D4.1.3 Information and Data Lifecycle Management: Design and open specification (Final)

WP4 Information and Data Lifecycle Management

Version: 1.0
Due Date: 30/04/2016
Delivery Date: 30/04/2016
Nature: Report
Dissemination Level: Public

Lead partner: IBM
Authors: Adnan Akbar (University of Surrey), Guy Hadash (IBM), Achilleas Marinakis (NTUA), Juan Rico (ATOS), Eran Rom (IBM), Juan Sancho (ATOS), Paula Ta-Shma (IBM), Gil Vernik (IBM)
Internal reviewers: Adnan Akbar (University of Surrey)

www.iot-cosmos.eu

The research leading to these results has received funding from the European Community's Seventh Framework Programme under grant agreement n° 609043

Version Control:

Version	Date	Author	Author's Organization	Changes
0.1	21/4/2016	Paula Ta-Shma	IBM	Initial version, based on last year's document. Created template for this year's document.
0.2	24/4/2016	Paula Ta-Shma and Guy Hadash	IBM	Updated sections 3, 4.1.6, 4.5.5, 4.6, 4.6.2, 5, and other minor updates
0.3	25/4/2016	Adnan Akbar	Univ Surrey	Added sections 4.5.6 and 4.5.7
0.4	25/4/2016	Paula Ta-Shma	IBM	Added section 3.1 and other minor updates. Released for internal review.
0.5	26/4/2016	Adnan Akbar	Univ Surrey	Univ Surrey review comments
1.0	28/4/2016	Paula Ta-Shma	IBM	Final version for submission to the EU

Table of Contents

Table of Figures

1 Overview

This work package includes COSMOS components dealing with Internet of Things (IoT) data management throughout the lifecycle of the system. For an IoT platform such as COSMOS, there are several key phases in the information lifecycle. Firstly, massive amounts of IoT data need to be ingested into the system. This data needs to be amenable both for analysis in real time as well as reliably stored for subsequent efficient batch analysis. In addition data needs to be managed over the long term in a scalable and cost effective manner, for example including policies for data reduction.

The IoT domain presents many challenges in the domain of information and data lifecycle management. The IoT domain requires large scale data management at low cost. Data will be generated by a large number of devices and will need to be ingested into the system reliably in real time. Moreover, incoming data needs to be analysed in real time and in a way that enables reacting to events detected by the analysis. In addition many kinds of analysis can only be done with data collected over a period of time. Therefore data needs to be collected and stored persistently in order to support search and analysis on historical data. Moreover the data needs to be stored in formats suitable for IoT data and amenable to analysis. Mechanisms should be developed which allow analytics frameworks to access the data in an efficient way, bringing computation close to the storage instead of moving the data to the framework requiring the computation.

In order to support low cost, a scale out architecture using commodity hardware components is warranted. In addition, new data will continually be born into the system and storing all raw data is costly. Therefore data reduction and archiving techniques are needed in order to reduce the cost of storing the data.

COSMOS can exploit the special nature of IoT workloads as we have seen in the COSMOS use cases. IoT workloads typically generate time series data and can be described by certain schema conventions. Moreover, time series data is append only so we can focus on data collection and analysis rather than transaction processing. This can enable dealing with the high scalability and throughput requirements of IoT workloads.

In summary, our approach for long term storage is to focus on low cost and scalable storage, formats which allow data reduction as well as analysis, and analysis close to the storage. In addition we focus on real time analysis of data and high throughput data ingestion. These aspects are key for our COSMOS use cases and we believe they also apply to a large class of IoT data management problems.

The purpose of this document is to describe the overall architecture for this work package and to describe the design of its various components and their interactions. Deliverable 4.2.3 will cover the implementation aspects of these components.

The year 3 revision of this document contains a new data flow architecture (section 3.1), and new and updated material in sections on the Scalable Data Mapper (section 4.1.6), on Integrating the Data Store with an Analytics Framework (section 4.5) including in particular subsections on projection and predicate pushdown (subsection 4.5.5) and its application to use cases (subsections 4.5.6 and 4.5.7) , as well as on the Kafka based Message Bus (subsection 4.6.2). The basic architecture has been enhanced and the focus in year 3 is on application to use cases.

2 Requirements of IoT workloads

For convenience, we list the requirements relevant to this work package here. The reader is referred to Annex 1 of Deliverable 2.2.1, which contains a list of requirements for all work packages in COSMOS. These requirements are addressed by the various components of the WP4 architecture, discussed in the next section.

UNI ID	Category	Description
4.1	Data Store	There must be a mechanism to collect raw data and make it persistent.
4.2	Data Store	There should be a mechanism to map raw data to a format that is suitable for subsequent search and analysis. This requires metadata extraction and possibly data transformation.
4.3	Data Store	There should be a mechanism to search for data according to its metadata.
4.4	Data Store	There should be a mechanism to perform data analysis.
4.5	Data Store	This mechanism would define APIs that are available to the application developer in order to implement application specific analysis.
4.6	Data Store	The mechanism for data analysis should enable computation to run close to the stored data in order to reduce the amount of data sent across the network.
4.7	CEP	Large amounts of raw data should be able to be processed in a real-time manner
4.8	CEP	A semantic analysis tool should be run prior to analyze a data stream
4.9	CEP	Publishing sub-system offer data broadcasting based on semantic analysis results
4.13	CEP	A rule-definition language should be specified along with a set of introductory rule examples
4.14	CEP	The system should be able to implement time-slicing windows for the analysis of temporal patterns
5.0	Data Set	The system must provide mechanisms in order to characterize objects (meta-data).
UNI.041	Data Set	COSMOS could provide historical information about the physical entity.
5.5	Data Distribution	COSMOS must provide mechanisms for distributed data-storage (Cloud Storage).

3 High Level Architecture

This work package includes components handling data management throughout the system lifecycle. The reader is also referred to deliverable D2.3.1 which discusses the overall COSMOS architecture, and here we focus on the data management components. Here we propose an architecture for scalable IoT data management as shown in Figure 1.

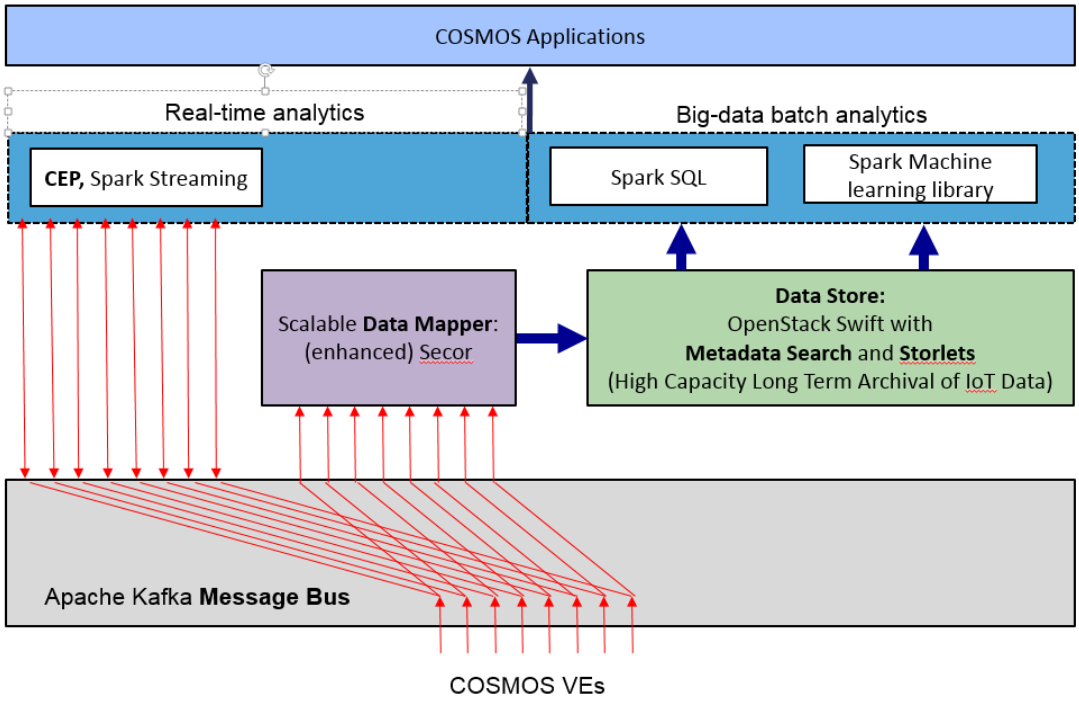


Figure 1: Scalable IoT Data Management Architecture Proposal

COSMOS data flows through the system via a Message Bus which is organized into topics, where each component can publish and/or subscribe to topics. In Year 2 we propose using Apache Kafka for the COSMOS Message Bus because of its performance and support for scalable ingestion of data to Swift. More details can be found in the Apache Kafka section 4.6.2.

The Complex Event Processing (CEP) component is responsible for processing data and analyzing it in real time, according to application specific logic. This component can subscribe to certain topics in the Message Bus and analyze the data flowing through these topics. It can also publish its output to (possibly different) topics in the Message Bus. For example, if a certain event is detected by CEP, this may trigger the generation of certain messages to a new topic. Applications and other components can subscribe to this topic in order to react to the event. The CEP component is described in detail in section 4.2.

The Data Mapper is responsible for persistently storing data flowing through the Message Bus in the Data Store. Certain topics in the Message Bus will be marked as persistent and these should be stored without losing messages. The Data Mapper will periodically collect data from

a message bus topic, extract metadata, transform it to a data layout suitable for subsequent metadata search and analysis, and upload the data to persistent storage. It may collect useful statistics about the data collected such as the data flow rate and data sources and topics. In years 2 and 3 we use the open source Secor tool as the basis for our Data Mapper, since it supports scalable data ingestion. More details can be found in section 4.1.6.

Further transformations can be done asynchronously by storlets, if needed, once the data resides in persistent storage. For example, storlets may be used to combine data from several different time series to form a merged time series, or to down sample time series data after a certain period of time has elapsed if high precision is no longer necessary.

IoT data is stored persistently in COSMOS in a reliable and scalable fashion using OpenStack Swift. The Data Store component is described in section 4.3.

The metadata search component allows applications, users and other components to search for COSMOS data according to metadata it was annotated with. This capability is important since there will be very large amounts of data and finding it without a search capability will not be feasible. This component is described in detail in sections 4.3.2 and 4.3.3 and it has a REST API which is described in Appendix 7.3.

Since massive amounts of data need to be stored and later analyzed, it is important to enable the analysis to take place close to the data where possible to avoid transferring large amounts of the data across the network. This will be done using the storlets framework. This is described in detail in section 4.4 and it has a REST API which is described in Appendix 7.4.

In order to enable big-data batch analytics on our IoT data, we integrate OpenStack Swift with the Apache Spark analytics framework. As a part of this integration, we enable storlets to be run close to the storage in order to pre-process data before transferring data to Spark. This can both reduce the amount of data transferred across the network and also be used for privacy preservation.

Recently there has been considerable development in the Apache Spark community including significant enhancements of a framework called Spark SQL which allows representing and querying/analyzing structured and semi-structured data within Spark. In year 2 we used Spark SQL to query and analyze IoT data stored in OpenStack Swift. For this purpose we use Apache Parquet, one of the formats supported by Spark SQL, as an open storage format for this data. We enabled the Data Mapper (Secor) to upload data into Swift in Parquet format. More details are provided in sections 4.1.6 and 4.5.8.2.

In addition, Spark SQL now supports an External Data Source API, which allows querying data sources external to Spark. We support our Data Store (OpenStack Swift with metadata search and storlets) as a Spark External Data Source. More details are provided in section 4.5.5, including showing how Spark SQL is used by our machine learning computations to access the data.

3.1 Data Flow Architecture

We developed a generic data flow architecture which is applicable to a wide variety of IoT use cases, and demonstrated it for both the Madrid Traffic use case and the III Taiwan use case. Figure 2 shows the data flow architecture for our approach. In real time, data flows from IoT devices to the COSMOS Message Bus (Kafka). The CEP component subscribes to this data feed and makes decisions according to CEP rules, possibly resulting in actuation of IoT devices. In order for the CEP component to have ***intelligent*** rules, we perform the batch flows which

apply machine learning to the entire history of the IoT devices. For example, in the Madrid Traffic use case, CEP rules use traffic speed and intensity thresholds, and these thresholds are generated by analyzing the historical data. This allows the CEP rules to automatically adjust the thresholds based on the history of a particular traffic sensor – because a speed of 30km/hour could be excellent for a busy city intersection at rush hour but could indicate unusual congestion on a freeway at night time. The use of machine learning allows automating the process of setting the "correct" thresholds, a task which is difficult and error prone to do manually.

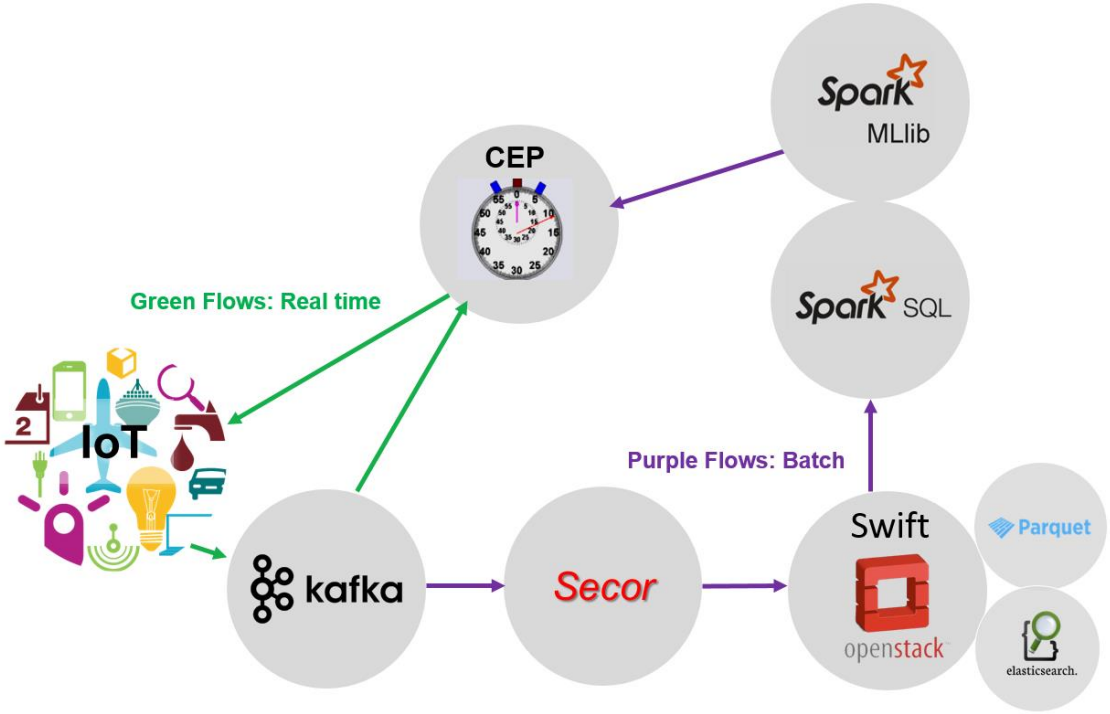


Figure 2 COSMOS Generic Data Flow Architecture

In order to enable machine learning on IoT historical data, the COSMOS Data Mapper (an extended version of Secor) subscribes to the Kafka feeds, aggregates them into objects with metadata and stores them in the COSMOS Object Store (OpenStack Swift) in Parquet format. Since the COSMOS Object Store supports metadata indexing and search (built on ElasticSearch), search indexes are also generated for the object metadata.

