SEVENTH FRAMEWORK PROGRAME

Specific Targeted Research Project

Project Number:	**FP7–SMARTCITIES–2013(ICT)**
Project Acronym:	**VITAL**
Project Number:	**608682**
Project Title:	**Virtualized programmable InTerfAces for innovative cost-effective IoT depLoyments in smart cities**

D4.2.2 Virtualized Filtering Mechanism

Document Id:	VITAL-D422-03032016-Draft
File Name:	VITAL-D422-03032016-Draft.pdf
Document reference:	Deliverable 4.2.2
Version:	Draft
Editors:	Valeria Loscri, Salvatore Guzzo Bonifacio, Riccardo Petrolo, Nathalie Mitton
Organisation:	Inria
Date:	05 / 03 / 2016
Document type:	Deliverable
Security:	PU (Public)

DOCUMENT HISTORY

Rev.	Author(s)	Organisation(s)	Date	Comments
V01	Salvatore Guzzo Bonifacio	Inria	16/11/2015	Initial table of contents and updates to previous version
V02	Salvatore Guzzo Bonifacio	Inria	17/11/2015	Updated architecture
V03	Salvatore Guzzo Bonifacio	Inria	25/11/2015	Updates on requirements and Threshold Filtering
V05	Elisa Herrmann	Atos	12/01/2016	Redefines of the Adaptive filtering: Design, functionality and rest services
V06	Elisa Herrmann	Atos	14/01/2016	Minor changes of the design and functionality
V07	Elisa Herrmann	Atos	15/01/2016	Fixed Figures and Tables Titles
V08	Salvatore Guzzo Bonifacio	Inria	15/01/2016	Update introduction, resampling, threshold and conclusion
V09	Elisa Herrmann	Atos	15/01/2016	Update of table 8 and 14 format.
V10	Salvatore Guzzo Bonifacio	Inria	18/01/2016	Minor Fixes
V11	Angelos Lenis	SILO	21/01/2016	Technical Review
V12	Elisa Herrmann	Atos	22/01/2016	Addressing Comments
V13	Riccardo Petrolo	Inria	25/01/2016	Addressing Comments
V14	Riccardo Petrolo	Inria	03/03/2016	Addressing Comments
V15	Martin Serrano	NUIG	05/03/2016	Circulated for Approval
Draft	Martin Serrano	NUIG	05/03/2016	EC Submitted

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TERMS AND ACRONYMS

ALE	Application Level Event
GSN	Global Sensor Network
ICO	Internet-Connected Object
IoT	Internet of Things
RDF	Resource Description Framework
RFID	Radio-Frequency Identification
SPARQL	SPARQL Protocol and RDF Query Language
W3C	World Wide Web Consortium
WSN	Wireless Sensor Network
XML	eXtensible Markup Language

1 INTRODUCTION

This deliverable specifies the architecture and the features of the Filtering module and its position within the VITAL framework. It also provides details about its implementation and the definition of services provided to other component of the framework as well as users and applications.

The list of proposed services is to be considered as an initial set of functions, in order to demonstrate the potentiality of a system, such as VITAL, and the benefits of federating data stemming from multiple sources. Even though this is the final version of the deliverable, the research process is active on the topic, allowing further development of the proposed functionalities after the end of the project, following the natural process of system's evolution.

In the final version of the document we finalized the structure of the operations that the Filtering, as a sub component of VITAL system, will provide to all other components. Such components are mainly other elements of the added value layer, but the Filtering can also be accessed, if needed, by components residing further up in the architecture (e.g., the Development Tools).

The deliverable is produced with different audiences in mind: first, VITAL consortium members, notably researchers and engineers, who require the functionalities of the Filtering module in order to develop their components; and second, external researchers and developers who want to use VITAL IoT resources in the development of their own applications.

The document is structured as follows. We first introduce the Filtering component and its features; we present the technologies we use in development; we show how Filtering is designed , implemented, finalised and connected to the other modules of the VITAL architecture. We conclude the deliverable with a summary and an outline of the next steps.

2 FILTERING LITERATURE REVIEW

The European Commission has predicted that by 2020 there will be 50 to 100 billion devices connected to the Internet. When large numbers of sensors are deployed and start collecting data, traditional application-based approaches become infeasible. Therefore, researchers have introduced significant amount of middleware solutions.

In the literature there are numerous descriptions of middleware to support wireless sensor networks in a broad spectrum of activities. IoT middleware solutions help users retrieve data from sensors and feed them into applications easily by acting as mediators between (remote) hardware and application(s). One example is Middleware Linking Applications and Networks (MiLAN) [Milan4], which describes a complex middleware aimed to assist the communication for application specific purposes over wireless sensor networks. The proposed architecture resides above operating system and below applications in order to abstract lower level functions, extending its scope down to the network layer. MiLAN receives information about the applications and about the sensors and resources available in the network.

It leverages such information to adapt the network configuration and at the same time meet the application's needs while extending the network lifetime. One of the filtering functions provided by MiLAN is mainly related to the selection of nodes actively involved. As an example, there may be scenarios where multiple sensors have overlapping coverage areas, hence producing redundant information. The ability to enable only a subset of nodes in the area can meet the application requirements and at the same time save energy and extend the network lifetime. To this purpose the system takes into account the power costs of using every node in the network, which include the power to run the device, the power to transmit its own and other nodes' data and the power needed to maintain a specific role in the network. The resources available in the network can be used to introduce Virtual Sensors obtained by mixing different data sources. Such functions can be exploited in the VITAL system.

Another example of proposed middleware is Garnet [Garnet03], where the architecture is mainly focused in data stream management. In the model depicted in Garnet many sensors transmit their data to a fixed network infrastructure via a wireless medium. The access to this network is granted through receivers. The relationship between sensor nodes and receivers is many to many, thus arriving data undergoes a filtering process. Such a process reconstructs the data stream by eliminating duplicates generated by the reception of the same information by multiple receivers.

Among the different middleware solutions proposed in literature we can find CASCoM [Cascom13], which is a middleware mainly focused to help non-IT experts in the configuration of sensors and data processing components. The proposed solution takes into account an existing middleware such as Global Sensor Networks (GSN) [GSN07]. The GSN platform consists of a sensor network, which internally can be arbitrarily configured, delivering sensor data to one or more sink nodes. Every sink node is connected with a base computer, which runs the GSN middleware. One of the key ideas introduced by GSN is the abstraction of a sensor device as “virtual sensor”, which can be any kind of data producer. The specification of virtual sensor characteristics includes an SQL-based definition of the operations to be performed on the received data by the virtual sensor itself. In order to support scalability they define GSN-light instances that can vary filtering condition complexity. Being based on GSN, CASCoM relies on the fact that sensor data can be processed in three different layers, namely "virtual sensor layer", "query processing layer" and "applications and services layer". For this kind of model the filtering functions can take place mainly in the first two layers. In the virtual sensor layer, filtering operations are applied over the sensor data whereas the query processing layer can perform filtering through SQL-like specifications. Such layered structure as well as the SQL-like filtering mechanism can be adopted in our system and will be discussed later in the deliverable.

Yet another available middleware instance is entitled CloUd-based Publish/Subscribe for the Internet of Things (CUPUS) [Cupus14] that is meant to work with mobile publish/subscribe services. CUPUS aims to enable the formulation of cloud-based applications based on mobile IoT sensor data, including flexible data filtering on mobile devices. They introduce the concept of mobile broker, whose purpose is to handle the sensing process on the mobile devices via appropriate processes. One more function is to face with the remote part of the system collecting sensed data.

Moreover the broker is capable to filter sensor data according to the global needs. So the acquired raw data are pre-processed, filtered and enriched with semantic information. Another important aspect is that those data should be sent to the interested receivers in near real time. To this end the publish/subscribe paradigm offers an appropriate solution. It offers a selective and flexible acquisition and filtering mechanism of sensed data on mobile devices. The dynamism and flexibility offered is because it takes into account user's preferences expressed as subscription. Furthermore, communication between publishers and subscribers is asynchronous, thus a device can be registered and disconnected at the same time. All the data that suits device's subscriptions can be sent as soon as it gets reconnected.

One last middleware is CA4IOT [CA4IOT14] which proposes an architecture to help the user in selection of the sensors according to the problem at hand.

Filtering has also been applied in several domains of Wireless Sensor Networks (WSN). One example is depicted in [Zoller13] where an on-mote filtering technique aided with a forecast mechanism is proposed. The main idea is driven by the need to reduce energy consumption in WSNs. Since data transmission accounts for the main component of energy consumption, filtering data directly on the mote leads to an enhancement in energy efficiency. A basic metric is used for filtering the value of information associated with sensor data, defining a “measurement execution and analysis” component. The aforementioned component can be user-defined so the number of data transmissions can be reduced while still providing enough data fidelity for user's perspective. Moreover the determination of information value is strongly application-specific. The basic task for the filtering consists of an appropriate interpretation of sensors measurements in order to define the value that such information has in respect to the user and the application. Subsequently to this evaluation, the filtering is applied by a "selective transmission" block, whose purpose is to take the final decision. This is accomplished with comparison between the value of information and the transmission cost. If the value of information overweighs the transmission cost the corresponding sensor data is transmitted, otherwise it gets filtered and discarded.

Except for middleware, there are also some other useful functions in the filtering area. They mainly involve the filtering of contents for user's requests and habits. One example is illustrated in [Ferman02] where a framework intended to filter contents of interest in a multimedia context is briefly introduced. The main idea is to analyse the user’s request history in order to define a profile. The history can be used by a profiling agent which, using structured data, extracts user’s preferences. The properties defined in such profile can be used with the metadata of different multimedia contents in order to filter the elements that best match user’s interests.

The techniques that aim to filter any kind of information in order to express recommendations for a selected user are commonly known as Collaborative Filtering. A good survey is in [Su09] that mainly focuses on the different techniques available to compute Collaborative Filtering. This kind of filtering is basically aimed to select one or more items to suggest to a user. The process involves a set of users and a set of items. The goal in this kind of filtering is, after properly categorizing both elements, to produce a list of items of interest for a user, based on his tastes ant those of similar users. There are many problems in this area e.g. the grey sheep and black sheep problem, the cold start problem, security and anonymity problem. The survey enlists various approaches and algorithms to address this problem.

Liu and Martonosi in [Impala03] propose a middleware architecture that enables application modularity, adaptability, and reparability in wireless sensor networks. The proposed Impala, allows software updates to be received via the node's wireless transceiver and to be applied to the running system dynamically. The filtering capabilities in this case are used in order to dispatch events to the above system units and initiate chains of processing.

