COSMOS

Cultivate resilient smart Objects for Sustainable city applicatiOnS

Grant Agreement Nº 609043

D4.1.1 Information and Data Lifecycle Management: Design and open specification (Initial)

WP4 Information and Data Lifecycle Management

Version: 1
Due Date: M8
Delivery Date: 2/5/2014
Nature: R
Dissemination Level: PU

Lead partner: IBM
Authors: Danny Harnik (IBM), Jozef Krempasky (ATOS), Achilleas Marinakis (NTUA), Eran Rom (IBM), Paula Ta-Shma (IBM)
Internal reviewers: Spyros Gogouvitis (NTUA), 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	22/4/2014	Paula Ta-Shma and co-authors	IBM	First version for internal review.
0.2	1/5/2014	Paula Ta-Shma and co-authors	IBM	Version for submission. Took internal review comments into account.

Annexes:

Nº	File Name	Title

Table of Contents

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 data need to be ingested and analyzed in real time. Secondly, data needs to be stored persistently in order to enable historical data analysis. Thirdly, data may need to be archived over time.

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 analyzed 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 persistently in order to support search and analysis on historical data.

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 information is costly. Therefore data reduction and archiving techniques are needed in order to reduce the cost of storing the data.

2 Requirements

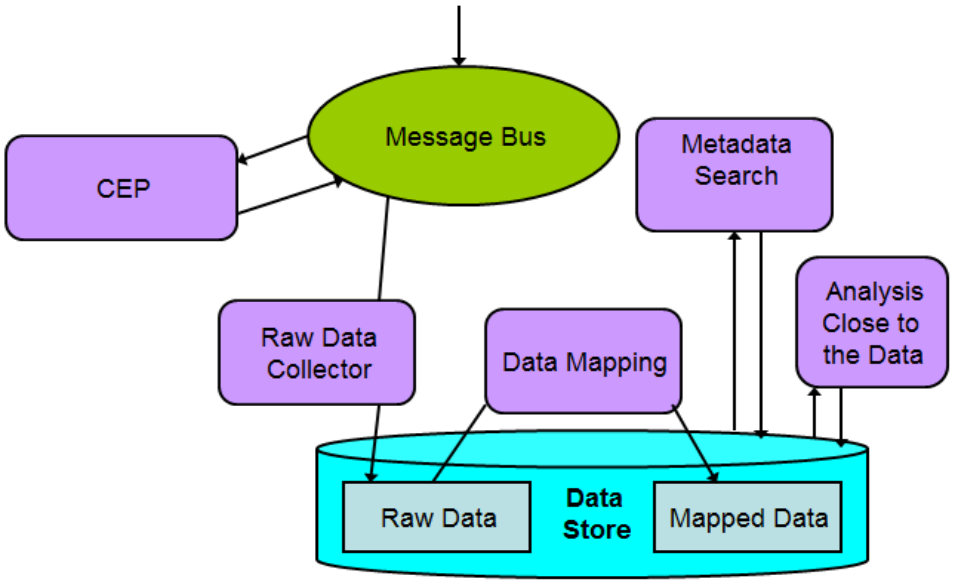
For convenience, we list the requirements relevant to this work package here. The reader is referred to Annex 1 of Deliverable 2.2.1, which contains a list of requirements for all work packages in COSMOS. These requirements are addressed by the various components of the WP4 architecture, discussed in 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	Raw stream data processing (predict anomalies or off-normal events) should be possible
4.8	CEP	System should offer CEP data persistence (post processing to detect behavior patterns).
4.9	CEP	Publishing sub-system offer data broadcasting based on semantic analysis results
4.10	CEP	System should provide meanings to define events taxonomy, including reasoning with unsafe/uncomplete events
4.11	CEP	System should provide the capability to define processing configurations/topologies, including fail safe configurations
4.12	CEP	CEP capability should provide support to be used as a situation awareness tool.
5.0	Data Set	The system must provide mechanisms in order to characterise 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. The architecture of this work package is summarized in the diagram below.



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.

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 Store persistently stores COSMOS data in a reliable and scalable fashion. The Data Store component is described in section 4.3.

The Raw Data Collector 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 Raw Data Collector will periodically dump the data to storage without analyzing it or changing its format. It will collect useful statistics about the data collected such as the data flow rate and data sources and topics.

The Data Mapping component will periodically process the Raw Data, extract metadata and transform it to a data layout suitable for subsequent metadata search and analysis. For example, the Data Mapping component may combine data from several different topics to form a merged data item. In Year 1, we plan to perform a very simple data mapping, therefore we combine the Raw Data Collector and the Data Mapping components into a single component which will persistently store data from the Message Bus into the Data Store. These

components are described in section 4.1. In future years we envision that this will be done by two separate components.

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 4.3.1 and 4.3.2 and it has a REST API which is described in Appendix 7.3.