The machine learning component accesses the data in the COSMOS object store using Spark SQL queries. These queries are made efficient by using metadata search to find those objects which are relevant to the query. We use the Spark machine learning library which has a large (and growing) number of machine learning algorithms to choose from. The output of the Spark machine learning process is information which can be used to construct intelligent CEP rules. In the Madrid Traffic use case, this is speed and intensity thresholds. In year 3 we are working to further develop this use case to handle additional data sources and build more complex rules, although the overall data flow architecture remains the same.

4 Component Descriptions

4.1 The Data Mapper

4.1.1. Data Rate Requirements

The Data mapper is responsible for the ingestion of IoT data flowing on the Message Bus into the cloud storage. In order to better understand the performance requirements for this component we looked at the ingestion requirements for the COSMOS use cases:

EMT Madrid

- Records typically contain around 86 bytes (including odometer data, GPS location etc.)
  - Note that this does not include alarms, accelerometer data and so on. Therefore this is a minimal estimate.
- 1800 buses
- 3 events per minute for each bus
- 18 hours per day of activity

This generates 478 MB per day of raw data.

The required ingestion rate ***during the hours of activity*** is 7.55 kB/s.

Camden

- Records typically contain around 150 bytes
- The data we currently have flowing in the Message Bus covers 3 buildings from the Ampthill estate, 21 storeys each, with around 800 residents in total.

The average data rate is 1.37 kB/s

This generates 116 MB of raw data on average per day.

Note that in general the Camden use case would generate more data – the above is just for the 3 buildings included in the pilot.

In-Synergy III Taiwan

- Records typically contain 95 bytes
- There are currently around 3000 households publishing data to the In-Synergy system
- The frequency of data sampling varies between every minute to every 30 minutes per household (5 minutes by default)

This generates between 13.04 MB and 391.39 MB of raw data per day.

This generates a data rate of between 0.15 kB/s and 4.63 kB/s.

Conclusion

COSMOS needs to handle multiple applications concurrently. This means that COSMOS needs to scale both in the number of applications and in the number of entities per application (e.g. buses or households). Clearly the sampling frequency greatly influences the data rate and the amount of raw data generated.

We plan to investigate what data ingestion throughput the Data Mapper can achieve.

4.1.2. Functional Overview

The Data Mapper is a component which subscribes to the topics which are flagged as persistent in the message bus, reads periodically data published from the Virtual Entities and transforms them into a format suitable for persistent storage in the cloud, annotating them with enriching metadata. Scalability and Reliability concerns are examined by the Scalable Data Mapper (see section 4.1.6), in order to be able to handle large amounts of data and to ensure that no message will be lost.

The main functionalities provided by the Data Mapper component are:

- Create objects with size relevant to cloud storage
- Extract metadata both form raw data and from the Social Analysis component described in D5.1.1 and D5.1.2

This component addresses requirement 4.1 by providing a mechanism to collect raw data and make it persistent. In addition, it meets requirement 4.2 since it provides a mechanism to map these raw data to a format that is suitable for subsequent search and analysis and also extracts metadata from them.

4.1.3. Design Decisions and Details

Here we discuss design decisions for the Data Mapper. Implementation details will appear in deliverable D4.2.3. For Year 1 we made the following design decisions:

- Data objects are stored in the cloud through Openstack/Swift component;
- Each account can be accessed by many users. Data Mapper component is one of these users;
- Containers correspond to the use cases;
- Many objects can correspond to one Virtual Entity;
- Each object is associated ,at least, with the following metadata:
    - The Id of the Virtual Entity which publishes the data (data type: string);
    - Timestamps (data type: date Time);
- Metadata data types can be string, integer, double, geospatial and date Time;

4.1.4. Use Cases for Data Mapper

The Y1 component's functionalities are explained using an example arising directly from the Camden use case:

In Camden, a flat can be considered as a Virtual Entity and each flat has some temperature sensors. The flats subscribe to the message bus and publish their data in a JSON format. Data Mapper component also subscribes to the message bus and reads periodically the messages coming from all the Virtual Entities (flats). The description of one object is shown below:

- User: Data Mapper
- Container: Camden
- Object Name: data_from_flat_109_from_2015-04-04T18_13_12_to_2015-04-04T18_14_11
- Metadata:
    - Start – d: 2015-04-04T18_13_12
    - End – d : 2015-04-04T18_14_11
    - Id: cuimnHeqgGSU

4.1.5. Communication with other Components

The Data Mapper component collaborates with the following components of the COSMOS project:

- Message bus (WP4): subscribes to the topics stored in the bus and consumes the data
- Social Analysis (WP5): requests for social metadata like Trust, Reputation, Reliability and Dependability indexes. The latter change over time since they are calculated based on feedback provided by VEs when they receive information (e.g. experience sharing) from other VEs. The dependability index, which is an aggregation of the other three ones, is a double number normalized between 0 and 1. This functionality will be implemented in year 2 and/or 3.
- Cloud Storage and Metadata Search (WP4): stores data objects, with their metadata, in the cloud storage.

4.1.6. Scalable Data Mapper

We built a scalable Data Mapper based on the open source Secor tool [10] developed by Pinterest. Secor is a service which allows persistently storing topics from Apache Kafka in Amazon S3 [11]. We enhanced Secor by enabling OpenStack Swift targets, so that data can be uploaded by Secor to Swift. This feature was contributed to the Secor community. In addition we enhanced Secor by enabling data to be stored in the Apache Parquet format, which is supported by Spark SQL. Moreover, we enhanced Secor to generate Swift objects with metadata.

We chose Secor because of the following key features:

- Horizontal scalability: Secor can be scaled out to handle increased load by starting additional Secor processes. It can also be distributed across multiple machines.
- Configurable upload policies: size based and time based policies are both supported.
- Output Partitioning: Data can be parsed and stored under partitioned Openstack Swift paths. This is useful since Spark SQL can access partitioned data in an optimized way.
- Reliability: Secor is fault tolerant and strongly consistent.

The data mapper is used to continuously upload the use case data from the COSMOS Message Bus (Kafka) to COSMOS object storage (OpenStack Swift).

Implementation details for the (scalable) Data Mapper can be found in the Data Mapper section (Section 3) of Deliverable D4.2.2.

In Year 3 we plan to assess the performance and scalability of this approach by measuring the ingestion throughput that can be achieved by using Apache Kafka and Secor to store IoT data in Swift.

4.2 Complex Event Processing

A Complex Event Processing engine (CEP engine) is a software component capable of detecting asynchronously, independent incoming *Events* of different types and generating a *Complex Event* –synthesized, mega, total– out of these events. In this sense, we can introduce *Complex Events* as the output generated after processing many small, independent incoming input events, which can be understood as a given collection of parameters at a certain temporal point. A CEP engine is commonly provided with a series of *plugins* or additional sub-components in order to improve the step of acquiring data from external data sources or provide a sort of business logic to the outputted, generated information. Thanks to its functionality, this tool provides the means for dealing with large amounts of raw data streams in a real-time manner, especially at the VE side, what is likely to be the case in both COSMOS use cases such as the buses in Madrid scenario and environmental sensors in the Camden one.

4.2.1. Functional Overview

In an effort to strengthen the deployment phase of the CEP engine, the proposed solution has been given a modular and configurable design. This way, it is possible to be deployed both as a VE level stream analysis tool and as a Platform level stream analysis tool, while also it is fully parameterizable so to be specifically adapted to the running environment. Three different scenarios are conceived for the deployment of the CEP engine:

1) **Internal event detection**: The Virtual Entity enhances its execution environment hosting a lightweight CEP engine flavour. Additionally to the inherent raw data provided by the local VE, a number of external data sources could feed the engine. Besides that, the output of the engine (complex events) can be locally consumed by the VE or even shared to other VEs or external components.
2) **Event detection as service**: Aims at providing stream analytics services to VE components that lack hardware resources or are incompatible for hosting an event processing solution. Depending on information exchange between VEs, other VEs can also subscribe to and consume results provided by this event detection service.
3) **Hybrid event detection**: Under certain circumstances, for instance performance and processing capabilities, it will be valuable to take advantage of the modularization of the CEP engine. The most common situation will be that in which a VE hosts locally an *Event Collector* module, while the *Complex Event Detector* and the *Complex Event Publisher* module may be hosted in a remote location, where more hardware resources are made available. This “modules” are explained later on this document.

4.2.2. Key Design Decisions

4.2.2.1. Rule-based inference engine

Due to the fact that complex event processing is a relatively recent discipline, there is no convergence to the type of language or methodology used for specification what complex events are. However, there are two main approaches:

- Continuous query in which user specifies SQL-like query, from which a continuous stream of results is obtained.
- Specification of rules by means of specific domain language as DOLCE [21] or Proton [27].

4.2.2.2. Knowledge inference as service

The COSMOS platform will offer event detection capabilities for external components and VEs. This custom knowledge inference service will be made available via two possible interfaces: an Admin REST API and a web-based wizard. This way, it will be possible to inject new or modify existing detection rules using the DOLCE Domain Language in an easier and controlled manner.

4.2.2.3. Distributed deployment

COSMOS is designed to coordinate huge number of heterogeneous IoT based systems. Therefore there is a large potential for utilization of a distributed CEP engine for independent detection of various situations in sub-networks and/or physical distribution of processing resources to several processing nodes. This also creates a possibility to apply load balancing tactics for optimized resource utilization.

4.2.2.4. Integration in COSMOS

The CEP engine will be used for a number of operations such as data analysis, VEs monitoring, context acquisition, prediction, etc. Being a key asset of the overall infrastructure, the CEP engine will be integrated with the Message Bus, which will then integrate COSMOS components with all external components.

4.2.3. System Architecture

The CEP engine being used in the COSMOS project is known as **µCEP** (micro CEP), which derives from the SOL/CEP engine provided by the FIWARE project [18]. From a conceptual and functional point of view, both SOL/CEP and µCEP are pretty much the same, although the latter has disassembled the functionality of the former in four different software modules, as can be viewed in Figure 3. Thanks to the modularity and configurable design of the µCEP, this engine is being used both at a VE level and at a Platform level, thus adapting to where it is deployed and the hardware resources available.

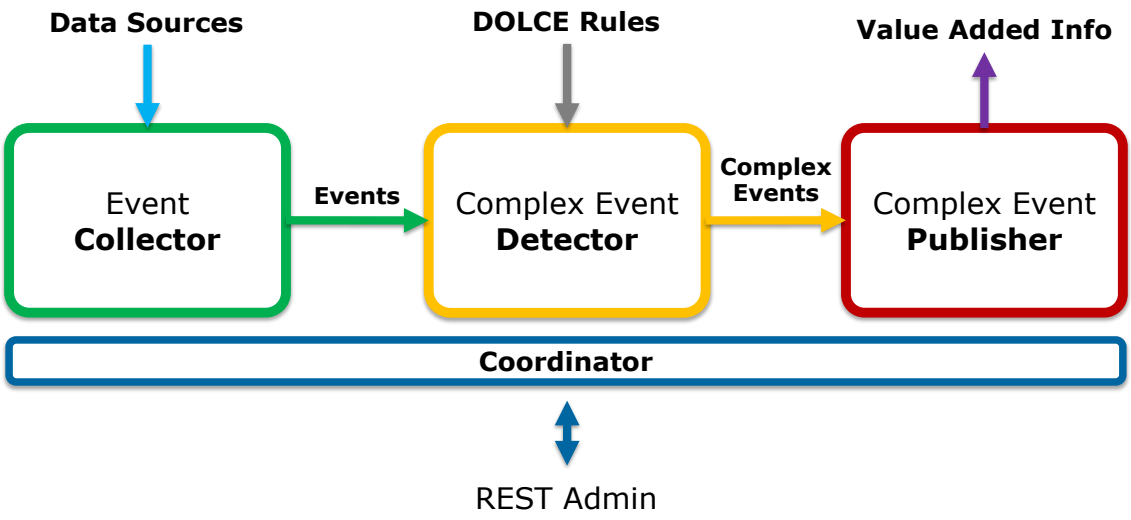


Figure 3: µCEP - System Architecture

Entering through the *Event Collector* and coming out the *Complex Event Publisher*, the *information* pass through the *Complex Event Detector*, wherein a *DOLCE Rules* file is applied. The four functional modules represented in this architecture are introduced in the following subchapters.

4.2.3.1. Event Collector

The entry point to the µCEP is called *Event Collector*, whose primary goal is gathering all the information coming from chosen sources, through different communication protocols and using varying data formats –this is known as data feeds or data sources–. Once the module has been fed with data, its secondary objective consists in transforming the information into a specific data format, an *Event*, which will be then outputted to the *Complex Event Detector* module. The following Figure 4 depicts its functionality.

