


FP7-ICT-2013-EU-Japan

ClouT: Cloud of Things for empowering the citizen clout in smart cities

FP7 contract number: 608641

NICT management number: 167 ア

Project Deliverable

D2.3 – CIaaS specification and reference implementation – second release

---

ABSTRACT

This deliverable includes the second version of the CIaaS specification: the functional blocks listed in D1.3 - "*Final requirements and Reference Architecture*" - are analyzed more in the details and their main interfaces are identified. For each main functional block, we focused on the implementations that have been deployed, to report the actual technical details of the interactions among the various block, with the existing technologies proposed by partners, and some considerations about security when it applies. In this second version of the specification we mainly addressed the evolution of the proposed components and the necessary wirings to make them properly interoperate.

The subsequent revision (objective of D2.4 – "*CIaaS final specification and reference implementation –review after field trials*") will take into account all the integration issues and the outcomes of the field trials, in order to improve and stabilize the current design, as well as some specification aspects that are still in a work in progress stage.

Disclaimer

The ClouT consortium is composed of the following institutions:

No.	Participant organization name	Short name	Country
1	Commissariat à l'énergie atomique et aux énergies alternatives	CEA (coordinator)	France
2	Engineering Ingegneria Informatica SpA	ENG	Italy
3	University of Cantabria	UC	Spain
4	STMicroelectronics S.r.l.	ST	Italy
5	Santander City Municipality	SAN	Spain
6	Genova Municipality	GEN	Italy
7	Nippon telegraph and telephone East Corporation (NTT East)	NTTE (coordinator)	Japan
8	Nippon Telegraph and telephone corporation (NTT R&D)	NTTRD	Japan
9	Keio University	KEIO	Japan
10	Panasonic System Solution	PANA	Japan
11	National Institute of Informatics	NII	Japan

EU Editor	Marco Grella, ST
JP Editor	Hiroyuki Maeomichi, NTTRD
Authors	Marco Grella, ST; Ciro Formisano, Daniele Pavia, Philip Wright, ENG; Levent Gürgen, Christophe Munilla, CEA; José Antonio Galache, UC; Hiroyuki Maeomichi, Takayuki Suyama, NTTRD; Takuro Yonezawa, KEIO
Internal reviewers	Ciro Formisano, Daniela Pavia, ENG
Deliverable type	O
Dissemination level (Confidentiality)	PU
Contractual Delivery Date	31/01/2015
Actual Delivery Date	30/04/2015
Keywords	ClouT, Cloud Computing, IoT, Smart Cities, CSaaS, CPaaS, CIaaS, System Requirements, Reusable Components, Reference Architecture, Functional Models, Field Trials

Revision history

Revision	Date	Description	Contributors
v0.1	15/12/2014	Table of Contents created	ST
v0.2	04/02/2015	Contribution to sections 2.1(ENG), 3.2/3.3 (UC), 3.3.2 (ST), 6.1 (ENG)	ST, UC, ENG
v0.3	09/02/2015	Contribution to sections 3 (ST), 3.3.3 (UC), 4.2/4.3 (KEIO), 6.1 (ENG), 7 (CEA)	ST, KEIO, ENG, UC, CEA
v0.4	12/02/2015	Contribution to sections 5 (NTTRD), section 3 and overall document (ST), section 3.4 (UC)	NTTRD, ST, UC
v0.5	27/02/2015	Contribution to sections 2/6 (ENG), sections 1/2/3/5/7 (ST), 5 (NTTRD), section 6 (UC)	NTTRD, ST, UC, ENG
v0.6	01/03/2015	Overall document fixing (ST), contribution to sections 7 (ST), drafted common sections (abstract, executive summary, conclusions)	ST
v0.7	06/03/2015	Overall document fixing, sections 2/Appendix A (ST), sections 3/5/7 (CEA), section 4 (KEIO), section 6 (ENG), sections 3/5 (UC)	ST, CEA, KEIO, ENG, UC
v0.8	11/03/2015	Section 7.3 (CEA), section 6 (UC), section 7/Appendix (ST), overall document fixing (ST)	CEA, UC, ST
v0.9	18/03/2015	Section 7.2 (UC), section 6 (ENG), section 5.3 (KEIO), sections 1, 2, 3, 5, 7 and overall document fix (ST)	UC, ENG, KEIO, ST

v1.0	23/03/2015	All sections about Security and Opportunities and Threats (UC). Overall document (ST). Sent to preliminary internal review.	UC, ST
v1.1	26/03/2015	Sections 3.6 and 7.5 (CEA), section 5.1 (NTTRD), small fixes and section 3.6 reorganization (ST)	CEA, NTTRD, ST
v1.2	20/04/2015	Addressing internal review comments	ENG, CEA, ST
v1.3	21/04/2015	Fixed internal reviewers comments and other typos (ST), Security in section 4 (KEIO).	ST, KEIO
v1.4	29/04/2015	IoT and cloud section (ENG), added pictures with mapping of components on the architecture (ENG, UC), final fixes (ST)	ENG, UC, ST
v1.5	30/04/2015	Added picture on ClouT product	ST
V1.6	30/04/2015	Final revision	CEA
V1.7	15/07/2015	Integration of a new table describing the differences wrt to the D2.2 (requested after the 2nd project review)	ST

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Abbreviations

TABLE 1 : LIST OF ABBREVIATIONS

6LoWPAN	IPv6 over Low Power Wireless Personal Area Networks
API	Application programming interface
CDMI	Cloud Data Management Interface
CIaaS	City Infrastructure as a Service
CIL	Common Intermediate Language
CLR	Common Language Runtime
CoAP	Constraint Application Protocol
CoRE	Constrained RESTful Environments
CPaaS	City Platform as a Service
CPU	Central Processing Unit
CSaaS	City Software as a Service
DoW	Description of Work, Annex II to the Grant Agreement
DTLS	Datagram Transport Layer Security
HAProxy	High Availability Proxy
HTTP	HyperText Transfer Protocol
ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IoT	Internet of Things
IoT-A	Internet of Things – Architecture
IPv4	Internet Protocol version 4
Ipv6	Internet Protocol version 6
IPSO	Internet Protocol for Smart Objects
JSDL	JSON Service Description Language
JSON	Javascript Object Notation
JSON-RPC	Remote Procedure Call protocol encoded in JSON
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
LUN	Logical Unit Number
LWM2M	Lightweight Machine To Machine (Device management protocol, by OMA)
NAS	Network Area Storage
NFS	Network File System
NIST	National Institute of Standards & Technology's
NPM	Node Packaged Module
OMA	Open Mobile Alliance
OS	Operative System
OSGi	Open Services Gateway Initiative
PHP	PHP: Hypertext Preprocessor (server side scripting language)
RDF	Resource Description Framework
REST	Representational State Transfer
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol

SPARQL	SPARQL Protocol and RDF Query Language
SPGW	Service Provision Gateway
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDDI	Universal Description, Discovery and Integration
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
USDL	Unified Service Description Language
VES	Virtual Execution system
VLAN	Virtual Local Area Network
VM	Virtual Machine
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
WSN	Wireless Sensor Network
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

TERMINOLOGY USED IN CLOUT

TABLE 2 : SERVICES TERMINOLOGY

Terminology	Definition	Examples
CIaaS	A model that provides virtualized city resources (sensors, actuators, storage, etc.) as a service.	Open city data as a service
CPaaS	A model that provides city computing platform (event processing, mash-up tools, etc.) as a service.	City application development tools
CSaaS	A model that provides city application software (city event analyser, public space management, etc.) as a service.	Participatory sensing, context-aware city applications
IaaS	A model that provides virtualized computer resources (CPU, network, storage, etc.) as a service.	Amazon EC2, Amazon S3
PaaS	A model that provides computing platforms (OS, Executable runtime, databases) as a service.	Google App Engine, Windows Azure
SaaS	A model that provides application software (CRM, Email, communication) as a service.	Salesforce.com, Gmail, Google docs
Service	A functionality offered by a computing entity via well-defined access interfaces to perform a given task	

TABLE 3 : ENTITIES TERMINOLOGY

Terminology	Definition	Examples
Action Resource	A City Resource which can be used as actuator in the city.	Virtual City Light
Actuator	An electronic hardware that acts on a physical property. The term actuator is used interchangeably as an IoT device with actuating capabilities.	Light, Display, Speaker, Shutter, heating devices
Actuatorised Web Application	A web application which is able to be used as a concrete actuator.	
Big Data	It identifies data, typically a large amount of data that needs “ad hoc” software to be	

Terminology	Definition	Examples
	managed.	
City Action	An action which controls action resource in the city.	turn-on, display
City Data	A generic term for representing data which is generated by sensors and web related to the city.	Presence information, city events, data captured by citizens
City Infrastructure Entity (CIE)	Representation of any physical or virtual entity providing data streams or actions.	IoT devices, web applications
City Infrastructure Service	A service that virtually represents a City Infrastructure Entity and exports means to access to Virtualized City Resources exposed by the service.	LightService, TemperatureService, PresenceService
City Resource	A resource of city which is represented by city infrastructure service. It is classified to Data Resource and Action Resource.	Virtual City Temperature, Virtual City Light
Cloud Storage	It is a service of data storage, typically file based, where the data is distributed into virtualized servers.	OpenStack Swift, Ceph, GlusterFS
Data Resource	A City Resource which can be used as city data generator.	Temperature, presence information, luminosity, humidity
Device	Any computing equipment with possible communicating and storage capabilities.	Computer device, IoT device, legacy device
IoT Device	A device that incorporates new generation, mostly wireless, communication capabilities, and possibly sensors and actuators.	Zigbee, 6LoWPAN devices, communicating home appliances, networked cameras, etc.
Legacy Device	A device that belongs to an "old" pre-existing infrastructure that is already deployed and will be reused after adaptation by the ClouT system. It can provide sensing or actuating capabilities	Traffic light system, SCADA systems, street light regulators
Legacy Web Application	A pre-existing web application that will be reused after adaptation by the ClouT system.	City web services
Metadata	A metadata is a collection of attributes that describes more in detail the data that it is	Timestamp, unit,

Terminology	Definition	Examples
	associated with.	owner, location
Networked Legacy Device	A legacy device which turned into being connected to the Internet.	
noSQL Database	It identifies a database that is not SQL based. It identifies a no relational database (NRDBMS).	Apache HBASE, Cassandra, MongoDB
Open Data	Data which is freely accessible through the Internet via well defined interfaces.	
Sensor	An electronic hardware that detects or measures a physical property. Here the term is used interchangeably as an IoT device with sensing capabilities.	Temperature, humidity, presence sensor
Sensor Data	Data stream which is generated by any city infrastructure entity (Sensors, sensorised web applications, sensorised legacy devices, etc.).	Temperature data, humidity, presence information flow, city events flow
Sensorised Web Application	A web application which is able to be used as a sensor.	Twitter, Facebook likes, etc.
Social Network Application	A web application which purpose is to connect people in consideration with their social relationship.	Facebook, Twitter, Foursquare
Virtualized Actuator	An entity which provides actuation function according to defined actuator capabilities. It is mapped to a concrete actuator at runtime.	
Virtualized City Resource	A virtualized resource in terms of either sensor data or an action exposed by City Infrastructure Service.	
Virtualized Sensor	An entity which provides sensor data stream according to defined sensor capabilities. It is mapped to concrete sensor at runtime.	
Virtualized Storage	An entity which provide sensor data history according to defined storage capabilities. It is mapped to concrete storage at runtime.	

TABLE 4 : TRANSFORM/FUNCTION/PROPERTIES TERMINOLOGY

Terminology	Definition	Input	Output
Abstraction	A method to abstract sensor node platform to provide programming capability by various languages.	Specific sensor node hardware resources	Abstracted sensor node
Actuatorisation	A transform method to change legacy web application to actuatorised web application (or actual transformation using the method).	Legacy web application	Actuatorised web application
Data Processing	A method to compute data.	Raw data	High level data
Decision Making	Decision making can be regarded as the cognitive process resulting in the selection of a course of action among several alternative scenarios.	Events, conditions and rules	Action or a choice
Dependability	Capability for monitoring, fault and error detecting and recovering mechanism.	Dependability requirements	Dependability ensurance
Event Processing	A method to process data/events to detect higher level event.	Raw events	Complex events
Hosting	A method to provide accessibility to sensor data stream and history.	Query	Sensor data streaming /history
Mapping	A method to map virtual resources.	Virtual resources	Physical resources
Mash-up	Creating a high level service by co-operating several services.	Different services	High level service
Self-binding	The process by the way of which the system ensures automatic connections of inter-dependent services in the system.	Required and exposed services	Bound services
Self-discovery	The process by the way of which the system ensures that all reachable and recognizable physical resources are discovered and integrated to the system.	Service needs, service repository	Discovered services

Terminology	Definition	Input	Output
Self-healing	A method to auto-repair an IoT error or failure.	Abnormal situation of sensors	Normal situation of sensors
Self-identification	The process by the way of which the system ensures the suitable virtual representation of a physical resource in the system (i.e. the features of the physical resource are accessible through its virtual counterpart) and provides a formalized description of those features allowing functionalities discovering.	Information on resources	Formalised descriptions
Semantic Interoperability	Interoperable capability on function among different formats of sensor data by utilizing meta data.	Different format of sensor data	Interoperable sensor data
Sensorisation	A transform method to change legacy web application to sensorised web application.	Legacy web application	Sensorised web application
Service Composition	A method to combine different services as one service.	Different services	Composite service
Syntactic Interoperability	Interoperable capability on communication among different operations of sensor nodes.	Different syntactic	Interoperable syntactic
Virtualization	A method to create virtual resources corresponding to physical resources.	Physical resources	Virtual resources

EXECUTIVE SUMMARY

The architecture redefinition between the 2.2 and 2.3 releases of this deliverable aims at streamlining the architecture, removing some non-functional components and refining the core components that were instantiated in the first release. In the course of the evolution some components have been moved into other parts of the ClouT architecture as well (see for example the IoT kernel). This evolution is summarized in the table below, in which the textual explanations are not exhaustive and are completed by the parallel presentation of the previous and the current release architectures:

From ***City Resource Management*** to ***City Infrastructure Management***: This block has been redefined for a better mapping of the functional components to the inner system's data structures hierarchy (the ClouT resource model), and so of a clarification of the responsibilities of each element of this hierarchy by specifying their management associated functionalities; From the Service Management perspective, different modules have been identified in order to carry out the search and discovery of the different deployed services, thus retrieving the dedicated resources and the specific APIs for accessing to the data associated to each of these services, from the corresponding (resources and APIs) repositories.

2.2 Release			2.3 Release		
Block	**Component**	**Sub-Component**	**Block**	**Component**	**Sub-Component**
City Resource Management	Universal Service Descriptions	Managing Profiles	City Infrastructure Management	Resource Access Management	
		Describing profiles		Service Management	
				City Entity Management	

From ***Interoperability*** to ***Interoperability & City Resource Virtualization***: Regarding the syntactic and semantic interoperability components the basically internal components and functionality have been kept the same as in the first release.

2.2 Release			2.3 Release		
Block	**Component**	**Sub-Component**	**Block**	**Component**	**Sub-Component**
Interoperability	Syntactic Interoperability	Data Transformation	Interoperability & City Resource	Semantic & Syntactic Interoperability	Data Transformation
		Syntax			Syntax Interpreter

		Interpreter	Virtualization		
		Syntax Verifier			Syntax Verifier
	Semantic Interoperability	Data Converter		City Entity Virtualization	Data Converter
		Component Repository			Component Repository
		Metadata Repository			Metadata Repository

From ***Virtualization and Hosting*** to ***Computing and Storage***: the *Virtualization & Hosting* functional block has been streamlined. Load balancing has been removed as it has been considered non-functional, even though its capabilities have been maintained in the Reference Architecture and reference implementation. The *Hosting* sub-component has been merged as its functionalities overlapped with the *Interoperability & Virtualised City Resources* and the *City Infrastructure Management* has it had partial support for sensor data interoperability and exposed cloud storage access; all those functionalities are present in ClouT's architecture. The actual Computing & Storage component reflects the idea of having computational and storage resources at City disposal that can be exploited "as a Service" thanks to the Cloud virtualization facilities.

2.2 Release			2.3 Release		
Block	**Component**	**Sub-Component**	**Block**	**Component**	**Sub-Component**
Virtualization & Hosting	Virtualization	Load Balancing	Computing & Storage	Computing as a service	
		IaaS Virtualization			
		NAS/Distributed Filesystem			
	Hosting	Translation Layer		Storage as a Service	
		Receiver Software			

IoT Kernel: In the first release a basic implementation of the IoT kernel block was provided on the basis of the original key building blocks (sensiNact and SmartSantander IoT Gateways). The current release provides functional extensions with more protocols supported such as XMPP and MQTT. The IoT Kernel block has been simplified, reflecting the migration of some functionality in the City Infrastructure Management block. Emphasis has been put on device management, with a sound common API and enrichment of the device wrapping layer.

2.2 Release			2.3 Release		
Block	**Component**	**Sub-Component**	**Block**	**Component**	**Sub-Component**
IoT Kernel	Standard-based IoT	Northbound Protocol	IoT Kernel	Uniform Access to IoT Devices	Common API

		Adapters			
		Service, Resource Discovery & Management		IoT Device Management	Resource Discovery
		Uniform Access to IoT Devices			Resource Directory
	Abstraction of IoT Devices	Virtualized Devices as Services		IoT Device Wrapping	Protocol Adapters (Protocol handlers + Link handlers)
		Device Discovery, Device Directory, Device Manager			
		IoT Protocol Adapters			

Sensorization and Actuorization: In the first release, we provided a first prototype of design and implementation for the sensorization/actuatorization functionality. Through our experiences with the first implementation, we analyzed more sophisticated way to design and implement of sensorization/actuatorization, especially for sensorization. In the current release we provide the first complete implementation of sensorizer with additional functionalities. Firstly, it has scalability function to cope with numerous number of WEBs. It adapts distributed mechanism to monitor WEBs for sensorization. Secondly, we designed and implemented automatic sensorization functionality which finds and sensorizes WEB information automatically. We opened our system through our web site and some of active developer inside/outside ClouT already uses our system.

2.2 Release			2.3 Release		
Block	**Component**	**Sub-Component**	**Block**	**Component**	**Sub-Component**
Sensorization and Actuorization	Uniform Access to Sensorized/ Actuorized Devices		Sensorization and Actuorization	Uniform Access to Sensorized/ Actuorized Web/SNS	Sensorized/ Actuorized Device Server
	Noise Reduction			Noise Reduction	Noise Reduction Engine
	Network Enabling			Networked Sensor / Actuator Enabler	Agent Software for Web monitoring
					Networked Sensor / Actuator Converter

The second version of the ClouT CIaaS specification follows the reference architecture identified in [D1.3]. For each of the five main functional blocks that compose the CIaaS layer the interfaces among the composing sub-blocks or the main blocks are reported, along with details on the proposed implementations, this document will particularly focus on.