Within the context of RFID, different filtering mechanisms have been introduced in literature; for example the EPCglobal standard specifies filtering mechanisms as part of the ALE (Application Level Event) layer of the architecture [Kefalakis09]. As specified by the EPCglobal standard in [EPCGlobal09], the role of the ALE interface within the EPCglobal Network Architecture is to provide independence among the infrastructure components that acquire the raw EPC data, the architectural component(s) that filter & count that data, and the applications that use the data. In detail, the ALE interface:

- Provides a means for clients to specify, in a high-level, declarative way, what EPC data they are interested in, without dictating an implementation.
- Provides a standardized format for reporting accumulated, filtered EPC data that is largely independent of where the EPC data originated or how it was processed.
- Abstracts the sources of EPC data into a higher-level notation of "logical reads", often synonymous with "location", hiding from clients the details of exactly what physical devices were used to gather EPC data relevant to a particular logical location.

Authors in [Kefalakis2011] introduce a middleware implementation of ALE mechanisms. The proposed AspireRFID (http://wiki.aspire.ow2.org/) extends the filtering function to any kind of data such as active sensor data, MAC addresses, phone numbers, etc.

The Cougar project [Cougar] aims to tasking sensor networks through declarative queries. Given a user query, a query optimizer generates an efficient query plan for in-network query processing, which can vastly reduce resource usage and thus extend the lifetime of a sensor network. In this case, the filtering capabilities of the system are used in order to filter the sensors interested by the query.

The main aspect that we addressed with this literature analysis was to identify a broad set of filtering problems and solutions. Unfortunately, due to the position of our filtering functions some of those problems cannot be properly addressed. This is because some of those functionalities are strictly connected with the sensing devices. This means that they have to be handled by entities, which reside inside the individual silos, before data are sent to the VITAL platform, like the filtering of duplicated values. Some other aspects are instead too specific to applications or users thus such operations should be left to the applications, which lie in an upper layer.

3 DESIGN OF INTELLIGENT MECHANISM OF FILTERING

The Filtering module is based on Java EE technology in order to implement a RESTful web service able to provide filtering services. The communication standard used to exchange data is JSON(-LD), the format in which these information will arrive as input to the filtering module. The received parameters will then be used to drive the filtering process itself.

In order to retrieve information, the Filtering module relies on the query endpoint provided by the Data Management Service (DMS). To this end, queries are defined using MongoDB Query standard. According to the standard, query parameters are exchanged using a JSON object containing specific criteria, or conditions, to be used for the selection of required results.

To give an example, the query in **Figure 1** returns all the values observed for `speed` by `sensor1`. Using such request it is possible to locate in the DMS observations fitting requested criteria. The result consists in a collection of JSON-LD objects, consisting of all data and metadata stored for the considered observation.

```
{
      "ssn:observedBy" : "http://example.org/resource/sensor1",
      "ssn:observationProperty.type" : "vital:Speed"
}
```

Figure 1: Basic query

```
                                  {
      "ssn:observedBy" : "http://example.org/resource/sensor1",
      "ssn:observationProperty.type" : "vital:Speed",
      "ssn:observationResult.ssn:hasValue.value" : { $gt : 30 }"
}
```

Figure 2: Query including speed limit

The query in Figure 2 extends the previous example by adding further criteria, to limit the observations to those that have measured value that goes above a specified threshold.

3.1 Requirements

The Filtering module, as part of the *Added Value Services* of VITAL (Figure 3), offers its services directly through a set of Virtualised Universal Access Interfaces (VUAIs).

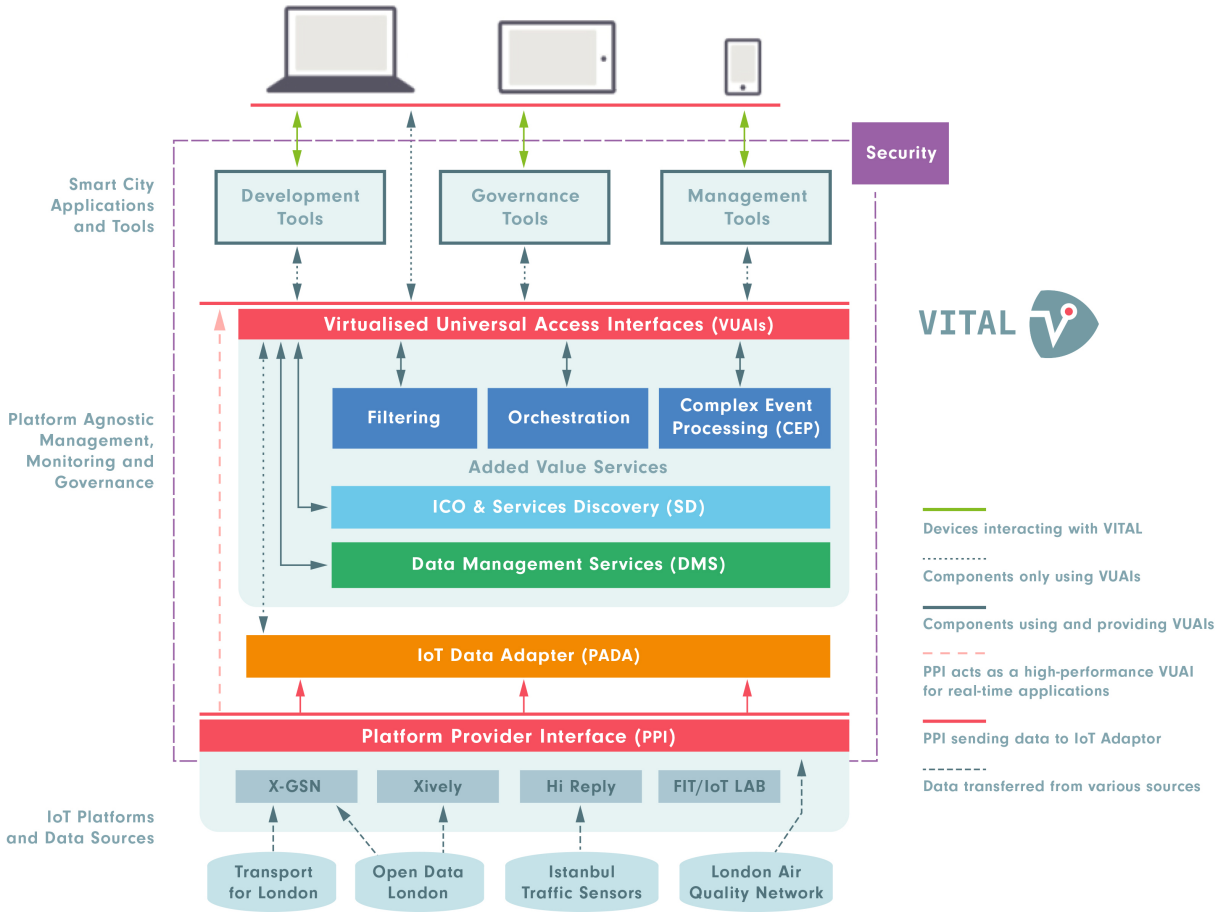

Figure 3: VITAL architecture

While Discoverer focuses on the definition of service for the discovery of systems, services and ICOs (See deliverable D4.1.2 for further details), Filtering component aims to provide functions that allow users and applications to select *observations* stored in the DMS in an easy and guided way.

An example of *observation* is represented in Table 1. In the example we can observe the different parameters that are available in the representation of a measurement performed by a sensor, i.e. observation identifier, type, reference to the observing sensor and observed value.

Table 1. Observation Example

```
{
    "@context": "http://vital-iot.eu/contexts/measurement.jsonld",
    "type": "ssn:Observation",
    "ssn:observedBy":"http://vital-
integration.atosresearch.eu:8180/camden-footfall-ppi/sensor/2",
```



```
    "id":  "http://vital-integration.atosresearch.eu:8180/camden-footfall-
ppi/measurement/2-1446807600000",
    "ssn:observationProperty": {
      "type": "vital:Footfall"
    },
    "ssn:observationResult": {
      "type": "ssn:SensorOutput",
      "ssn:hasValue": {
        "value": 2237,
        "type": "ssn:ObservationValue"
      }
    },
    "ssn:observationResultTime": {
      "time:inXSDDateTime": "2015-11-06T12:00:00+01"
    }
  }
```


Filtering is accessible through a RESTful interface, which exposes information like the context, a description, its status and the operations it offers. It is characterized as a system registered in VITAL, as well as all other added value functionalities available in the platform. To be compliant with the structure defined for VITAL components in D3.2.2, Filtering exposes his PPI through a set of endpoints. Required services are available at the addresses listed in Table 2.

Table 2. Filtering PPI endpoints

Service	***Address***
Get IoT system metadata	`FILTERING_BASE_URL/ppi/metadata`
Get IoT system status	`FILTERING _BASE_URL/ppi/status`
Get IoT service metadata	`FILTERING _BASE_URL/ppi/service/metadata`
Get IoT sensor metadata	`FILTERING _BASE_URL/ppi/sensor/metadata`

3.2 Design

3.2.1 Threshold Filtering

Threshold filtering is a function that allows users and application to retrieve observations based on the comparison of the measured value of a property and a threshold value defined in the request.

Figure 4 shows an example of the interactions between the Filtering module and the Orchestrator, involving also the Discovery and the DMS, a possible sequence of interaction between components of VITAL system. In this example, the Orchestrator

directly activates the Filtering with the objective to filter observation for a specific ICO.