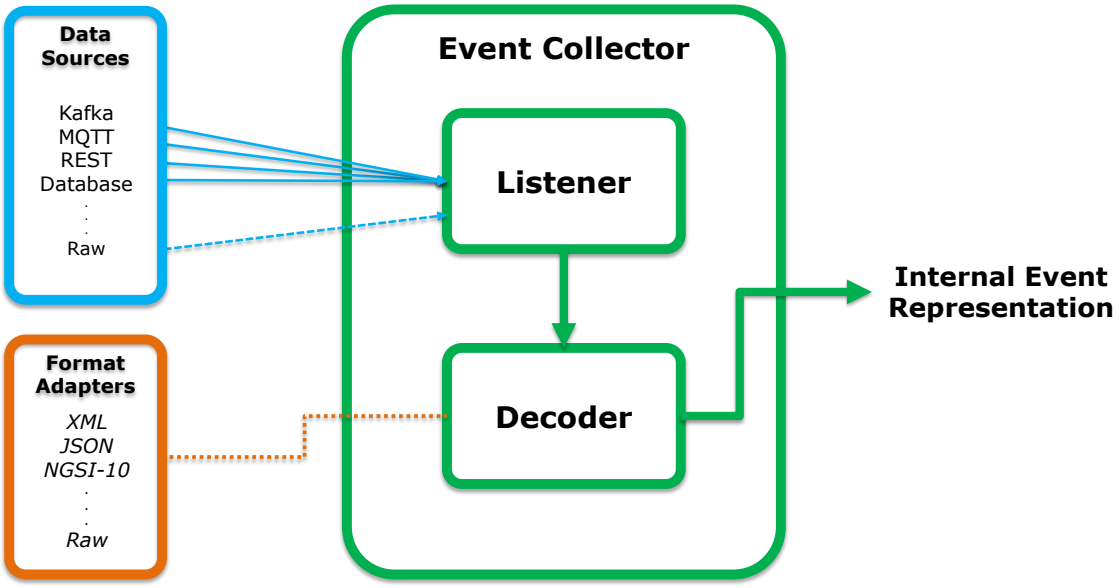


Figure 4: µCEP - Event Collector module

Extracting the *Event Collector* module out of the kernel's engine provides better flexibility at the development stage. It is up to the *system developer* to choose which messaging protocol subscribe to –Apache Kafka, MQTT, AMQP, XMPP, etc. – or even provide the internal *Listener* sub-module the ability to request data periodically to a REST API data feed, for example. In this sense, when running the engine in a constrained hardware board, the *Event Collector* module can be written in a way that implements the less required functionality (software libraries *per se*), thus maximizing the amount of memory needed to operate. Concerning the *Decoder* sub-module, its purpose is to translate the acquired data into the internal representation understood by the µCEP, what in turn must match the *Event* clause as defined in the *DOLCE Rules* file –what will be explained in the following subchapter.

4.2.3.2. Complex Event Detector

The *Complex Event Detector* module can be understood as the kernel of the CEP engine. It controls event detection and production of expected results by using temporal persistence of volatile events until constraints of a rule(s) are entirely satisfied.

Being extracted from the previous SOL/CEP engine, this module has been specifically re-written so to be implemented solely using the standard C++ library, thus getting rid of unnecessary external libraries inherit from its big brother. With this small, code-efficient implementation –less than 300 KB once compiled– we are able to take advantage of a powerful module for real-time analysis of streaming data, while at the same time deploy it in a very easier manner through the variety of IoT-enabled hardware boards. In this sense, the engine is capable to run in major hardware boards running C++ compatible operating system, for instance the Raspberry Pi [19] or the UDOO board [20].

Figure 5 depicts, from a high level perspective, the internal submodules that take part on the complex event detection process.

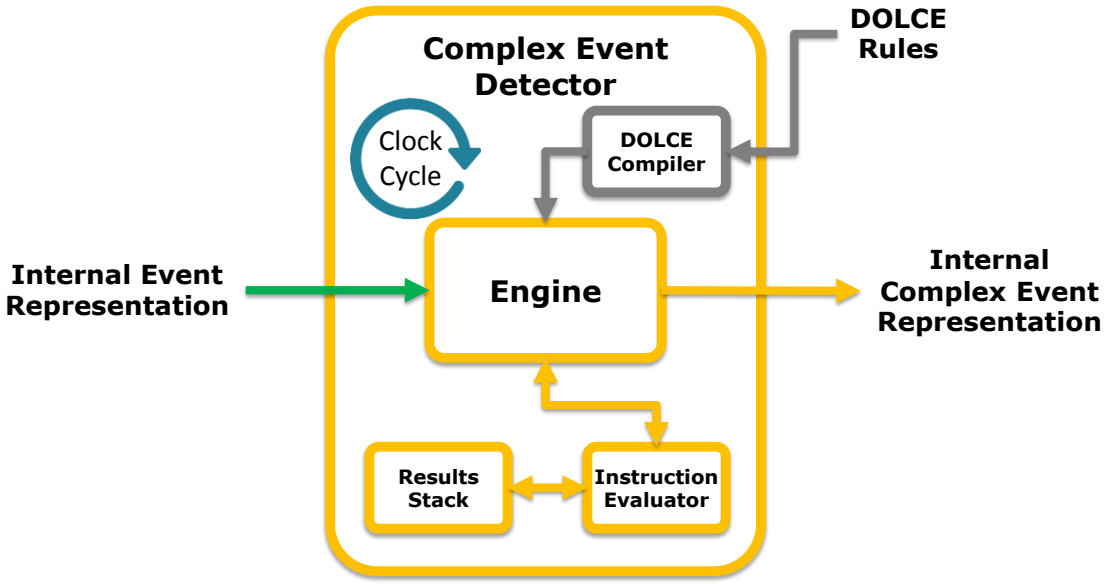


Figure 5: µCEP – Complex Event Detector module

Each *DOLCE Rules* file served to the µCEP suffers a process of compilation, so that the converted rules are processed in a more efficient manner during runtime. Meanwhile, every time the internal clock dispatches a *tick,* the *Engine* manages the sliding windows in combination with the *Instructor Evaluator*, where every *Complex clause* is checked –as defined in the rules file. When a *Detect clause* is evaluated to *true* the engine composes a *Complex Event* integrating the parameters included in the *Payload clause*, and outputs it to the *Complex Event Publisher* module.

The entire lifecycle process can be better understood by referring to the DOLCE Language Specification, which has been summarized and adapted in the following subchapter.

4.2.3.2.1. DOLCE Rules

DOLCE [21] stands for "Description Language for Complex Events". It was designed for the CEP engine created by the Smart Objects Lab, part of ARI Research & Innovation, a division of ATOS Spain S.L.

DOLCE language follows two main lines of development. On the one hand, it is conceived as a declarative language so as to minimize programming expertise to be able of using it. On the other hand, it considers real life scenarios from the design phase to make it simpler to use. This includes built-in types that deal with location and temporal awareness, as well as basic functions that facilitate the integration of business logic in the component.

As it has been already mentioned, the CEP engine is based on the analysis of *Events* and the generation of *Complex Events*. The first are the basic inputs derived from the event collector, the former are the result from executing the rules included in the *Complex Event Detector*, and are delivered to the *Complex Event Publisher*. What DOLCE does is providing an easy way to trigger these *Complex Events* based on the execution of a business logic that considers as input the simple *Events* detected by the system.

In order to better explain how DOLCE works, this section will follow an example increasing the functionality and power step by step, starting from the detection of an event and finishing with an advanced example of CEP functionalities.

Event declaration

The following minimal DOLCE program detects an event called *TemperatureReading*.

```
event TemperatureReading
{
}
```

An event is declared using the **event** keyword, followed by an identifier stating the name of the event which the CEP must detect.

The previous program tells the CEP to consume and interpret the events. However, it is of no use, since it does not have any knowledge about complex events. The following code adds a complex event called *HeatWave*, by means of the **complex** keyword.

```
event TemperatureReading
{
}
complex HeatWave
{
}
```

With the previous code there is still no complex event generated, it is just the declaration of a complex event. For the generation of a complex event it is necessary to detect at least a simple event, this is declare by using **detect** key word as follows.

```
event TemperatureReading
{
}
complex HeatWave
{
        detect TemperatureReading;
}
```

This code generates a complex event every time a *TemperatureReading* event comes in. Once the definition and declaration of events and complex events is done, it is necessary to dig in them so as to provide a most accurate definition of the business logic and rules that system will follow.

Event attribute and filtering

In DOLCE, events can include several attributes as parameters, the way it is implemented is by the key word **use**. But events are quite generic and due to the large amount of different entities that can generate one, they can be filtered so as to precisely collect the input event that is relevant for a specific rule. The filtering is done using **accept** reserved word.

```
event TemperatureReading
{
        use
        {
                int SensorId,
```

```
                int Temperature,
        };

        accept { SensorId == 17 }; // only accept
        events from sensor #17

}
```

It is also possible to include arithmetic operations in all the statements of the DOLCE code, moreover sub-expression grouping is possible using brackets and following operation hierarchy.

Sliding Time Window

For triggering complex events, many times it is not enough to have a single event compliant with certain rule, but requires a set of them during a limited period of time. This is known as Sliding Time Window, the window start at the current point and goes back the amount of time defined in the condition.

```
event TemperatureReading
{
        use
        {
                int SensorId,
                int Temperature,
        };
        accept { SensorId = 17 };
}
complex HeatWave
{
        detect TemperatureReading
        where Temperature > 36
        in [ 3 days ];

}
```

The **in** statement is the keyword that is required for the specification of the time units that cover the time where the events will be considered. Besides *days*, other temporal units are hours, minutes, seconds, years and months, together with their linguistic singular counterpart's *day, hour, minute*, etc. In a graphical way, the Figure 6 displays how this feature works.

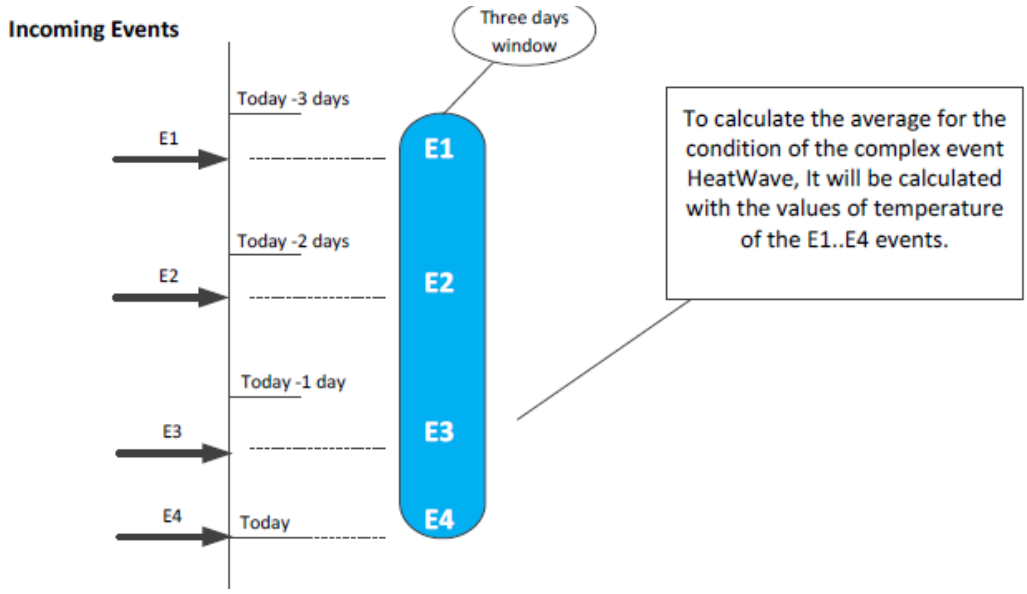


Figure 6: Time Sliding Windows representation

Sliding Tuple Window

In order to select a group of events it is also possible to pick the last n events instead all happened in a time window. The declaration is also using the keyword **in**, but in this case just a scalar number is included without any time unit.

The name is implied from that used in the detect statement. However, this notation is not allowed if more than one event where involved in the detection, because the CEP would not know for which event to follow the occurrences.

Complex event functions

The time and tuple window provide a list of events that requires a processing action before generating the complex event. In this sense, there are several actions that can be performed. The most representative ones are presented in the following table:

Statement	Description
avg	This function calculates the average of one or several attributes received by the CEP
count	This function counts the occurrences of events
sum	This function reflects the total result of all the values of particular event attribute
diff	This function calculates the relative difference between two expressions or an event value

External Variables

The DOLCE language provides a feature that accepts external variables that can be transmitted to the CEP at runtime. These external variables can be used to influence the behavior of the events and complex events, without having to modify their definitions.

An external variable is declared by using the *external* keyword, followed by the type and name of the variable and a mandatory default value. As a convention, external variables are spelled in capital letters.

Data Types

DOLCE is able to deal with several data types. As it was said at the beginning of this description this programming language focuses on real life situation, thus the data types allowed are presented in the following table:

Data Type	Description
int	A signed integer value, whose range is limited by the architecture on which the CEP is deployed
float	Floating point number, whose range is limited by the machine architecture on which the CEP is deployed
string	0-terminated string of 8-bit characters
duration	Period of certain length

Constants

When declaring external variables and complex event payloads, they can be assigned values of a certain type. The following example includes the declaration of a constant *sensorId* and also a external variable *MINIMUM_TEMP*.

```
complex MyAlert
{
        payload
        {
                int sensorId = 2334;
        };
}
external int MINIMUM_TEMP = 15;
```

All the information presented above represents the high level specification of DOLCE programming Language. Starting from this basic information, it is possible to grow in complexity and create advanced business logic rules by combining them.

4.2.3.3. Complex Event Publisher

The last step of the data lifecycle consists in traversing the *Complex Event Publisher* module. This module receives *Complex Events* and delivers them to the selected *Data Sinks*, reformatting the *Complex Event* portion into the data format selected by the *system developer* according to the expected application-defined external components. It can be seen as the opposite functionality provided by the *Event Collector* module, as it also supports various communication protocols and data formats, as seen in Figure 7.

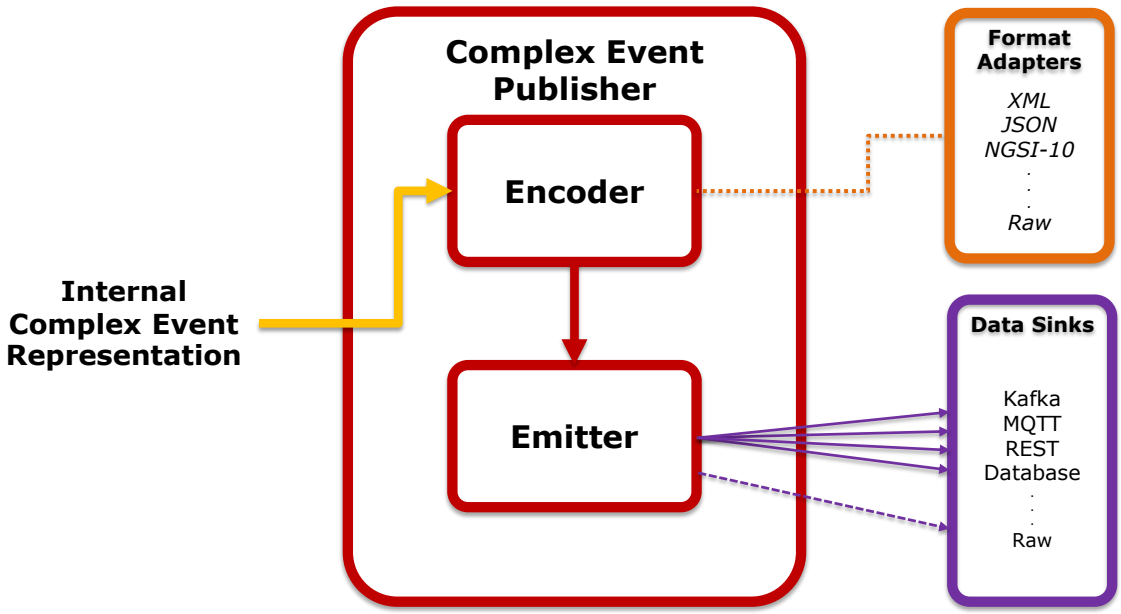


Figure 7: µCEP – Complex Event Publisher module

4.2.3.4. Coordinator

There have been developed a set of management functionalities, but instead of providing this intelligence in each module, a dedicated *Coordinator* module orchestrates the correct behavior of the µCEP as a whole. Its main duties comprise the management of:

- The execution state of the µCEP
- The correct intra-communication between the internal modules
- The provision of proper DOLCE rules file(s)
- The maintenance of an internal knowledge base of the µCEP
- The profiling of 'execution schemas' to ensure safety/fallback against over aggressive conditions

In order to provide these features, the *Coordinator* module relies on configuration files. In an effort to facilitate external and remote operation of the µCEP, a specific *REST Admin* interface is provided. Annex 7.2 introduces the API and next versions of this document and the prototype report will complement its description.