The ***IoT Kernel*** block is standardized starting from two main existing and consolidated implementations (sensiNact and SmartSantander IoT Gateways), extended to include more protocols (XMPP, MQTT), to interoperate one with the other and with new emerging M2M solutions like the IPSO Smart Object profile on top of OMA LWM2M. This block deals with low level interaction with ClouT IoT Devices: device discovery and relevant directory, access to the resources exposed by the different sensors/actuators, device management tasks.

The ***Sensorisation and Actuatorisation*** block is in charge of the access to resources like legacy devices, sensor data reported on web pages or social networks, in order to expose these data sources to the Interoperability block with the same interface as the ClouT IoT devices. Access to web data takes advantages of the development of a specific extension to Chrome web browser. A proper noise filter is needed to access social networks data, in order to extract only meaningful information.

The ***Interoperability and City Resource Virtualisation*** block is in charge of granting both syntactic and semantic compatibility among data coming from different devices of different vendors. This block takes care also of the conversion among the supported protocols and data format into a common internal solution (ICDF), starting from widely adopted standards like XML, JSON, TLV, ASN.1. The Virtualisation tasks grants the access to the virtualized resources by the City Infrastructure Management block, leveraging on the main implementations (sensiNact and SmartSantander IoT Gateways).

The ***Computing and Storage*** block offers a virtualisation infrastructure, with high availability, scalability and resource exploitation properties. It leverages on solutions like Openstack Swift, Hypertable, Orion context Broker and Cosmos Big Data (from the FI-WARE generic enablers) for the Storage access, and on the *Openstack Cloud Computing Platform* for the computing part.

The ***City Infrastructure Management*** block implements the highest layer of the ClouT CIaaS, granting the CPaaS block the relevant API to access the City Infrastructure Virtualisation by means of search capabilities and event management. Also in this case the sensiNact gateway, thanks to its OGSi architecture, is used as a building block to expose City Resources in terms of Services.

1. INTRODUCTION

1.1. Scope of the Document

The main objective of this deliverable is to consolidate the specification for the City Infrastructure as a Service (CIaaS) layer of the ClouT system, along with details about its implementation. This document takes into account the final reference architecture as reported in ClouT Deliverable 1.3 "*Final requirements and Reference Architecture*" and describes more in details the interactions and interfaces of the various functional blocks.

As illustrated in Figure 1, the CIaaS layer is logically situated below the City Platform as a Service (CPaaS) layer, and is mainly in charge of the data retrieval, both from physical or virtual devices, of granting their interoperability and of offering to the above layer adequate API to interface with the City Services. CPaaS is in charge of the data processing and exploiting.

A Security&Dependability block is transversal to the whole reference architecture.

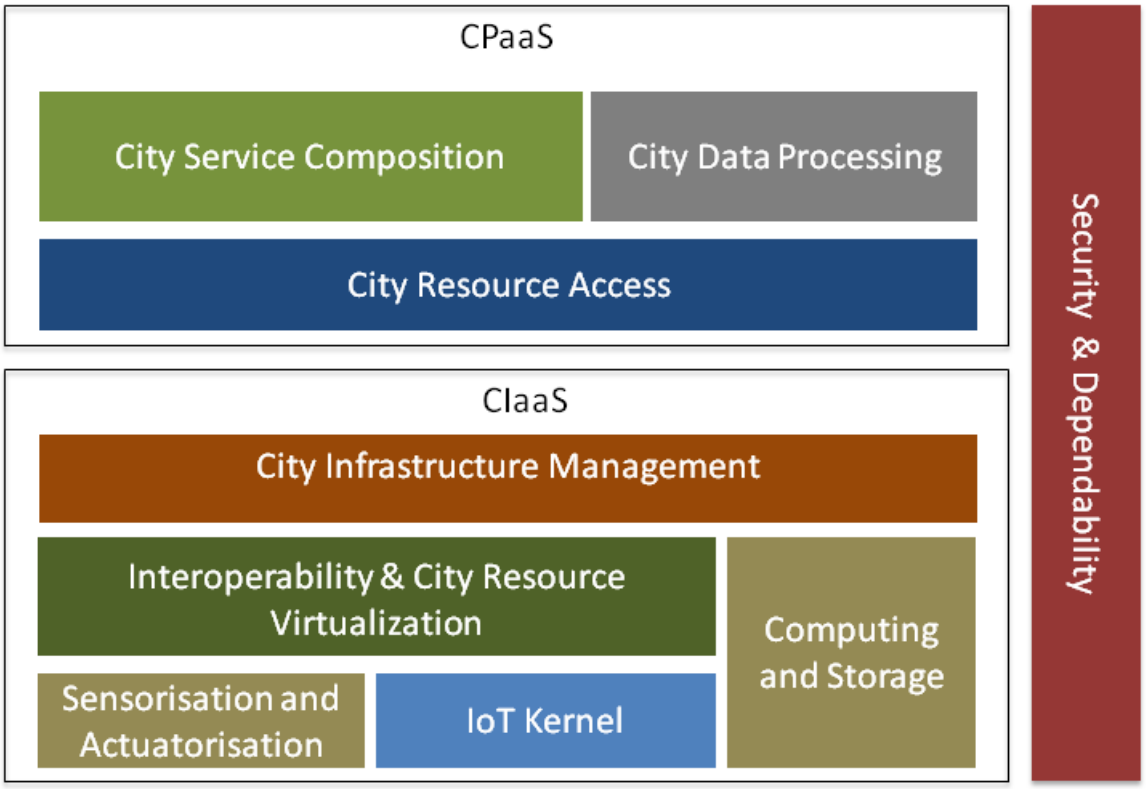

FIGURE 1: MAIN FUNCTIONAL BLOCKS FOR CLOUT REFERENCE ARCHITECTURE

1.2. Target Audience

The document is addressed to the following target groups:

- Smart City Ecosystem integrators: each person who is looking to implement a Smart City Ecosystem based on ClouT Reference Architecture. The content of this document can give a detailed view on the APIs to interact with the different ClouT components at different layers.
- ClouT project members/developers: this document describes the functional block interfaces and implementation details, so it will be the reference to implement new functionalities or to modify the already developed tools.

1.3. Structure of the Document

Chapter 2 of this document recalls the overall description of the CIaaS Reference architecture, as identified in the deliverable [D1.3].

The five subsequent chapters are devoted to each of the main functional blocks that compose the CIaaS layer; in each chapter for each of the sub-blocks that compose the main task, three sections are reported:

- First section describes the ***Architecture*** of the sub-block itself by means of communication diagrams that recall the functional sub-blocks along with the generic interactions among them.
- Second section details the ***Specification*** of the sub-blocks, describing its offered interfaces, without entering in the details, for instance, of the used or proposed programming language; in this section, the System Requirements addressed by this sub-block are also recalled. Figures that maps the proposed components (reusable or developed) to the ClouT reference architecture are also provided in this section.
- Finally ***Implementation*** section gives the details of some possible implementations that partially or totally fulfil the specified functionalities of the components.

For some of the main block (for example, IoT Kernel), a more comprehensive implementation section may be reported, in the cases where it makes more sense from a logically point of view.

A section devoted to highlight the reuse of the different components is also reported, with 2 tables: 1 table clarifying the components that will be reused / extended from the state of the art and the components that will be developed in the project, and the 2nd table giving opportunities/threats analysis.

For each chapter, a brief paragraph about security considerations is also reported, knowing that these aspects will be covered more tightly in the last year of the project.

To summarize: chapter 3 covers the "IoT Kernel" block; chapter 4 covers the "Sensorisation and Actuatorisation" block; chapter 5 covers the "Interoperability & City Resource Virtualisation" block; chapter 6 covers the "Computing And Storage" block; chapter 7 covers the "City Infrastructure Management" block.

Eventually, chapter 8 draws the conclusions of this document and highlights the main achievements along with the open issues and the next steps.

2. CITY INFRASTRUCTURE AS A SERVICE: OVERALL ARCHITECTURE

The final version of the reference architecture of the ClouT has been defined in the deliverable 1.3. Figure 2 illustrates the two main technical layers of the architecture, namely CIaaS and CPaaS, giving an overview on their internal functional blocks.

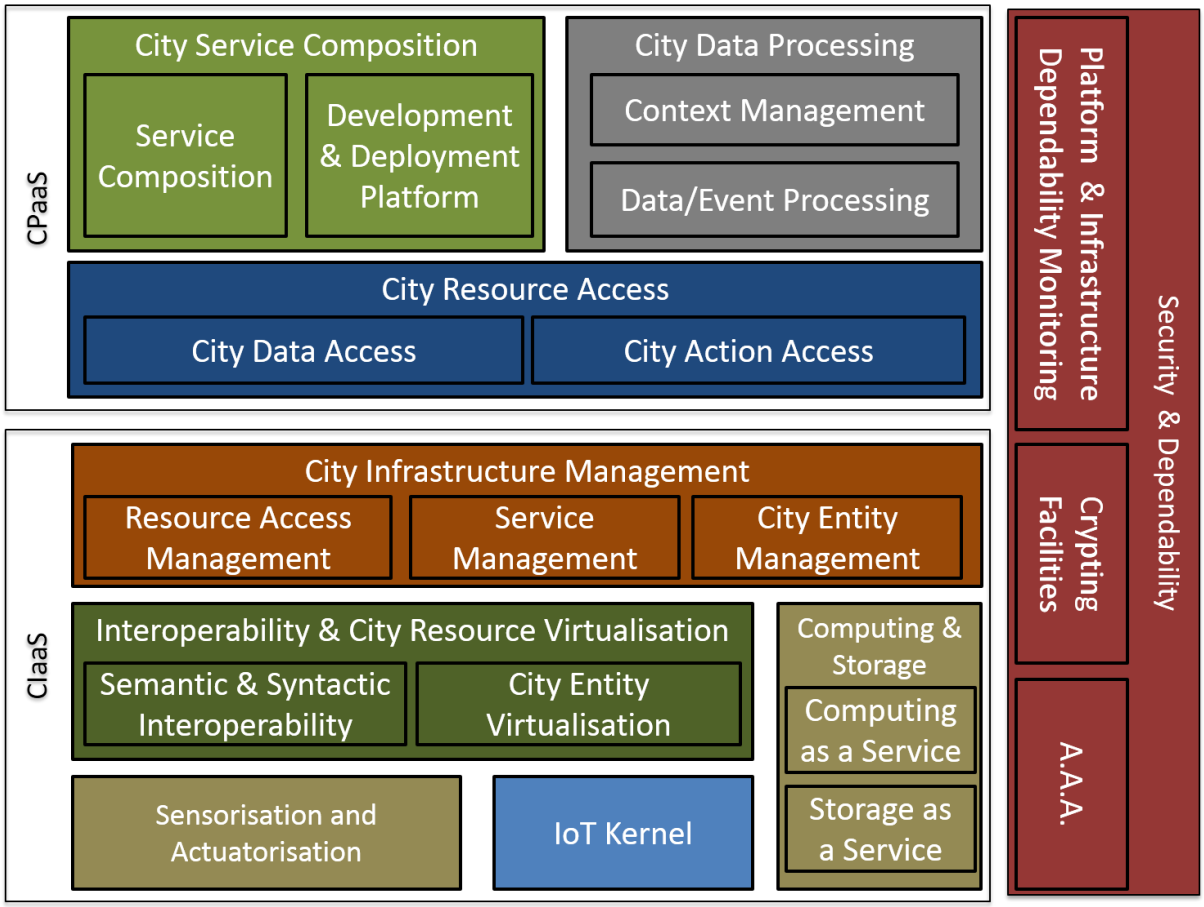

FIGURE 2: THE ARCHITECTURAL OVERVIEW OF THE CLOUT SYSTEM

This deliverable focuses on the CIaaS layer and specifies the necessary functions that are necessary to fulfil the requirements we have identified in the deliverable 1.3.

The CIaaS layer is composed of five main functional blocks:

- The ***IoT Kernel*** block deals with the access to the physical sensor and actuator devices, taking into account the different media and protocols they rely on, and offering to the other blocks a common abstract interface to their resources; it also deals with the device management aspects.
- The ***Sensorisation and Actuatorisation*** block completes the connectivity and abstraction task, granting access to legacy devices as well as to social networks and other web applications (or sensor data exported on web pages).
- The ***Computing and Storage*** block provides a virtualization infrastructure, with high availability and scalability properties, and storage capabilities to safely store all the data retrieved from the city, including sensor data and object data (such as images, documents or email).
- The ***Interoperability & City Resource Virtualisation*** block takes care of the necessary translations needed to make data coming from different sources understandable and meaningful to different consumers, providing semantic and syntactic capabilities.
- The ***City Infrastructure Management*** block grants access to City Services (providing discovery and search capabilities) and Resources (with the relevant API to get historical data, or on demand or, finally, subscribing to some event).

2.1. Cloud for IoT in ClouT

Cloud technologies provide the following useful features in IoT context:

- **scalability**
- **pay as you go**
- **resiliency**
- **rapid development via PaaS.**

In the following part of this section these features will be described in general and the additional value that these features bring to the IoT context is explained.

Scalability. Given the Internet of Things and the Internet of People's nature, billion of networked devices and millions of people constantly generating data, storage and computational requirements to store, process data, build and monitor services upon such data are steep. By its own definition, the Cloud is virtually resource limitless. That's due to the fact that Cloud systems are typically scalable: if the need for more physical resources arises, providers can add more physical nodes, rapidly increasing system capabilities. Usually, Cloud providers have plenty of physical resources already at hand which, together with overbooking policies, means that extending storage and computational resources is just a mere request away. Cloud bursting mechanisms further add to Cloud's scale out capabilities, rendering, for example, a hybrid Cloud approach interesting to those cities that already have a private cloud infrastructure in place. Such scalability features are a perfect match for a smart city that wishes to build services that

are capable of hosting and processing enormous amounts of data in quasi-real-time. Moreover, Cloud services brokerage and citizen-location-aware services can make data and services available where the user needs them. Finally, due to the Platform as a Service scalability properties, Cloud-enabled smart city applications can dynamically scale in and out resource allocation, therefore becoming capable of efficiently handling spikes in workload. This applies all Cloud-based applications and services developed for the smart cities.

Pay as you go. One peculiar concern that may prevent municipalities to deploy a smart city infrastructure is the involved cost. The need to invest a relevant quantity of money to set up and maintain a sensor and processing infrastructure may drive potential adopters away. However, thanks to the Cloud pay-as-you-go model and ClouT's architecture flexibility, the deployment of the components that are necessary to instantiate a processing architecture can happen within a short time frame and with a minimal investment. Moreover, thanks to ClouT sensorisation capabilities, which turn nearly every user generated content (for e.g. a participatory sensing initiative) available on the internet into a possible source of data – just as a smart networked sensor – a municipality may find itself already in possession of a significant amount of data, with no need to deploy any physical sensor. Such scenario obviously has its limits, but it goes to show that every municipality, even the smallest, can adopt ClouT architecture with a limited initial investment, increasing its spending only when the need for more resources and more complex services arises.

Resiliency. Cloud services are geo-location independent, which makes public Cloud based smart city services reliable in the event of a local disaster. That's in stark contrast to a local, municipality owned computing infrastructure. Moreover, disaster recovery facilities and raw data backup are demanded to the Cloud service providers, rendering the smart city infrastructure more robust and further freeing municipality resources for actual service development. Cloud Bursting and data replication further extends the ability to keep the smart city services up when the local municipality infrastructure is under strain, running out of available resources or struck by disaster, making it possible for the smart city infrastructure to rapidly offload to public Cloud services.

Rapid Development via PaaS. Thanks to the Platform as a Service components included in ClouT's architecture, city application developers can easily customize their development environment. All the common operative systems, application servers and middleware software used to develop Cloud-enabled applications are just a few clicks away, just as replicating environment instances for development, testing and production deployment. This makes for an easier and more efficient development of smart city services.

3. IOT KERNEL

The IoT Kernel main block is composed of three main sub-blocks:

- *IoT Device Wrapping*, that is in charge of implementing the low level connectivity and specific profile details in order to interact with different devices and different physical media;

- *IoT Device Management*, that is in charge of the task devoted to device identification, registration, naming and management;
- *Uniform Access to IoT Devices*, that implements a generic and unique abstraction API to expose all the underlying devices to the upper layers.

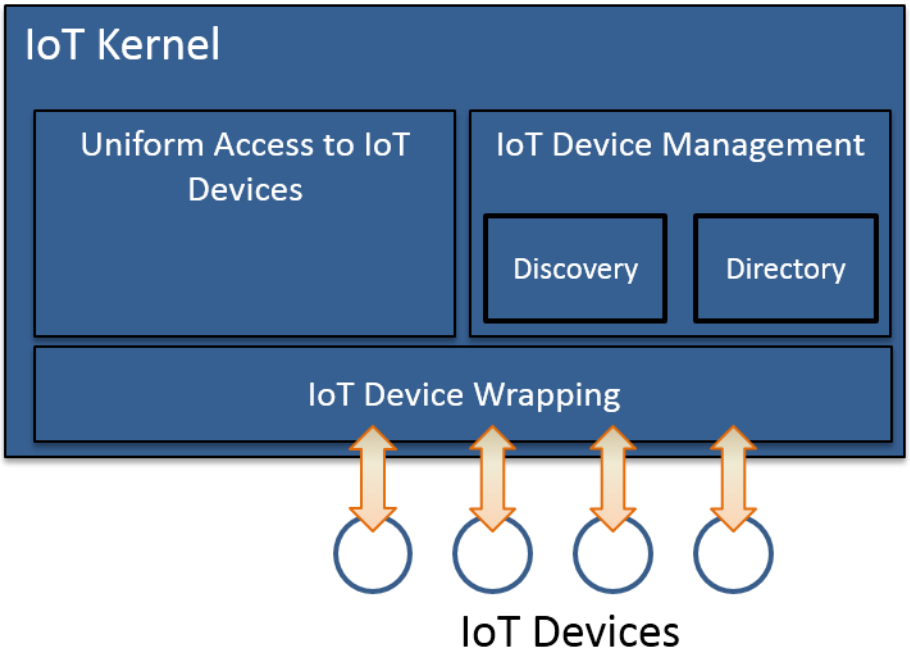

FIGURE 3 : IOT KERNEL COMPONENTS

3.1. IoT Device Wrapping

This is the lowest layer in the IoT Kernel, and is in charge of accessing the different IoT Devices by different communication media and using different application profiles. In this block resides all the specific knowledge for a given protocol (i.e. an instance for IPSO Smart Objects will exist, as well as another one for Bluetooth LE, …).

3.1.1. Architecture

In Figure 4 we report a more detailed picture of the structure of the Protocol Adapters that build the IoT Device Wrapping layer. They are logically split in two, the lower part (link handler) being intended to deal with the actual physical medium used to communicate and the upper one (protocol handler) that deals with the actual transport/application protocol termination and handling of the supported profile/data format. The protocol handler is also in charge of creating a virtual representation of the supported (and discovered) devices, in the internal format used by the IoT Gateway.