The Analysis Close to the Data component supports analysis of the persistent data. 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 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.

4 Component Descriptions

4.1 Raw Data Collector and Data Mapping (combined)

4.1.1. Functional Overview

Raw Data Collector and Data Mapping 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 going to be examined in year 2 and/or 3, but the general idea is to have multiple components in order to be able to handle huge amounts of data and to ensure that no message will be lost.

The main functionalities provided by Raw Data Collector and Data Mapping Component are:

- Create objects with size relevant to cloud storage
- Calculate some simple statistics
- Extract metadata both form raw data and from the Virtual Entity semantic registry described in D5.1.1

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.

4.1.2. Design Decisions and Details

For Year 1 the component has a static configuration, whose details are shown below:

- Data objects are stored in the cloud through Openstack/Swift component;
- Two accounts are created, one for each use case;
- Each account can be accessed by many users. Raw Data Collector and Data Mapping Component is one of these users;
- Containers correspond to the topics, stored in the message bus;
- 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: integer);
    - Timestamps (data type: date Time);
- Metadata extracted from the Virtual Entity semantic registry can be updated;
- Metadata data types can be string, integer and date Time;

4.1.3. Use Cases for Raw Data collector and Data Mapping

The component's functionalities are explained using examples arising from the Use Cases:

Madrid Use Case

In Madrid Use Case a bus can be considered as a Virtual Entity and each bus is supplied with a GPS module, so a bus exposes the IoT service called "provide the location of the bus constantly". A bus subscribes to the topic *transportation* in the message data bus and publishes its data (time series of its location) in a JSON format. Raw data collector and Data Mapping component also subscribes to the same topic and reads periodically the messages coming from all the Virtual Entities (buses). Firstly, the component separates the messages by their origin (Virtual Entity) and calculates some stats like: the bus #1 has published 3% of the whole amount of messages. Then, it aggregates the data in a single object, until the latter has a size suitable for the cloud storage. Finally, the component communicates with the Virtual Entity Registry component so as to retrieve the social metadata with which the object is annotated before being stored in the cloud storage. The object description is shown below:

Account: Madrid Use Case

User: Raw Data Collector and Data Mapping component

Container: Transportation

Object Name: location of the bus #1 on 10/4/2014

Metadata:

1. Id = 1
2. Start time: 2014-04-10T06:00:00
3. End time: 2014-04-10T23:00:00
4. Trust & Reputation index: +1

London Use Case

In London Use Case a building can be considered as a Virtual Entity and each building has some temperature sensors, so a building exposes the IoT service "provide the average indoor temperature of the bus". A building subscribes to the topic *temperature* in the message data bus and publishes its data (time series of temperature) in a JSON format. Raw data collector and Data Mapping component also subscribes to the same topic and reads periodically the messages coming from all the Virtual Entities (buildings). Like above, the object description is:

Account: London Use Case

User: Raw Data Collector and Data Mapping component

Container: Temperature

Object name: Average indoor temperature of the building #2 on 10/4/2014

Metadata:

1. Id = 2
2. Start time: 2014-04-10T00:00:00

3. End time: 2014-04-10T24:00:00
4. Location: “42, Oxford Street”

4.1.4. Communication with other Components

Raw Data Collector and Data Mapping 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

VE registry (WP5): requests for social metadata stored in the triple stores during the VE registry

Cloud Storage and Metadata Search (WP4): stores data objects, with their metadata, in the cloud storage.

4.2 Complex Event Processing

4.2.1. Functional Overview

The Complex Event Processing is a component responsible for extraction of valuable information through a real-time analysis of the temporal and structural relations between various information flows within COSMOS.

The core functionalities provided by Complex Event Processing are:

- Pre-processing of the basic events.
- Real time analysis of event streams originating from multiple clients.
- Situational knowledge acquisition for COSMOS based on the streaming data.
- Adaptable event detection based on cognitive feedback loop provided by COSMOS.

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 for example by means of specific domain language as in [Dolce] or [Proton].

Since rule based inference engines are most widely used and proved, COSMOS will also integrate CEP with a rule-based inference engine.

4.2.2.2. Knowledge inference as service

The COSMOS platform will offer event detection capabilities for external components (Virtual Entities). This custom knowledge inference service will be available via dedicated REST API. It will be possible to inject new or modify existing detection rules. The particular rules will be injected in the Dolce language format. The purpose of the Dolce domain language is to provide a link between human and machine understanding of different situations. The language is specialized to an IoT domain. The language is designed for defining, modelling and analysis of

user defined context based on available streams of events. More detailed information about language as well as practical example can be seen in [D6.1.1].

4.2.2.3. Distributed deployment

COSMOS is designed to coordinate huge number of heterogeneous IoT based systems. Therefore there is there is a large potential for utilization of a distributed Complex Event Processor 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 with message bus

For analysis, monitoring and situation awareness of overall infrastructure, the Complex Event Processor will be integrated with the message bus. This message bus will integrate COSMOS components with all external components.