4.2.4. Communication Interfaces

This section describes the intra-communication between the different modules of the µCEP and the inter-communication of the µCEP with the rest of COSMOS components.

4.2.4.1. Interface between internal modules

The disaggregation of µCEP in multiple modules allows the distributed implementation of the whole system. However, this implies several requirements in terms of communication and information sharing. Moreover, one of the key objectives of the approach we are following is the minimization of latency and UDP [22] is the best protocol for real time low latency communications.

UDP communications expose several threats that must be considered in the design of the system. In order to provide a stronger security scheme it is possible to include security layers on top of UDP stack such as DTLS [23] which is quite light and still valid for simple machines.

This last point reflects an important aspect of the approach followed in the implementation of the µCEP. The split of functional components in different machines allows the simplification of the engine to the minimal expression, thus having a more powerful external collector and publisher with actuation capabilities that goes far beyond the ones provided by constrained machines.

Scalability issues will be explained below in section 4.2.5, but for depicting how powerful this solution is Figure 8 shows how it is possible to take advantage of simple communication paradigms for building a reliable structure.

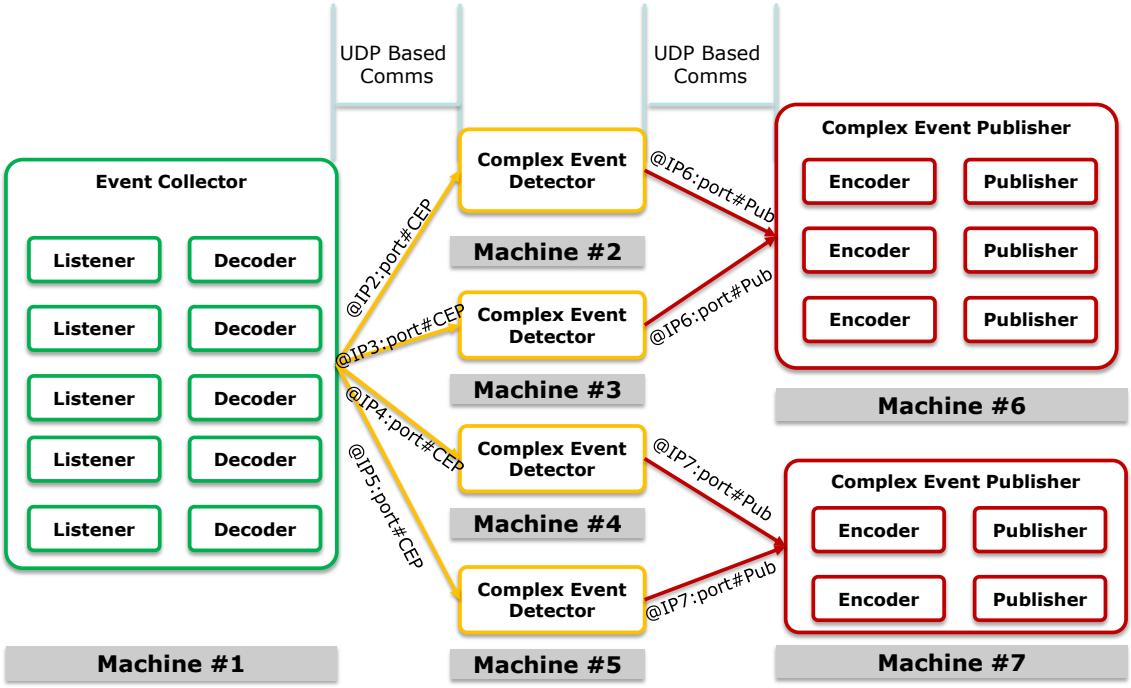


Figure 8: Generic communication scheme of CEP

4.2.4.2. Interface between µCEP and other COSMOS components

The µCEP engine is a component intended to receive information from varying data sources such as weather channels, traffic state feeds, or geolocalization data from VEs, but also will receive inputs from other COSMOS components, for instance the Planner. In this sense, the communication of the µCEP relies on the Message Bus, being the *Event Collector* and the *Complex Event Publisher* modules the ones who have to subscribe and publish, respectively, to dedicated *topics*.

Given that the Message Bus was implemented using RabbitMQ during the first year of the project, we are providing an adapter that can be used with the underlying protocol managed by this framework, the AMQP pub/sub protocol [24]. Thanks to the modular implementation of the µCEP, it has been possible to provide and additional *Event Collector* module for the new

implementation of the Message Bus occurred during project's year two, the Apache Kafka pub/sub messaging [25]. All in all, there can be multiple *Event Collector* modules listening to different data sources, since it is just a matter of managing accordingly which one feeds *Events* to a certain µCEP running instance.

In order to ease the communication of the µCEP with the rest of components, Node-red is introduced [26]. Node-RED is a tool for wiring together hardware devices, APIs and online services in new and interesting ways, providing a browser-based flow editor that makes it easy to wire together flows using the wide range nodes in the palette. Flows can be then deployed to the runtime in a single-click. This tool is running at http://lab.iot-cosmos.eu:1880/nodered. Once logged, the 'Inter-communication' workspace shows a deployment where a number of *Input Nodes* have been connected to the *Event Collector* module, represented by a *Subflow* that has the logic to translate the needed data formats. The same applies for the *Output Nodes*. Once the workspace is completed, it is quite easy to go node by node setting the needed configuration requirements accordingly. Figure 9 shows the resultant Inter-communication workspace.

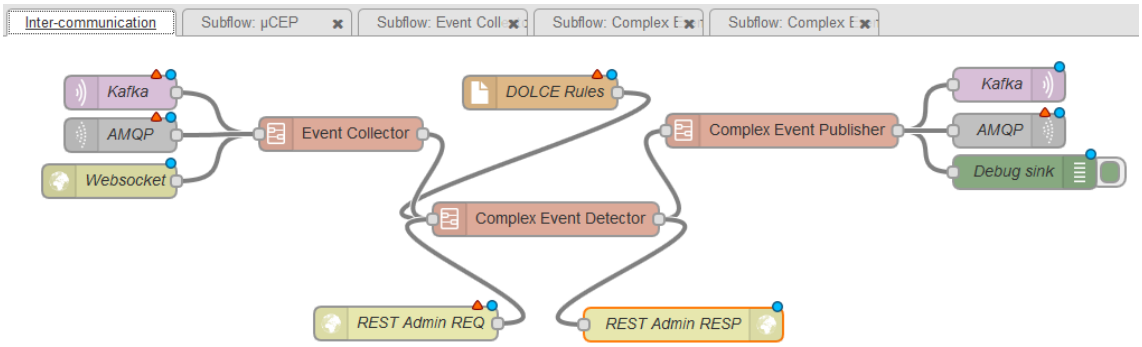


Figure 9: Inter-communication µCEP workspace on Node-RED

4.2.5. Scalability

One of the key aspects of any CEP engine is how to address the large amount of entities that will take part in IoT scenarios in the near future. The possibilities flow in two directions. On one hand, increasing the processing capabilities of individual VEs, so to have full CEPs with all the functionalities in each one. On the other hand, the minimization of complexity in devices and having dedicated machines running efficiently part of the functionalities.

COSMOS has chosen the second option so as to be reused in any scenario, since the first approach can be seen as a particularization of the second. Figure 10 depicts an example of a µCEP instance where *Event Collector* is running in a machine (machine#1) and the *Complex Event Detector* module and *Complex Event Publisher* module are running in (machine#2).

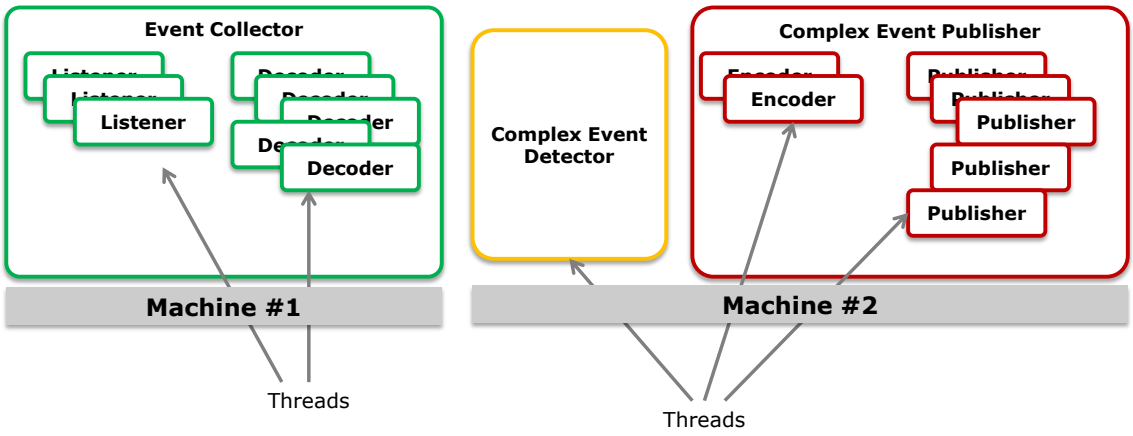


Figure 10: De-coupling of modules

µCEP is conceived as a staged pipeline and the architecture has been designed to work in a multithreading environment so as to allow each actor of the stage to be instantiated as many times as necessary thus allowing parallel processing. Finally, this modular distributed approach requires being non-blocking and lock-free messaging that guarantees efficient CPU usage and performance.

µCEP targets IoT devices, which is expected to cope with many different requirements and cover a broad range of hardware and software components. Going from the simplest meter to really complex powerful machines, however COSMOS and many applications focuses on the first group due to challenges they represent. Far from seeing their constraints as a problem, they introduce a lot of technical challenges and improvement opportunities.

Taking into account the future scenarios and the µCEP architecture, scalability is assured by disassembling the functionalities and linking instances to different µCEPs which are the most critical components and whose individual implementation has been reduced to require the minimum resources possible.

4.3 Data Store

The COSMOS data store is used in order to store historical IoT data. Important functionality related to the Data Store includes

- Representing data as objects and an ability to create, access and delete those objects
- Enabling efficient search for data objects
- Enabling computation on this data to occur close to the storage
- Integrating with analytics frameworks

We now discuss the Data Store itself in more detail. The purpose of the COSMOS Data Store component is to persistently store COSMOS data and make it available for search and analysis. The open source OpenStack Swift object storage software will be used in order to implement the COSMOS Data Store. Object storage is a cost effective way to store large volumes of data.

Object storage allows defining CRUD operations (Create, Read, Update, Delete) on entire objects, and write-in-place is not supported. This can be suitable for storing historical IoT data (typically time series data) which does not change over time. Objects can be organized into

containers, and each container belongs to an account. Account, container and object CRUD operations can be performed using the Swift REST API [1].

Objects, containers and accounts in Swift can be annotated with metadata key-value pairs, and this metadata can be updated. Metadata updates rewrite the entire set of key-value pairs, so in order to update a single key-value pair it is necessary to perform read-modify-write of the metadata.

This component will persistently record historical information about COSMOS Virtual Entities, and therefore addresses requirement UNI 041. It will be implemented using distributed cloud storage frameworks such as OpenStack Swift, and therefore addresses requirement 5.5. OpenStack Swift supports annotating data with metadata, this capability can be used to address requirement 5.0.

Object storage is suitable for long term storage and archiving of IoT data as well as batch analytics on it. Together with CEP (the COSMOS component which processes IoT data in real time), object storage has been sufficient for the COSMOS use cases. We note that some IoT use cases may also require a low latency data store, although since this was not required by our use cases, this is beyond the scope of COSMOS.

4.3.1. Data Representation

Recall from the architecture diagram in figure 1 that data flows from VEs via the Message Bus and Data Mapper into the Data Store. Regarding the data representation, we assume the following for COSMOS:

1. Each COSMOS application is mapped to a Swift account. For example the EMT bus application could be mapped to one account and the Camden city council application could be mapped to another account.
2. The data from a VE in a COSMOS application is typically published to a particular Message Bus topic. If this topic is specified as persistent by the Registry component then it is mapped to a Swift container under the corresponding account.
3. VEs periodically publish ‘messages’ to the Message Bus in json format. Multiple such messages are collected by the Data Mapper and published as a single Swift object which contains multiple ‘records’.
4. In addition objects can be annotated with metadata such as the start and end timestamps for a Swift object.
5. Objects may be stored in their original JSON format or they may be transformed into another format such as Parquet.
6. The Cloud Storage can optionally store a schema for Message Bus data on a per topic basis. This schema should be specified using the Avro format [28]. Schemas allows the data to be analysed effectively using Spark SQL.

4.3.2. Metadata Search

In order to make metadata useful for applications one needs the ability to search for objects (or containers, accounts) based on their metadata key-value pairs. This functionality is not

supported by Swift today. Currently, Swift stores objects as files and metadata as extended attributes of those files. This means that in order to find objects (or containers) with particular metadata key-value pairs one would need to iterate through large numbers of objects (containers) while filtering them according to their extended attributes, resulting in very large amounts of disk I/O, which is not feasible. Therefore we extend Swift with an ability to search for objects (containers, accounts) according to their metadata keys and values.

For COSMOS, we have the following requirements for metadata search

- The architectural approach needs to be scalable since we expect large amounts of data to be indexed.
- Loosely coupled integration with Swift is preferable to reduce dependencies.
- Indexing metadata should be done asynchronously to object/container creation requests so as not to increase the latency of such requests.
- In order to support efficient ingest of metadata into the index, the updates can be batched.
- It is reasonable for the index to be slightly out of date with respect to the object storage metadata, which may happen as the result of asynchronous operations and batching.
- COSMOS data often involves timestamps, geo-spatial coordinates, numerical measurements (temperature, speed, energy usage etc.). Therefore data types for this kind of data should be supported. They are needed in order to search the data correctly. In Year 1 we supported string, number, date and geo-point data types in metadata search.
- Range searches should be supported (i.e. allow searching for values in certain ranges), for example to search for objects containing temperature readings within a certain time interval.
- In year 1 we also supported bounding box searches for geospatial metadata.
- An extension to the Swift REST API should be provided which supports metadata search.