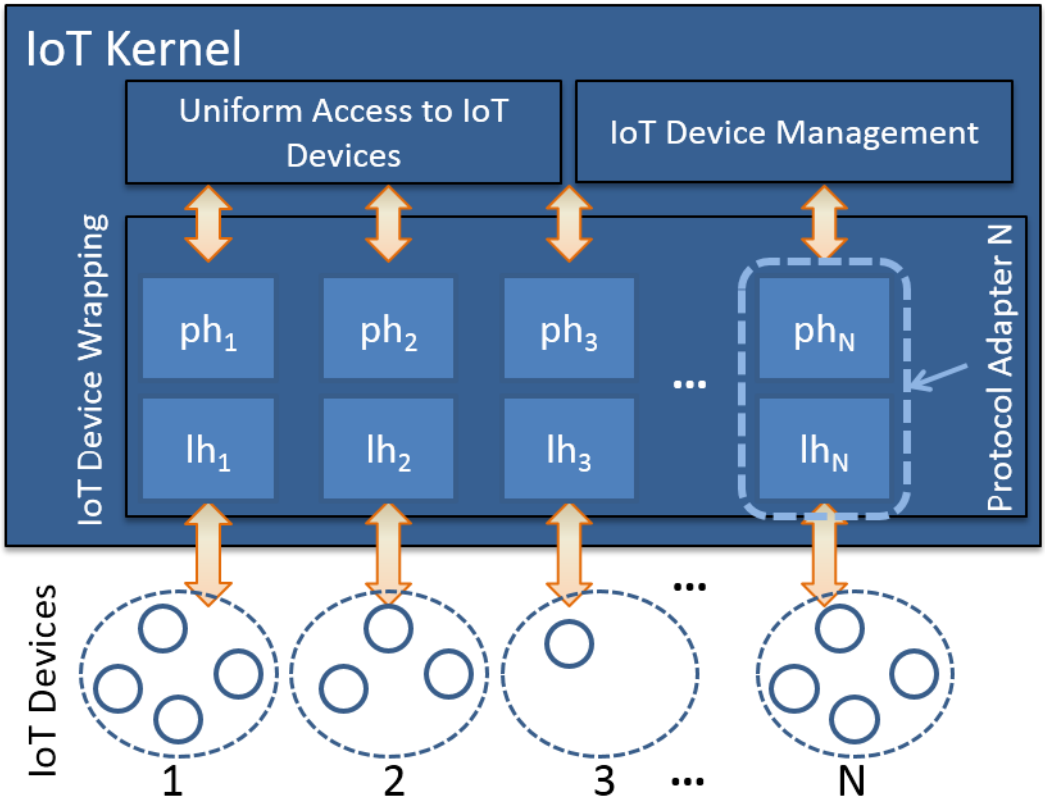

FIGURE 4 : IOT DEVICE WRAPPING ARCHITECTURE

3.1.2. Specification

In Figure 5 we report a communication diagram about the main interactions of the IoT Device Wrapping block with the other blocks of the ClouT IoT Kernel.

In Table 5 the main generic functions of this block are reported, too: we remand for the details to the Implementation section.

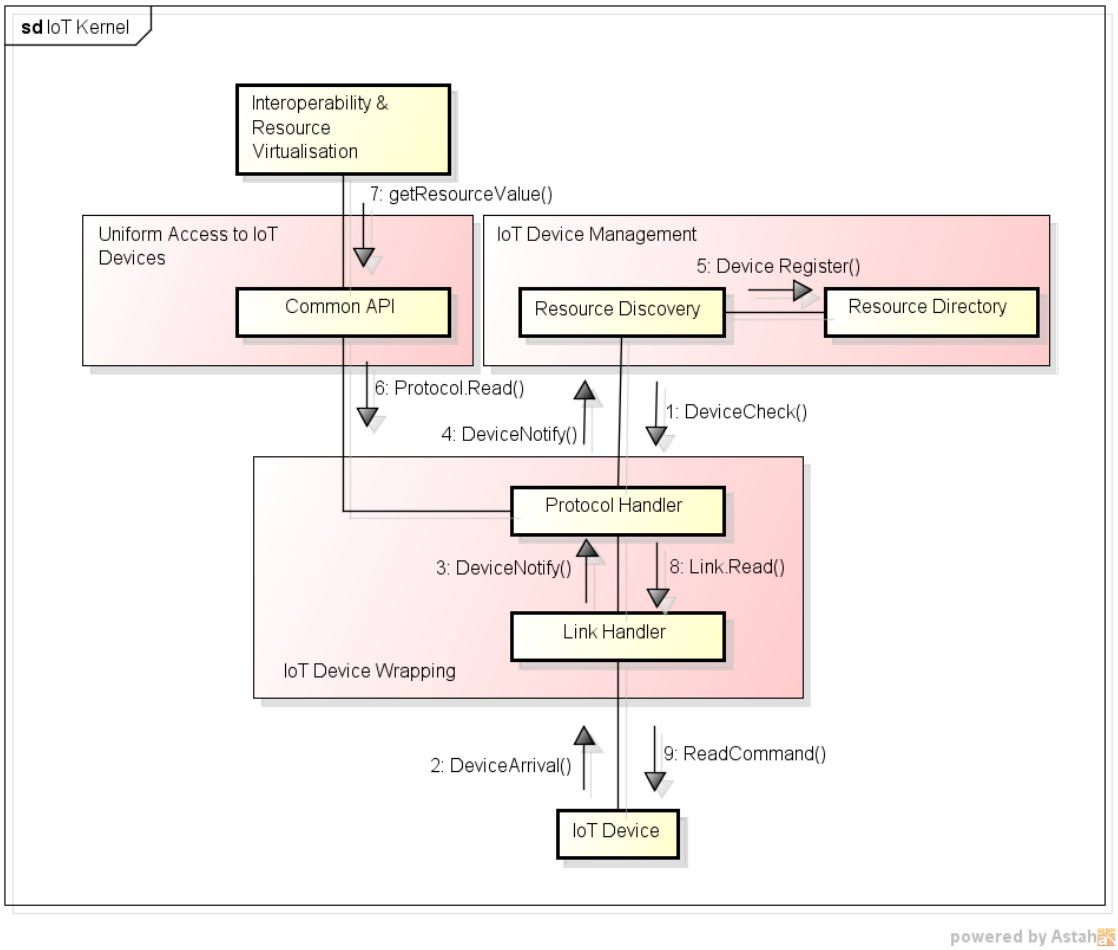

FIGURE 5 : IOT KERNEL INTERACTIONS

TABLE 5 : IOT DEVICE WRAPPING

Modifier and Type	Method and Description
Void	**DeviceCheck()** Polls the Device Wrapping layer about the presence of a new device. This mechanism can be implemented differently with regards to the different protocol, profiles and standards that are used by the IoT devices.

Modifier and Type	Method and Description
Void	**DeviceArrival(<Dev_ID>)** Notification, specific to a given protocol/profile, of the presence of a new Device. Dev_ID is a unique identifier for the device, i.e. the IPv6 address of a Device in a 6LoWPAN WSN.
Void	**Protocol.ACT()** Where ACT can be Create, Read, Update, Delete on a specific resource hosted on the final IoT Device. This generic API will be translated in the proper command by the Protocol Handler (i.e. CoAP GET) and forwarded to the IoT Device by the Link Handler (i.e. by encapsulating the CoAP command in a 802.15.4 packet).

Relevant system requirements, as from [D1.3]: REQ_CIAAS_19, REQ_CIAAS_22, REQ_CIAAS_27, REQ_CIAAS_28, REQ_CIAAS_29.

3.1.3. Implementation

The implementation of the blocks that compose the IoT Kernel is reported at the end of this chapter because we have several different solutions that covers different functionalities, so it is clearer to describe them one by one, and highlighting the roles and mapping of each blocks with regards to ClouT reference architecture tasks.

3.2. IoT Device Management

In order to manage the IoT devices, either physical or virtual, it is needed firstly to identify the nodes with a unique identifier for addressing them in a univocal way, secondly to store the information, such as capabilities or location, associated to the node, and finally to manage and handle the different events associated to these devices, such as installation of a new node, addition of a new capability in an existing node, node removal. In the next sections, they are going to be addressed the three aforementioned issues.

3.2.1. Architecture

In Figure 6 the overall architecture of the IoT Kernel block is reported, showing in particular the links between the IoT Device Management block and the upper layers, and the two main functions that belong to this block: Device Discovery and Device Directory.

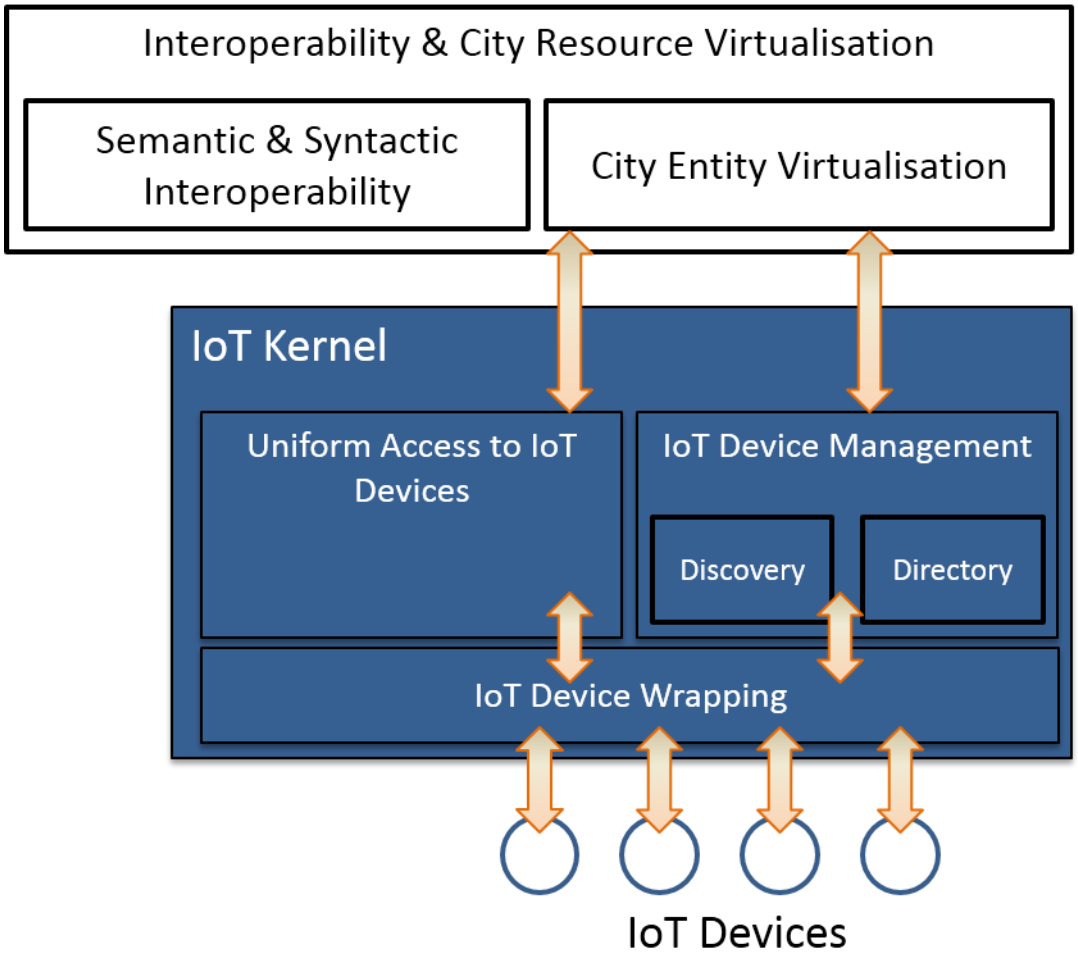

FIGURE 6 : ACCESS TO IOT KERNEL FROM OTHER LAYERS

These two blocks are in charge of detecting the presence of a new device (this process can be initiated by the IoT device or by the IoT Gateway) and of storing the related information in a repository called « Device Directory » that keeps an up to date track of the IoT Devices along with their exposed resources.

3.2.2. Specification

In Figure 7 we report the main interactions related to the IoT Device Management.

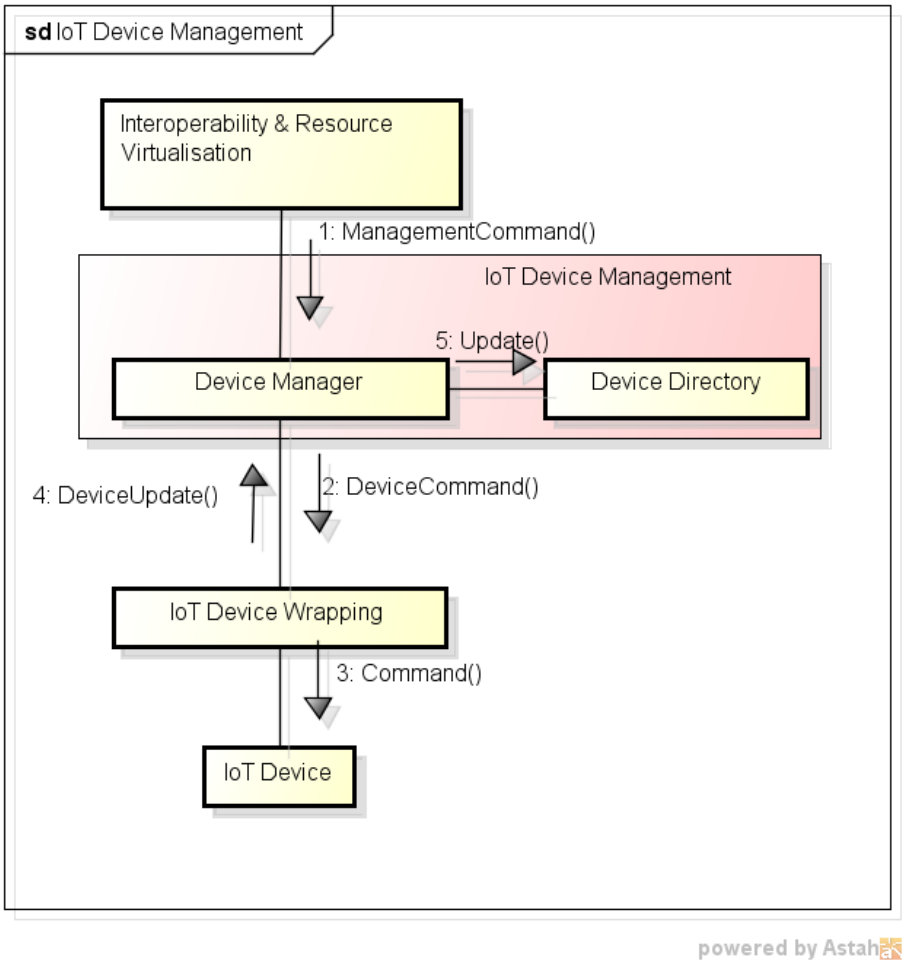

FIGURE 7 : IOT DEVICE MANAGEMENT COMMAND FLOW

Other functionalities that are covered by the IoT Device Management function are

- Device naming: In order to identify the nodes, either real or virtual, physically installed or sensorised/actuatorised, it is needed to define a procedure to assign them a unique identifier, which allows define it.
- Device Description: in order to have identified and stored the different characteristics (measurement capabilities, location, …) associated to all the devices, it is needed to determine a common structure for the description of these devices in order to uniform and homogenize, both searching and interpretation of the description of the different devices.

TABLE 6 : IOT DEVICE MANAGEMENT

Modifier and Type	Method and Description
Status	**DeviceReset** (String name) If applicable, resets the device identified by its name.
Status	**DeviceReprogram** (String name, File firmware) Updates the firmware on a device, identified by its name
List<**Objects**>	**DeviceGetInfo** (String name, object command) Gets some per-device class specific information, for instance on the available programs flashed in the memory or on specific management parameters
Status	**DeviceSetInfo** (String name, object command) Sets some per-device class specific management parameters
Status	**DeviceUpdate** (Object device) Receives a notification of the update of the status of a pre-registered device. This function can trigger the modification/deregistration of the device in/from the Device Directory module.

Relevant system requirements, as from [D1.3]: REQ_CIAAS_22, REQ_CIAAS_23, REQ_CIAAS_4.

3.2.3. Implementation

The implementation of the blocks that compose the IoT Kernel is reported at the end of this chapter because we have several different solutions that cover different functionalities, so it is clearer to describe them one by one, highlighting the roles and mapping of each blocks with regards to ClouT reference architecture tasks.

3.3. Uniform access to IoT Devices

This block is the interface of the internal device representation with the upper layers of the GW stack or with the northbound adapters used to remotely access the gateway.

3.3.1. Architecture

For a general view on the architecture and interactions of this block with the other that compose the IoT Kernel, the reader can refer to Figure 6 .

3.3.2. Specification

The Uniform Access to IoT Devices block implements the common API used by the layers above ClouT IoT Kernel to access the devices. An example of the *getResourceValue()* function, mapped to the a "Read" of a specific resource, has already been illustrated in Figure 5.

In Table 7 the generic common APIs are reported.

TABLE 7 : UNIFORM ACCESS TO IOT DEVICES

Modifier and Type	Method and Description
Data	**getResourceValue** (String resource) Gets the current data value of the given resource.
Status	**setResourceValue** (String resource, Object value) Sets the given data value to the given resource.
Status	**actResourceProperty** (String resource, [Set<parameter>]) Actuates on a resource that can be invoked with optional parameters.
Subscription	**subscribeResourceValue** (String resource, [Object rule]) Asks to be notified of changes of the data value of a given resource. An optional parameter can specify additional constraint in terms of periodicity of the reporting or threshold of the data value.
Void	**unSubscribeResourceValue** (Subscription subscription) Cancel a notification request that was previously initiated.

Relevant system requirements, as from [D1.3]: REQ_CIAAS_11, REQ_CIAAS_14, REQ_CIAAS_22, REQ_CIAAS_23, REQ_CIAAS_25, REQ_CIAAS_26, REQ_CIAAS_4.

3.3.3. Implementation

The implementation of the blocks that compose the IoT Kernel is reported at the end of this chapter because we have several different solutions that covers different functionalities, so it is clearer to describe them one by one, and highlighting the roles and mapping of each blocks with regards to ClouT reference architecture tasks.

3.4. IoT Kernel implementations

In this section we will illustrate the main solutions that we use to implement the IoT Kernel of the ClouT project.

The relevant mapping of the used components to the ClouT reference architecture modules is highlighted in each section.

3.4.1. CEA Gateway Solution

The OSGi Gateway developed by CEA, called sensiNact, can be used as a basis to build the IoT Gateway for the ClouT project, as it already implements the basic blocks for connectivity, service abstraction, device management, virtualization and remote access.