4.2.2.5. Adaptive feedback loop

For runtime analysis and monitoring of situations, appropriate knowledge is required. Considering the fact that COSMOS will provide some predictive analysis on historical data, we also provide a possibility to update CEP runtime analysis. This adaptive feedback loop will provide a possibility to predict particular situation in future in real time. The focus of this chapter is to describe a technical solution for injecting, updating and removing CEP analysis rules at runtime. For information about how automatic adaptive mechanisms build new event detection rules, please refer to [D6.1.1].

An adaptive feedback loop as well as VEs and other potential REST clients can integrate with CEP using a REST interface explained in more detail in following chapters especially chapter [7.2].

4.2.3. Context View

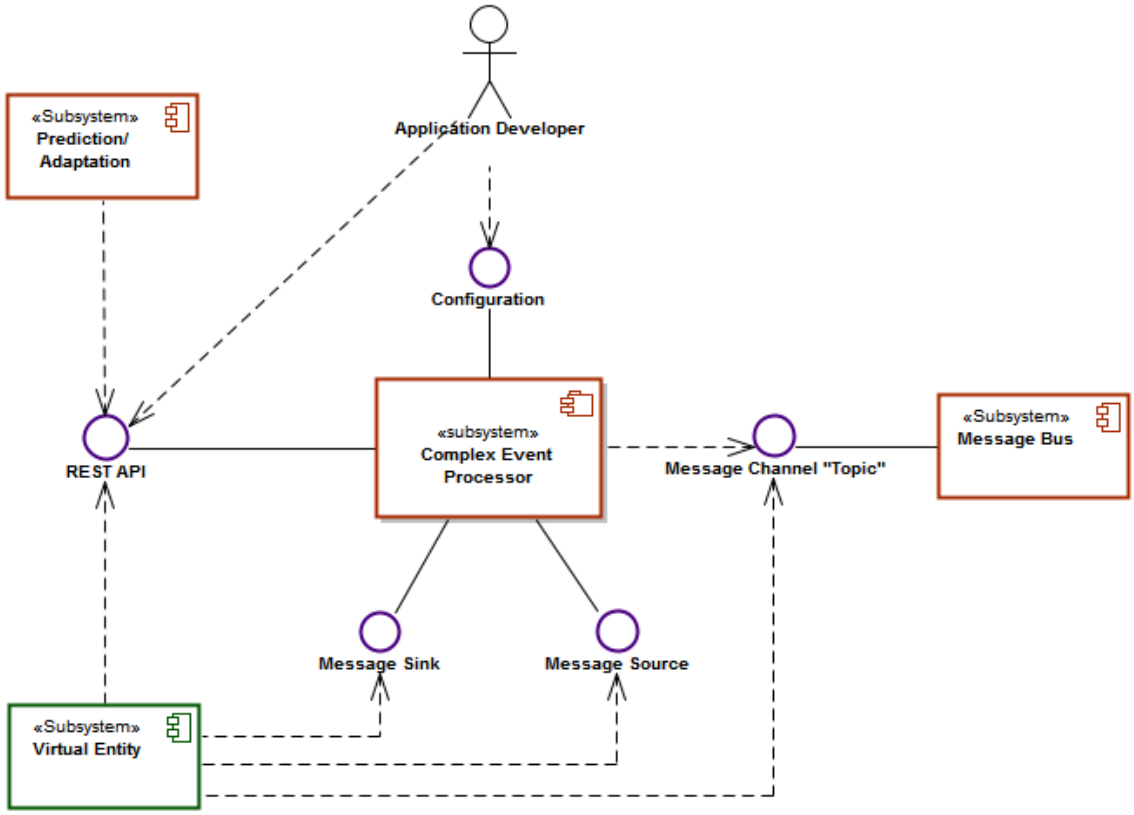


Figure 1: *Complex Event Processor External Interfacing*

This diagram shows subsystems and actors and communication interfaces related to Complex Event Detector subsystem. In this high level view, Complex Event Processor is considered as "black-box" which captures interoperability with other COSMOS components.

4.2.3.1. Actors

The following actors and roles interact with CEP subsystem:

- **Application Developer**

4.2.3.2. Sub-systems

Following sub-systems can be integrated and communicate together with CEP:

- **Complex Event Processor**: A main subsystem described in this chapter.
- **Message Bus**: The message bus responsible for propagating messages between various COSMOS components.
- **Virtual Entity**: Representation of physical objects in the heterogeneous IoT world. These are also main sources of messages within COSMOS platform.
- **Adaptation**: A component responsible for update of topic detection based on COSMOS cognitive feedback loop.