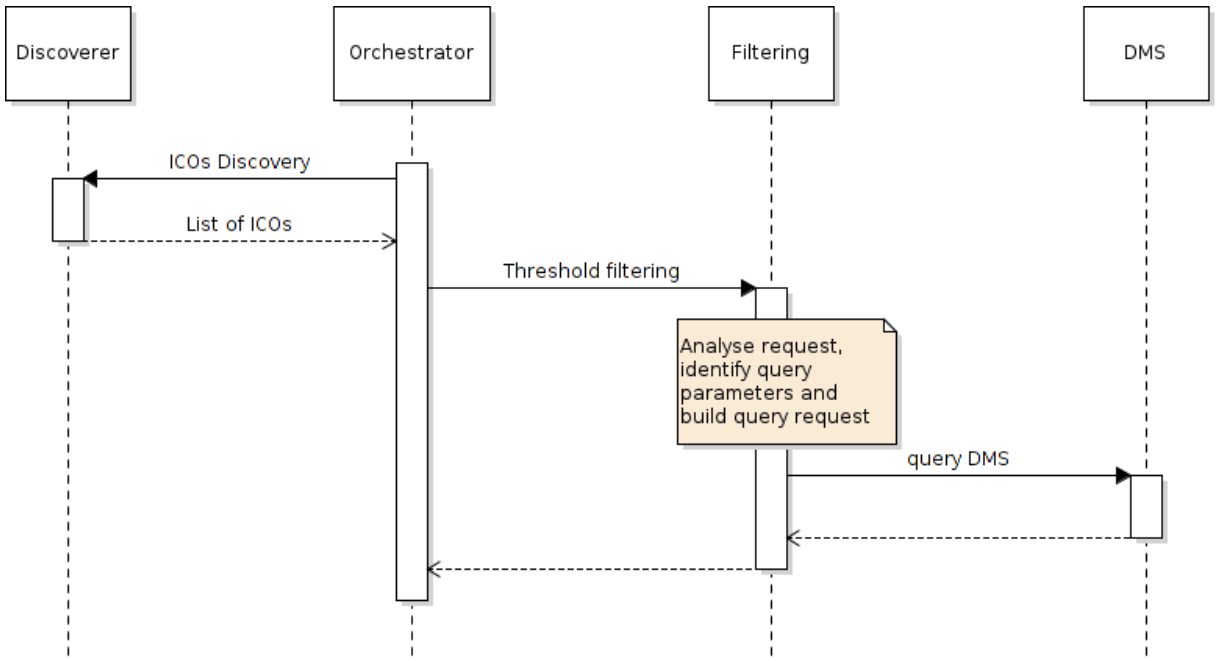

Figure 4: Threshold filtering sequence diagram

The function receives as input a JSON object containing a set of keys like ICO identifier and threshold numerical value. The request object is enriched with other options in order to configure the filtering process to operate on a wider set of parameters. A list of keys that can be defined in a Threshold Filtering request are summarized in Table 3.

Table 3. Threshold filtering object's JSON format

Name	***Type***	***Description***
`Ico`	String	Identifier of an ICO. Represent the reference to the component that observed the measure
`Position`	Object	Object containing keys to define a spatial region over which perform the filtering
`observationProperty`	String	Represents the observation type of the observations of interest
`Inequality`	String	Defines the inequality relationship between the observed value and the defined threshold
`Value`	Number	Represent the numerical value of the desired threshold
`From`	String	Represents the start of time interval over which perform the filtering

`To`	String	Represents the end of time interval over which perform the filtering

In the following we give some more details about the available options in the filtering request and how they can be configured to change the filtering logic.

The key `position` has been included to allow the filtering in a specific spatial region. The aforementioned key is associated to a JSON object with the triple `latitude, longitude` and `radius`. Even though `position` is optional, the triple is mandatory. If a user wants to restrict the filtering process over a specific area, all the elements of the triple must be provided. If one of the element is missing the request is considered as malformed and a status code 400 (*`Bad Request`*) is returned.

The keys *`ico`* and *`position`* are meant to be used alternatively. If the request object contains the *`ico`* key, the filtering is performed only over the observations measured by the selected internet connected object. If, instead, the request is characterized by the definition of *`position`* option, the filtering process is performed over the observations measured by all the internet connected object that are registered in the geographical region specified in the request.

The option *`inequality`* is defines the relationship between the threshold value and the observed value. Supported values are:

- `lt`: lower than (<)
- `gt`: greater than (>)
- `lte`: lower than or equal to (≤)
- `gte`: greater than or equal to (≥)

Keys *`from`* and *`to`* defines a time interval over which the filtering is performed and expects input values in the XSD format. The limitation over a time interval is optional, and the filtering behaviour changes according to the received configuration. If both time boundaries are defined the filtering is limited on the observations within the selected range. If only *`from`* is provided, *`to`* is automatically set to the time value at the moment of the request. If no time range is defined the filtering process is performed over all the observation in the system.

To further illustrate the filtering mechanism, in a visual manner, we can consider all available observations stored in the DMS, for a specific ICO, regarding speed measurement. Such observations can be plot as illustrated in Figure 5

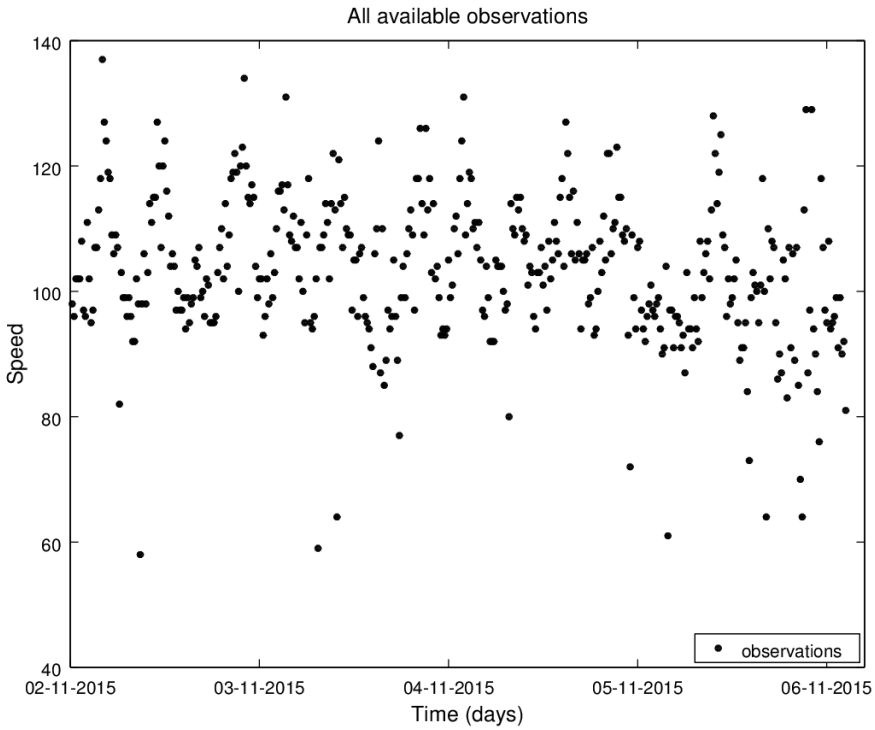

Figure 5: All speed observations of an ICO

Starting from such data distribution we can consider a filtering request based only in the definition of a threshold. More specifically, performing a request to filter all observations above 100 Km/h will generate an output as illustrated in Figure 6. In this case, observations illustrated in red are used to build the result set, whereas observations illustrated in black will be discarded.

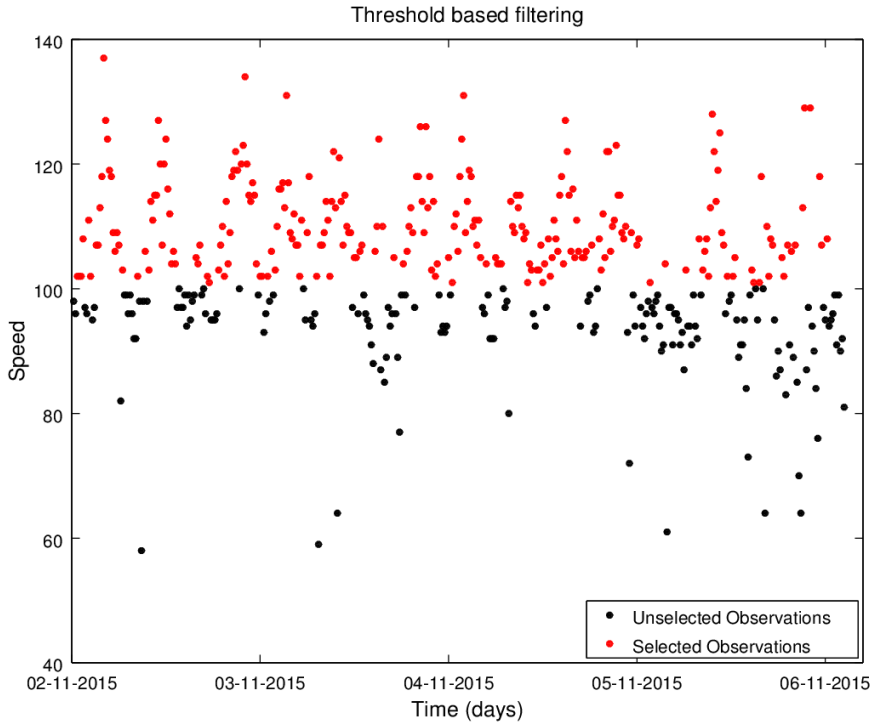

Figure 6: Selection by threshold

Considering instead the distribution as shown in Figure 5, if the filtering is requested with the definition of both threshold and time interval, the corresponding output will be populated by all elements highlighted in red as shown in Figure 7.

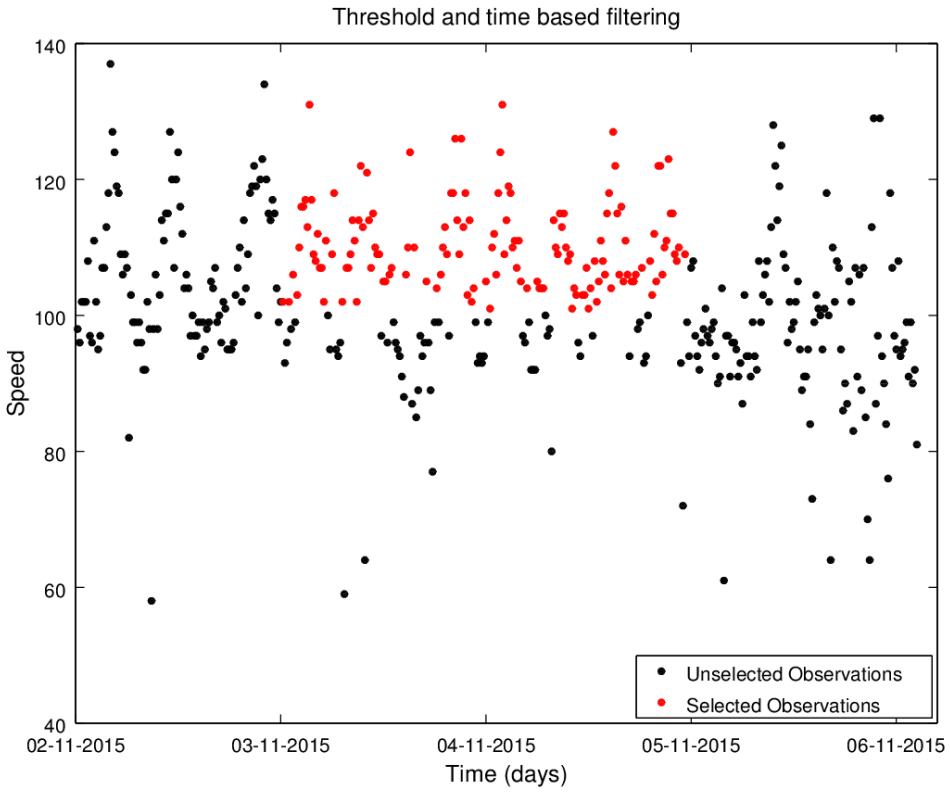

Figure 7: Selection by threshold and time interval

An example of how to request the service, including the definition of input parameter and an extract of a possible output of the threshold filtering function is presented in Table 4.