The metadata search component meets requirement 4.3, since it provides a mechanism to search for data according to its metadata.

4.3.3. Metadata Search Architecture

Our approach is to use an open source search engine called Elastic Search (ES). ES is built on top of the Java Lucene search library, and provides a REST API, logging, scale out and resiliency. The integration with Swift is done using Swift proxy server middleware, which allows plugging in user defined code as part of the request flow. Metadata search has two main flows, an indexing flow as shown in Figure 11 and a search flow as shown in Figure 12. For each of these cases, we plug in specific code for metadata indexing/search.

The indexing flow intercepts regular Swift creation (PUT), update (POST) or delete (DELETE) requests. In order to allow indexing of metadata to happen reliably and asynchronously to Swift creation, update or delete requests, a persistent message queue (Rabbit MQ) is used. Note that this could be a separate deployment of Rabbit MQ or it could use a separate Rabbit MQ Exchange within the same deployment as the Message Bus component. If the request succeeds, associated metadata is sent to the ES index via the message queue. The pink arrows below denote the parts of the flow that are added for metadata indexing. The HEAD request retrieves metadata from Swift for indexing. Note that the response to the Swift request can be returned before the metadata reaches the index.

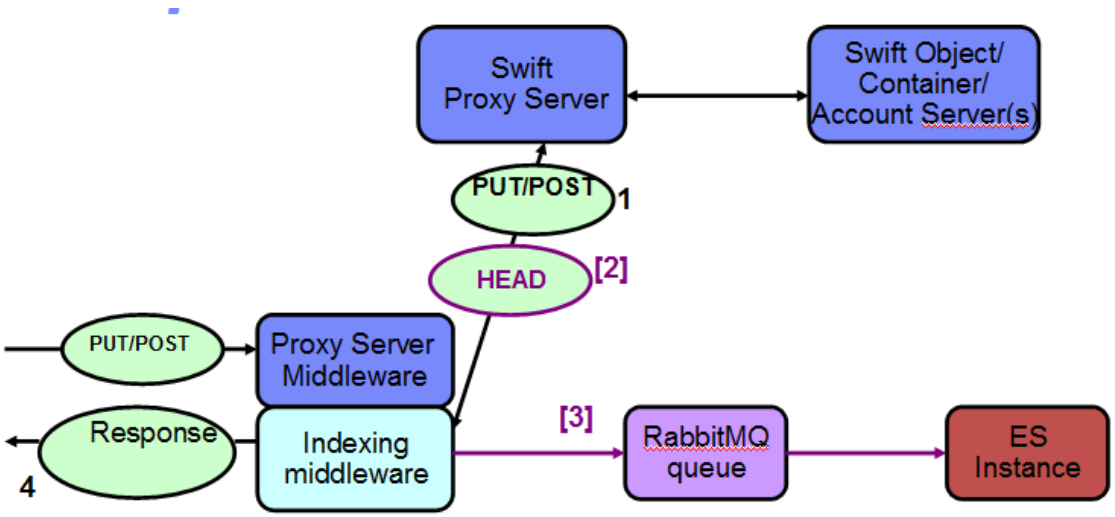


Figure 11: Metadata Indexing Flow

The search flow is a Swift GET request with a header identifying it as metadata search. In these cases, the metadata search middleware plugin is activated and diverts the request to ES after converting it to an appropriate ES search. The search results are returned to the user. Note that in this case the request does not reach Swift, and uses an extended API specifically for metadata search.

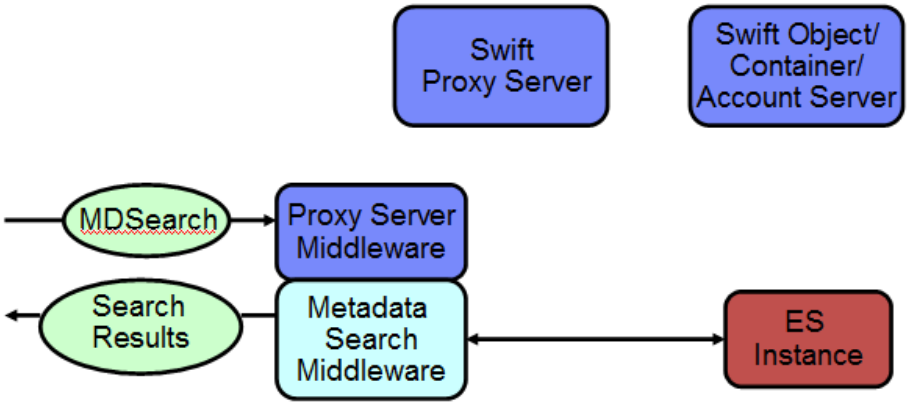


Figure 12: Metadata Search Flow

The metadata search API appears in Appendix 7.3

In year 2 we used metadata search in order to support predicate pushdown in Spark SQL using the External Data Source API. For more details see 4.5.5.

4.4 Storlets

In COSMOS we use object storage to economically store large amount of IoT data. When processing this data it is advantageous to run some of the computation close to the storage instead of transferring the entire dataset to the location of the computation. This can

significantly reduce the amount of data transferred over the network. This is particularly relevant when integrating the COSMOS Data Store with analytics frameworks such as Apache Spark. Besides reducing network bandwidth, another use case for storlets which is relevant to COSMOS is ensuring that private data does not leave the storage cloud, by applying privacy filtering storlets which run close to the storage. Storlets can also be useful for format conversion – in particular for time series and geospatial data conversion – and for generating metadata than can later be used for metadata search.

4.4.1. Overview

Storlets are computational objects that run inside the object store system. Conceptually, they can be thought of the object store equivalent of database store procedures. The basic idea behind storlets of performing the computation near the storage is saving on the network bandwidth required to bring the data to the computation. Computation near storage is mostly appealing in the following cases:

1. When operating on a single huge object, as with e.g. healthcare imaging.
2. When operating on a large number of objects in parallel, as e.g. with a lot of time series archived data.

The storlet functionality in COSMOS is developed in the context of the Openstack Swift object store[1]. The high level architecture section below describes how we integrate the storlet functionality into Swift.

Running a computation inside a storage system, involves two major aspects: one is resource isolation and the other is data isolation. Resource isolation has to do with making sure the computation does not consume too many resources, so that the storage system stability and on-going operation are not compromised. Data isolation has to do with making sure that the computation can access only the data it is supposed to access. Achieving resource and data isolation is done by sandboxing the computation. The sandboxing technology section below describes in more details the way in which the storlets computation is isolated.

Our storlets implementation supports two scenarios referred to as the PUT and GET scenarios. In the PUT scenario the storlet is invoked during object upload, where instead of keeping the data (and user metadata) as they are being uploaded, the storage system keeps the result of the storlet invocation over the uploaded data and metadata. This scenario is useful for e.g. metadata extraction: consider a case where the uploaded data is a .jpg, the storlet can extract the jpg information (resolution, geospatial coordinates, etc.), and keep it as Swift metadata.

In the GET scenario the storlet is invoked during object retrieval, where instead of getting the object's data (and metadata) as kept in the object store, the user gets back the result of the storlet invocation on the object's data (and metadata). This scenario is useful for e.g. analytics pre-filtering: consider an analytics program that is done over logs, where we are only interested in 'ERROR' lines. A storlet that runs near the data can filter out all other lines resulting in a reduced bandwidth usage between the store and the analytics engine.

This component addresses requirement 4.4 by providing a mechanism for data analysis, as well as requirement 4.6, by enabling computation to run close to the storage. In addition, APIs are defined in the appendix which allows developers to write application specific analysis using storlets. This meets requirement 4.5.

[1] https://wiki.openstack.org/wiki/Swift

4.4.2. High Level Architecture

4.4.2.1. Openstack Swift Architecture Essentials

At a high level Openstack Swift has two layers: A proxy layer in the front end, and a storage layer at the back end. Users interact with the proxy servers that route requests to the backend storage layer. A typical flow of a request that operates on some object is as follows: the request hits the proxy server that (1) authorizes the request and (2) looks up in which storage server the requested data is kept. Then the proxy server forwards the request to the designated storage server (also known as object server). See Figure 13 which describes the Swift request flow.

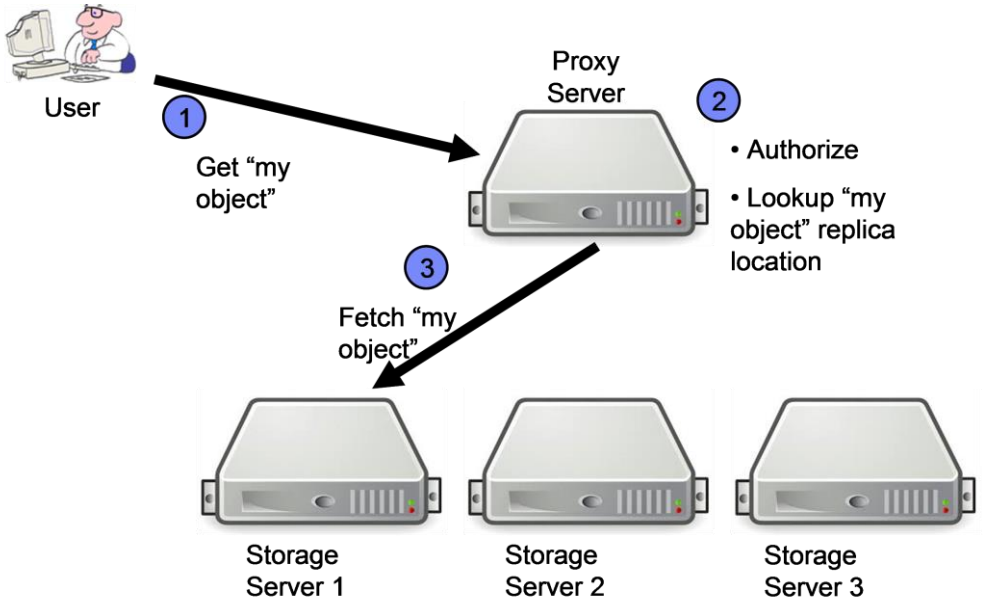


Figure 13: Request Flow in Swift

Swift is implemented using WSGI [8] technology that allows to plug-in functionality into the request processing. Each request hitting a WSGI based server goes through a pipeline of such plug-ins, called middleware. For example, amongst the plug-ins that consist the pipeline at the proxy server are an authorization middleware, quota related middleware and a 'router' middleware that forwards the request to the appropriate server according to the location of the request target resource.

4.4.2.2. Storlets' Architecture Components

The storlet functionality implementation consists of 2 WSGI middleware plug-ins: one in the proxy server pipeline and the other in the storage server pipeline. Alongside the middleware, each storage server runs a sandboxed daemon process where the storlet code is executed. See Figure 14 which illustrates this.

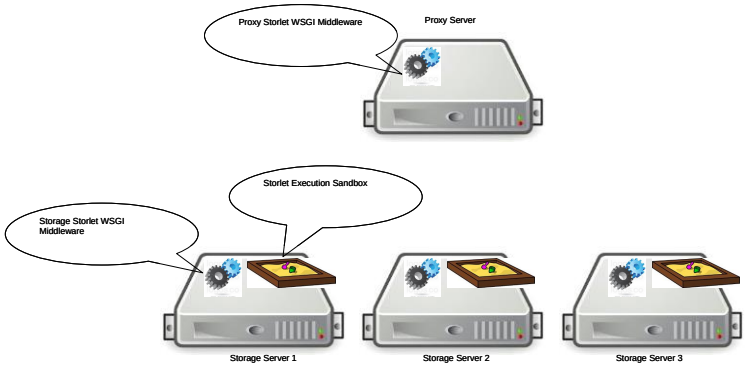


Figure 14: The storlets' high level components: WSGI middleware in the proxy and storage servers, and a sandbox in each of the storage servers. Each storlet is executed in a daemon that runs inside the sandbox

4.4.2.3. Storlets' Invocation Flow in the Get Scenario

A storlet is invoked by adding a designated header to the Swift GET request. When such a request hits the proxy server, the proxy storlet middleware validates that the issuing user has access to the required storlet. As descried in the above flow, the proxy server then routes the request to a storage server where the requested object resides.

The storage service middleware that runs on the server where the object resides opens the object's file and passes its file descriptor to a daemon that executes the storlet. Together with the object's file descriptor, the storage server storlet middleware passes a pipe file descriptor through which the storlet can send back the computation results. The computation results are then sent back to the user. Figure 15 describes the interaction between the storage server storlet middleware and the daemon running the storlet code.

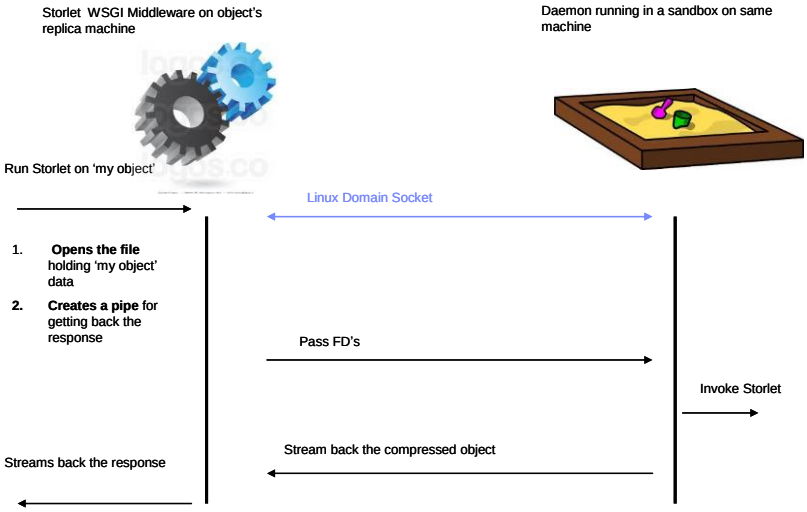


Figure 15: The interaction between the storlet middleware on the storage server and the sandbox running on the same machine. The middleware got a request for running a storlet on an object named 'my object'. The middleware is communicating with the sandbox via a Linux domain socket to pass the designated file descriptors

4.4.2.4. Storlets' Invocation Flow in the Put Scenario

A storlet is invoked by adding a designated header to the Swift PUT request. When such a request hits the proxy server, the proxy storlet middleware validates that the issuing user has access to the required storlet. The storlet middleware in the proxy server then passes the

request data to daemon that executes the storlet in the proxy server. As with the GET scenario together with the request data, the proxy server storlet middleware passes a pipe file descriptor through which the storlet can send back the computation results. The computation results are then continue with the Swift PUT flow, that creates several copies of the data.