The sensiNact Gateway allows interconnection of different networks to achieve access and communication with embedded devices. It is composed of five **functional groups** and their relative interfaces:

- The **Device Protocol Adapter** abstracts the specific connectivity technology of wireless sensor networks. It is composed of the bridges associated to protocol stacks. All the bridges comply with a generic Device Access API used to interact with northbound sensiNact's services.
- The **Smart Object Access and Control** implements the core functionalities of sensiNact like discovering devices and resources and, securing communication among devices and consumers of their services.
- The **Consumer API** is protocol agnostic and exposes services of the Smart Object Access and Control functional to Consumers.
- The **Consumer Protocol Adapter** consists of a set of protocol bridges, translating the **Consumer API** interface into specific application protocols.
- The **Gateway Management** functional group includes all the components needed to ease management of devices connected to sensiNact, regardless of their underlying technologies. A **Device Management API** is used for this purpose. This functional group also contains the components managing cache, resource directory and security services. These management features are exposed by means of the **Gateway Management API**.
- And finally the **Manager Protocol Adapter** allows adapting the Gateway Management API to the specific protocols used by different external management entities.

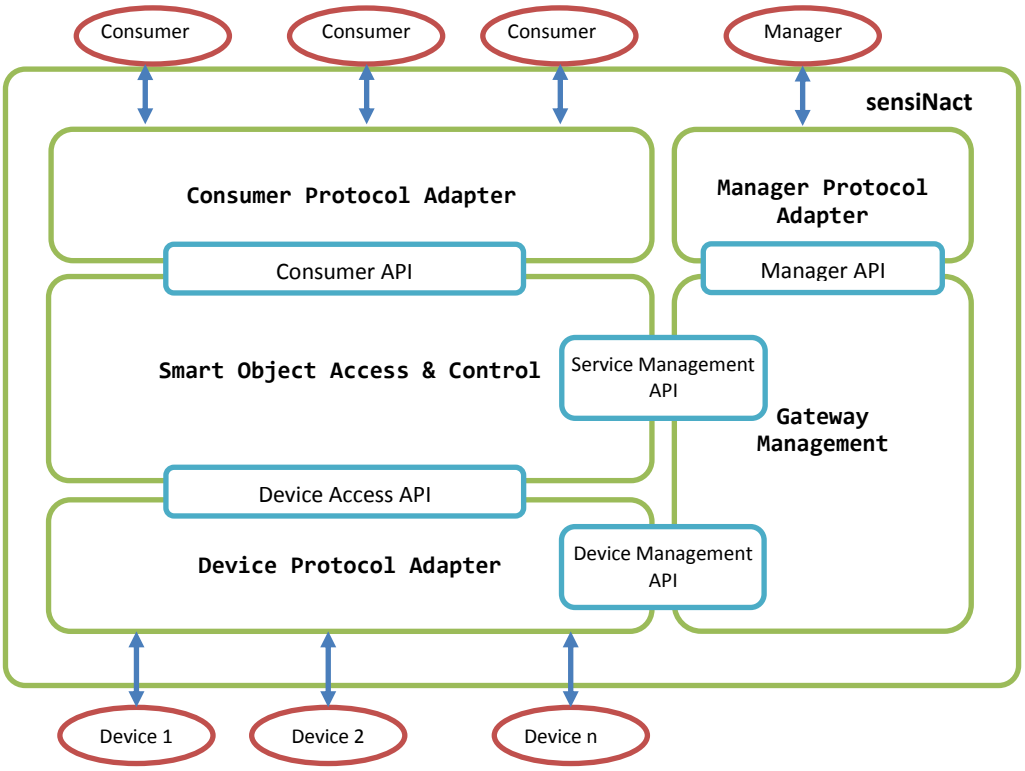

FIGURE 8: SENSINACT GATEWAY FUNCTIONAL COMPONENTS

In terms of connectivity:

On the southbound side the sensiNact gateway allows to cope both with “physical device” protocols and “virtual device” ones, allowing a uniform and transparent access to an XBee network, or an HTTP Restful web service for example. Here's pell-mell a non-exhaustive list of supported protocols:

- **EnOcean**, concerting energy harvesting wireless sensor technology (ultra-low-power radio technology for free wireless sensors), and protocols in use to interact with those sensors;
- **Bluetooth Low Energy** , which is a personal area network, low power protocol designed mainly for healthcare or entertainment type of applications;
- **MQTT**, which is a machine-to-machine protocol, lightweight publish/subscribe messaging transport, useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium;
- **ZigBee based** protocols (**XBee** for example);
- **CoAP** which is REST application protocol, designed to be *“the HTTP for constrained networks and devices”* whose concept originated from the idea that "the Internet Protocol could and should be applied even to the smallest devices," and that low-power devices with limited processing capabilities should be able to participate in the Internet of Things; it is usually used on top of a 6LoWPAN network, but it may travel regular IP networks as well (it is used by the OMA LWM2M protocol, for instance);

- **MQTT**, which is a machine-to-machine protocol, designed as an extremely lightweight publish/subscribe messaging transport, useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium;

On the northbound side the sensiNact gateway provides both client/server and publish/subscribe access protocols:

- **MQTT**;
- **JSON-RPC** (1.0 and 2.0);
- **HTTP Restful**

Moreover a **CDMI** bridge can be configured to feed a CDMI Cloud Storage as it is the case in the ClouT platform

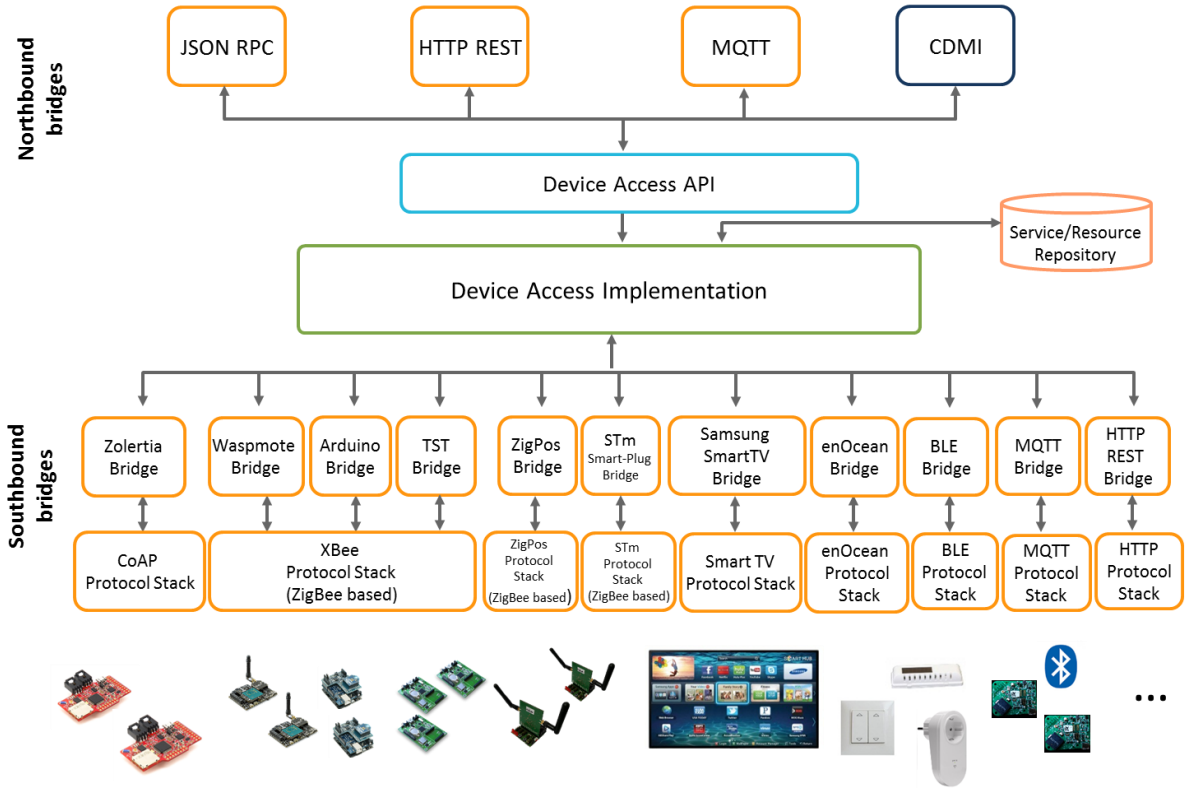

FIGURE 9: SENSINACT GATEWAY BRIDGES

To be able to use the sensiNact gateway in the ClouT platform some northbound and southbound bridges have to be finalized, allowing a complete integration with the different components bring by all partners; particularly, XMPP and OMA/NGSI (Orion) ones.

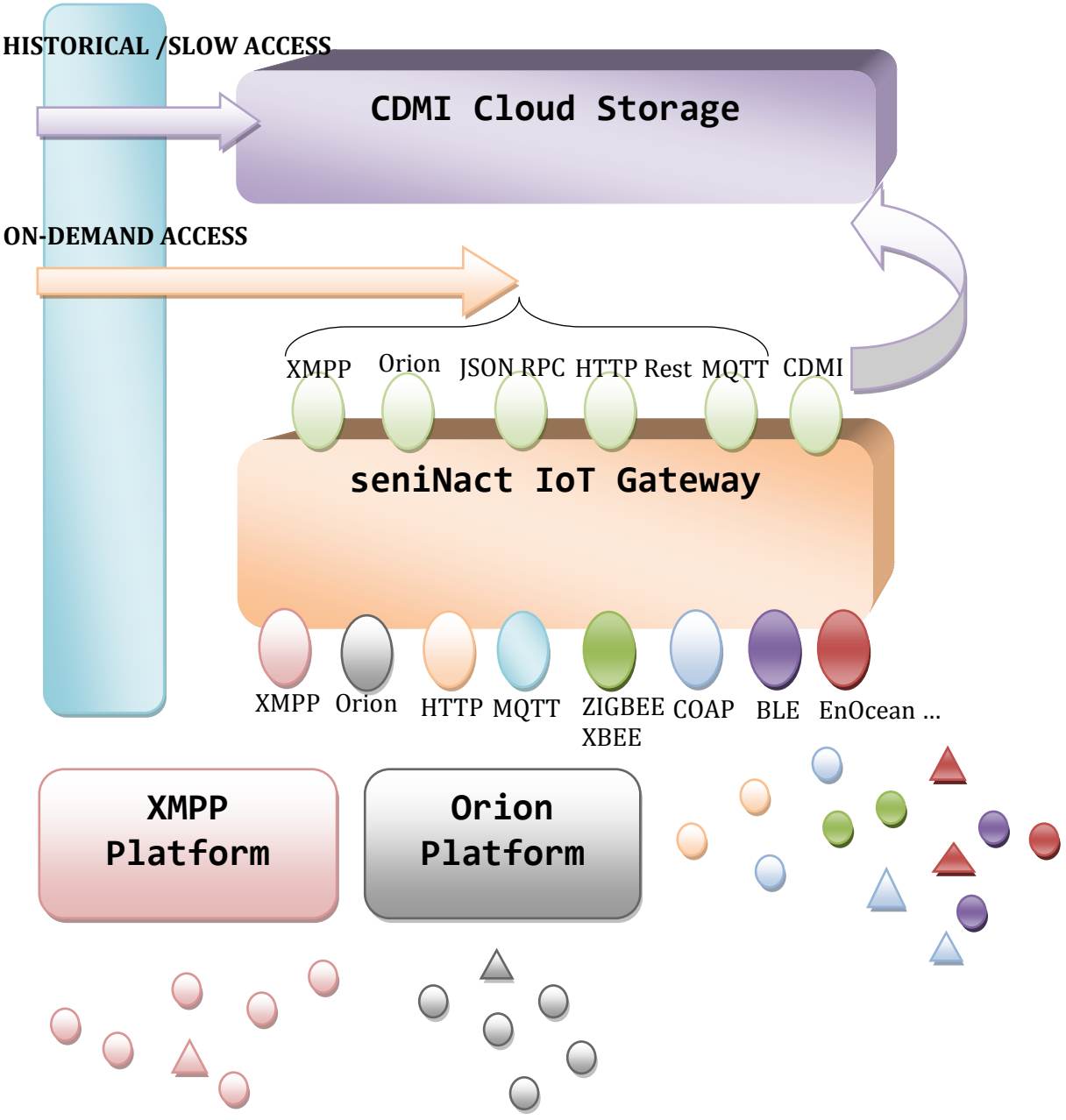

FIGURE 10: SENSINACT GATEWAY SOLUTION IN THE CLOUT PLATFORM

The **Smart Object Access and Control** functional group described above in this section includes a large number of functionalities:

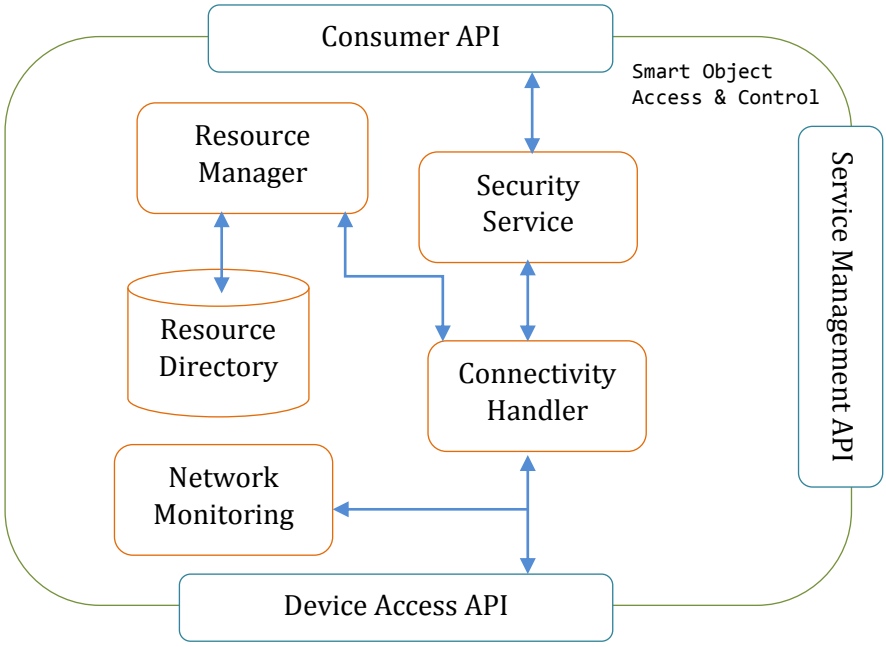

FIGURE 11: SMART OBJECT ACCESS AND CONTROL FUNCTIONAL GROUP

- It handles the communication with the Consumer Protocol Adapter (REST API, JSON RPC, etc.) and IoT (and non-IoT) devices, providing URI mapping, incoming data/messages translation in an internal format and outgoing data/messages translation in Consumer format.

  Whenever a Consumer tries to access a resource via Consumer API, the requested URI is forwarded to the Resource Manager in order to check if a specific resource descriptor exists or not inside the Resource Directory and to verify its accessibility status. If a resource descriptor doesn't exist, a message response with error code is returned to the Consumer API. Otherwise, the request is forwarded to the right interface. At the same time whenever response is originated form IoT device (or abstract IoT device), it will be also forwarded to its logical counterpart in order to update the resource representation in the gateway.

- It manages the subscription/notification phases towards the Consumer, if it is not handled by the targeted device (service) itself

- It supports Devices and Resource Discovery and Resource Management capabilities, to keep track of IoT Resource descriptions that reflect those resources that are reachable via the gateway. These can be both IoT Resources, or resources hosted by legacy devices that are exposed as abstracted IoT Resources. Moreover, resources can be hosted on the gateway itself. The Resource Management functionality enables to publish resources in sensiNact, and also for the Consumer to discover what resources are actually available

from the gateway; **sensiNact Service and Resource model** allows exposing the resources provided by an individual service. The latter, characterized by a service identifier, represents a concrete physical device or a logical entity not directly bound to any device. Each service exposes resources and could use resources provided by other services. Figure 12 below depicts the Service and Resource model:

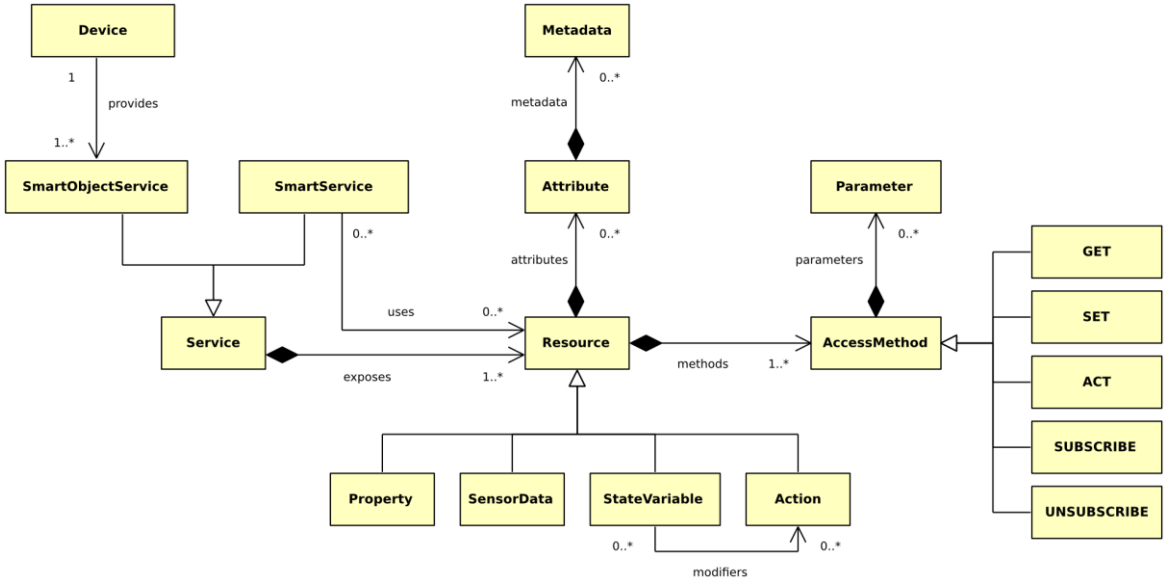

FIGURE 12: SERVICE AND RESOURCE MODEL

The **Resource Directory** allows storing information, i.e. resource descriptions, about the resources provided by individual devices connected to sensiNact. It also supports looking up resource descriptions, as well as publishing, updating and removing resource descriptions to it.

Discovering and using resources exposed by Services is a favored approach for avoiding using static service interfaces and then increase interoperability. Therefore, sensiNact Services and their exposed resources are registered into Service/Resource repositories. The gateway uses the OSGi service registry as Service/Resource repository, where resources are registered as service properties. Clients ask the Service/Resource repository for resources fulfilling a set of specified properties (defined by LDAP filters). In response, the Service/Resource repository sends clients the list of service references that expose the requested and authorized resources. Clients can then access/manipulate the resources exposed by their selected service objects.