Table 4. Threshold filtering example

	Threshold
Description	Retrieves observations according to the comparison of measured value with a specified threshold
URL	BASE_FILTERING_URL/threshold
Method	POST

Input	**Example:**
	
	{
	"ico" : "http://vital-integration.atosresearch.eu:8180/camden-footfall-ppi/sensor/2",
	"observationProperty" : "vital:Footfall",
	"inequality" : "gt",
	"value" : 1000
	}

Output	[{ "@context": "http://vital-iot.eu/contexts/measurement.jsonld",

```
    "type": "ssn:Observation",
    "ssn:observedBy": "http://vital-
integration.atosresearch.eu:8180/camden-footfall-ppi/sensor/2",
    "id": "http://vital-integration.atosresearch.eu:8180/camden-footfall-
ppi/measurement/2-1448283600000",
    "ssn:observationProperty": {
      "type": "vital:Footfall"
    },
    "ssn:observationResult": {
      "type": "ssn:SensorOutput",
      "ssn:hasValue": {
        "value": 2294,
        "type": "ssn:ObservationValue"
      }
    },
    "ssn:observationResultTime": {
      "time:inXSDDateTime": "2015-11-23T14:00:00+01"
    }
  },
  {
    "@context": "http://vital-iot.eu/contexts/measurement.jsonld",
    "type": "ssn:Observation",
    "ssn:observedBy": "http://vital-
integration.atosresearch.eu:8180/camden-footfall-ppi/sensor/2",
    "id": "http://vital-integration.atosresearch.eu:8180/camden-footfall-
ppi/measurement/2-1448539200000",
    "ssn:observationProperty": {
      "type": "vital:Footfall"
    },
    "ssn:observationResult": {
      "type": "ssn:SensorOutput",
      "ssn:hasValue": {
        "value": 2217,
        "type": "ssn:ObservationValue"
      }
    },
    "ssn:observationResultTime": {
      "time:inXSDDateTime": "2015-11-26T13:00:00+01"
    }
  }
]
```


3.2.2 Resampling

In the digital signal processing theory a signal can be sampled on a specific sample rate, measuring a physical value on regular time intervals. Once the signal have been sampled, the need to change this time interval can occur. This change can be indicated as upsampling when the number of samples is increased and downsampling when the number of sampling is decreased.

Upsampling operations are usually executed using an interpolation process, whereas the downsampling operations are conducted through a decimation process. In the case where both operations are available the resultant is known as resampling. For the filtering functionalities we wanted to extend this concept also to data measured by an internet connected object. Through the use of the resampling function, observations stored in the DMS can be used as data source to create a set of observation, which are separated in time by a constant value.

To provide an example, suppose that an application need to calculate the speed measured by an ICO with an interval of 15 minutes, while data are measured with a interval than can vary from 8 to 12 minutes. To this end the application can request,

through the VUAI, a resampling operation to the Filtering obtaining a list of observations compliant with the requested characteristic. Figure 8 shows a sequence diagram related to the aforementioned example.

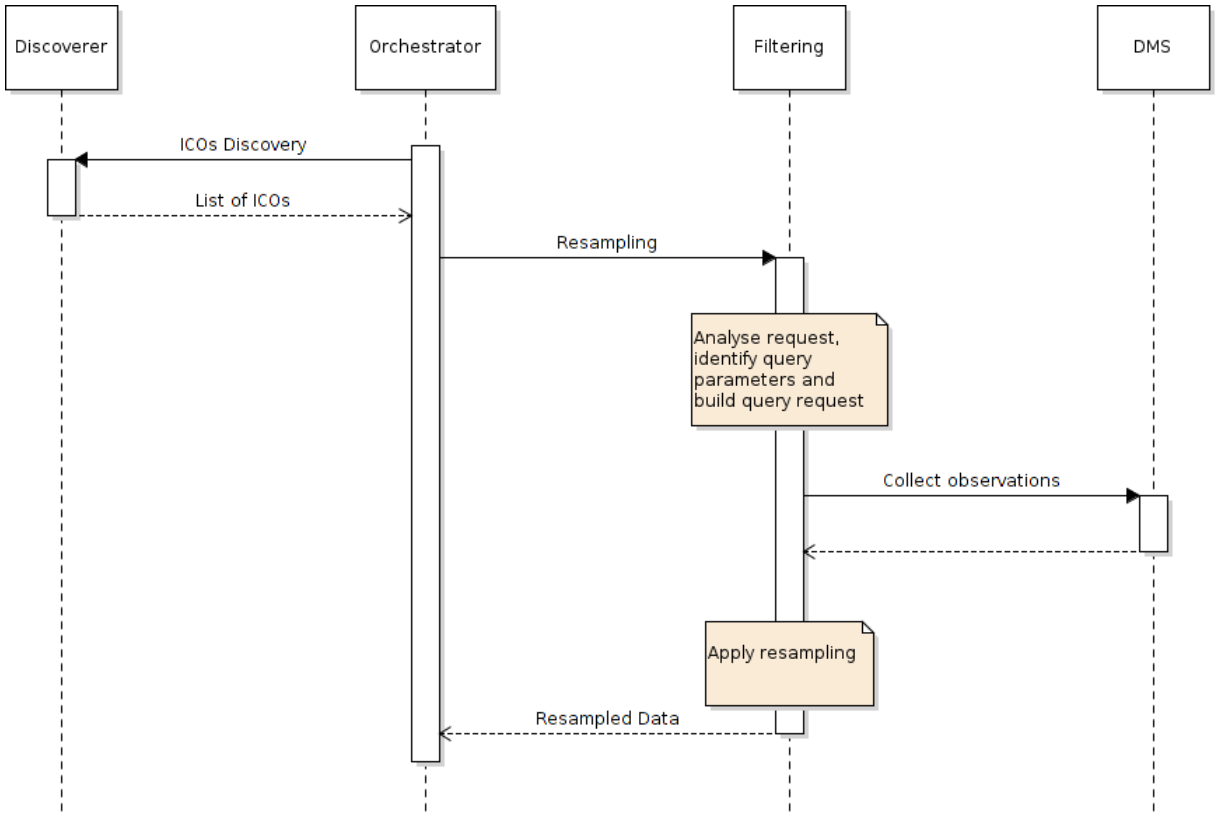

Figure 8: Resampling sequence diagram

All parameters needed to set up the resampling operation are received through the use of a JSON object received as Content-Type in the body request. Table 5 illustrates the different keys that define a resampling request object.

Table 5: Resampling request JSON object

Name	***Type***	***Description***
`Ico`	String	Identifier of an ICO. Represent the reference to the component that observed the measure
`observationProperty`	String	Represents the observation type of the observations of interest
`timeValue`	Number	Defines the numeric value of the time period requested for the resampled output
`timeUnit`	String	Is the time unit associated to the time value of interest
`From`	String	Represents the start of time interval over which perform the resampling
`To`	String	Represents the end of time interval over which perform the resampling

In the following, we provide some more details about time parameters, in terms of requested time period and time interval, as required in the JSON object.
Time period is the amount of time between two subsequent observations contained in the response, whereas the time interval is a time window used to collect available observations to be used as data source for the resampling process.
The key `timeUnit` can assume one of the following values:

- minute
- hour
- day

The union of `timeValue` and `timeUnit` gives the time period that will be taken into account to produce the result. Referring to the previous example, to request a resampling with a time period of 15 minutes the JSON object should contain a key `"timeValue"` with the value `15` and a key `"timeUnit"` with the value `"minute"`.
The couple `from` and `to` defines the time interval to use to collect observations from the DMS in order to build the data source for the process. Both parameters are mandatory and their value is expected in the XSD time format specified by the OWL Time ontology as defined in D3.1.2.
The resampling of measured observation takes place through the use of an interpolation process. Interpolation is a method to evaluate new points in a plane starting from a finite set of given points, with the assumption that all these points refer to the same function. Within interpolation theory different techniques are available, e.g. linear interpolation, polynomial interpolation, spline interpolation. For the implementation of resampling we adopt the *Spline Interpolation*. Spline interpolation is a specific kind of interpolation which is based on spline functions. Differently from polynomial interpolation, which uses a unique polynomial to approximate the function over the entire definition range, Spline Interpolation is obtained dividing the aforementioned range in sub-intervals. For every sub-interval a polynomial function is generated. Polynomials corresponding to two subsequent intervals are matched observing continuity of derivatives. Thanks to this property Spline Interpolation allows to generate interpolation functions that have smoother transitions between input samples and incurs in a smaller error.

The sequence of main operations necessary to generate the resampled set of observations is illustrated in Figure 9.

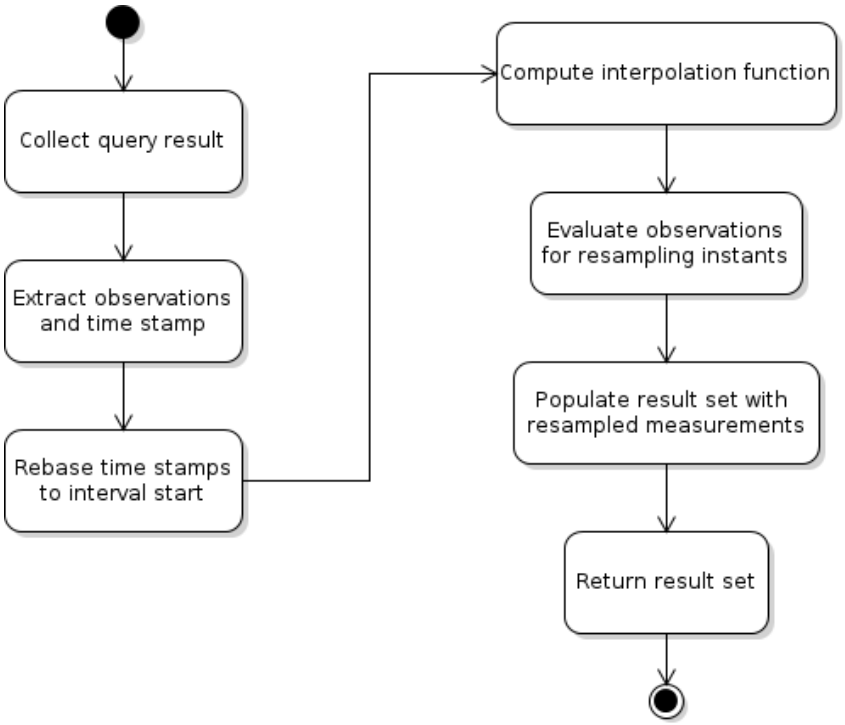

Figure 9: Resampling Activity Diagram

Data collected from the DMS are analysed in order to extract the measured value of an observation and the corresponding time. Since all timestamps are represented in an absolute time value, a time rebase operation is performed, to represent time using as relative reference the selected resampling interval. Observation measurements and rebased time instants are then used to compute an interpolation function over the resampling period of interest. To complete the resampling process observations are evaluated through the interpolated function according to the required interval. Finally the result set is returned in JSON-LD format.