4.2.3.3. Interfaces

- **Message channel**: a general publish/subscribe interface providing means to write or read messages of particular interest.
- **REST API**: a REST interface for administration of Event Detection based on available streams of events. The core functionality includes CRUD operations for event detection rules and events in Dolce format.
- **Configuration**: an XML and Environment based interfaces for modification of Complex Event Processor behavior.
- **Message Sink**: an alternative interface for direct collection of messages. Optionally used for messages which are not transported over message bus.
- **Message Source**: an alternative interface for direct broadcast of detected messages.

4.2.4. Message bus integration

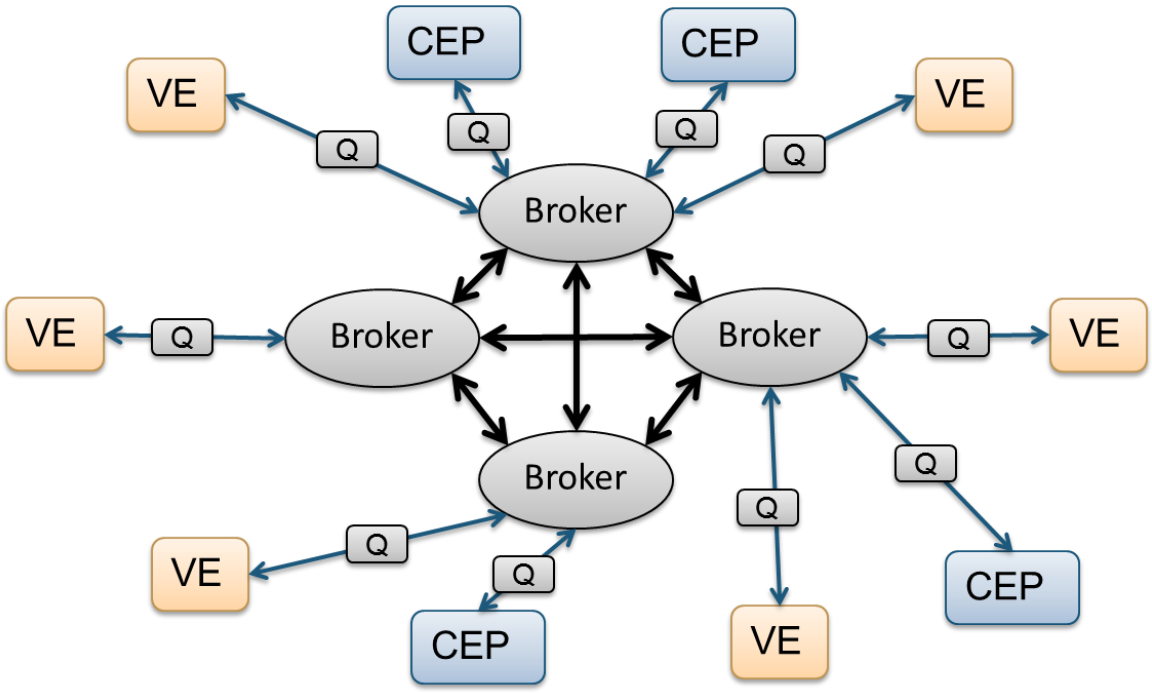


Figure 2: *Integration of Complex Event Processor with broker based message bus*

This figure 2 shows an example of integration of distributed Complex Event Processor nodes with broker based message bus such as [RabbitMQ].

Two message buses have been considered for COSMOS

- Broker-based message bus
- Distributed queue middleware aka broker-less.

4.2.5. Architectural Decomposition

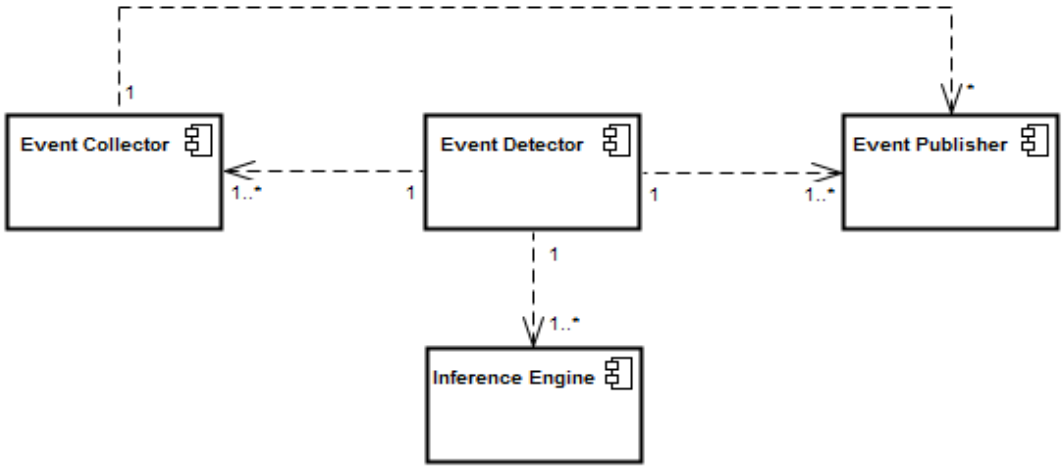


Figure 3: *CEP Subsystems*

For better understanding of overall functionality and configuration and communication possibilities of the Complex Event Processor, main functional components are described individually.