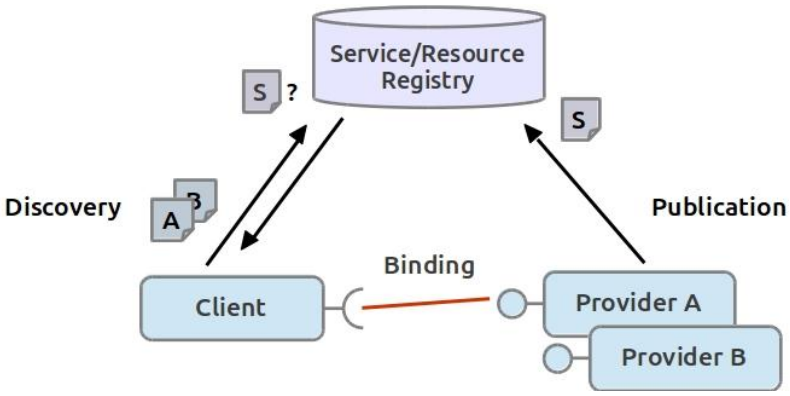

FIGURE 13: SERVICE AND RESOURCE REGISTRATION

Resources and services can be exposed for remote discovery and access using different communication protocols, such as HTTP REST, JSON-RPC, etc., and advanced features may also be supported (as semantic-based lookup).

Resources are classified in the following types:

TABLE 8 : RESOURCES TYPES

Type	Description
SensorData	Sensory data provided by a service. This is real-time information provided, for example, by the SmartObject that measures physical quantities.
Action	Functionality provided by a service. This is mostly an actuation on the physical environment via an actuator SmartObject supporting this functionality (turn on light, open door, etc.) but can also be a request to do a virtual action (play a multimedia on a TV, make a parking space reservation, etc.)
StateVariable	Information representing a SmartObject state variable of the service. This variable is most likely to be modified by an action (turn on light modifies the light state, opening door changes the door state, etc.) but also to intrinsic conditions associated to the working procedure of the service
Property	Property exposed by a service. This is information which is likely to be static (owner, model, vendor, static location, etc.). In some cases, this property can be allowed to be modified.

Resources have attributes that represent intrinsic information about resources. An attribute is characterized by a name, a type, a value and a set of metadata.

Access methods are classified in the following types:

TABLE 9 : RESOURCE'S ACCESS METHODS

Type	Description
GET	Get the value attribute of the resource
SET	Sets a given new value as the data value of the resource
ACT	Invokes the resource (method execution) with a set of defined parameters
SUBSCRIBE	Subscribes to the resource with optional condition and periodicity
UNSUBSCRIBE	Remove an existing subscription

The access methods that can be associated to a resource depend on the resource type, for example, a GET method can only be associated to resources of type Property, StateVariable and SensorData. A SET method can only be associated to StateVariable and modifiable Property resources. An ACT method can only be associated to Action resources. SUBSCRIBE and UNSUBSCRIBE methods can be associated to any resource type.

3.4.2. IPSO Smart Objects for STM32 ODE

IPSO is a technology alliance which promotes the use of open standards and IP based communication in networks composed by constrained devices.

ST is member of the IPSO Alliance and active in the standardization of the Smart Objects as a common standard to describe devices and their resources, helping in the development of this standard and working on its preliminary implementation, on hardware prototyping platforms produced by ST. The main focus in term of connectivity is the 802.15.4 sub GHz radio standard.

Compared to what has been described in D2.2, we are now adopting in the context of the ClouT project the new specification proposed by IPSO, called IPSO Smart Objects.

This addresses a more structured and extensible profile, based on a set of XML files describing the devices along with their resources, sensors and actuators.

The low level mechanism to access the WSN is the same as per the previous specification and preliminary implementation (D2.2), based on the use of a RPL Border Router firmware running on a dongle-form factor device attached to a USB port of the IoT Gateway.

In this document we will focus on the differences brought by the IPSO Smart Objects specification itself: this is based on the OMA LWM2M standard that provides the relevant API and bindings for the communication between nodes and a Server.

In Figure 14 we report a mapping of the components that compose the solution based on IPSO Smart Objects with regards to the ClouT reference architecture.

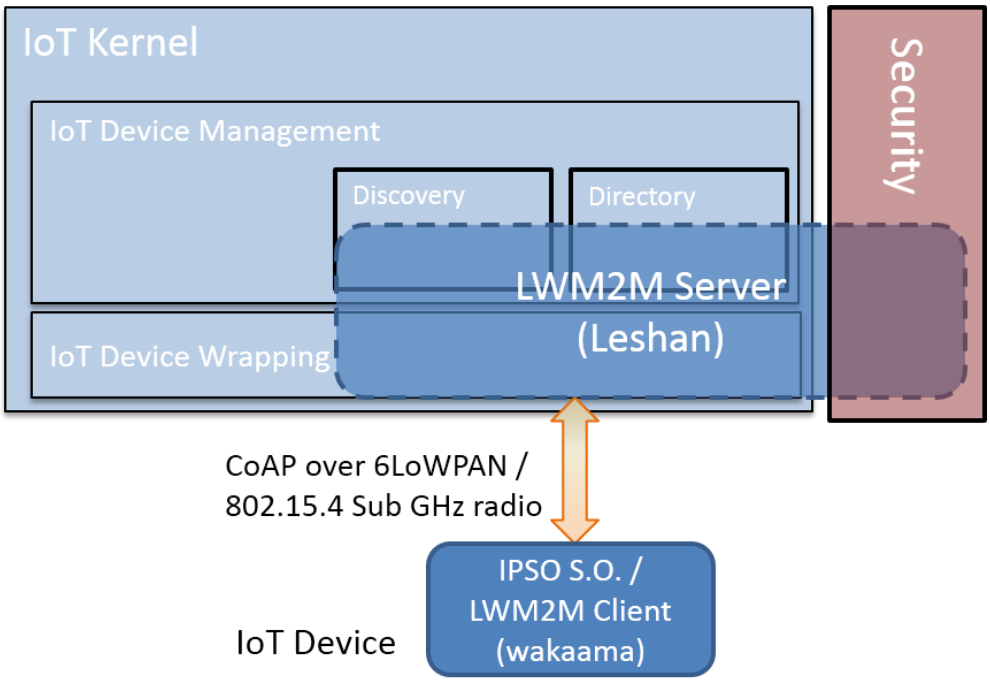

FIGURE 14 : IPSO SMART OBJECTS SOLUTION AND CLOUT REFERENCE ARCHITECTURE

The implementation proposed by ST is currently based on:

- A tool that can be used to automatically generate, starting from proper XML descriptions, stubs of C code for the IoT Devices:
    - this code will use and extend the *wakaama* open source implementation of OMA LWM2M Client
    - *wakaama* is, in turn, based on the *Erbium* CoAP C framework (integrated in Contiki)
- A server side solution that extends the *Leshan* open source implementation of OMA LWM2M Server:
    - *Leshan* is, in turn, based on the *Californium* CoAP Java framework

The LWM2M server can be hosted on the local IoT Gateway or can be remotely deployed in the cloud.

As it can be inferred by the name, OMA LWM2M is intended to be a lightweight specification that describes only the minimum set of requirements for the nodes communication and management. It should also be light enough to be run on devices with minimum amount of resources. The main application protocol that OMA LWM2M is intended to rely on is CoAP, even if a binding to HTTP is also possible but it won't be considered here.

In Figure 15 we report the overall stack of protocols involved in a full implementation of devices with resources specified accordingly to IPSO Objects.

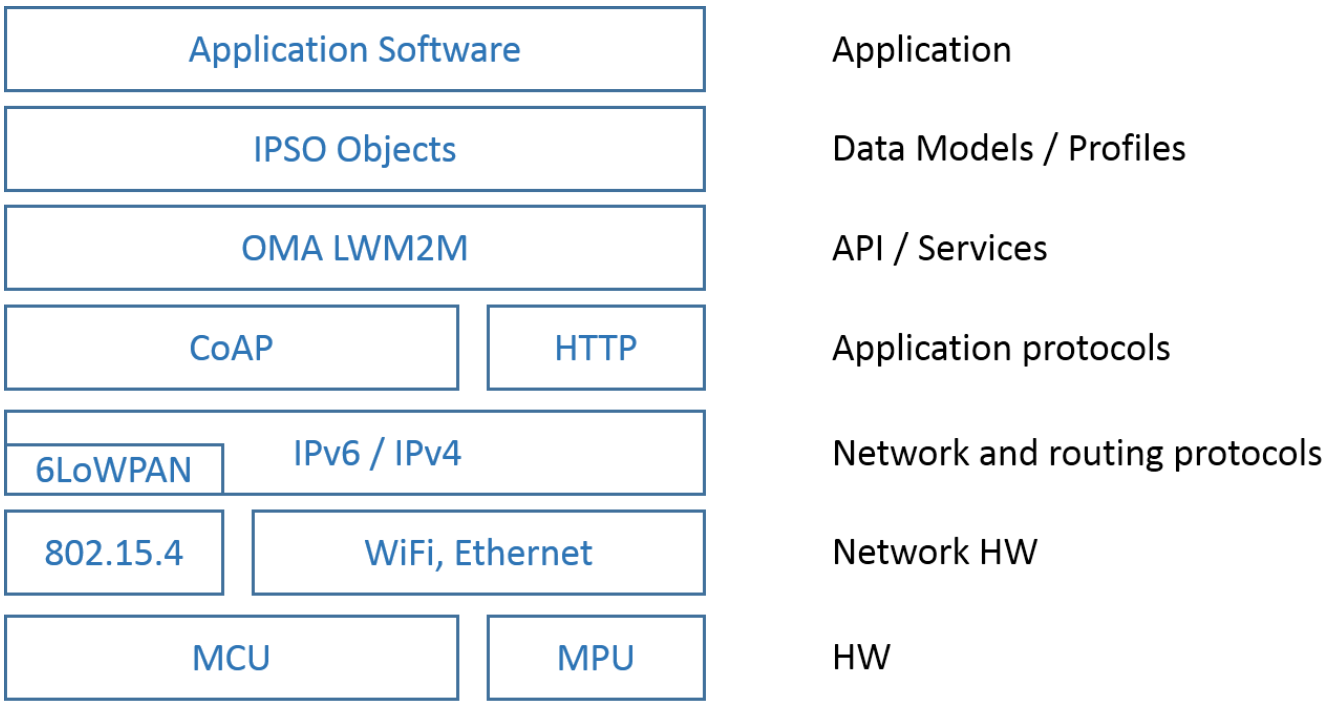

FIGURE 15 : PROTOCOL STACK FOR AN IPSO SMART OBJECT BASED DEVICE

An IoT Device compliant with the IPSO Objects specification is implemented as a OMA LWM2M Client, i.e. it follows the Object Model defined in the latter specification, organized in Resources and Objects.

A LWM2M Client may have any number of Resources, each of which belongs to an Object. Resources and Objects have the capability to have multiple instances of the Resource or Object.

An overview of the OMA LWM2M standard, along with details on how the IPSO Smart Objects are mapped on top of it, can be found in Appendix A – OMA LWM2M.

As a result of the generic API of OMA LWM2M specification (that maps on top of CoAP requests), and of the choice of the IPSO Smart Objects formalism to represent the resources on the IoT device, in the following table we report the Request and Response messages used to read the Sensor Value Resource (/5700) of a specific instance (/1) of a IPSO Temperature Object (/3003). The Response carries the 5.7 °C value encoded in the proper TLV format.

It is assumed that the device has already registered with the LWM2M Server and the LWM2M Server is authorized to read sensor data.

TABLE 10 : REQUEST/RESPONSE EXAMPLE FOR A IPSO TEMPERATURE SMART OBJECT

Request:	CoAP GET /3303/1/5700/ Token: 0x7 *Read the value of resource 5700 (Sensor Value) instance #1 of object 3303 (IPSO Temperature)*

<table>
<tr><td>Response:</td><td>CoAP ACK
Token: 0x7
Payload:
<table>
<tr><th colspan="2">Type Byte</th><th>Identifier</th><th>Value</th></tr>
<tr><td colspan="2">0b11 1 00 100</td><td rowspan="5">5700
(16 bit integer in network byte order)</td><td rowspan="5">5.7
(32 bit float as per IEEE 754-2008)</td></tr>
<tr><td>Type</td><td>0b11 (Resource with value)</td></tr>
<tr><td>ID length</td><td>0b1 (16 bits)</td></tr>
<tr><td>Length field</td><td>0b00 (no length field, the next 3 bits indicate value length)</td></tr>
<tr><td>Length</td><td>0b100 (4 bytes)</td></tr>
</table></td></tr>
</table>

The solution based on IPSO Smart Objects running on STM32 based platforms will be integrated in the ClouT IoT Gateway based on the sensiNact solution developed by CEA.

3.4.3. SmartSantander IoT Kernel Implementation

Within this section, they are presented the different information management procedures as well as the operating modules inherited from SmartSantander, for naming, description, management and sending of information. Regarding to naming it would be presented an approach intended to assign a unique identifier to all the resources within the CLouT project. In order to have unified information of all the available resources, it is described a structure for describing the capabilities and characteristics associated to each of these resources. In this sense, for maintaining this information updated, they are presented different modules, methods and interfaces addressing resource management. Finally, it is indicated the way in which IoT kernel transform information coming from subjacent resources into a common data format for easing the different interaction process with them.

3.4.3.1. SmartSantander IoT device naming

Derived from the experience in the installation carried out in the SmartSantander project, and also attending to some of the considerations indicated in RFC2141 [RFC-2141] and RFC3406 [RFC-3406], it has been defined a set of rules for building the corresponding identifier.

First of all, regarding to the naming and structure some restrictions should be considered:

- **Valid characters**: ASCII table and characters ".", "-" and "_". Character ":" used as levels separator.
- **No case sensitive**. Recommended to use URN in lower-case characters.
- **Length**: In principle bounded to 255 characters.
- URN will be composed of four parts: prefix, domain, provider_index and local_id.

- **Prefix**: It will be a common identifier for all the resources. For this purpose "x-iot" has been selected as prefix, thus indicating an IoT node used in an experimental way.
- **Domain**: Associated to each of the different instances of the project. In this sense, domain could make reference to the project and the different cities/partners within the consortium. Something similar to this "clout:fujisawa", when referring to a resource located in Fujisawa.

- **Provider/Manufacturer/Owner index**: This could be related with the company that provides the service, the manufacturer of the devices, … In this sense, it will be assigned a random alphanumeric or a sequential numeric value to each of the providers.
- **Local ID**: proprietary ID established by the owner. Its format and content is free, but respecting the restrictions previously defined.

Some examples of this naming are indicated next:

```
"urn:x-iot:clout:genova:39af2t:camera23
```

```
"urn:x-iot:clout:cea:000021:temp32"
```

As you can observe, the first example defines a node installed in the city of Genova, indicating the corresponding manufacturer as a random alphanumeric identifier and local id would be camera23, as the proprietary id used by the corresponding manufacturer. In the second example, the node would be owned by CEA with a manufacturer associated to a sequential identifier and the corresponding local_id defining a temperature sensor.

3.4.3.2. SmartSantander IoT device description

Apart from the unique identifier associated to the node, it is also needed to store the information related to the node description, indicating the following issues:

- Type of node: Nodes could be classified attending to their mobility: fixed, mobile, semi mobile, as well as to its abstraction status: real, sensorised or virtual. In this sense, semi mobile nodes could be defined as nodes that are taking measurements during a specific amount of time in a fixed location, but that can be moved to another location to keep on taking measurements at this new fixed position during a specific period of time.
- Location: GPS coordinates of the device (only valid for fixed and semi mobile nodes)
- Short description: It includes the main characteristics of the device in terms of sensing capabilities, processing and memory capacity, estimated battery lifetime, name of the manufacturer, communication interfaces and others.
- Status (up, down, …): It indicates the operational status of the node, indicating if it is working properly, being replaced, turned off or other specific status that could be defined and added.
- Capabilities: Number of measuring attributes associated to a determined node. For each of these capabilities, they will be indicated the following parameters:
    - Id: Identifier of the capability according to a specific dictionary that assures the univocal identification of a determined capability, independent from the device, location where it is being measured (i.e. temperature as the common identifier for ambient temperature measured by a node)
    - Phenomenon: This field describes the specific phenomenon that is specifically measured.
    - Type: Unique urn describing the type of data.

- Unit of Measurement: It indicates the unit used for storing the measured data.
- Data: It describes the data format (integer, string, char, float, …), the data will be represented.

- Additional specific properties (can receive remote commands, can be remotely updated, …): This field indicates the additional functionalities associated to the node, in terms of remotely sending/receiving commands and updating the firmware.

In the next table, it is included an example of the generic structure of the resource description:

```
module.exports = {
  attributes: {
    _id: {
      type: String,
      lowercase: true,
      trim: true
    },
    created_at: Date,
    updated_at: Date,
    type: {
      type: String,
      lowercase: true,
      enum: ['testbed_server', 'gateway', 'fixed_node', 'mobile_node'],
    },
    parent_id: {
      type: String,
      trim: true
    },
    childs: {
      type: Number,
      default: 0
    },
    status: {
      type: String,
      lowercase: true,
      enum: ['enabled', 'disabled'],
      required: true
```

```
},
location: {
    index: '2dsphere',
    type: [Number]
},
description: {
    /**
     * Human readable description of the resource
     */
    short_desc: {
        type: String,
        trim: true
    },
    /**
     * Wheter or not the resource allows native experimentation
     */
    allow_native_exp: {
        type: Boolean,
        default: false
    },
    /**
     * Wheter or not the resource allows transmission/reception of commands
     */
    allow_remote_cmds: {
        type: Boolean,
        default: false
    },
    /**
     * This field is optional and will only probably exist in 'fixed_node' or
     * 'mobile_node' resources
     */
    capabilities: [{
        _id: false,
        phenomenon: String,
```

```
        uom: String,
        type: String,
        data: Boolean,
    }],
}
```

For each of the registered devices, all the aforementioned information will be generated and stored within the resource directory, either when a new device is installed or created (virtual device), a modification in an already registered device is carried out (including/removing new capabilities, changing its status) or removing the description of a stored node when this is uninstalled or does not work properly.

3.4.3.3. SmartSantander IoT resource manager

In order to update the IoT resource description, addition, removing and change in the resource characteristics should be dynamically managed. SmartSantander resource manager module, shown in next figure, supports these functionalities.