We mention that in both GET and PUT the daemon is sandboxed in a way that it cannot access any I/O devices of the storage or proxy servers. The only communication channels it has with the outside world are the file descriptors it is given from the middleware.

4.4.3. The Sandboxing Technology

4.4.3.1. Linux Containers

For the sandboxing technology we chose to use Docker [14] which is based on Linux containers [9]. In contrast to traditional virtualization Linux containers provide an operating system level virtualization rather than hardware virtualization which makes them lightweight. Linux containers are based on two Linux kernel features:

1. Control Groups. Control groups allow controlling the resource consumption at the level of a process. For example they can be used for limiting a process to use only a subset of the machine's core, set a limit on the IO bandwidth a process can use with a certain device, and completely block the access to certain hardware devices.
2. Namespaces. Namespaces allow wrapping a global system resource so that it appears to a process as if it has its own instance of the resource. For example, a mount namespace provides a process with what looks like a root file system while effectively it sees only a portion of the host's root file system. Another important type of namespaces is the user namespace. User namespaces allow a process within the namespace to have the root user id (0) and have root privileges, while outside of the namespace it has no special privileges. Thus, a process running as root in some user namespace could send signals to other processes running in the same namespace, while it will not be able to send any signals to processes running outside of the namespace.

Docker is one of many tools that helps manage Linux containers. The merit of docker over other tools is its ability to easily build and deploy applications to be executed inside Linux containers

4.4.3.2. Other Sandboxing Technologies

Other possible sandboxing technologies include

1. ZeroVM. ZeroVM are very secure and lightweight VMs which are based on the Google NaCL project aimed at sandboxing code that is downloaded from web servers and running inside the Chrome browser. The major drawback of ZeroVM is that code that runs within it must be written in "C" and compiled using special compiler and linker.
2. Traditional virtual machines (VMware, KVM, etc.). Traditional virtual machines are notoriously slow when it comes to I/O intensive workloads. Also, VM uptime requires much more processing then a container uptime which shares the same kernel and hardware as the host.

3. Java VM isolation. Java VM isolation is good only for code written in Java. Also, the java security does not allow a fine grained control over the hardware devices processes can access as Linux containers give.

4.4.3.3. Uses of Storlets in COSMOS

In year 1 of COSMOS we demonstrated the following storlets

1. A storlet which converts geospatial locations from one coordinate reference system (UTM) to another (lat-long) and also generates metadata. The geospatial conversion was used in order to use the EMT Madrid data provided to us using UTM together with the Leaflet Javascript library which requires lat-long. This storlet also generated a geospatial bounding box as metadata for each bus trip segment. This metadata was used to demonstrate geospatial metadata search over the EMT Madrid bus trip segments collected in the COSMOS Data Store.
2. A storlet which pre-processes the office occupancy data collected by the University of Surrey and stored in the COSMOS Data Store. This pre-processing aggregated several readings into a single reading, taking the average. An occupancy detection algorithm running in Apache Spark accessed the transformed data and loaded it for further processing. This storlet significantly reduced the amount of data sent from the COSMOS Data Store (OpenStack Swift) to Apache Spark without significantly reducing the accuracy of the algorithm.
3. A storlet which performs facial blurring to images in the COSMOS Data Store. In our demonstration we used fabricated images of passengers travelling on an EMT Madrid bus. The storlet uses the OpenCV image processing library and shows an example of how libraries from other programming languages other than Java such as C++ can be used to build storlets. This storlet demonstrates the idea of privacy preserving storlets where private data does not leave the Data Store. The concept is more general than the specific example of facial blurring.

In years 2 and 3 of COSMOS we focused on integration with an analytics framework which can be combined with the use of storlets (see subsections 4.5.3 and 4.5.4).

4.5 Integrating the Data Store with an Analytics Framework

4.5.1. Introduction

An important motivation of our work is to build a data store that is optimized for IoT workloads. In order to gain value from the historical IoT data we need to be able to store and analyze it effectively. We focus here on accessing the historical IoT data for analytics purposes. We discuss several aspects in this section:

- Connecting the data store (OpenStack Swift) with an analytics framework
- Optimizing the operation of analytics on the data store
- Usage of the open Parquet format for storing data in Swift
- Data Reduction

4.5.2. Integration of OpenStack Swift with Apache Spark

Apache Spark [16] is a general purpose, fast, analytics engine that can process large amounts of data from various data sources and is recently gaining significant attention and traction. It

performs especially well for multi-pass applications which include many machine learning algorithms. Spark maintains an abstraction called Resilient Distributed Datasets (RDDs) which can be stored in memory without requiring replication and still are fault tolerant.

During the analytics stages or at task completion, Spark may persist analytics results (RDDs) back into the underlying storage. Persisting results is needed by Spark if the analytic results from one job should be used as an input to another analytics task. Effective, fault tolerant sharing of the results between analytics jobs is one of the unique features of Spark.

Spark may be configured to use a local file system, HDFS, Amazon S3, Cassandra, MongoDB, etc. Spark has the ability to use various data sources concurrently. Spark provides a single API that can be used to access data from various data sources. Figure 16 shows the Spark ecosystem.

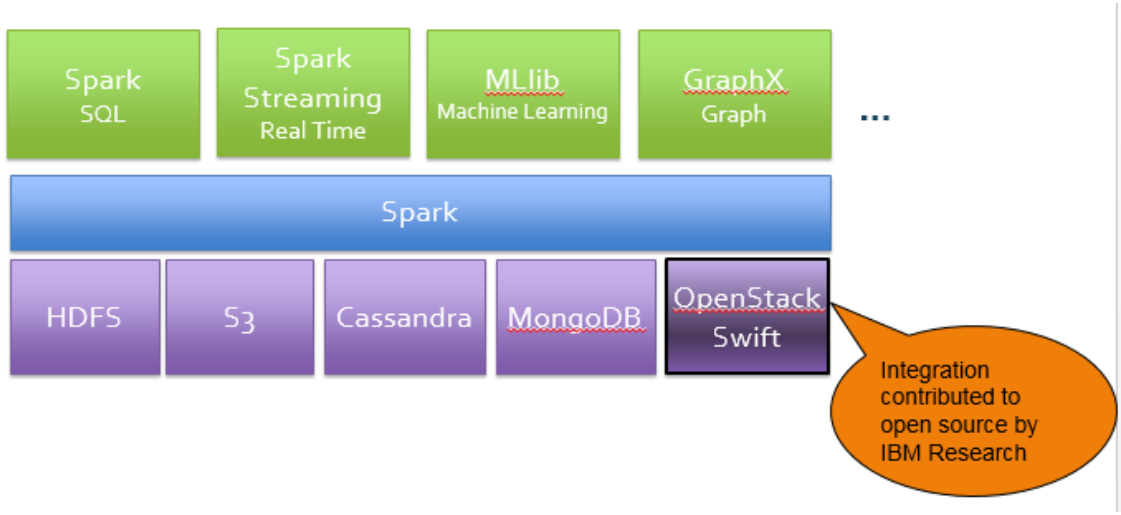


Figure 16 Spark Ecosystem

We extended Spark allowing it to analyze data stored in OpenStack Swift. This basic integration allows Spark to read data from Swift and persist final or intermediate results back into Swift. To access objects in Swift, Spark needs to use the "swift://" namespace that uniquely identifies the Swift data source. Figure 17 illustrates this. We contributed the findings to the Spark community and it's part of Spark official distributions [15].

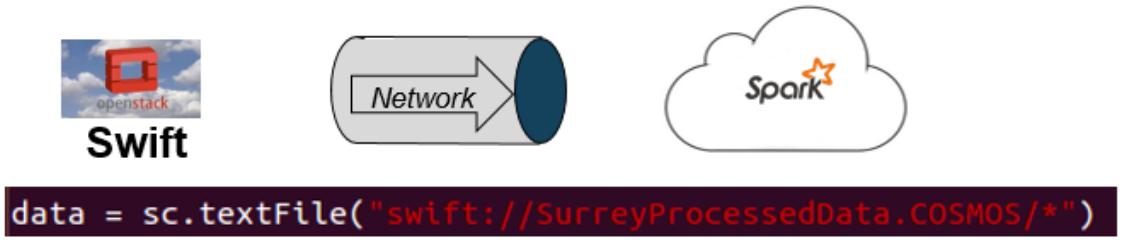


Figure 17 Accessing Swift Data from Spark

We also put effort into investigating the interaction between Swift and Spark and identified various items that may improve the interaction between Spark and Swift. One of the discovered items is related to how Spark persists its RDDs in Swift. During data save stage,

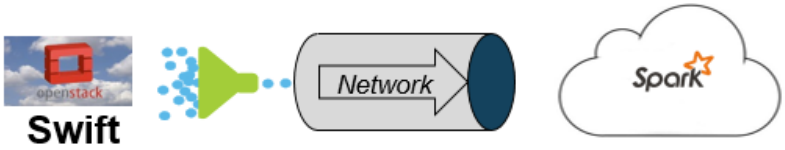
Spark generates various temporary objects and directories that later are used by Spark to build a final result. We observed that temporary file generation produces too many requests to Swift. This happens since Spark generates temporary folders in Swift as zero length objects. We improved the above behavior and greatly reduced number of requests Swift receives during the data generation phase. We plan to contribute these findings to open source.

4.5.3. Using storlets for filtering and aggregation

Storlets bring additional benefits to Spark's integration with Swift. As discussed in section 4.4 and appendix 7.4, every storlet is plain Java code that is deployed in Swift and executed when triggered on Swift's storage nodes. The Storlet Engine is the storlet execution mechanism that is responsible for executing storlets and registering them during the storlet registration phase. The ability to trigger storlet execution directly on Swift's storage nodes has great benefit for analytics use cases, in particular for Spark. We extended the basic Spark-Swift integration by allowing Spark to trigger storlets.

For example, in the Occupancy Detection use case, an aggregation storlet computes averages for multiple records, where each record contains light/power readings at a particular timestamp. For this use case, this provides sufficient accuracy and significantly reduces the amount of data sent across the network to Spark.

To trigger storlet execution from Spark one needs to proxide the x-run parameter together with the storlet name (in this case 'aggregationstorlet') immediately after the container name in the URL, as is shown in Figure 18.




```
data = sc.textFile("swift://SurreyProcessedData-x-run-aggregationstorlet.COSMOS/*")
```

Figure 18 Applying storlets on data in Swift when using Spark

4.5.4. A machine learning use case: occupancy detection

At the start of the project, we used a smart energy scenario in order to demonstrate the use of different supervised machine learning algorithms using Apache Spark. The data used is from university of Surrey IoT-test bed for the following two main reasons.

1) The controlled environment in the university makes it possible to gather labelled data with the users feedback for supervised machine learning methods and for validating the performance of statistical inference methods;

2) The nature of data is quite similar to London use case scenario and III Taipei scenario and the methods applied are generic and can easily be applied to other use case scenarios.

One of the core objective of COSMOS is to provide truly smart building capabilities by contributing towards more automated and innovative applications. In this regard, occupancy detection plays an important role in many smart building applications such as controlling heating, ventilation and air conditioning (HVAC) systems, monitoring systems and managing lighting systems. Most of the current techniques for detecting occupancy require multiple sensors fusion for attaining acceptable performance. These techniques come with an increased

cost and incur extra expenses of installation and maintenance as well. All of these methods are intended to deal with only two states; when a user is present or absent and control the system accordingly. In our work, we have proposed a non-intrusive approach to detect an occupancy state in a smart office using electricity consumption data by exploring pattern recognition techniques. The contextual nature of pattern recognition techniques enabled us to introduce a novel concept of third state as standby state which can be defined as

"A state when a user leaves his work desk temporary for a short period of time and switching off HVAC and other equipment associated with occupancy state is not optimal choice"

The intuition behind our approach is that the pattern of electricity data of user will be different when the user will be at his desk and using his appliances as compared to other states. And if our algorithm recognize the pattern, it can infer the state of user as well. We have used Spark Machine Learning Library (MLib) for the implementation of different pattern recognition algorithms. Figure 19 shows the current and power variations for random samples from the observation data.

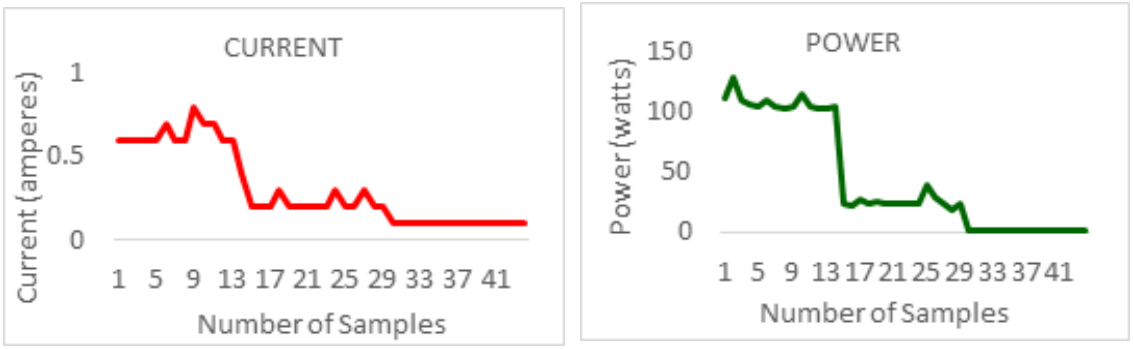


Figure 19: input Current and Power waveforms

Furthermore, our proposed solution does not require extra equipment or sensors to deploy for occupancy detection as smart energy meters are already being deployed in most of the smart buildings.

4.5.5. Projection and predicate pushdown

Spark SQL [17] is a Spark component which allows processing of (semi) structured data, and it allows relational queries expressed in SQL to be executed using Spark. A DataFrame is a structured RDD i.e. an RDD with a schema. Registering a DataFrame as a table allows SQL queries to be run against it.

Projection and predicate push down are well known techniques in relational databases. Spark uses these techniques to integrate with various data sources efficiently, for example Parquet files. Both projection and predicate push down are used when Spark queries a Parquet file, thus only relevant information is transferred from Parquet to Spark. Starting from Spark 1.2.0, Spark had an ability to access data from external data sources, allowing projection and predicate pushdown to those data sources [29]. In year 2, we implemented a similar mechanism for Swift, where Spark is able to push down projection and predicate evaluation into Swift. This is done using metadata search (see section 4.3.2) for predicate pushdown, and using Parquet format (see section 4.5.8.2) for projection pushdown. More details on our

design and implementation can be found in the section on Integration with an Analytics Framework (section 7) of deliverable 4.2.2.