Component Name	**Description**
Event Collector	Reads all information coming from different sources using different communication protocols and data formats. It also acts as de-multiplexer, receiving events from multiple sources and sends them in proper format to the next component.
Event Detector	Controls event detection and production of expected results by delegation of particular rules to inference engine(s). Event Detector is also responsible for maintaining internal knowledge base and ensuring safety/fallback against over aggressive conditions.
Inference Engine	Is responsible for detection of particular situation by using temporal persistence of volatile events until constraints of a rule(s) are entirely satisfied.
Event Publisher	Is in charge of delivering the detected information to the expected external listeners by providing support for various communication protocols and data formats.

4.2.6. Component Communication

The communication between Event Collector, Event Detector and Event Publisher components is performed using Publish/Subscribe communication stack on top of TCP/IP network protocol. Therefore these components can be distributed on different physical processing nodes when appropriate.

For communication between Event Detector and Inference Engine an inter-process communication mechanisms are used.

4.3 Data Store

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. In year 2 and/or 3 of the project, the question of whether additional cloud storage frameworks are needed, in addition to object storage, will be examined.

Object storage allows defining CRUD operations (Create, Read, Update, Delete) on entire objects, and write-in-place is not supported. Objects can be organized into containers, and each container belongs to an account. A possible mapping of COSMOS use case data to accounts, containers and objects was described in the Data Mapping section. Account, container and object CRUD operations can be performed using the Swift REST API, as discussed in the appendix.

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.

4.3.1. 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 plan to support string, number and date 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.
- 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.

4.3.2. 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 and a search flow. 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.

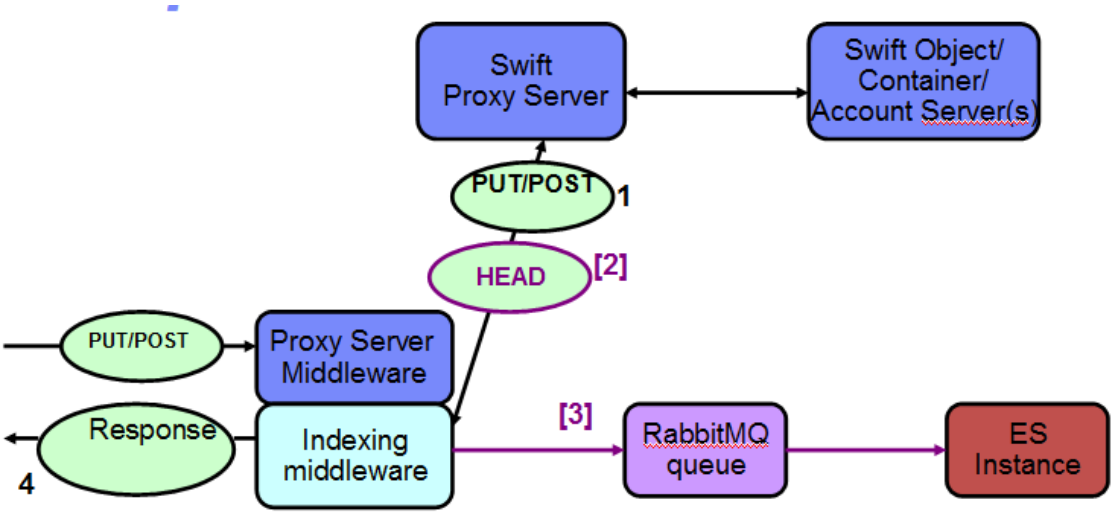


Figure 4: *Metadata Indexing Flow*

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.

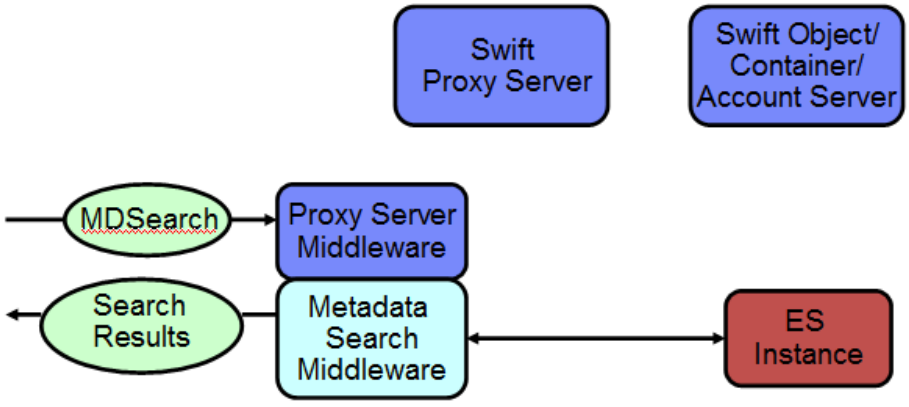


Figure 5: *Metadata Search Flow*

4.4 Storlets

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.