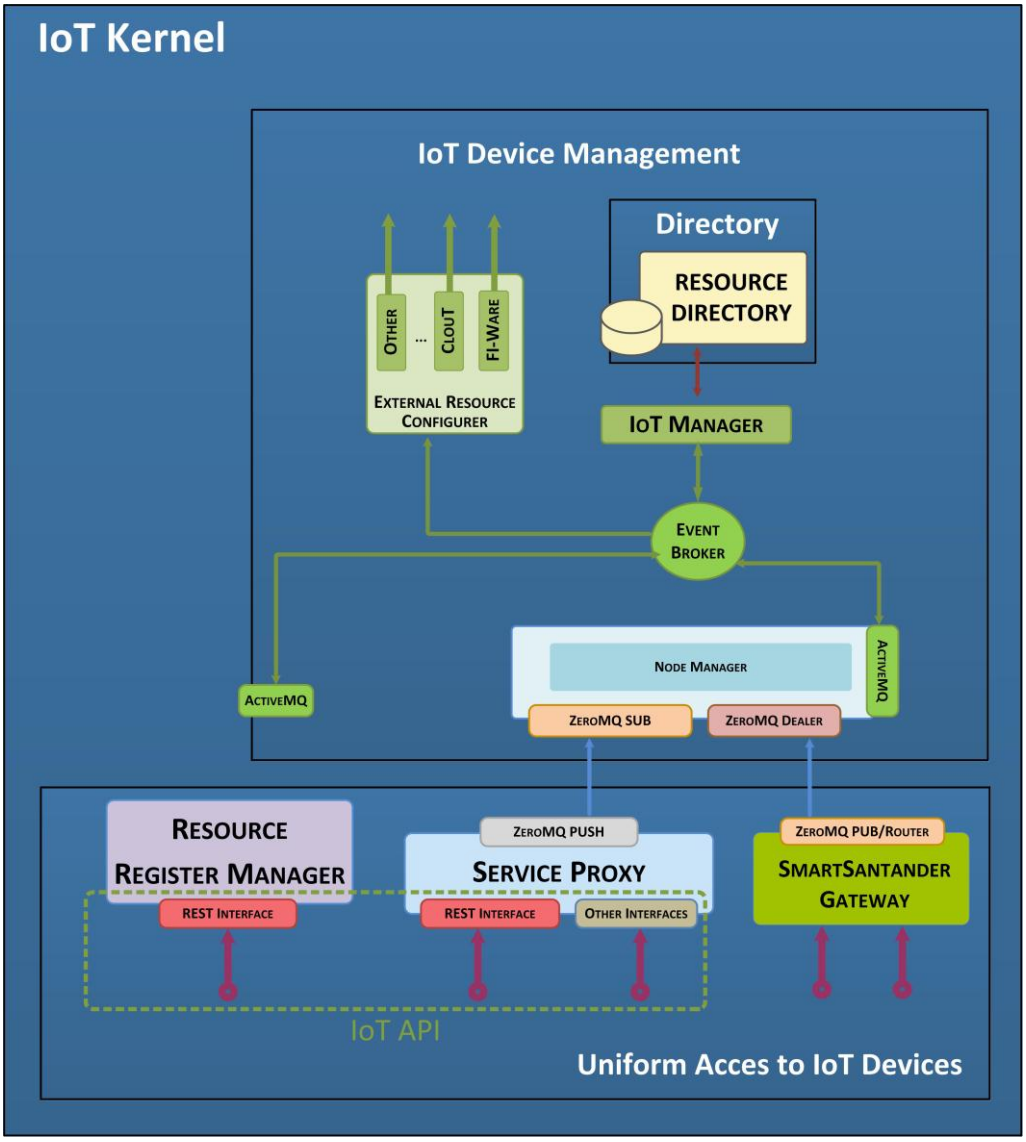

FIGURE 16 : SMARTSANTANDER RESOURCE MANAGER ARCHITECTURE

As it can be derived from the figure, SmartSantander resource manager module is composed of different components:

- **Resource register manager**: This is the entry point for the registration of new nodes that start injecting information to the SmartSantander architecture, thus receiving and processing the device identification.
- **Node Manager**: this module is in charge of overhearing transmissions from IoT devices and gateways, thus allowing the discovery of new devices, as well as their corresponding management according to the messages sent and the interval among these transmissions. This information will feed the Event broker and the rest of entities within the resource management module.
- **Resource directory**: it stands as a directory including the attributes (available sensing capabilities, location, type) associated to the resources offered by the deployed platform.

- **IoT Manager**: this module is in charge of managing registration of new nodes to the network, as well as the maintenance of the corresponding timers and events for calculating the validity time of a node in the network through notification messages sent by nodes. Once determined the node status, the IoT Manager updates the Resource Directory accordingly, with the corresponding registration, deregistration or change of attributes of a determined node or set of nodes within the network.

- **External resource configurator**: component in charge of updating external platforms (DCA-IDAS, ClouT data storage), indicating new, changing and removed ones, in order to keep updated the set of nodes that are sending and storing measurements within the different storage platforms.

- **Event broker**: component that manages all the events that take place related to the resource management within the network, mainly those implying registration/deregistration of nodes to/from the platform. This module interacts with IoT Manager, TR Configurator and IDAS Configurator.

The interaction among these modules is as follows: information provided both by resource register manager in order to add new resources, as well as by the node manager in order to update device status, feed the event broker, which interacts with IoT manager in order to access to the Resource Directory and update it accordingly. Apart from updating resource directory, event broker is also in charge of updating external platforms, through external resource configurator.

3.4.3.4. SmartSantander Gateway: Uniform Access to IoT Devices

SmartSantander Gateway allows the access to the subjacent devices deployed within the SmartSantander architecture in a bidirectional way, thus allowing both to take data retrieved from them and send to the corresponding data repository, as well as to send/receive commands to/from those devices able to be managed. SmartSantander Wireless Sensor Network API (iWSN API) represents the back-end implementation of the set of functionalities required for interacting with the IoT nodes with commands such as reset, reprogram, check if a node is alive, add/remove virtual links and many others. The iWSN API provides also the implementation of a channel for exchanging debug and control message with IoT nodes. There exist several instances of the iWSN API at server, gateway and node level, all of them implementing the set of functionalities associated to each of the architectural levels (CIaaS and CPaaS), thus allowing the management of the subjacent devices.

In order to manage both 802.15.4 and Digimesh interfaces, the CommServer module developed within the SmartSantander project, provides communication with the SmartSantander GarewaySPGW (Service Provision Gateway) through the use of a library called ZeroMQ, which is able to carry atomic messages using different transport protocols including TCP, multicast, in-process and inter-process. It is licensed under a LGPL license, allowing developer to use it freely as long as possible, apply modifications on top of the library, improving its behavior, are release under the same license.

ZeroMQ library offers an intelligent management of TCP sockets, allowing multiple socket types such as PUSH/PULL, PUB/SUB, REQUEST/DEALER, etc. PUSH/PULL sockets follow a one way only producer/consumer policy where each service generator sends messages to an endpoint

who is listening to incoming messages. Consumer peer does not need to be alive to start connectivity, as the client can queue messages until the consumer comes back, which becomes one of the main benefit of this communication method. Multiple clients can connect to the consumer without restriction as well.

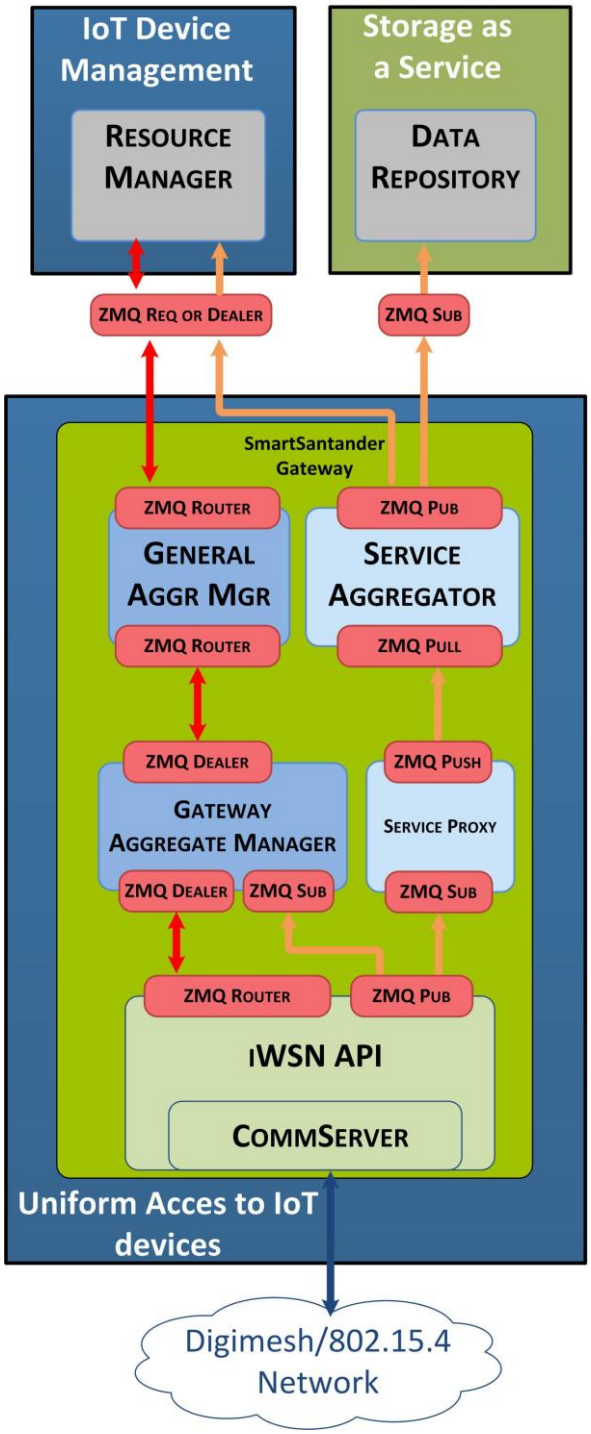

FIGURE 17 : 802.15.4/DIGIMESH PROTOCOL ADAPTER FUNCTIONAL BLOCK

As it can be derived from Figure 17, the commserver module gets the information from the subjacent devices (manging both Digimesh and 802.15.4 radio interfaces), thus publishing the corresponding information towards the service proxy and service aggregator in order to access to the data repository. On the other hand, information related with the data management is biased on the one hand by the iWSN API instance , sending data to the gateway and general aggregate manager, and on the other hand service information will allow to determine if a node is correctly sending (periodically) the corresponding values associated to it.

3.4.3.5. SmartSantander IoT API

SmartSantander IoT API is the main interface to access to the platform tier from deployed elements, external sensors and experimenters or applications. This component is meant to provide a common and homogeneous access to the registered resources within SmartSantander platform. Access to the data stored in SmartSantander, management of the resources and new data injection is carried out by using https RESTful webservices, which include authentication and authorization.

- Resource management: the API exposes a set of primitives to control and modify the description and status of the resources defined in SmartSantander. These resources do not necessary have to be part of the SmartSantander deployment and can be owned by external providers which uses the SmartSantander platform to manage their generated data.
- Data access: the IoT API exposes the appropriate webservices to access to historical data as well as real time data. Historical data can be accessed using a set of primitives that allow distinguishing by geographical position (e.g. mobile nodes which are in a certain location), time frame of the generated data and sensor types. Furthermore, real time data access is performed using a REST HOOKs implementation. IoT API implements a pub/sub interface using webservices, where external applications can subscribe.
- Data injection: IoT API considers the possibility of injecting new data to the platform. While deployed sensors within the SmartSantander testbed that use the SmartSantander gateways send measurement data through zeromq, external sensors must use this interface, as it provides the proper elements to identify and authorize resources registered in the platform.

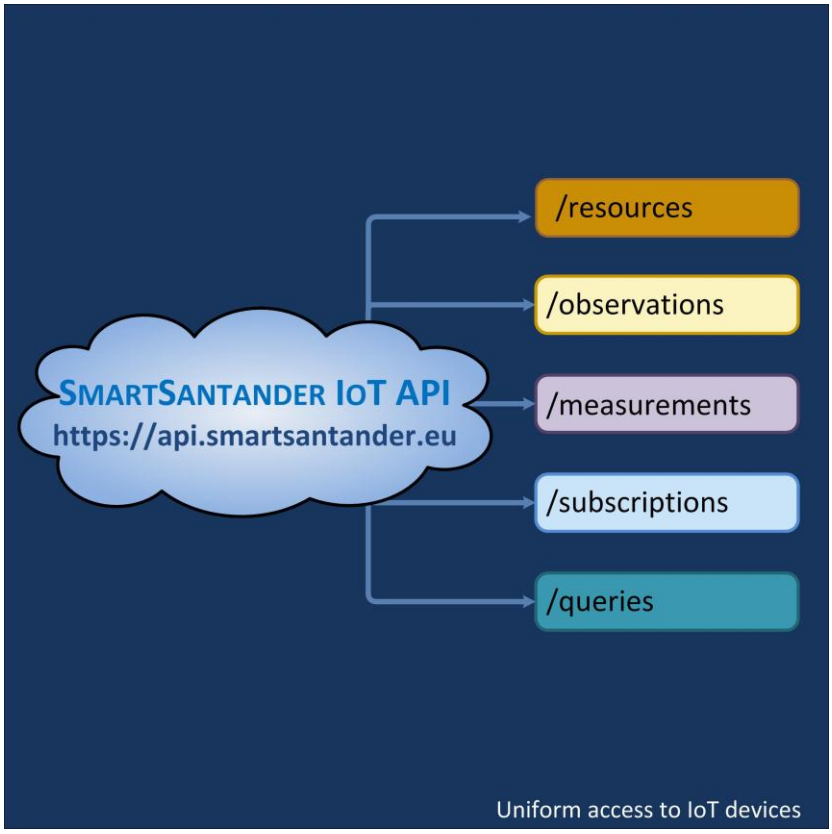

FIGURE 18 : IOT API ENDPOINTS ROOTS

Figure 18 shows the different endpoints mapping the functionalities offered from the IoT API interface. The management of the resources is made through the /resources root, while the access to the historical values is made through the /observations root, which may content several measurements in each one, and the /measurements root, with only specific measurements. Specific requests can be made using the /queries root and subscription through /subscriptions. In addition, the /observation root is also used to inject new data to SmartSantander.

3.4.3.6. SmartSantander IoT device measurements

In order to send the corresponding values measured by the sensors each node is provided with, it is needed to define a specific and standardized data format to send this information towards upper layers. In this sense, a JSON format could be used as shown next:

```
"urn":"urn:x-iot:clout:fujisawa:12345:truck12",
      "observations":[
            {
                  "timestamp":"2014-04-30T11:41:01.123+02:00",
                  "location":
                  {
                   "coordinates":
                      [-3.8643275,43.4664959],
```

```
                    "type":"Point"
                  }"measurements":[
                        {
                              "type":"temperature",
                              "value":12.4,
                              "unit":"celsius"
                        },{
                              "type":"batteryCharge",
                              "value":73,
                              "unit":"percent"
                        },{
                              "type":"luminosity",
                              "value":3245,
                              "unit":"lumen"
                        }
                  ]
            },{
                  "timestamp":"2014-04-30T11:46:00.123+02:00",
                  "latitude":"43.4764962",
                  "longitude":"-3.8543257",
                  "measurements":[
                        {
                              "type":"temperature",
                              "value":12.4,
                              "unit":"celsius"
                        }, {
                              "type":"batteryCharge",
                              "value":72,
                              "unit":"percent"
                        }, {
                              "type":"luminosity",
                              "value":1234,
                              "unit":"lumen"
                        }
                  ]
            }
      ]
```

As it can be observed, this observation is related to a mobile node (a garbage truck in Fujisawa), that is retrieving two measurements of three sensors (temperature, battery and luminosity), indication the corresponding GPS coordinates where these measurements have been gathered.

In order to maintain a standardized format for coding the information some considerations should be taken into account:

- Every observation including a timestamp with notation YYYY-MM-DDThh:mm:ss.sTZD.
- All observations geolocated.
- Each capability including type and measurement unit.
- Several observations in a JSON.
- Several measurements per observation.
- Multi-level JSON (if not supported, it could be adapted to single level JSON).

The protocol used to send this formatted information, from both SmartSantander Gateway and IoT API is ZeroMQ, as previously defined.

3.5. Opportunities/Threats Analysis

In the following subsections we provide 2 tables for the two main implementations, clarifying the components that will be reused / extended from the state of the art and the components that will be developed in the project, and a final table giving opportunities/threats analysis, for each component solution.

3.5.1. Opportunities/Threats Analysis: CEA/ST implementation

In Table 11 we report, for each functional block of IoT Kernel, the reusable component analysis for the solution developed by CEA in the context of the BUTLER project.

TABLE 11: [IOT KERNEL] (CEA) EXTERNAL COMPONENTS VS CLOUT DEVELOPED COMPONENTS

Functional block	External component to be reused or extended	Components being developed in the ClouT
Uniform access to IoT devices	sensiNact Device Access API	The sensiNact GW API has been improved to facilitate integration of new IoT devices
IoT Device Management	sensiNact local Device Management; Leshan LWM2M Server; Wakaama LWM2M Client	The original sensiNact management API is being enriched with other functionalities like the OMA LWM2M interfaces, that have to be ported and adapted to the ClouT Gateway and ST Devices
IoT Device Wrapping	sensiNact GW supports: Xbee, CoAP, enOcean; Leshan LWM2M Server; Wakaama LWM2M Client	The support for IPSO Smart Objects/OMA LWM2M bindings, XMPP, MQTT, ZeroMQ protocols has been or is being added. Integration of STM32 based platforms is also covered in this task.

3.5.2. Opportunities/Threats Analysis: UC implementation

A similar analysis has been done starting from the solution developed by UC in the context of the SmartSantander project: it is reported in Table 12.

TABLE 12: [IOT KERNEL] (UC) EXTERNAL COMPONENTS VS CLOUT DEVELOPED COMPONENTS

Functional block	External component to be reused or extended	Components being developed in the ClouT
Uniform access to IoT devices	SmartSantander Gateway	Extensions for including new devices (through a public REST interface) deployed or provided within the ClouT project.
IoT Device Management	SmartSantander Resource manager	Addition of new resources, both physical and virtual, generated within the ClouT project in the four pilot cities.
IoT Device Wrapping	SmartSantander Gateway	Addition of new IoT devices (through a public REST interface).

In Table 13 we summarize opportunities and threats for the identified implementation solutions.

TABLE 13 : [IOT KERNEL] OPPORTUNITIES VS THREATS ANALYSIS OF REUSABLE COMPONENTS

Existing work	Corresponding ClouT components	Opportunities	Threats	Notes
sensiNact IoT GW	All IoT Kernel	Scalable IoT gateway solution, open source, developed by one member of ClouT consortium. Addresses all the functionalities we need and can be easily extended for the missing protocols.	To be verified how it will be maintained and the licenses policy that will be adopted	We believe it is a plus to reuse components from other EU funded projects, granting a sort of synergy to these activities
SmartSantander Gateway, SmartSantander IoT resource	All IoT Kernel	Solution already deployed and validated in the context of the	Nothing specific	This is part of the deployments in a city where

manager		SmartSantander project.		we have the field trials, we have to deal with this solution in terms of gateway and cloud interaction
Leshan / wakaama OMA LWM2M solutions	IoT Device Management; IoT Device Wrapping	Open source state of the art implementations, widely adopted and with clear API	Nothing specific	
IoT Device Naming and Description	IoT Device Management	Solution for associating univocal identifiers and uniform descriptions to IoT devices.	Difficult to converge in the device description and to generate identifiers offering information on the device.	To be agreed between European and Japanese approaches.

3.6. Security Considerations

3.6.1. SensiNact Gateway Security Management

The sensiNact gateway is OSGi based. A first level of security is reached by the way of some of available security "tools" in this environment: *ServicePermission* and *ConditionalPermissionAdmin*

The *ServicePermission* is a module's authority to register or use a service.

