for user-deployed code that allows composition but that is completely driven by the scalability and efficiency requirements of an Internet-scale service for the IoT. It possibly has more in common with the Twitter architecture, as tweets and sensor updates don't differ by that much, than to the traditional approach used by the service composition community.

The data plane in COMPOSE is not intended to exist in isolation, but it is expected to be complemented by other components that mainly fall in two major categories: those that allow for a graphical creation of processing graphs and those created to allow the connection with a large number of low-level specific protocols used by devices. The most prominent example for the former categorie is Node RED [6], which allows for the visual creation of data processing pipelines: the COMPOSE platform can act as the high-performance execution platform for data flows created in Node RED. A representative example of the latter category is thethingsystem [11], which allows for a simple connection to many well known devices, and provides an architecture that allows for a direct connection to the COMPOSE platform once the corresponding plug-in will be developed.

8. CONCLUSIONS

In this paper we have introduced the COMPOSE API for Service Objects, from the specification of the existing methods and operations to the details of the actual implementation using stream processing technologies. The Service Objects abstraction represent the virtual counter-parts of any existing physical device. Service Objects API provides the interfaces to store, retrieve and process data associated to one physical device. They are also used to dynamically construct the COMPOSE data plane that can ingest, transform and output sensor updates as they arrive into the platform. Although we do not include performance numbers about the current implementations, this space is currently under evaluation, both in terms of latency and scalabiliy of the systems.

Acknowledgments

This work is partially supported by the Ministry of Science and Technology of Spain under contract TIN2012-34557, by the BSC-CNS Severo Ochoa program (SEV-2011-00067), and by the by the European Commission IST activity of the 7th Framework Program under contract number 317862 (COMPOSE).

9. REFERENCES

[1] Apache activeMQ, http://activemq.apache.org
[2] Apache S4, http://incubator.apache.org/s4
[3] CouchBase, http://couchbase.com
[4] ElasticSearch, http://elasticsearch.org
[5] MQTT, http://mqtt.org
[6] Node RED, http://nodered.org/
[7] PubSubHubBub, https://code.google.com/p/pubsubhubbub/
[8] Sentilo, http://sentilo.io
[9] servIoTicy, http://www.servioticy.com
[10] Storm, http://storm-project.net
[11] the thing system, http://thethingsystem.com
[12] The webSocket API, http://dev.w3.org/html5/websockets
[13] Goessner, S.: JSONPath (2007), http://goessner.net/articles/JsonPath
[14] Gregorio, J., de hOra, B.: The Atom Publishing Protocol. RFC 5023 (Proposed Standard) (2007), http://www.ietf.org/rfc/rfc5023.txt
[15] Pautasso, C.: Composing restful services with jopera. In: Bergel, A., Fabry, J. (eds.) Software Composition, Lecture Notes in Computer Science, vol. 5634, pp. 142–159. Springer Berlin Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02655-3_11
[16] Pautasso, C.: RESTful web service composition with BPEL for REST. Data Knowledge Engineering 68(9), 851 – 866 (2009), http://www.sciencedirect.com/science/article/pii/S0169023X09000366, sixth International Conference on Business Process Management (BPM 2008)
[17] Pautasso, C., Wilde, E.: Push-enabling restful business processes. In: Proceedings of the 9th International Conference on Service-Oriented Computing. pp. 32–46. ICSOC'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-25535-9_3
[18] Pedrinaci, C., Liu, D., Maleshkova, M., Lambert, D., Kopecky, J., Domingue, J.: iserve: a linked services publishing platform. In: The 7th Extended Semantic Web Ontology Repositories and Editors for the Semantic Web Workshop. vol. 596 (June 2010), http://oro.open.ac.uk/23093/