In the first phase of the storlets implementation we support what we call the 'GET scenario'. In this scenario the storlet is invoked during object retrieval, where instead of getting the object's data as kept in the object store, the user gets back the result of the storlet invocation on the object's data.

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 allow developers to write application specific analysis using storlets. This meets requirement 4.5.

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 6: *Request Flow in Swift*.

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

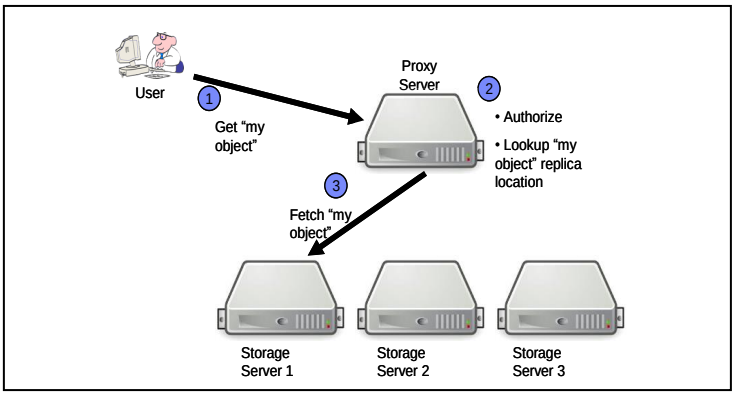


Figure 6: *Request Flow in Swift*

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.

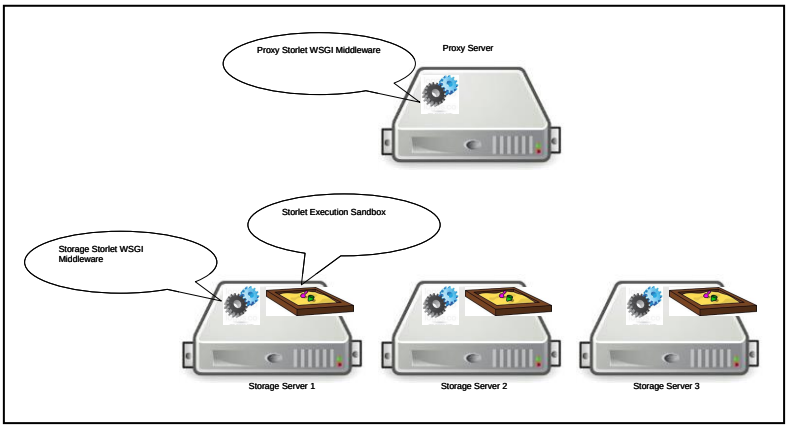


Figure 7: *The storlets' high level components: WSGI middleware in the proxy and storage servers, and a sandbox in each of the storage servers. Each storlet is executed in a daemon that runs inside the sandbox*

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 following figure describes the interaction between the storage server storlet middleware and the daemon running the storlet code.

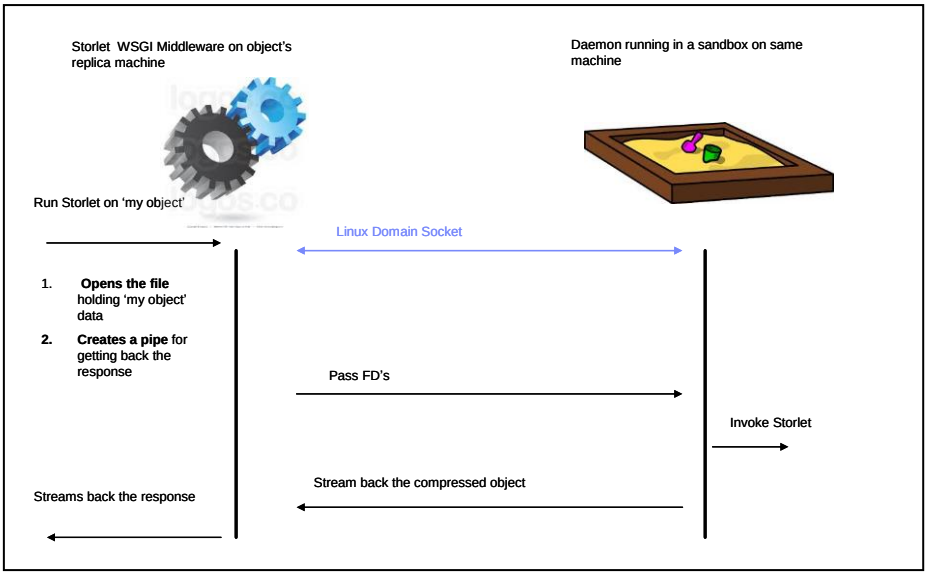


Figure 8: *The interaction between the storlet middleware on the storage server and the sandbox running on the same machine. The middleware got a request for running a storlet on an object named 'my object'. The middleware is communicating with the sandbox via a Linux domain socket to pass the designated file descriptors.*