- The register action allows a module to register a service on the specified names.
- The get action allows a module to detect a service and use it.

Permission to use a service is required in order to detect events regarding the service. Untrusted modules should not be able to detect the presence of certain services unless they have the appropriate *ServicePermission* to use the specific one. The *ConditionalPermissionAdmin* is framework service to administer conditional permissions that can be added to, retrieved from, and removed from the framework.

The sensiNact gateway defines service permissions in such a way that access to the ones it provides is forbidden excepted if a specific condition is met (a sensiNact specific conditional permission). This condition being that the requirer is the sensiNact *SecuredAccess* service. Even sensiNact services have to use the *SecuredAccess* one to be able to "talk" to each other's; Modalities of such exchanges depend on the *UserProfile* of the user of these services (the user can be the system itself). A *UserProfile* can be defined at each level of the hierarchical sensiNact resource model: *ServiceProvider*, *Service*, *and Resource*. Five *UserProfiles* exist for which predefined access rights are defined: Owner, Administrator, Authenticated, Anonymous, and Unauthorized.

When asking for a data structure of the sensiNact resource model, the access rights of the user are retrieved; the set of this user's accessible AccessMethods for the specific data structure is built and returned as part of the description object. Each future potential interaction of the user on the data structure will be made by the way of this description object. For a remote access a security token is also generated and transmitted to the user, to avoid repeating the security policy processing. A token is defined for a user and a data structure (and so it previously created description object).

The Security & Dependability functional block is used for authentication and to retrieve identity material from which it will be possible to associate a user and a sensiNact resource model data structure to a *UserProfile*.

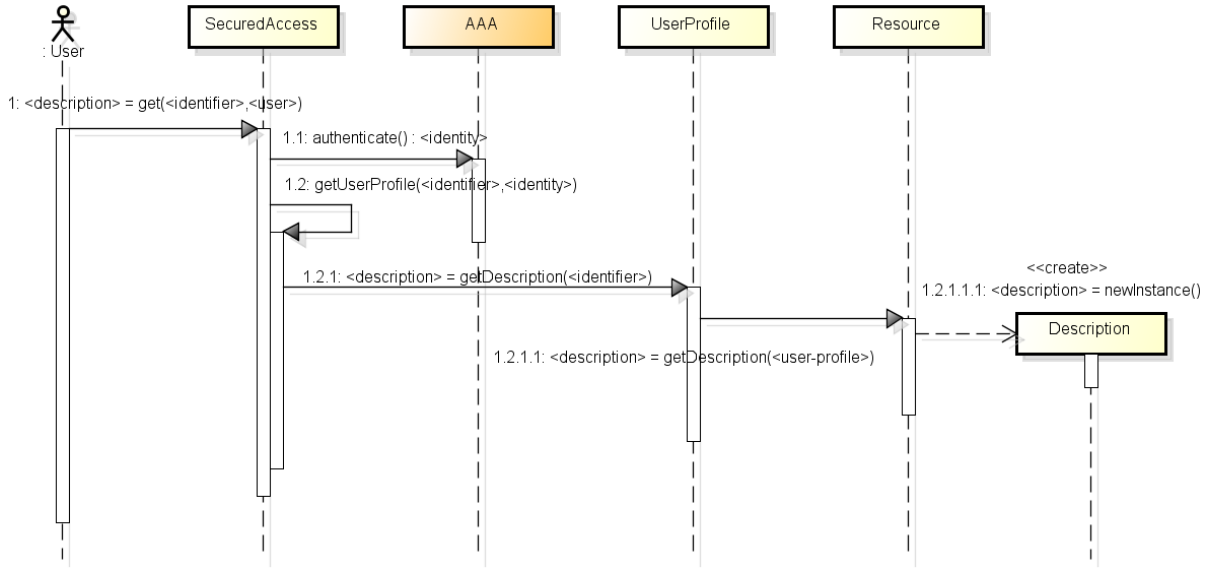

FIGURE 19: SECUREDACCESS SEQUENCE DIAGRAM

In addition to the database managed by the Security & Dependability functional block, used to authenticate a user and to retrieve its identity in the system, the sensiNact platform manages an internal database allowing to link this identity to a *UserProfile* for a specific data structure. For all data structures for which the user has not been registered the *Anonymous UserProfile* is used by default (except if the owner of a resource has defined this default profile to another one). The internal database also gathers information relative to the minimum required *UserProfile* to access to data structures. This definition can be made at each level of the resource model, knowing that if no *UserProfile* is defined for a data structure, the one specified for its parent is used.

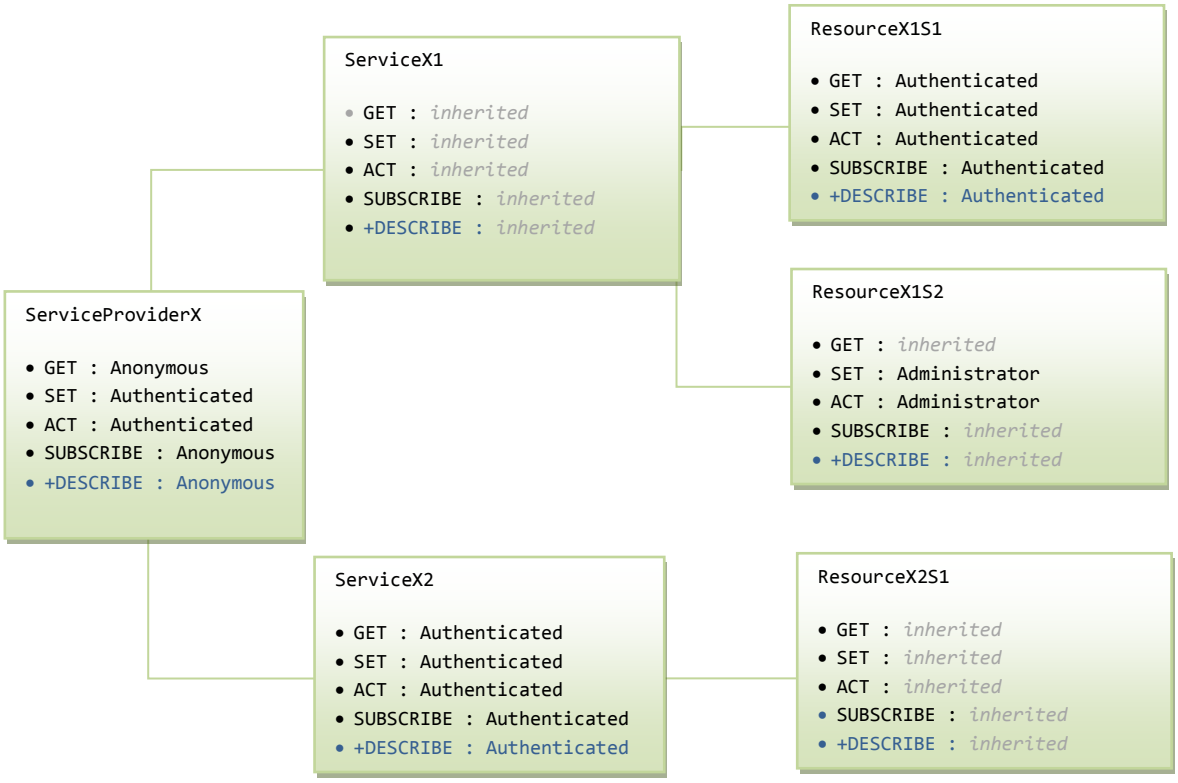

FIGURE 20: ACCESS RIGHT INHERITANCE DIAGRAM EXAMPLE

For example, according to the diagram shown above, a user trying to access to the *ServiceProviderX* for which its *UserProfile* is *Anonymous* will receive a description object in which only one *Service* will be referenced (*ServiceX1*), containing a single *Resource* (*ResourceX1S2*) providing two *AccessMethod*s, GET and SUBSCRIBE.

3.6.2. IPSO Smart Objects security considerations

IPSO Smart Objects implemented by ST will use the DTLS protocol as an end to end secure channel, as per the CoAP current specification. Additional Access Control grants are covered by the OMA LWM2M specification, with regards to the Access Control Object implemented in any OMA LWM2M client (see section 7 of [OMA-LWM2M]). This aspect is not already covered by the implementation at the time of writing.

3.6.3. SmartSantander Security considerations

Authentication of the different accesses to the data using the IoT API is carried out by the use of API keys that must be included in the headers of every http request made. API keys are randomly generated and assigned to different users depending on their needs, thus generating X.509 based certificates. The use of API keys allows the network managers to ban users from accessing platform in case of detection of a misuse of the API. Additionally to authentication, it is also being addressed the authorization control, thus providing users with the corresponding permissions for accessing the platform according to their role.

4. SENSORISATION AND ACTUATORISATION

4.1. Architecture

In Figure 21 we report a more detailed picture of the structure of the Sensorisation and Actuatorisation. The architecture is logically split in three layers:

- Networked Sensor/Actuator Enabler: this function can transform legacy sensors/actuator and social network sensors, especially immersed web data from the devices, into IoT sensors/actuators.
- Noise Reduction: this function can remove noises in social network/web data to use them as sensors.
- Uniform access to Sensorized/Actuatorized web/SNS: this block provides uniform APIs to access virtualized sensorised/actuatorised devices/web.

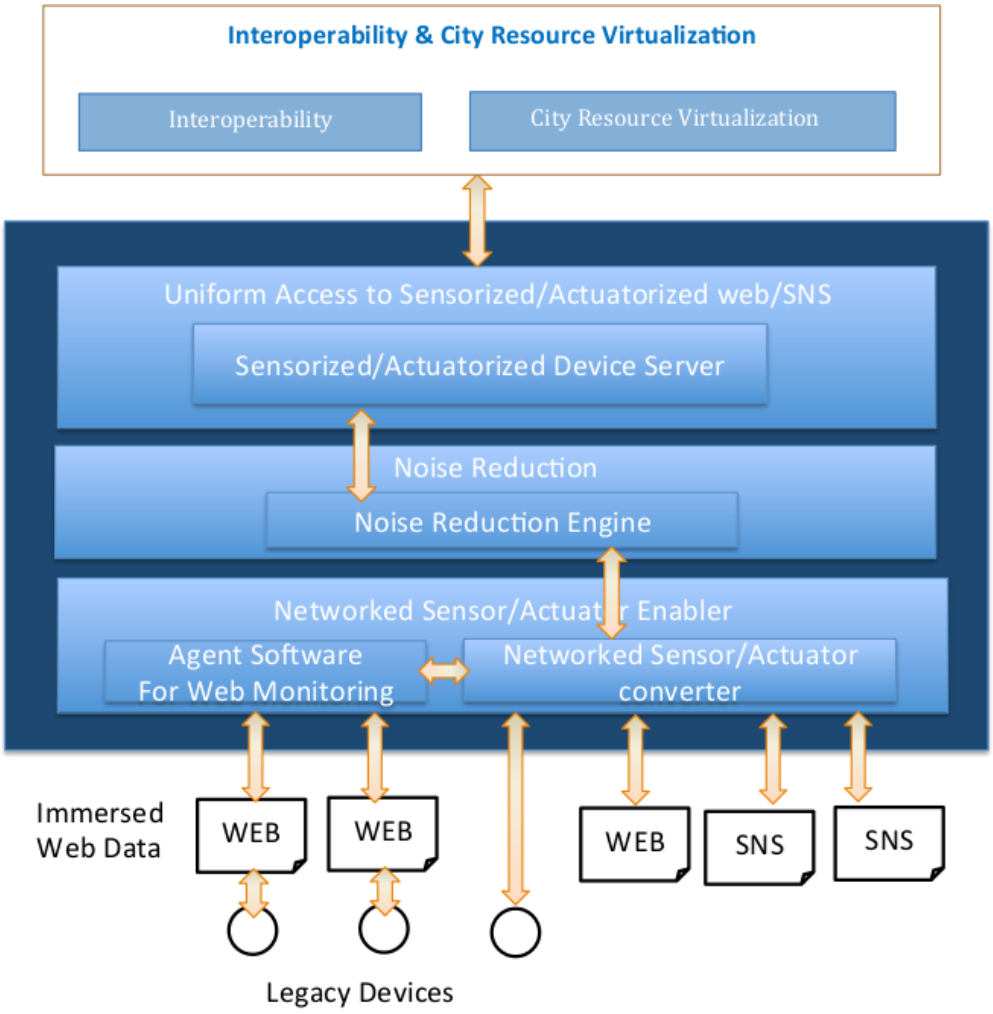

FIGURE 21: SENSORISATION AND ACTUATORISATION ARCHITECTURE

4.2. Specification

4.2.1. Networked Sensor/Actuator Enabler

Various kinds of cyber physical data are put on the web. These data are acquired from the real world via many kind of sensor devices (hard sensors), analysis software (soft sensors), interviews to human, and so on. In most cases, these data are left on the web without any API that enables computer programs to directly consume them. In addition, they are updated periodically, making the past data totally inaccessible. For example, AirNow (http://www.airnow.gov) is a web site that discloses current air quality in cities in the US. Due to the lack of APIs, when an application processes the air quality value on the page, it needs to download the whole web page, and scrape the data from the page. This difficulty of data consumption has been resulting in huge amount of cyber physical data left unconsumed on the web, which we term Entombed Web Contents (EWC). To allow applications to leverage EWC, we have been investigating on an architecture that exacavates EWC and makes them accessible through a common standard application protocol. We call such an architecture Sensorizer/Actuatorizer architecture in that the mechanism senses EWC, which were originally the data sensed once from the physical world in the past. Especially, Sensorizer is a set of (1) an authoring tool with which arbitrary elements on a web page can be defined as an EWC container, (2) a probing tool that periodically mines current value from the container, and (3) a data transmission middleware that uses XMPP over HTTP.

To achieve the purpose, browser Extension enables users to generate an EWC definition that contains the URL of a web page with EWC and their attributes, such as the name, unit, udpate rate, and their path. Its architectural role is to crowd-source EWC finding and meta-data association to them. Leveraging human supervision and knowledge for this allows the architecture precise excavation of EWC. The browser extension registers the generated EWC definition to the probe using a communication protocol, typically HTTP. This process is what we term sensorizing, in that users generate a virtual sensor on the HTML element that contains an EWC.

TABLE 14 : NETWORKED SENSOR/ACTUATOR ENABLER

Modifier and Type	Method and Description
Void	**registSensor** (String sensor_node_name, Configuration configuration) Register a new sensor node with its meta information and monitoring rule

Modifier and Type	Method and Description
Void	**unregistSensor** (String sensor_node_name) Unregister a registered sensor node

Relevant system requirements, as from [D1.3]: REQ_CIAAS_11, REQ_CIAAS_14, REQ_CIAAS_26, REQ_CIAAS_6, REQ_CIAAS_7

4.2.2. Noise Reduction

Noise Reduction functional block provides noise reduction mechanism for social network data. Social network data has many noisy data, so it is important to remove the noise in order to use the data as sensors. We assume that it can also act as simple data filtering system.

Here is an example of API interactions:

TABLE 15 : NOISE REDUCTION

Modifier and Type	Method and Description
Boolean	**isNoise** (String sns_data) Judges whether the sns_data is noise or not
Datastream	**getNoiseReductedDataStream** (Datastream stream) Returns noise-reducted data stream.

Relevant system requirements, as from [D1.3]: REQ_CIAAS_26, REQ_CIAAS_5

4.2.3. Uniform Access to Sensorised/Actuatorised web/SNS

Uniform access to Sensorised/Actuatorised web/SNS functional block provides API for accessing data. We consider various users share the same sensorised/actuatorised web/SNS, so that we leverage publish/subscribe model to access the information. Users can subscribe virtual sensor node, and receive sensor data in a uniform manner. Sensor data is also formatted with unified schema. It enables users to understand information correctly, and use the data for various applications.

TABLE 16 : UNIFORM ACCESS TO SENSORISED/ACTUATORISED WEB/SNS

Action	Method and Description
Void	**subscribe** (String sensor_node_id) Subscribe sensorised web/sns. Users can receive sensor data when a sensor data is published.
Void	**unsubscribe** (String sensor_node_id) Unsubscribe sensorised web/sns.
Void	**publish** (String actuator_node_id, Command command) Publish command to actuatorised web/sns.

Relevant system requirements, as from [D1.3]: REQ_CIAAS_19, REQ_CIAAS_11, REQ_CIAAS_14

4.3. Implementation

Before providing some implementation details, the following subsection provides 2 tables: 1 table clarifying the components that will be reused / extended from the state of the art and the components that will be developed in the project, and the 2nd table giving opportunities/threats analysis, for each component solution.

4.3.1. Opportunities/Threats Analysis

The Virtualisation and Hosting of City Resources block makes use of the Openstack cloud OS as its main reusable component and it uses some well-known open standard, like CDMI and XMPP, to achieve interoperability.

In Table 17 we report, for each functional block of Virtualisation and Hosting of City Resources, the reusable component that can be used to implement it.

TABLE 17: [SENSORISATION AND ACTUATORISATION] EXTERNAL COMPONENTS VS CLOUT DEVELOPED COMPONENTS

Functional block	External component to be reused or extended	Components being developed in the ClouT
Networked Sensor/Actuator enabler	Phantom.js Smack	User interface for defining sensor/actuator from WEB Publishing the extracted data to virtual sensor node

Noise Reduction	Place-triggered geo-tag tweet extraction technology	
Uniform Access to S/A web/SNS	ejabberd Smack	Library for several languages to access data without knowledge of XMPP/XML technology.

In Table 18 we summarize opportunities and threats for the identified components.

TABLE 18 : [SENSORISATION AND ACTUATORISATION] OPPORTUNITIES VS THREATS ANALYSIS OF REUSABLE COMPONENTS

Solution	Corresponding ClouT components	Opportunities	Threats	Notes
Smack	Networked sensor/actuator enabler, Uniform acess to S/A web/SNS	Useful for communicating XMPP server such as ejabberd. It also manages various languages such as C, Java, Javascript, etc.	Only provides basic API to communicate XMPP server.	We extend Smack solution for understandin g sensor and actuator information.

4.3.2. Networked Sensor/Actuator Enabler

We have implemented this component to Google Chrome, whose screenshot is shown in Figure 22. The idea of this extension is to let users to create virtual sensor nodes (VSN) on a page. Each VSN can contain multiple virtual transducers, each of which is associated to a single EWC. This allows users to compose multiple data elements related to a single entity into a single virtual node. For example, each row in the table in Figure 23 contains multiple interests about the air quality in a single city, such as the current AQI, (Air Quality Index) and two-day forecast. The screenshot shown in Figure 22 depicts this; the window shows the virtual sensor node for Birmingham consisting of three transducers, namely current, tomorrow, and the day after. Doing this allows applications to receive these data through a single datastream.