This capability allowed the machine learning required by our use cases to efficiently access the data in object storage and analyze it. For example, the Madrid Traffic use case requires accessing all traffic history for each sensor according to four time periods : morning, afternoon, evening and night time. For example to get all data for morning traffic, defined at between 8am and 12pm, the Spark SQL query is:

```
SELECT codigo, intensidad, velocidad FROM madridtraffic
WHERE tf >= '08:00:00' AND tf <= '12:00:00'
```

Since we annotated objects with their minimum and maximum timestamps using the Data Mapper, and indexed them using metadata search, our Spark SQL external data source driver is able to use these components to restrict the SQL query to only those objects overlapping the time interval quested by the query. After collecting roughly one year's worth of Madrid traffic data, our Spark SQL driver gave an improvement of 21.5 times compared to the naïve method of accessing all the data without using metadata search. The naïve method resulted in 13,425 Swift requests whereas using our Spark SQL driver resulted in 616 Swift requests.

Similar queries were required by the III use case which uses anomaly detection to alert for abnormal appliance behavior.

4.5.6. A machine learning use case: differentiating between good and bad traffic

Our proposed architecture is optimized for big data applications and provide a generic solution which can be applied to a set of different scenarios. We demonstrated our solution using Madrid use-case in order to detect the traffic state automatically in an optimized manner. The city of Madrid has deployed hundreds of traffic sensors on different locations across the city. These sensors provide real-time information about the traffic in the city using traffic parameters such as average traffic speed, average traffic intensity, type of traffic or type of road etc.

As mentioned earlier, CEP has the potential to provide an optimized solution for analysing, correlating and inferring high-level knowledge from this large amount of data in near real-time. The core of CEP is a rule-based engine which requires rules for extracting complex patterns. These rules are based on different threshold values. For example, traffic speed can be analysed using a simple rule as "if current speed is less than a threshold speed; generate slow speed event". The calibration of rules e.g. by setting a threshold, requires the system administrator to have prior knowledge about the system which is not always available and prone to error. We collected and exploited a large historical dataset and proposed a novel method based on a machine learning approach to find the optimal parameters for CEP rules automatically.

In our method, we used a clustering method called k-means clustering from the Spark MLlib library. Clustering refers to an unsupervised machine learning method which is used for grouping similar objects on the basis of a predefined metric such as distance or density. Clustering is a general technique used widely for knowledge discovery in big data sets. Clustering separates the data into different possible events and groups the data points from the same event together. Applying clustering on the Madrid traffic data groups the traffic parameters from the good traffic together and bad traffic parameters in a separate cluster.

The midpoint between both cluster centres provides the boundary separating the clusters and serves a threshold for CEP rules. More details can be found in D6.1.3.

We used different models for different time periods such as morning, afternoon and evening. Spark SQL enables us to extract the relevant historical data in an optimized manner using Spark SQL queries as described in 4.5.5.

4.5.7. A machine learning use case: anomaly detection

As described earlier, our proposed architecture is generic and can be applied to a range of applications. We used our architecture to detect anomalies automatically and demonstrated it on two use-case scenarios which are;

1) Detecting anomalies from electricity consumption data (III Taipei)
2) Extracting unusual events from twitter data feed

The intuition of our approach is to exploit historical data and learn normal patterns from historical data using statistical methods (3 standard deviation) and machine learning methods (k-means clustering). The range of normal patterns acts as threshold values for CEP which can analyze and detect the anomalies in near real-time. A small description for both scenarios is given below.

4.5.7.1. III Taipei Scenario

Taipei Scenario consists of fifty (50) volunteer households where III's In-Synergy suite (incl. home gateway, smart plugs, and smart strips) enables users to monitor energy usage and control/set up/use appliances in a smarter way. Different appliances are connected to smart gateways with the help of smart plugs. Smart plugs monitor real-time electricity consumption data which is provided as a web service from III. COSMOS accesses this data in near real-time using credentials provided by III and published to a specific topic in Apache Kafka. Real-time data is being monitored with the help of CEP in COSMOS to detect anomalies. Historical data is analysed in Apache Spark using different statistical and ML methods in order to calculate threshold values for µCEP rules.

4.5.7.2. Twitter Events

In year 3, we also analyzed social media data using the Node-Red API to access real time twitter feeds. We defined a geographical area and extract only tweets from that area. In our case, we extracted all tweets from Madrid City surrounding the region where traffic sensors are deployed and store the total number of tweets per unit time in the COSMOS data store. The analytics on the gathered historical data enables us to learn the normal pattern for the total number of tweets and using CEP rules, we can trigger an event if the pattern deviates from the normal pattern.

4.5.8. Data Format

4.5.8.1. Schemas

We described our general assumptions for data mapping and representation in section 4.3.1. In order for the COSMOS IoT data to be amenable to analysis using Spark SQL, it needs to be described by a schema. We propose optionally associating each persistent Message Bus topic with a schema. This schema can be stored in a dedicated container for schemas in Swift and a reference to the schema location can be stored in the COSMOS Registry. If a schema is not available then Spark SQL needs to infer the schema automatically. This can be done in Spark SQL for JSON data but it requires a full scan of the data and is therefore computationally expensive. Therefore the availability of a schema can significantly improve performance.

4.5.8.2. The Parquet Data Format

Apache Parquet [12] is an open source file format designed for Hadoop that provides columnar storage. This is a well known technique for optimizing analytical workloads in recent relational database systems. Using this techniques, data for each column of a table is physically stored together, instead of the classical technique where data is physically organized by rows.

Columnar storage has two main advantages for IoT workloads. Firstly, organizing the data by column allows for better compression. Note that using this technique each column can be compressed independently using a different encoding scheme. For IoT workloads, many columns will typically contain readings which do not change rapidly over time for example temperature readings. For this kind of data some kind of delta encoding scheme could significantly save space. Secondly, organizing the data according to columns allows for projection pushdown all the way down to the physical I/O level. This means that if certain columns are not requested by a query then they do not need to be accessed on disk at all. This is unlike the classical case where data is organized by rows and all columns are accessed together. This can significantly reduce the amount of I/O in addition to reducing the amount of network bandwidth required (the latter can be achieved using without columnar storage).

4.5.9. Data Reduction

In the initial project period we identified the requirements and opportunities in the realm of data reduction. The premise for data reduction in COSMOS is that the IoT setting accumulates large amounts of data. While this data may be small individually, the sheer time and scope of the collection would eventually result in the need to collect and analyze very large amounts of data. There are multiple opportunities to reduce the amounts of raw data in each and every step of the IoT process, starting from the IoT collection devices through the network and aggregation mechanisms. In this work we chose to focus on reducing data at the storage and analytics side where the data is accumulated. This has two beneficial effects: a) as data mounts up, the potential for compressing it grows due to similarity between objects, and the ability to invest more resources towards this end. b) At the Data Store, data can be viewed according to its relative importance, and in the long run over time can be diluted or compressed in a lossy fashion according to retention policies and results of analysis executed on it.

Regarding lossless data reduction, starting from year 2 we stored COSMOS data in Parquet data format, which is column based and supports column wise compression. This means that compression schemes can be specified on a per-column level. Parquet supports multiple encodings including “plain”, dictionary encoding, run length encoding, delta encoding, delta

length byte array and delta strings. In addition Parquet allows new compression schemes to be added [13]. The Parquet data format is supported by Spark SQL so analytics can be applied to data compressed when using this format.

Regarding lossy data reduction, we think that policies for down sampling and eventual deletion of time series data over time in order to save storage space are important for IoT use cases in general. However, so far this has not been a priority for COSMOS use cases since the amount of data stored has been manageable in terms of storage space (see section 4.1.1).

4.6 Message Bus

A Message Bus has been selected for interconnection of distributed COSMOS components as well as external clients – Virtual Entities. This was shown in Figure 1, where the Message Bus allows data to flow from virtual entities to other COSMOS components such as CEP and the Data Store (via the Data Mapper). We have identified following main design drivers for proper selection of technology for COSMOS communication platform bus:

- Decoupled communication model. Data producers and consumers should not depend on each other. They still have to depend on structural and sematic aspect of exchanged information which is necessary for interoperability.
- Secure and Fast/Scalable and Reliable information exchange.
- Simple and convenient data exchange solution. The effort to connect, send and receive data should be minimized.
- Support for management features such as orchestration, intelligent routing and provisioning.

The COSMOS architecture requires a Message Bus although messaging is not one of the topics planned to be researched in COSMOS directly. Therefore the COSMOS partners agreed to adopt an open source framework for messaging in COSMOS 'as is' without making enhancements or research contributions in this area.

In relation to the message payloads that will be transmitted through the Message Bus, a JSON data format has been selected since it is widely adopted and provides required flexibility and portability. In the original version of this document, we also proposed compressing messages using the open source MessagePack [4] data serialization library although we decided not to implement this since it has not proven essential for the COSMOS use cases.

In year 1 we used RabbitMQ as the framework underlying the Message Bus, and in year 2 we evaluated the use of Apache Kafka. The following sections describe these technologies.

More details about the use of the Message Bus in COSMOS can be found in Deliverable 2.3.2.

4.6.1. RabbitMQ

A RabbitMQ [2] messaging solution was selected as the COSMOS Message Bus during the first year of the project. It offers reliable messaging, flexible routing, high availability and support for wide range of communication protocols and programming language bindings. It also decouples publishers and consumers and has built-in support for offline applications through late delivery feature.

The RabbitMQ implementation follows a message broker architecture built on top of the AMQP [24] communication protocol. This connects clients through common platform for sending and receiving messages. In RabbitMQ all messages are transferred asynchronously, therefore they have to be serializable and immutable.

4.6.1.1. Message Exchange Strategies

RabbitMQ provides three different routing algorithms, each of them serving different types of message exchange provided by the protocol. Exchanges control the routing of messages to subscribers.

- **Direct exchange** – the direct exchange is the simplest one. Messages are identified with a routing key. If the routing key matches, then the message is delivered to the corresponding subscribers.

- **Fanout exchange –** this exchange will multicast the received message to all subscribed listeners. A Fanout exchange is useful for facilitating the publish-subscribe communication pattern.

- **Topic exchange –** works similarly to direct exchange, but it allows subscribers to match on portions of a routing key. A topic exchange is useful for directing messages based on multiple categories or for routing messages originating from multiple sources.

4.6.2. Apache Kafka

Apache Kafka [25] is an open source message broker originally developed by LinkedIn. Kafka is designed to allow a single cluster to serve as the central messaging backbone for a large organization. Kafka emphasizes high throughput messaging, scalability, and durability. Although Kafka is less mature than Rabbit MQ, it supports an order of magnitude higher throughput messaging [3]. Moreover, Kafka supports both batch consumers that may be offline, or online consumers that require low latency. Importantly Kafka can handle large backlogs of messages to handle periodic ingestion from systems such as Secor, and allows consumers to re-read messages if necessary. This scenario is important for COSMOS.

For these reasons, we evaluated the use of Kafka for the COSMOS Message Bus in order to enable a high throughput ingestion path of data from VEs via the Message Bus through the Data Mapper (based on Secor) to object storage. This flow appears in the architecture diagram of Figure 1. We found Kafka to be suitable for our use cases, and it integrates well with our most of our components. During our evaluation, version 0.9 of Kafka was released, which supports security features. Moreover, a new framework called Kafka Connect was released which is interesting for future work. Since this evaluation was successful, Kafka was adopted as the framework underlying the COSMOS Message Bus.

5 Results and Conclusions

Data management for IoT is a very important area since the amount of data which will be generated by IoT devices is massive and continually increasing. We address certain central aspects of data management in COSMOS. We define an overall data management architecture, which includes data flow in the system using the Message Bus and its ingestion into persistent storage using the Data Mapper component. We also address how analytics can be done both in real time using Complex Event Processing, and on accumulated data, by supporting metadata search and storlets in the Data Store, and by integrating with the Apache Spark analytics framework. These are critical pieces of a data management platform for IoT applications.

This document describes requirements, architecture and component design for the Information and Data Lifecycle Management Work Package. This is the final plan for our work in COSMOS. Many aspects of this work have been implemented. The final version of the deployment and implementation details will be described in deliverable 4.2.3.

The framework described in this document has been successfully used to implement COSMOS use cases such as the Madrid Traffic and III Taiwan scenarios. The Madrid Traffic scenario has been ported to the IBM Bluemix platform and demonstrated there. The COSMOS architecture and the Madrid Traffic use case have been demonstrated in many forums including the Apache Spark Summit in Europe, the IBM Insights conference and the IBM Interconnect conference. We have also shown that our architecture is applicable to a wide variety of IoT use cases. We will further develop the Madrid Traffic and III Taiwan use cases in the remainder of the project, in order to deal with richer data sources, multiple data sources and more complex CEP rules, although we expect the same basic architecture described in this document will remain relevant and applicable.

6 References

[1] OpenStack Object Storage API v1 Reference http://docs.openstack.org/api/openstack-object-storage/1.0/content/index.html

[2] RabbitMQ https://www.rabbitmq.com/

[3] Kreps, Narkhede, Rao, "Kafka: a Distributed Messaging System for Log Processing", NetDB'11, Athens, Greece

[4] Message Pack http://msgpack.org/

[5] Protocol Buffers https://code.google.com/p/protobuf/

[6] Cap'n Proto http://kentonv.github.io/capnproto/

[7] ZeroVM http://zerovm.org/

[8] WSGI http://wsgi.readthedocs.org/en/latest/

[9] Linux containers https://linuxcontainers.org/

[10] Secor https://github.com/pinterest/secor

[11] Amazon S3 http://aws.amazon.com/s3/

[12] Apache Parquet https://parquet.incubator.apache.org/

[13] Apache Parquet Encoding Schemes https://github.com/Parquet/parquet-format/blob/master/Encodings.md

[14] Docker https://www.docker.com/

[15] Accessing OpenStack Swift from Spark https://spark.apache.org/docs/latest/storage-openstack-swift.html

[16] Apache Spark https://spark.apache.org/

[17] Spark SQL Programming Guide https://spark.apache.org/docs/1.3.0/sql-programming-guide.html

[18] ATOS SOL/CEP engine in FIWARE https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Atos_SOL

[19] Raspberry Pi board https://www.raspberrypi.org/

[20] UDOO board http://www.udoo.org/

[21] DOLCE Language Specification, FIWARE, Nov. 2012 https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Atos_SOL#Dolce_Language

[22] User Datagram Protocol IETF RFC 768 https://tools.ietf.org/html/rfc768

[23] Datagram Transport Layer Security IETF RFC 4347 https://tools.ietf.org/html/rfc4347

[24] Advanced Message Queueing Protocol v1.0 http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-complete-v1.0-os.pdf

[25] Apache Kafka http://kafka.apache.org/documentation.html

[26] Node-RED – A visual tool for wiring the IoT http://nodered.org/

[27] Proton Programming Language http://mirrors.apple2.org.za/apple.cabi.net/Languages.Programming/PROTON.COMPLETE/PROTLANG.TXT

[28] Apache Avro https://avro.apache.org/

[29] External Data Sources in Spark 1.2.0 http://www.river-of-bytes.com/2014/12/external-data-sources-in-spark-120.html

7 Appendix

7.1 Data Mapper API

7.1.1. JSON format

As it is mentioned in the section 4.1.3, the component expects to receive JSON messages which should contain, at least, the Id field and those related to the timestamps. These fields are stored as metadata of the data objects in the cloud storage. The VE developer is responsible for filling in a configuration file, containing all this information (please see section 7.1.2). For instance, a message that was produced in Y1 is the following:

```
{
   "estate":"Dalehead",
   "hid":"cPGKKhiI4q6Y",
   "ts":"1405578854",
   "instant":"226",
   "returnTemp":"72.3",
   "flowTemp":"72.4",
   "flowRate":"742",
   "cumulative":"15703"
}
```

The mapping between the fields above and the Data Mapper's configuration is explained in the next section.