To summarize input parameters and output format an example is shown in Table 6.

Table 6: Resampling Example

	Resampling
Description	Resamples observations of an ICO
URL	BASE_FILTERING_URL/resample
Method	POST

Input	**Example:**
	
	{
	"ico":"http://vital-integration.atosresearch.eu:8180/hireplyppi/sensor/vital2-I_TrS_304",
	"observationProperty":"vital:Speed",
	"timeValue":15,
	"timeUnit":"minute",
	"from":"2015-11-09T17:21:03+01:00",
	"to":"2015-11-09T23:25:03+01:00"
	
	}

Output	[

```
{
    "dul:hasLocation": {
      "type": "geo:Point",
      "geo:long": 28.9371815,
      "geo:lat": 41.03869375,
      "geo:alt": 0
    },
    "@context": "http://vital-iot.eu/contexts/measurement.jsonld",
    "type": "ssn:Observation",
    "ssn:observedBy": "5.79.79.172:8180/filtering",
    "id": "5.79.79.172:8180/filtering/1449044646713",
    "ssn:observationProperty": {
      "type": "vital:Speed"
    },
    "ssn:observationResult": {
      "type": "ssn:SensorOutput",
      "ssn:hasValue": {
        "qudt:unit": "qudt:KilometerPerHour",
        "value": 77,
        "type": "ssn:ObservationValue"
      }
    },
    "ssn:observationResultTime": {
      "time:inXSDDateTime": "2015-11-09T17:36:03+01"
    }
  },
  {
    "dul:hasLocation": {
      "type": "geo:Point",
      "geo:long": 28.9371815,
      "geo:lat": 41.03869375,
      "geo:alt": 0
    },
    "@context": "http://vital-iot.eu/contexts/measurement.jsonld",
    "type": "ssn:Observation",
    "ssn:observedBy": "5.79.79.172:8180/filtering",
    "id": "5.79.79.172:8180/filtering/1449044646713",
    "ssn:observationProperty": {
      "type": "vital:Speed"
    },
    "ssn:observationResult": {
      "type": "ssn:SensorOutput",
      "ssn:hasValue": {
        "qudt:unit": "qudt:KilometerPerHour",
        "value": 80,
        "type": "ssn:ObservationValue"
      }
    },
…
]
```


The quality of resampled observations depends on the cardinality of the available observations and the resampling interval requested by the user. In general, if the resampling is based on decimation the result set can be considered accurate. For resampling requests with a time interval too small, compared with measurement period performed by ICOs, the interpolation process may not include variations occurred between two successive input samples.

4 ADAPTIVE FILTERING FOR IOT DATA DRIVEN PROCESSING

4.1 Introduction

Adaptive filtering provides the capability of creating customized filters that supports complex filtering pattern by means of a Complex Event Processing. The filter takes streams of incoming events, in the form of input observations and evaluates whether those events meets the rules specified by the user and then publishes the resulting events to observers.

The VITAL CEP components for the different adaptive filter are implemented based on the ATOS reference architecture for distributed, scalable and "cloudified" complex event processing [Atos16]. Figure 10 shows the functional blocks of the ATOS references architecture that have been implemented for VITAL and explained in the next section.

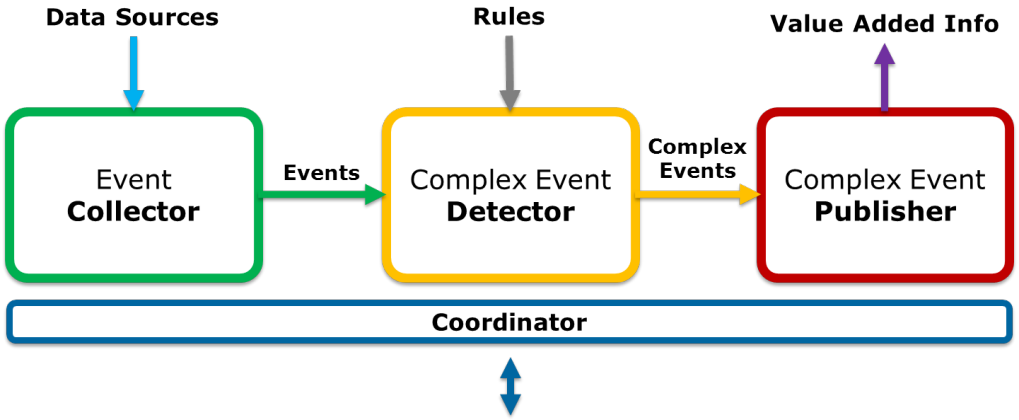

Figure 10: ATOS functional blocks of the reference architecture for Complex Event Processing

Due to the modularity of the Atos CEP architecture, it is possible to implement many different kinds of adaptive filters providing different implementations of the functional blocks of the architecture.

For the Complex Event Detector functionality we use the µCEP [Atos16] implementation adapted and extended for VITAL, and for the Event Collector, the Complex Event Publisher, and the Coordinator functionalities we have implemented the components from scratch according to vital needs. The µCEP is a rule engine that uses a rule specification language named DOLCE [Atos14]. The expressiveness of the DOLCE rules language allows to apply different filtering algorithms to fulfil user requirements. For example, filters that compare events to events in same or different stream, or compare events to some aggregated values. A typical example of event filtering is capturing sensor readings where values average fall outside of expected range.

An event stream can have an infinite number of events. Windows provide a means to select event subsets for complex event detection. Out of many possible ways to select events, two of the most basic mechanisms include:

- **Time Windows**: In some specific cases, it is not sufficient to detect a complex event when certain values are detected. There is a need to take time range into account as well. For example, detecting events when certain values exceed or fall below some threshold within a specified time period. These time periods are usually represented by fixed time windows or sliding time windows. For example, sliding time window can collect all sensor readings during the last two hours and fixed time window collects readings once every second hour;
- **Tuple Windows**: Instead of measuring elapsed time, tuple windows select events based on number of occurrences of particular events within an input stream. A typical example of tuple window is collection of last twenty sensor readings for further evaluation.

The typical transient event lasts for infinite small period. However real world scenarios require support for so called “long lasting events”. The duration of such events is for a fixed amount of time or until another event arrives.

In VITAL Event Filtering, the idea of join events is to match events coming from different input streams and produce new complex event stream. A join between two data streams necessarily involves at least one window. To perform a join, it is usually necessary to wait for events on the corresponding stream or to perform aggregation on selected events. This is what windows do. The most complex event processing implementations support window to window joins, outer joins or stream to database joins. The joins are used for example to correlate information across different security devices for sophisticated intrusion detection mechanism and response.

Real world tasks require support for more sophisticated complex event detection mechanisms. Patterns and sequences match conditions that happen over time. For example, “a fire” is detected when within a 10 minute interval sensor A detects smoke, followed by same event from either sensor B or C, followed by absence of event from sensor D. In addition to that, all events may be related to each other in some way.

4.2 Design and Implementation

The adaptive IoT event filtering mechanism is implemented using the ATOS reference architecture for distributed, scalable and “cloudified” complex event processing [Atos16] as a base. The implementation of this architecture has been customized and extended to satisfy the needs of the different filtering mechanism and also to conform to the VITAL specification of the components in terms of the VUAI described in D322.

The CEP reference architecture Figure 10 is composed of 4 functional blocks as listed below;

- Coordinator
- Event Collector
- Complex Event Detector
- Complex Event Publisher

This reference architecture is shown in different VITAL deliverables (specifically detailed in D4.3.1) to facilitate documents understanding. Each deliverable explains how the functional blocks are specifically implemented according to the concrete

requirements or expected functionality (Static Data Filters, Static Query Filters and Continuous Filters). In concrete, in this document about Adaptive IoT filters.

4.2.1 Filtering Administrator Component

The Filtering Administrator Component is the implementation of the coordinator functional block of the ATOS reference architecture.

The Filtering Administrator Component is implemented as a RESTFul API in java by means of the Jersey[1] library to provide the functionality to create, read, update, and delete the different filters types provided for VITAL. The methods implemented in this component are explained in detail in section 4.4

The coordinator functional block is also in charge of managing the CEP engine instances and managing the internal processes to transform the input data stream into the output data stream. To manage the internal estate of the filtering process the component uses a MongoDB[2] instance by means of the mongodb-driver for java.

The component has been deployed in the VITAL integration server my means of a ".war" file into the Wildfly[3] application server.

4.2.2 Event Collector Component

The Event Collector functional block goal is to gather information from different sources with different data format and transform the data into events to feed the Complex Event Detector functional block. This functional block is composed by two inner blocks, the "Listener" to gather the information from different sources and the "Decoder" to transform the input data into the "event" data format of the DOLCE specification.

The "**Listener**" component is present in the Continuous Filter implementation (4.3.2) to collect data streams from the VITAL observation sources and forwards the received data to the "Decoder" component by means of a pulling mechanism implemented for that purpose. This component has been redefined for the Static Query Filter (4.3.1.2) implementation in order to query the DMS to obtain the input observations to be filtered.

The "Decoder" is the component that reads all the events information coming from the "Listener" and has been implemented to transform the received observations data formatted in JSON-LD, specific for VITAL, into the µCEP engine readable format.