The daemon is sandboxed in a way that it cannot access any I/O devices of the storage server. 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 Linux containers [9]. In contrast to traditional virtualization Linux containers provide an operating system level virtualization rather then 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.

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.

4.5 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.

In general the data reduction component is planned to be based on storlets, but will likely not belong to a specific component in the architecture as it may be initiated by multiple events. Either trigger the compression/decompression by specific operations such as uploads downloads, analytics jobs etc, or they will be triggered by timed events such as periodic tests or data life cycle management items. We plan to create flexible support for data reduction purposes in the data store that will allow the deployment of multiple compression techniques as well as the transcoding between various techniques. Ideally data will be stored in compressed format and a user/job will have the ability to vary its access pattern between reading compressed data (suitable for transfer), reading uncompressed data (suitable for analytics) or transcoding (for example using lossy compression in case that the analysis at hand does not require the entirety of the data in order to achieve its goals and statistics).

The work in the realm of the IoT introduces new challenges related to compression methods that we will strive to address. Specifically, data from IoT typically compresses better with *domain specific compression*. Ideally, by deploying background or sampling tasks at the data store, one can divulge the optimal compression technique for specific data and later transcode to it. Another opportunity that the IoT setting brings is the ability to use the interplay between analytics and data reduction, so that once meaningful analytics have been obtained, some (or most) of the data at hand can be dramatically reduced, leaving only traces for further analytics or verification of analytics that were obtained.

4.6 Message Bus

A Message Bus has been selected for interconnection of distributed COSMOS components as well as external clients – Virtual Entities.

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.

4.6.1. RabbitMQ

A RabbitMQ [3] messaging solution has been selected for COSMOS message bus as 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 follows messaging broker architecture build on top of AMQP [2] 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

A RabbitMQ provides three different routing algorithms. Each of them serves different type 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 exchanges are useful for directing messages based on multiple categories or for routing messages originating from multiple sources.

4.6.2. Data Format

As JSON data format is widely adopted and provides required flexibility and portability. Therefore, it has been selected for message exchange within COSMOS message bus.

4.6.3. Binary Data Serialization

A data serialization mechanism can effectively compress the data. This is very useful because smaller payload means more effective message based communication especially for message broker architectures. On the other hand, the major drawback of binary serialization is that they are neither human-readable nor human-editable.

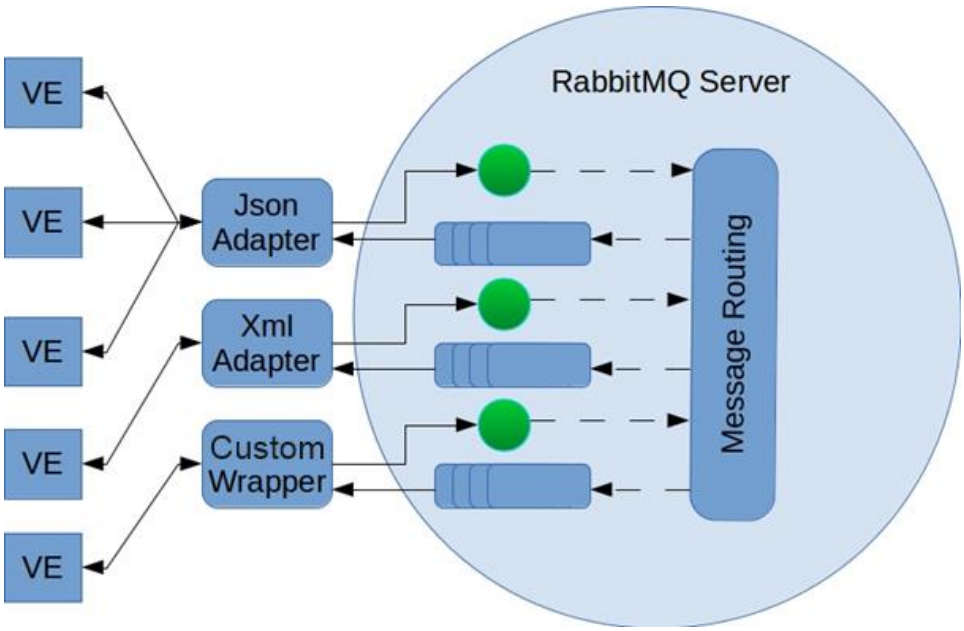


Figure 4 - The integration of message serialization adapters with a message bus.