7.1.2. Configuration

The main parameters that are associated with the component's configuration are the **period** (measured in minutes) in which it reads the messages from the message bus and the lowest **size** (measured in kilobytes) of the data object that is going to be stored in the cloud storage.

The configuration file used in Y1, regarding Camden use case, is the following:

```
{
   "period":"1",
   "size":"1",
   "mandatory_metadata":{
      "Timestamp":"ts",
      "Id":"hid"
   },
   "optional_metadata":{
    "Estate":"estate"
   }
}
```

The mandatory metadata mean that the Data Mapper does not store any data that do not contain this kind of information, whereas for the optional ones, we give the VE Developer the capability to annotate the data with enriching metadata. Metadata checks (for the mandatory parts) are applied in all cases of testing.

7.2 µCEP REST Admin API

The µCEP solution will provide REST API built on pragmatic RESTful design principles. The API will use resource-oriented URLs that leverage built in features of HTTP like authentication, verbs and response codes.

For compatibility with other COSMOS components, all request and response bodies will be JSON encoded, including error responses. Any off-the-shelf HTTP client should be able to communicate with the API.

7.2.1. Uniform Resource Identifier

The base URL for API is https://{serverRoot}/ucep/v1.

The serverRoot is the address of the machine hosting the µCEP instance. The address as well as listening port can be configured outside of REST API.

7.2.2. Authentication

The API will be authenticated using HTTP Basic Access Authentication method over HTTPS. Any request over plain HTTP will fail.

7.2.3. HTTP Verbs

HTTP Verb Name	*Description*
GET	To retrieve a resource or a collection of resources.
POST	To create a new resource.
PUT	To set an existing resource. A whole representation of resource is required.
DELETE	To delete an existing resource.

7.2.4. JSON Bodies

All CRUD HTTP requests must be JSON encoded and must have a content type of application/json, otherwise the API will fail with a return of 415 “Unsupported Media Type” status code.

The response will be JSON encoded as well. The response will always return updated representation for creation and update of operations.

7.2.5. Supported HTTP Status Codes

HTTP Status Code	*Description*
200 OK	Request succeeded.
201 Created	Resource created. URL to new resource in header.
400 Bad Request	Error in the request.
401 Unauthorized	Authentication failed.
403 Forbidden	Client does not have access.
404 Not Found	Resource could not be found.
415 Unsupported Media Type	Missing application/json content type.
500 Internal Server Error	An internal error occurred.

7.2.6. Result Filtering

All responses from the API can limit results to only those that are actually needed.

For example `GET /ucep/v1/rules?name=TrafficJam`

7.2.7. Events

Events define messages that should be collected by CEP. Events can be listed using API:

`GET /ucep/v1/events`

A single existing event can be obtained usin API:

`GET /ucep/v1/events/{EventName}`

A structure of message will follow DOLCE domain language format:

```
{
  “name”: “RoomTemperature”,
  “use”: {
    “int”: “SensorId”,
    “float”: “Value”,
    “time”: “TimeStamp”,
  },
  “accept”:{
    “SensorId”: “42”
  },
}
```

7.2.8. Rules

Rules are accessible via API:

`GET /ucep/v1/rules`

A single existing event can be obtained usin API:

`GET /ucep/v1/rules/{RuleName}`

A structure of the rule will follow DOLCE domain language format:

```
{
  “name”: “SmogAlert”,
  “payload”: {
    “level”: “avg(SmogLevel)”,
    “position”: “SensorLocation”,
  },
  “detect”:{
    “name”: “TrafficSensorReport”,
    “where”: “sum(NumberOfCars) > 1000”,
    “in”: “60 minutes”
  },
}
```

7.3 Cloud Storage and Metadata search API

The OpenStack Swift REST API [1] can be used for CRUD operations on containers and objects, and also supports annotating containers and objects with metadata. However Swift does not support metadata search. We describe our extension of the Swift API to support metadata search here.

Metadata search is performed by a GET request having an X-Context header with the value of search, which specifies that this is a metadata search request. The syntax of a metadata search is specified below

GET endpoint/<Object Storage API version> [/<account>[/<container>[/<object>]]] ?

[&query=[(]<query expr1>[%20AND%20<query expr2>][)][%20AND%20…]]

[&format=json|xml|plain]

[&type=container|object]

[&sort=<query attr> asc|desc [,<query attr> asc|desc]*]

[&start=<int>]

[&limit=<int>]

[&recursive= True | False]

This is a regular Swift GET request [1], with some additional parameters described below.

The query parameter describes the metadata being searched for, using a conjunction of query expressions, where

- <query expr> = <query attr><operator><query value>
- <query attr> = A custom metadata attribute to be compared against the <query value> as a query criterion.
    - Note that we use a naming convention for <query attr> (metadata attribute names) which allows their data types to be derived (see the Data Types section below). This data type should usually correspond to the data type of the <query value>.
- <operator> = The query operation to perform against the <query attr> and <query value>, one of:
    - ***= (equals exactly)***
    - ***!= (does not equal)***
    - ***< (less than)***
    - ***<= (less than or equal to)***
    - ***> (greater than)***
    - ***>= (greater than or equal to)***
    - ***in (within a geo bounding box)***

- ***~ (free text matching)***

    - Range operators (<, <=, >, >=) can be applied to attributes with types: string, free text, date, timecode, integer and float.
    - The in operator should be applied to an attribute with geo-point type.
    - The ~ operator should be applied to an attribute with free text type. The <query value> will go through an analysis process defined by Elastic Search which involves tokenization and normalization. For example, see http://www.elasticsearch.org/guide/en/elasticsearch/guide/current/analysis-intro.html
    - Note regarding the != operator:
        - <query attr> != <query value> also returns true in cases where <query attr> is not defined on the container/object

- <query value> = The value to compare against the <query attr> using the <operator>. The value is either a numeric integer or decimal value without quotes (e.g. -127, 3.14159), or a date or a string enclosed in single or double quotes.
    - For geo bounding box queries (when the operator "in" is used) the value should be a geo bounding box. Geo Bounding Boxes use the format [TopLeft], [BottomRight] where TopLeft and BottomRight are geo-points. See the section on data types below for the definition of geo-points. For example [50,-3.58],[0,20.5] is a Geo Bounding Box. Note that Geo Bounding Boxes are used in searches only (not for attribute values).

format

- Used to specify the format of the query results. Default is plain

type

- Used to restrict results to objects or containers only. Default: no restriction.

sort

- Used to sort the query results according to one or more metadata attributes. Default is no specified sort order.

  Note regarding sort
    - Objects which don't have they query attribute requested will always sort last, irrespective of whether asc or desc is chosen

start

- Used to request returning search results starting from a certain result number. The default is 0.

limit

- Used to limit the number of search results. The default is 100

recursive

- Defines whether or not the search will be limited to the given URI exactly (recursive=False) or whether the URI specifies a prefix of what is being searched for (recursive=True). Default: True
    - For example, if recursive = True, searching with URI: /v1/my_account/cars will also match the container my_account/cars2 and the objects in it. Also the container my_account/cars itself will match.
    - If recursive = False, searching with URI: /v1/my_account/cars will not match container my_account/cars2 or the objects in that container. Also the container my_account/cars itself will not match, only the objects in that container will match.

Data Types

Correct mapping of metadata attributes to data types is essential for metadata search to work correctly. User metadata attributes are mapped to data types using a naming convention as follows:

- *-d are dates
- *-c are timecodes
- *-i are integers
- *-f are floats
- *-g are geo-points
- *-t is free text
- *-b is boolean (future)
- everything else is a string (exact search)

Metadata attribute values with incorrect format (according to their data type mapping) may not be indexed and therefore may not appear in search results.

Data Type Formats

- **Strings**
    - Strings (with exact search) are the default data type and are assumed when no naming convention is used.
    - Strings should be enclosed in single or double quotes.
    - Certain characters need to be URL encoded. These characters are & # % + ; `
    - Double quotes as part of the query string need to be escaped using \.
- **Dates**
    - Dates should be enclosed in single or double quotes.
    - Dates (with optional time values) should use the ElasticSearch default date/time format. This format is documented here as dateOptionalTimeParser http://joda-time.sourceforge.net/api-release/org/joda/time/format/ISODateTimeFormat.html#dateOptionalTimeParser()
    - For example 2014-07-21T15:42:00.00
- **Timecodes**
    - Timecodes should be enclosed in single or double quotes.
    - Timecodes use the format **HH:mm:ss.SSS** - a two digit hour of day, two digit minute of hour, two digit second of minute, and three digits (000-999) to represent the frame. Note that each of these digits is manadatory. Therefore 19:53:26.018 and 09:53:26.018 are valid timecodes, whereas 9:53:26.018, 09:53:26 and 19:53:26.18 are **not** valid timecodes. Also HH needs to represent a valid hour, similarly for mm (minute) and ss (second). Therefore 30:53:26.018 and 19:63:26.018 are **not** valid timecodes.
- **Integers**
    - A positive or negative integer or zero e.g. 352, -990, 0
- **Floats**
    - A numeric decimal value e.g. 456.9759, -0.4234. Note we do not currently support exponential notation.
- **Geo-points**

- Geo-points use the format: latitude, longitude (where each of these is a numeric decimal value). For example 50,-3.58

- **Free Text**

  - Free text attribute values have the same format as strings but are indexed and searched differently. Free text attributes should be searched using the ~ operator. Before being indexed, free text attribute values pass through an analysis process defined by Elastic Search which includes tokenization and normalization.

7.4 Storlets API

There are three APIs of interest in the context of storlets: The API one needs to implement when writing a storlet, the API for deploying a storlet and the API of invoking a storlet.

7.4.1. Storlets API

Currently, storlets can be written in Java. In order to write a storlet one needs to implement the com.ibm.storlet.common.IStorlet API given below:

```
public void invoke(StorletInputStream[] inStreams,
                   StorletOutputStream[] outStreams,
                   Map<String,String> parameters,
                   StorletLogger logger)
```

When invoked via the Swift **GET** REST API, the invoke method will be called as follows:

1. The first element in the inStreams array would include an instance of StorleInputStream representing the object appearing in the request's URI.
2. The outStreams would include a single element of type StorleObjectOutputStream representing the response returned to the user. Anything written to the output stream is effectively written to the response body returned to the user's GET request.
3. The parameters map includes execution parameters sent. These parameters can be specified in the storlet execution request as described in the execution section below.
4. A StorletLogger instance.

When invoked via the Swift **PUT** REST API, the invoke method will be called as follows:

1. The inStreams array would include a single element of type StorleInputStream representing the object to read.
2. The outStreams would include a single element which is an instance of StorletObjectOutputStream. Metadata and data written to using this object will make it to the store, under the name provided in the original request URI.
3. The parameters, and StorletLogger as in the GET call.

7.4.2. Deploying a Storlet

Any interesting storlet would require dependencies on libraries that might not exist on the Linux container. Thus, a storlet writer can declare that a certain storlet is dependant in external libraries, and deploy them as part of deploying a storlet. Storlet deployment is essentially uploading the storlet and its dependencies to designated containers in the Swift account we are working with. While a storlet and a dependency are regular Swift objects, they must carry some metadata used by the storlet engine. When a storlet is first executed, the engine fetches the necessary objects from Swift and 'installs' them in the Linux container.

7.4.2.1. Storlet Deployment

Storlets are deployed using Swift's PUT object API to a designated container. This container is where the storlet middleware will look for storlets code. Any PUT to the storlet container must carry the following headers:

1. X-Object-Meta-Storlet-Language - currently must be **'java'**
2. X-Object-Meta-Storlet-Interface-Version - currenltly we have a single version **'1.0'**
3. X-Object-Meta-Storlet-Dependency - A comma separated list of dependent jars.
4. X-Object-Meta-Storlet-Object-Metadata – This is an optimization flag, that indicates whether the storlet requires the objet's metadata for its execution. Only if the value is yes, would the storage middleware parse the object's metadata and pass it to the storlet. This option is currently not operational.
5. X-Object-Meta-Storlet-Main - The name of the class that implements the IStorlet API.

7.4.2.2. Dependency Deployment

Dependencies are deployed using Swift's PUT object API to a designated container. This container is where the storlet middleware will look for the declared dependencies. Any PUT to the dependency container must carry the following headers:

1. X-Object-Meta-Storlet-Dependency-Version - While the engine currently does not parse this header, it must appear.
2. X-Object-Meta-Storlet-Dependency-Permissions - An optional metadata field, where the user can state the permissions given to the dependency when it is copied to the Linux container. This is helpful for binary dependencies invoked by the storlet. For a binary dependency once can specify: '0755'. This allows the storlet code to execute the dependency.

7.4.3. Storlet Invocation API

In both GET and PUT scenarios, a storlet in invoked using a regular Swift GET/PUT operations complemented with the following headers:

- X-run-storlet : <storlet name>
- X-generate-Log: <True / False>

In addition one can pass argumets to the storlet invocation using query string parameters, as follows: Suppose that we want to run the storlet **MyStorlet.jar** over an object name called **MyObject** that resides in a Swift account called **MyAccount**, inside a container called

MyContyainer. Furthermore, suppose that we want to pass two parameters: **param1** having value **val1** and **param2** having value **val2**. Here is how the command looks like:

```
GET
http://<SwiftProxyhostName>/MyAccount/MyContainer/MyObject?param1=val1
&param2=val2

X-run-storlet: MyStorlet.jar

X-generate-log: True
```

Suppose that we want to run the storlet **MyStorlet.jar** during the upload of an object that will be called **MyObject** and will reside in a Swift account called **MyAccount**, inside a container called **MyContyainer**. Furthermore, suppose that we want to pass two parameters: **param1** having value **val1** and **param2** having value **val2**. This time, however, we do not want to generate logs. Here is how the command looks like

```
PUT
http://<SwiftProxyhostName>/MyAccount/MyContainer/MyObject?param1=val1
&param2=val2

X-run-storlet: MyStorlet.jar
```