The Event Collector functional block is also responsible for sending the decode data to the Complex Event Detector functional block. For that purpose the Event Collector component implements the functionality to send the input events to the Complex Event Detector Component using the MQTT[4] protocol to publish the events to the Mosquitto[5] message broker used for the communication between them. This

[1] Jersey library: https://jersey.java.net/

[2] Mongodb database: https://www.mongodb.org/

[3] Wildfly application server: http://wildfly.org/

[4] MQTT protocol: http://mqtt.org/

[5] Mosquitto message broker: http://mosquitto.org/

component is implemented in java and uses the Paho[6] Java Client library to connect to the Mosquitto message broker.

4.2.3 Complex Event Detector Component

Complex Event Detector functional block is responsible for filtering incoming data using temporal persistence of volatile events until constraints of a rule(s) are entirely satisfied. The component is also responsible for production of expected results and sending the results to next component. For the VITAL Adaptive filtering we are using an ATOS solution called µCEP. The µCEP consist of a C++ program that implements the CEP engine based on the DOLCE rules specification language and uses the Mosquitto message broker in order to subscribe to the input data and to publish the output data to be used by others components.

Other added functionalities to µCEP regarding VITAL needs are:

- New supported types of data: string, char, float, pos, area
- Include the tuple sliding window; and improve the time sliding window; that are used to store temporally a pool of events related to a complex event
- Complex functions: count, sum, avg, diff, inarea; these functions process the data stored in the event pool of each complex event, to evaluate if a complex event is published
- Event operators: after, during and absent supported in the detect clause of the complex events, to relate more than an event with a complex event
- Offset operator (@) inside the time & tuple windows, to enable the direct access to an item of the CEP engine (complex event detector) event pool
- Allow complex events in the detect clause of other complex event, to enable define complex events which have as input other complex event triggered
- Group clause to support the groups of events in the event pool of the complex events, to enable classify the events by specific attribute
- Allow complex functions in the payload clause of a complex event definition; to enable publish the outcome of complex functions
- Basic operations (== and !=) for string data type

4.2.4 Event Publisher Component

The Event Publisher functional block is in charge of receives the output complex events of the Complex Event Detector functional block and delivers them to the selected data sink. This functional block is composed by two blocks, the “Encoder” to transform the DOLCE complex events into the required output data format and the “Emitter” to push the output data to the selected data sinks.

The Event Publisher Component has been implemented to receive the complex events from Complex Event Detector component by subscribing to the Mosquitto message broker.
The “**Encoder**” component has been implemented for transforming the DOLCE complex events into VITAL observations in JSON-LD format. The “**Emitter**” component has been implemented for the Static Data Filter (4.3.1.1) and the Static Query Filter (4.3.1.2) in order to retrieve the filtering results to the DMS and also to the filter observer in the response of the filter creation request. The “Emitter”

[6] Paho Java Client: https://eclipse.org/paho/clients/java/

implemented for the Continuous Filter (4.3.2) has been modified to push the resulted observations to the DMS all along the life cycle of the filter.

4.3 Functionalities

4.3.1 Static filtering

Static filtering is the capability of running filtering algorithms on finite input data provided directly by the user or by the result of a query. In a single API request, a new instance of CEP filter is created to apply the DOLCE rules to the input data. Then, the result is sent to the user in the response through the Event Publisher component that transform the result data into JSON-LD VITAL observations while in parallel the result is forwarded to the DMS to be persisted. Once the service has been completed and the results have been persisted and sent to the requester the CEP instance is removed to release resources as it is no longer needed.

4.3.1.1 Static Data Filter

The Static Data Filter obtains new complex observations from filtering a set of input observations based on the DOLCE rules provided. The input observations and the DOLCE rules are provided as input parameters into the filter request and the resulted observations are sent to the user in the response and sent to the DMS to be persisted. Figure 11 shows the different components of the implementation of the Static Data Filter and the interactions among them.

The Filtering Administrator component is implemented to accept user requests from the user applications or form others VITAL modules and also is define to coordinate the whole filtering process. This component creates the instance of the CEP static data filter and then calls to the Event Collector and the Decoder to transform observations input data into DOLCE events and send those events to the CEP static data filter by means of the third party message broker (Mosquitto). The Filtering Administrator component also calls to the Event Publisher component to receive the complex events returned by the CEP through the message broker and to transform them into the JSON-LD Vital Observations by means of the Encoder. Finally, the Emitter persist the observations into the DMS and also return them to the Filter Administrator to be sent to the user application in the response of the request and then it removes the CEP instance that is no longer needed.

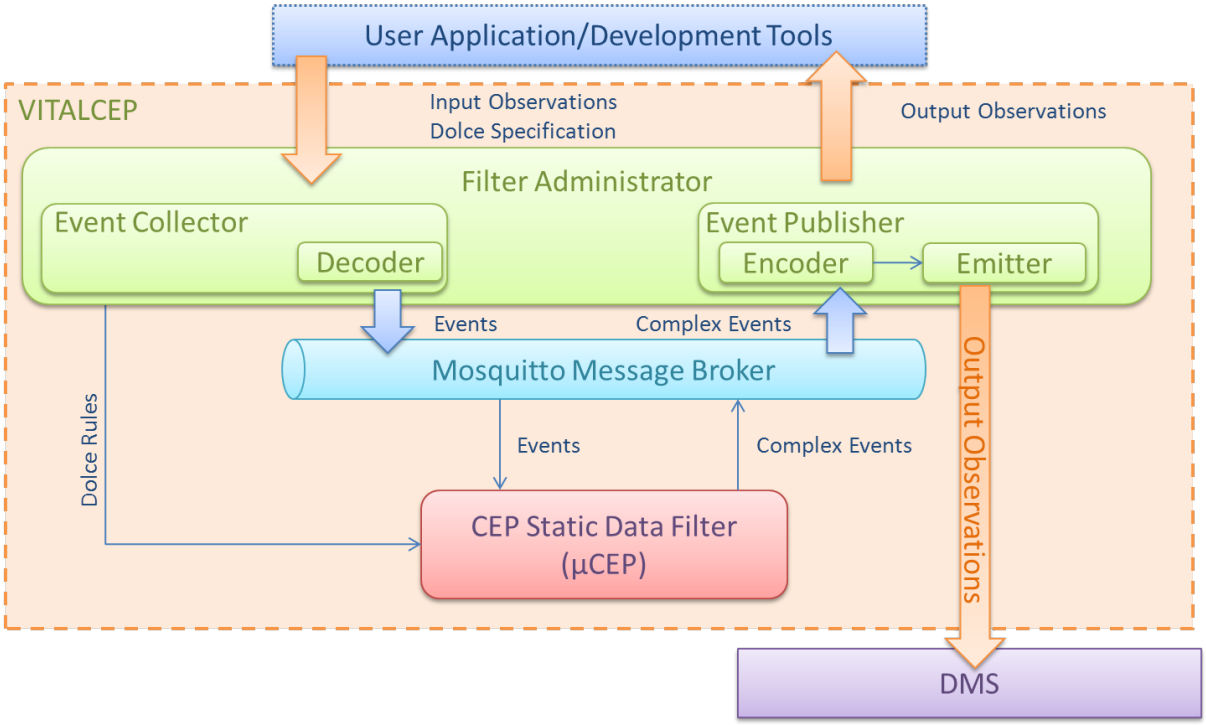

Figure 11: CEP Static Data Filter Architecture

4.3.1.2Static Query Filter

The Static Query Filter obtains new complex observations from filtering a set of observations retrieved by querying the DMS. The MongoDB query and the DOLCE rules are provided as input parameters into the filter request and the resulted observations are sent to the user. Furthermore, the process of filtering includes to query the DMS for providing input observations, obtain the results based on DOLCE specification provided, and to send the results to the DMS to be persisted. Once the service has been completed the CEP instance is removed to release resources.

The Filtering Administrator component calls to the Event Collector that has been implemented for Static Query Filter with a “Listener” and a “Decoder”. The “Listener” queries the DMS with the embedded query in the user request for obtaining the input observations, then the input observations are sent to the “Decoder” for transforming the VITAL observations into DOLCE events and sending them to the CEP static query filter. The input events are sent to the CEP static query filter by means of the Mosquitto message broker. The Filtering Administrator component also calls to the Event Publisher component to receive the complex events returned by the CEP through the message broker and to transform them into the JSON-LD Vital Observations. The event publisher has been implemented with an “Encoder” and an “Emitter” same as the Static Data Filter (4.3.1.1) to persist the observations into the DMS and return them to the user application in the response of the request. Once the filter is no longer needed the CEP instance is removed to release resources.

The Figure 12 shows the different components of the implementation of the Static Query Filter and the interactions among them.

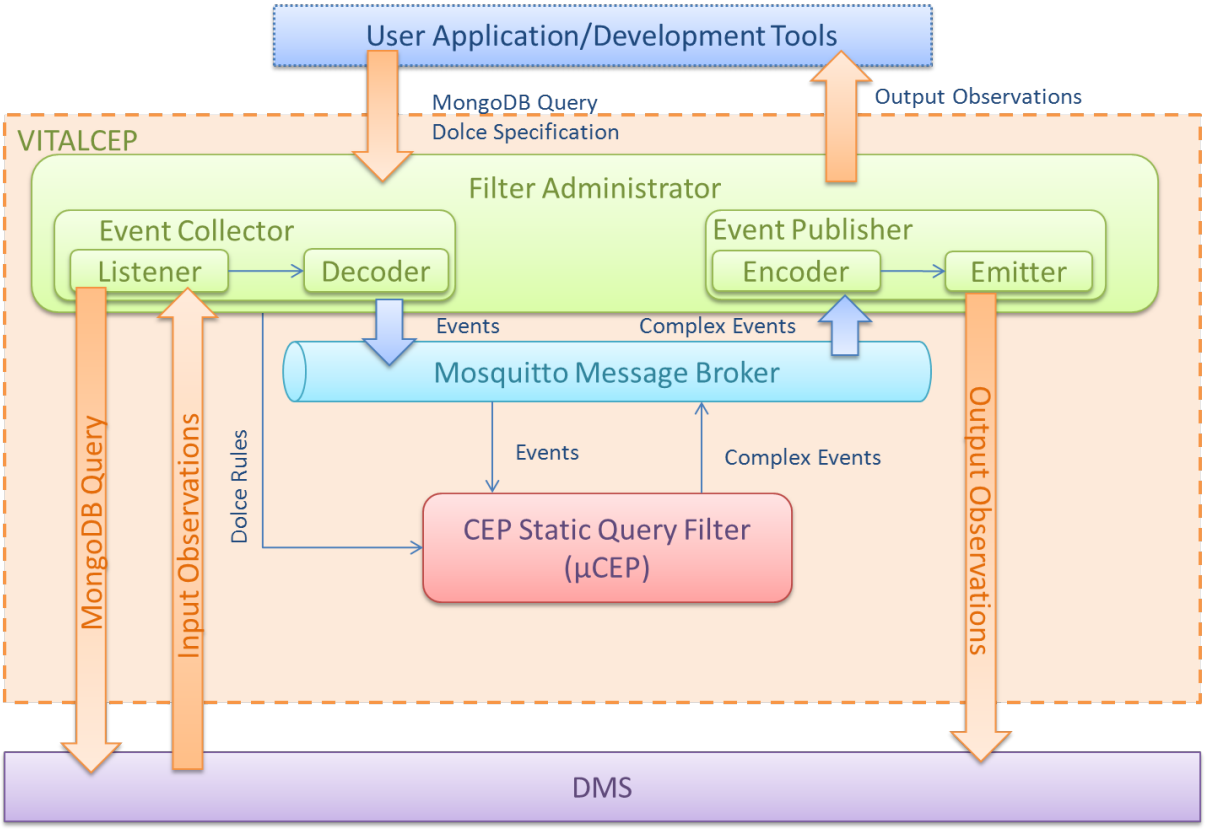

Figure 12: CEP Static Query Filter Architecture

4.3.2 Continuous filtering

A continuous filtering is the capability of attaching filtering to a data stream and receiving filtered results by listening to observations of different sources collected in the DMS till a delete filter request tells the filtering mechanism to stop. This filtering mechanism will create a CEP as a data stream endpoint, and will publish the results as “observations” continuously to the DMS until the filter is no longer needed and is stopped by the user application.