4.6.3.1. Message Pack

A several binary serialization formats (incl. Message Pack [4], Protocol Buffers [5] and Cap'n Proto [6]) were considered for the COSMOS platform. Message Pack has been selected as:

- It is fully transparent to the JSON specification (even more transparent than BSON).
- It has support for a wide range of programming languages.
- Type checking and streaming API are built-in.

COSMOS will extend the RabbitMQ message bus with support for message serialization by utilizing the MessagePack library.

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 Raw Data Collector and Data Mapping components. 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. 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 initial plan for our work in COSMOS, which will be revised in years 2 and 3 of the project.

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] Advanced Message Queuing Protocol http://www.amqp.org/

[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/

7 Appendix

7.1 Raw Data Collector and Data Mapping (combined) API

7.1.1. JSON format

As it is mentioned in the section 4.1.2, the component expects to receive JSON files which should contain, at least, the Id field and those related to the timestamps. These fields are stored as metadata in the cloud storage and the others as the body of the data object. For instance, a file that could be acceptable regarding the example from London Use Case, as it is written in the section 4.1.3, is the following:


```
{
   "Id": "2",
   "Start time": "2014-04-10T00:00:00",
   "End time": "2014-04-10T24:00:00",

   "Temperatures": [
      { "time": "2014-04-10T00:00:00", "temperature": "10 C" },
      { "time": "2014-04-10T02:00:00", "temperature": "09 C" },
      { "time": "2014-04-10T04:00:00", "temperature": "08 C" },
      { "time": "2014-04-10T06:00:00", "temperature": "10 C" },
      { "time": "2014-04-10T08:00:00", "temperature": "11 C" },
      { "time": "2014-04-10T10:00:00", "temperature": "12 C" },
      { "time": "2014-04-10T12:00:00", "temperature": "13 C" },
      { "time": "2014-04-10T14:00:00", "temperature": "14 C" },
      { "time": "2014-04-10T16:00:00", "temperature": "13 C" },
      { "time": "2014-04-10T18:00:00", "temperature": "12 C" },
      { "time": "2014-04-10T20:00:00", "temperature": "11 C" },
      { "time": "2014-04-10T22:00:00", "temperature": "10 C" },
      { "time": "2014-04-10T24:00:00", "temperature": "10 C" }

   ]
}
```

7.1.2. Configuration

The main parameters that are associated with the component's configuration are the **period** in which it reads the messages from the data bus and the **size** of the data object that is going to be stored in the cloud storage. For year 1 both parameters are static and will be defined during the implementation phase of the component depending on real data coming directly from the use cases. In year 2 and/or 3, users will be allowed to configure the component according to their specific application.

7.2 Complex Event Processing API

The Complex Event Processing solution will provide REST API build on pragmatic RESTful design principles. The API will use resource-oriented URLs that leverage build 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}/solcep/v1.

The serverRoot is the address of the machine hosting CEP 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 /solcep/v1/rules?name=TrafficJam`

7.2.7. Events

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

`GET /solcep/v1/events`

A single existing event can be obtained usin API:

`GET /solcep/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 /solcep/v1/rules

A single existing event can be obtained usin API:

GET /solcep/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]

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 system and/or custom metadata key to be compared against the <query value> as a query criterion.

<query value> = The value to compare against the <query attr> using the <operator>. The value is either a numeric decimal value without quotes, or a string enclosed in single quotes.

<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)

The format parameter specifies whether the output should be returned in json, xml or plain text format. The default is plain text.

The type parameter specifies whether to search for containers or objects. The default is both containers and objects.

Date, number and string data types will be supported for metadata values. Users will be able to specify that certain metadata keys are associated with one of these data types (default is string). This is needed in order to support range searches on date and number types.

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)
```

Once a user invokes a storlet via the Swift GET REST API, the invoke method will be called as follows:

1. The inStreams array would include a single element representing the object to read. A StorletInputStream has a member 'stream' that can be accessed using getStream(). This member is of type java.io.InputStream on which one can do read() to get the object's data.

2. The outStreams would include a single element representing the response returned to the user. A StorletOutputStream has a 'member' stream that can be accessed using getStream(). This member is of type java.io.OutputStream on which one can do write(). 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. The StorletLogger class supports a single method called emitLog, and accepts only String type. Each invocation of the storlet would result in a newly created object that contains the emitted logs. Creating an object containing the logs per request has its overhead. Thus, the actual creation of the logs object is controlled by a header supplied during storlet invocation. More information is given in the storlet execution section below.

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.

7.4.3. Storlet Invocation API

In the GET scenario, a storlet in invoked using a regular Swift GET operation 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 looke like:

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

```
X-run-storlet: MyStorlet.jar
X-generate-log: True
```