The components of the Continuous Filter are depicted in Figure 13. The Filter Administrator for the continuous filter has been redefine to create, read, update, and delete continuous filters as virtual sensors. In this case the Event Collector the “Listener” component is totally redefine for subscribing observations sources from the DMS allowing to gather historical and real time observations to be filtered. The listener process has been implemented for pulling observations from the source sensors until the user send a request to stop the filter. Since the filter is going to last longer than the creation request the Event Collector, and the Event Publisher are redefined to runs in different threads from the Filtering Administrator component as they are going to be needed until the filter is stopped. Once the CEP instance is created and the components are running the response to the user application is the identification of this new virtual sensor in order to query the filtered observations or to manage the continuous filter.

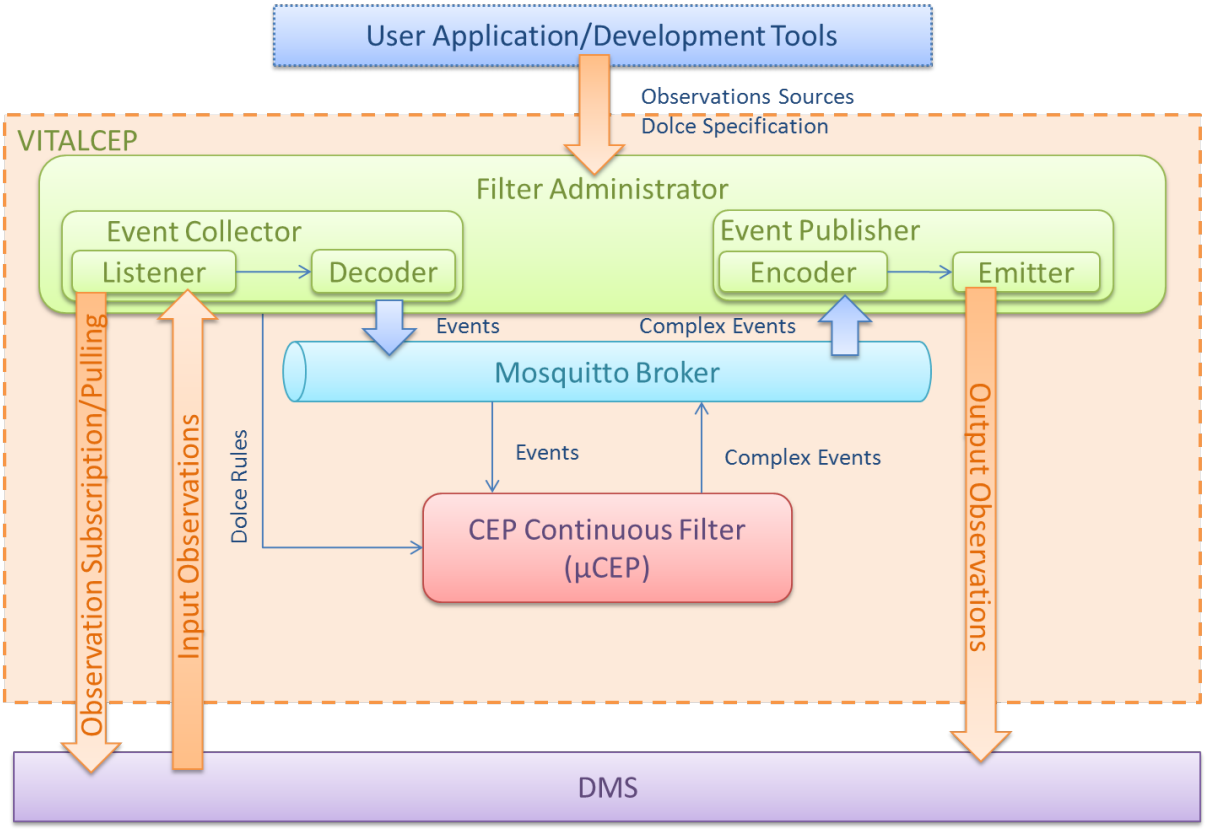

Figure 13: CEP Continuous Filter Architecture

4.4 REST Interfaces

The VITAL Event Filtering functionality is exposed through set of RESTFul Services through the Filtering administrator for managing the different type of filters it provides. These services are;

Static Query Filters:

- **FilterStaticData**: Creates a Static Data Filter to filter the provided data using the provided DOLCE rules, and returns the filtered data set.
- **FilterStaticQuery**: Creates a Static Query Filter to filter the result of a query using the provided DOLCE rules, and returns the filtered data set.

Continuous Filter:

- **GetContinuousFilters**: Returns information about previously generated continuous filters that are running in the system.
- **GetContinuousFilter**: Returns metadata about a specified continuous filter defined by its URI.
- **CreateContinuousFilter**: Creates a Continuous filter that takes data sources and DOLCE specifications as input and provides continuous observations results until the filter is no longer needed and disposed.
- **DeleteContinuousFilter**: Deletes a continuous filter based on the input id.

4.4.1 FilterStaticData Service

Table 7 shows the Rest API for filtering a static data with an example of the input request and the response. In this case the system accepts a POST request that contains an array of the observations to be filtered, and the rules that define the filtering process described in DOLCE language. The service will create an instance of the filter, provide the data to evaluate and return the output observations. At the end of the request the filter will save the observations into the DMS and the filter will be disposed.

Table 7: Filter Static Data Service

	Filter Static Data
Description	Filters the provided static data
URL	BASE_FILTERING_URL/filterstaticdata/
Method	POST

Input	Data List and DOLCE rules{
	"data": [{…
	"type": "ssn:Observation",
	"ssn:observedBy":"http://vital-integration.atosresearch.eu:8180/hireplyppi/sensor/vital2-I_TrS_136",
	"ssn:observationProperty": {
	"type": "vital:Speed" },
	"ssn:observationResult": {
	"type": "ssn:SensorOutput",
	"ssn:hasValue": {
	"type": "ssn:ObservationValue",
	"value": 82,
	"qudt:unit": "qudt:KilometerPerHour" }} …}, {…}, …],
	"dolceSpecification": {
	"id":"ppDoce",
	"complex": [
	{"id": "trafficJam",
	"definition":"group id; payload{string id = id, float value = value, pos location = location}; detect Speed where (avg(value)<80 && count(Speed) > 3) in [40 seconds];"}],
	"event": [
	{"definition": "use {string id,pos location,float value};",
	"id": "Speed"}]}
	}

Output	[{
	"ssn:observedBy": "http:// vital-integration.atosresearch.eu:8180/cep/sensor/d7e22877-f683-45ba-91b6-d487a8cf0d16",
	"ssn:observationProperty": {
	"type": "vital:ComplexEvent"
	},
	"ssn:observationResult": {
	"ssn:hasValue": {
	"type": "ssn:ObservationValue",
	"value": {
	"complexEvent": "trafficJam",
	"ssn:observedBy": "http://vital-integration.atosresearch.eu:8180/hireplyppi/sensor/vital2-I_TrS_143"
	}
	},
	"ssn:observationResultTime": {
	"time:inXSDDateTime": "2015-11-07T20:04:56+01:00"
	},
	},{…},{…}]

The request of the service requires mandatory fields, the “data” field that specifies a set of sensors or virtual sensors observations, and the “dolceSpecification” to create the CEP dolce rules. See Table 8 for details.

Table 8: Static Data Filter mandatory input fields

Field		Description
data		This field specifies an array of observations described in D3.1.2 section 3.4
dolceSpecification		This field is a JSON element to specify the DOCE rules that define the filter behaviour and is composed by “complex”, “event”, “external” and “id”. The structure of the Dolce rules are specified in detail in the D4.3.1, section 3.2.3 and in [Atos14]
	complex	JSON array of elements to specify the complex events in DOLCE composed by the “definition” of the complex rule and the “id”
	event	JSON array of elements to specify the events in DOLCE and are composed by the “definition” of the event and the “id”
	external	JSON array of elements to specify the external in DOLCE and are composed by the “definition” of the external and the “id”. The external element is optional for a dolceSpecification
	id	String Identifier for the DOLCE specification

4.4.2 FilterStaticQuery Service

The Static Query Filtering functionality is offered to apply event filtering over data stored into DMS, in this case, the parameters of the request also contains the rules to be applied for the filtering, but instead of a list with the input data as the previous filter, this request contains the MongoDB query needed to obtain the data from DMS.

The Rest API for a static query filtering is described in Table 9 with an example of the input request and response.

Table 9: Filter Static Query Service

	Filter Static Query
Description	Filters the provided static data
URL	BASE_FILTERING_URL`/filterstaticquery`

Method	POST

Input	{
	"query": {"hasLastKnownLocation.geo:lat" : {$gt : 41.06},
	"hasLastKnownLocation.geo:lon" : {$lt : 28.4} }
	
	",
	"dolceSpecification": {
	"id":"ppDoce",
	"complex": [
	{"id": "trafficJam",
	"definition":"group id; payload{string id = id, float
	value = value, pos location = location}; detect Speed where
	(avg(value)<80 && count(Speed) > 3) in [40 seconds];"}],
	"event": [
	{"definition": "use {string id,pos location,float value};",
	"id": "Speed"}]}}
	}

Output	[{
	"ssn:observedBy": "http:// vital-
	integration.atosresearch.eu:8180/cep/sensor/d7e22877-f683-45ba-91b6-
	d487a8cf0d16",
	"ssn:observationProperty": {
	"type": "vital:ComplexEvent"
	},
	"ssn:observationResult": {
	"ssn:hasValue": {
	"type": "ssn:ObservationValue",
	"value": {
	"complexEvent": "trafficJam",
	"ssn:observedBy": "http://vital-
	integration.atosresearch.eu:8180/hireplyppi/sensor/vital2-I_TrS_143"
	}
	},
	"ssn:observationResultTime": {
	"time:inXSDDateTime": "2015-11-07T20:04:56+01:00"
	},
	},{…},{…}]

The mandatory field of a Static Query Filter are “query” that, specifies the query to obtain the input data and “dolceSpecification”. See Table 10.

Table 10: Static Query Filter mandatory input fields

Field	Description
Query	This field specifies a MonngoDB query to the DMS to get the input observations to be filtered. The query is going to be executed as a request to the “DMS_BASE_URL/query” service described in D3.2.3, section 4.1.2.3
dolceSpecification	See Table 8

4.4.3 GetContinuousFilters

This method offers the functionality of getting all the continuous filters created, that are running at the time of the request, expressed in JSONLD. This method does not need any input parameter.

Table 11: Get Continuous Filters details list

	Get Continuous Filter Specifications
Description	Gets details list of all continuous filters created by user
URL	BASE_FILTERING_URL/getcontinuousfilters
Method	GET
Input	-

Output	**Example**
	[
	{
	"@context": "http://vital-iot.eu/contexts/sensor.jsonld",
	"id":"http:// http://vital-integration.atosresearch.eu:180/cep/sensor/d7e22877-f683-45ba-91b6-d487a8cf0d16",
	"name": " Traffic Incidents Continuous Filter.",
	"type": " CEPSensor ",
	"description": "Continuous Filter for Traffic Incidents.",
	}
	]

4.4.4 GetContinuousFilter

This method is used to retrieve the metadata of a specific Continuous Filter. The required input parameter is the URI of the filter that was been sent in the response to a filter creation request. It returns all data related to the filter, including the rules applied in the filtering process expressed in Dolce language.

Table 12: Get Continuous Filter details

	Get Continuous Filter Specification
Description	Gets details of a specified continuous filter
URL	BASE_FILTERING_URL/getcontinuousfilter
Method	POST

Input	{
	"id":"http:// http://vital-integration.atosresearch.eu:180/cep/sensor/d7e22877-f683-45ba-91b6-d487a8cf0d16"
	}

Output	**Example** { "@context": "http://vital-iot.eu/contexts/sensor.jsonld", "id":"http:// http://vital-integration.atosresearch.eu:180/cep/sensor/d7e22877-f683-45ba-91b6-d487a8cf0d16" , "name": "Live Traffic Data.", "type": "CEPFilterSensor", "description": "CEPICO for Live Traffic Data.", "ssn:observes": [{ "type": "vital:ComplexEvent",

<table>
<tr><td></td><td><pre>
        "uri": "http://localhost:8180/cep/sensor/e6c69c9c-6aa8-42cd-9174-1fda1a4f0f87/trafficJam"
      }
    ],
     "source":"http://vital-integration.atosresearch.eu:8180/hireplyppi/sensor/vital2-I_TrS_143" ,
"dolceSpecification": {
      "id":"ppDoce",
"complex": [
         {"id": "trafficJam",
                "definition":"group id; payload{string id = id, float value = value, pos location = location}; detect Speed where (avg(value)<80 && count(Speed) > 3) in [40 seconds];"}],
"event": [
         {"definition": "use {string id,pos location,float value};",
"id": "Speed"}]}
   }
}
</pre></td></tr>
</table>

4.4.5 DeleteContinuousFilter

This method deletes a filter. The input is the URI of the filter that was sent in the response to a filter creation request.

Table 13: Delete a Continuous Filter

	Delete Continuos Filter Specification
Description	Deletes a specified continuous filter specification
URL	BASE_FILTERING_URL/deletecontinuousfilter
Method	POST
Input	{ "id":"http:// http://vital-integration.atosresearch.eu:180/cep/sensor/d7e22877-f683-45ba-91b6-d487a8cf0d16"}
Output	-

4.4.6 Create/UpdateContinuousFilter

The main input parameters of this method are the sources for subscribing to the input observations and the rules of the filtering expressed in Dolce. These two parameters are mandatory. See Table 14.

Table 14: Static Query Filter mandatory input fields

Field	Description
Sources	This field specifies an array of sensors or virtual sensors in order to get those sensors observations from the DMS (D3.2.3) as input data for the filtering.
dolceSpecification	See Table 8

This method has a parameter “id” which is optional. If this parameter is empty in the request, the Filter Admin module will create a new filter to attend the request. Or if

the “id” parameter contains a valid value that point to a running filter, this filter will be updated with the new data.

Table 15: Create or Update a Continuous Filter

	Create / Update Continuous Filter Specification
Description	Creates or Updates the specified continuos filter
URL	BASE_FILTERING_URL/createcontinuousfilter
Method	POST

Input	{
	"name": " Live Traffic Data.",
	
	"description": "Continuous Filter for Traffic Data.",
	"source":"http://vital-integration.atosresearch.eu:8180/hireplyppi/sensor/vital2-I_TrS_143",
	"dolceSpecification": {
	"id":"ppDoce",
	"complex": [
	{"id": "trafficJam",
	"definition":"group id; payload{string id = id, float value = value, pos location = location}; detect Speed where (avg(value)<80 && count(Speed) > 3) in [40 seconds];"}],
	"event": [
	{"definition": "use {string id,pos location,float value};",
	"id": "Speed"}]}}}

Output	{
	"id":"http://
	http://vital-integration.atosresearch.eu:180/cep/sensor/d7e22877-f683-45ba-91b6-d487a8cf0d16"
	}

5 CONCLUSION

In the final release of this deliverable, we have introduced the Filtering mechanisms that will be available in the VITAL architecture. This enables the filtering of data and streams, as well as the generation of events according to user or applications needs. We analysed the state of the art, to identify the different filtering activities that can be performed. We have positioned it within the global architecture, presented the requirements and expectations of the module and explained our reasoning for the technical implementation choices we have made. We described involved technologies, main interfaces, current status of the service and the interactions between Filtering operations and the other elements of the platform.

This document describes the interfaces and the functions that the Filtering module is going to provide to other VITAL modules or to the user applications. Such descriptions include the characterization of prerequisites, in term of input required by the various operation, and the format of the produced output. We also included a description of the internal structure of the components that have been developed, the design and the implementations details.

6 REFERENCES

[Ca4iot12] C. Perera, A. Zaslavsky, P. Christen, D. Georgakopoulos, "CA4IOT: Context Awareness for Internet of Things" IEEE International Conference on Green Computing and Communications (GreenCom), 2012

[Cascom13] C. Perera, A. Zaslavsky, M. Compton, P. Christen, D. Georgakopulos, "Semantic-driven Configuration of Internet of Things Middleware". Ninth International Conference on Semantics, Knowledge and Grids (SKG), 2013

[Garnet03] L. St Ville and P. Dickman, “Garnet: A Middleware Architecture for Distributing Data Systems Originatingin Wireless Sensor Networks”. In the Proceedings of the23rd International Conference on Distributed Computing Systems workshops. IEEE 2003.

[Cougar] Cougar Project.www.cs.cornell.edu/database/cougar

[Gsn07]K. Aberer, M. Hauswirth, and A. Salehi, “Infrastructure for data processing in large-scale interconnected sensor networks”. International Conference on Mobile Data Management, May 2007

[Cupus14] I.P. Zarko, K. Pripuzic, M. Serrano, M. Hauswirth, "IoT Data Management Methods and Optimisation Algorithms for Mobile Publish/Subscribe Services in Cloud Environments". European Conference on Networks and Communications (EuCNC), 2014

[EPCGlobal09] The Application Level Events (ALE) Specification, Version 1.1.1 Part I: Core Specification," EPCglobal Ratified Standard (2009)

[Ferman02] A. M. Ferman , J. H. Errico , P. Van Beek , M. I. Sezan, "Content-Based Filtering and Personalization Using Structured Metadata". In Proceedings of 5the Second ACM/IEEE-CS Joint Conference on Digital Libraries, 2002

[Impala03] Ting Liu and Margaret Martonosi. 2003. Impala: a middleware system for managing autonomic, parallel sensor systems. SIGPLAN Not. 38, 10 (June 2003), 107-118. DOI=10.1145/966049.781516 http://doi.acm.org/10.1145/966049.781516

[Kefalakis09] Nikos Kefalakis, Nektarios Leontiadis, John Soldatos, Didier Donsez: Middleware Building Blocks for Architecting RFID Systems. MOBILIGHT 2009: 325-336

[Kefalakis2011] Nikos Kefalakis, John Soldatos, Nikolaos Konstantinou, Neeli R. Prasad, APDL: A reference XML schema for process-centered definition of RFID solutions, Journal of Systems and Software, Volume 84, Issue 7, July 2011, Pages 1244-1259, ISSN 0164-1212, http://dx.doi.org/10.1016/j.jss.2011.02.036.

[Milan04] W.B. Heinzelman, A.L Murphy, H.S Carvalho and M.A Perillo, “Middleware to Support Sensor Network Applications”. IEEE Network Magazine Special Issue, Jan. 2004

[Su09] X. Su, T. M. Khoshgoftaar "A Survey of Collaborative Filtering Techniques". Advances in Artificial Intelligence Volume 2009, 2009

[W3C08] SPARQL Query Language for RDF - *W3C Recommendation 15 January 2008* – Available online at: http://www.w3.org/TR/rdf-sparql-query/

[Zoller13] S. Zoller, C. Vollmer, M. Wachtel, R. Steinmetz, A. Reinhardt, "Data Filtering for Wireless Sensor Networks Using Forecasting and Value of Information". IEEE 38th Conference on Local Computer Networks (LCN), 2013

[Atos14] Atos Research and Innovation, “Dolce Language Specification”, https://repository.atosresearch.eu/index.php/s/ygH9764F9uiuI7T, 2014

[Atos16] Atos Research and Innovation (Internet of Everything Lab), “ATOS reference architecture for distributed, scalable and “cloudified” complex event processing”, https://repository.atosresearch.eu/index.php/s/TNzdI7WeAg7Zhfz, 2016

Vital 2016