The user interaction with this extension is as follows. First a user can refer to the list of VSNs that are already defined by other users on the current web page. This list is acquired from the Probe server by the extension when the extension is shown. Second, the user creates a VSN by right-clicking the mouse on the page. The extension automatically set the text content, if any, below the mouse to the name of the VSN. In the case of Figure 23, the user is expected to right-click the mouse on a city name. He then adds virtual transducers to the VSN by right-clicking the mouse on the EWC that he wants to add. The XPath of the EWC is automatically detected by the extension, and he is requested to complement the name of the transducer and the unit of the data contained in the EWC.

The update rate needs to be specified with UNIX crontab format, which consists of minute, hour, day, month, and year. This means the maximum scraping frequency allowed in the current Sensorizer implementation is once a minute. Finally, he checks the radio button to the top left in the popup to register the EWC definition in the server. Figure 24 shows the EWC definition for VSN.

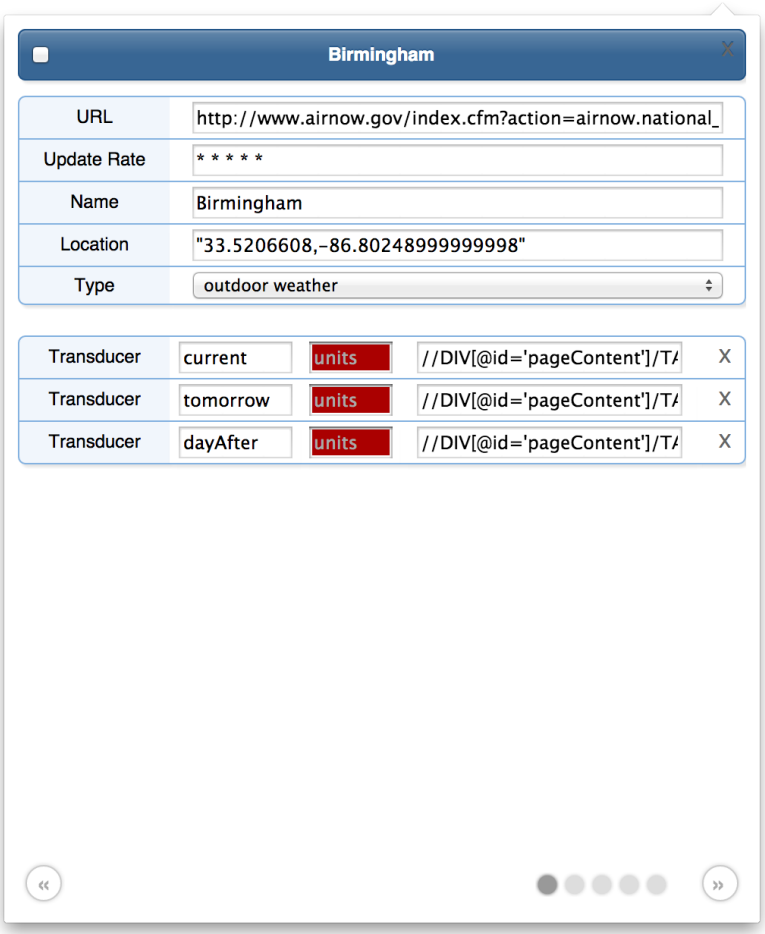

FIGURE 22: SCREENSHOT OF SENSORIZER

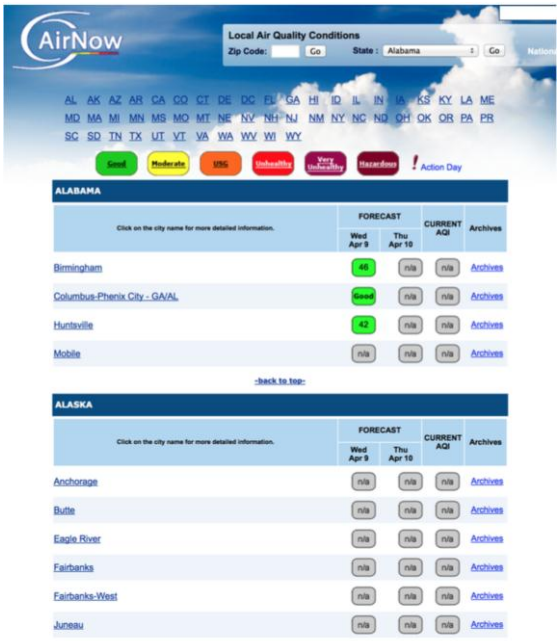

FIGURE 23: AIRNOW WEBSITE

```
{ "device":
  { "name":"Birmingham",
    "type":"outdoor weather",
    "nodeName":"Birmingham",
    "transducers":[
      {"name":"tomorrow",
       "id":"tomorrow",
       "value":{"path":"xpath://div[@id='pageco..."}
      },
      {"name":"dayAfter",
       "id":"dayAfter",
       "value":{"path":"xpath://div[@id='pageco..."}
      },
      {"name":"current",
       "id":"current",
       "value":{"path":"xpath://div[@id='pageco..."}
      }
  ]},
  "source":{"url":"http://www.airnow.gov/index....",
    "update":"* * * * *"},
  "location":{"latitude":"33.5206608",
    "longitude":"-86.80248999999998"}
  }
}
```

FIGURE 24: SENSOR DEFINITION

A monitoring agent called Probe periodically downloads the web pages from the URLs specified in the registered EWC definitions. It uses the update rate in the definition to determine the period of web page downloads. After downloading the pages, it scrapes them to extract the EWC from the path also specified in the EWC definitions. Finally, it transmits the EWC to a network.

The role of Probe in the architecture is to generate data streams from a web page. As discussed above, EWC are updated periodically. An EWC on a web page can be thus considered to be a snapshot of the original data stream. Probe allows applications to consume an EWC as a stream, instead of an element packed in a HTML document. It should be noted, however, that temporal granularity in the stream would be coarse, such as a data per one minute at finest. Since this component fetches web pages continuously from a web server, it should avoid overloading the server.

We implemented this component with Javascript program running on Phantomjs (http://phantomjs.org), which is a headless web browser. This program scrapes EWC, whose paths are specified in the EWC definition. We leverage Phantomjs since it can execute Javascript programs referred within HTML files of a web page. There exists many web pages that are generated on browsers with Javascript programs. If a VSN definition is generated from such a script-generated web page, the XPath in the definition is extracted from the DOM structure generated by the script. Scraping meaningful data from such a page thus needs to run the scripts to generate the DOM structure equivalent to the above. When sensor data are successfully acquired from a web page, the probe finally transmits the data to the Internet. For this transmission, we leverage Sensor-Over-XMPP (SOX) to achieve higher availability of sensor data over the Internet.

4.3.3. Noise Reduction

For Noise Reduction functional block, we leveraged our technology called extracting of place-triggered geo-tagged tweets. Recently, many related works address to detect real world events from social media such as Twitter. However, geotagged tweets often contain noise, which means tweets which are not content-wise related to users' location. This noise is a problem for detecting real world events. To address and solve the problem, we define the Place-Triggered Geo- tagged Tweet, meaning tweets which have both geotag and content-based relation to users' location. We designed and implemented a keyword-based matching technique to detect and classify place-triggered geotagged tweets. We basically use the reusable component in our implementation. In addition to the component, we are planning to create further functions which reduce the data stream of sensorised web. Some of web is dynamically changing, but other web is static for some period. For such web, we will implement checking agent which monitors whatever the web is updated or not.

4.3.4. Uniform Access to Sensorised/Actuatorised web/SNS

We provide complementary client-side APIs for different programming languages including Java, Javascript, and Python. Those APIs are object oriented, and share almost the same class

hierarchy. The APIs support both publishing and subscription of VSNs, though in this section we focus on the latter.

Figure 25 shows a simple Javascript code that subscribes multiple VSNs specified with their names in var deviceNames = ["name1", ...,"nameN"];. The API is event-driven. The program first connects to the server after the page is loaded. If the connection is successful, the event handler *connected* is invoked, inside which the program subscribes to the VSNs whose names are specified in the *deviceNames* array. If the subscription is successful, the program periodically receives the data and meta-data published by the VSNs through the event handler *sensorDataReceived* and *metaDataReceived*.

Using this API, programmers can easily create server-side web services or client-side web applications that, for example, mashup EWC distributed over multiple web pages without coping with their web servers manually.

We also provide Java API for stand-alone applications and server-side web services. Programmers can leverage multi-thread capability of the language to handle a number of data streams with Java API. It also can be used in Java Server Pages (JSP) within Apache Tomcat.

In both languages, the XML notation used in XMPP payload is hidden from applications. The developers thus can concentrate on the core body of the programs. In total, we support stand-alone, server-side web, and client-side web applications for consuming EWC.

```
var serv = "sox.ht.sfc.keio.ac.jp";
var bosh = "http://"+serv+":5280/http-bind/";
var jid = "guest@sox.ht.sfc.keio.ac.jp";
var pw = "miroguest";

window.onload = function() {
    var client = new SoxClient(bosh, serv, jid, pwd);
    client.unsubscribeAll();

    var soxEventListener = new SoxEventListener();
    soxEventListener.connected = function(e) {
        console.log("[main.js] Connected "+e.soxClient);
        status("Connected: "+e.soxClient);

        var deviceNames = ["name1", ..., "nameN"];
        deviceNames.forEach(function(name){
            /* subscribe all the devices above*/
            var device = new Device(name);
            if(!client.subscribeDevice(device)){
                status("Couldn't subscribe: "+device);
            }
        });
    };
    soxEventListener.subscribed = function(e){
        status("Subscribed: "+e.device);
    };
    soxEventListener.metaDataReceived = function(e){
        /* called when meta data are received */
        status("Meta data received: "+e.device);
    };
    soxEventListener.sensorDataReceived = function(e){
        /* called when sensor data are received */
        status("Sensor data received: "+e.device);
    };

    client.setSoxEventListener(soxEventListener);
    client.connect();
};
function status(message){
    $("#s").html(message+"<hr>\n"+$("#s").html());
}
```

FIGURE 25: SENSORIZER APPLICATION CODE IN JAVASCRIPT

4.4. Security considerations

For the Uniform access to Sensorised/Actuatorised web/SNS the XMPP interface is used, so XMPP security function is available. If sensor data is possible to reveal the users' privacy, it is necessary to set appropriate access model to the virtual sensor node. We leveraged XMPP security feature to access pubsub event node. We can set access model as following:

In terms of subscribing:

- **Open**: anyone may subscribe and retrieve items.
- **Authorize**: subscription requests must be approved by an owner and only subscribers may retrieve items.
- **Whitelist**: only those on a whitelist may subscribe and retrieve items.
- **Presence**: entities that are subscribed to the node owner's presence may subscribe to the node and retrieve items from the node.
- **Roster**: entities that are subscribed to the node owner's presence and in the specified roster group(s) may subscribe to the node and retrieve items from the node.

In terms of publishing:

- **Open**: anyone may publish items to the node.
- **Publishers**: owners and publishers are allowed to publish items to the node.
- **Subscribers**: owners, publishers and subscribers are allowed to publish items to the node.

These features make the access to sensor data more secure.

5. INTEROPERABILITY AND CITY RESOURCE VIRTUALISATION

5.1. Architecture

The set of interoperable city data components aims to provide a cross-application domain interoperability that will make the system capable of cooperating with the solutions of different device vendors, service providers or application integrators. It is responsible of the passage of raw unstructured city data to meaningful contextualized one. This transformation follows from the cooperation of a syntactic interoperability processing with a semantic interoperability one.

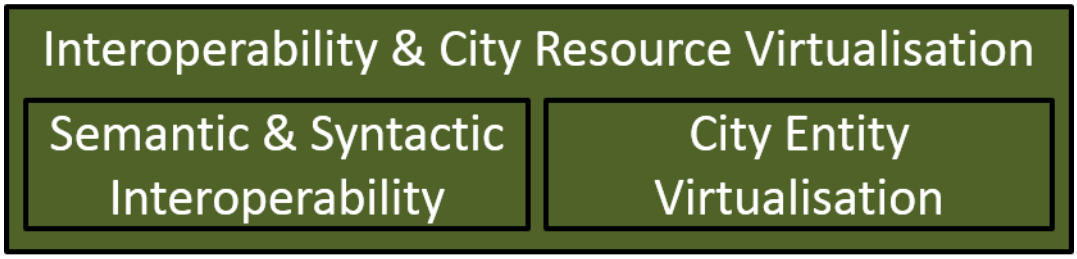

FIGURE 26: INTEROPERABLE CITY DATA

There are several types of mismatches about sensory data between sending side and receiving side (Figure 27). They include protocol, format, label, sampling rate, error rate, accuracy, the difference of mesureing object and policy. Upperside elements of the figure are resolved by syntactic interoperability, and downward side elements are resolved by semantic interoperability.

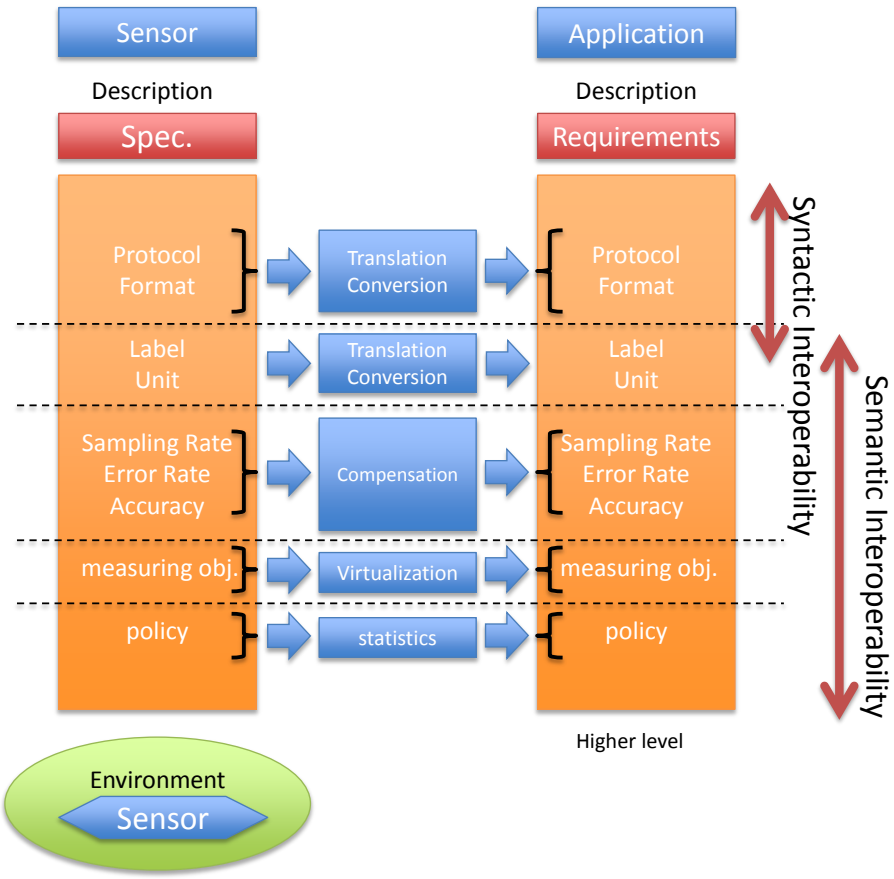

FIGURE 27 : CATEGORY OF MISMATCH

A more concrete, component level arcthicture is shown in Figure 28. After receieving data from sender, the initial data converter in syntactic conversion converts the data to internal common data format (ICDF) and following conversion components convet the data format onto the ICDF. Lastly the component in the syntactic conversion, converts ICDF to the outernal data format. With this conversion model, it is possible to increase the reusability of conversion components and it is expected to reduce developing duration and cost.

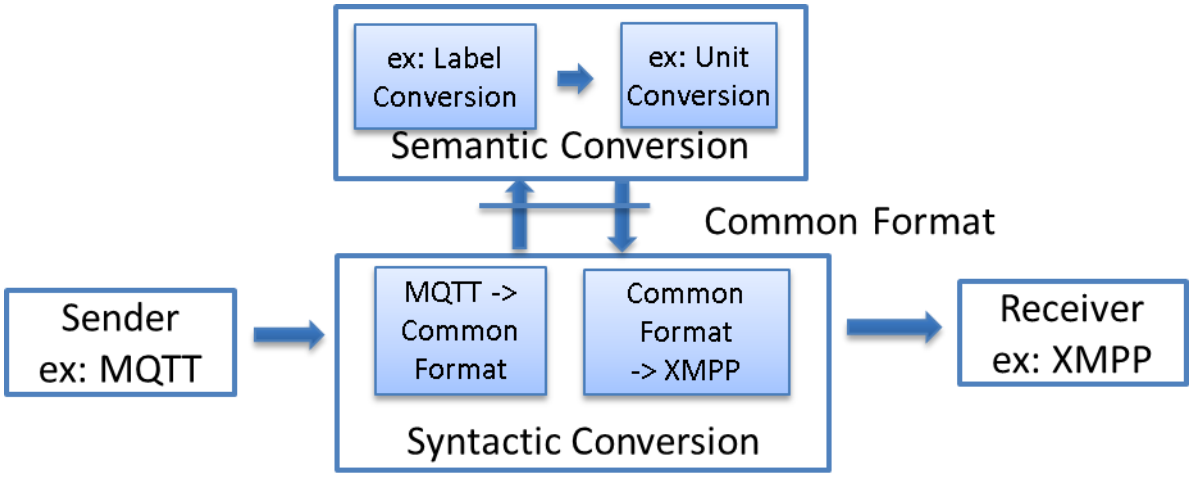

FIGURE 28 : COMPONENT LEVEL ARCHITECTURE FOR INTEROPERABILITY

Reusability of conversion components and Internal Common Data Format.

In case of n protocols to m protocols conversion, direct conversion requires n x m conversion functions (Figure 29). With indirect conversion via common data format, we can reduce the number of required components to (m + n) (Figure 30).

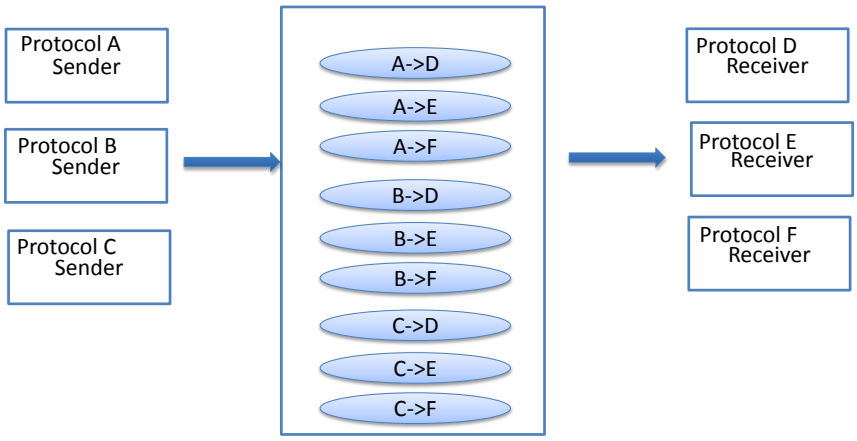

FIGURE 29 : DIRECT CONVERSION FROM 3 PROTOCOLS TO 3 PROTOCOLS

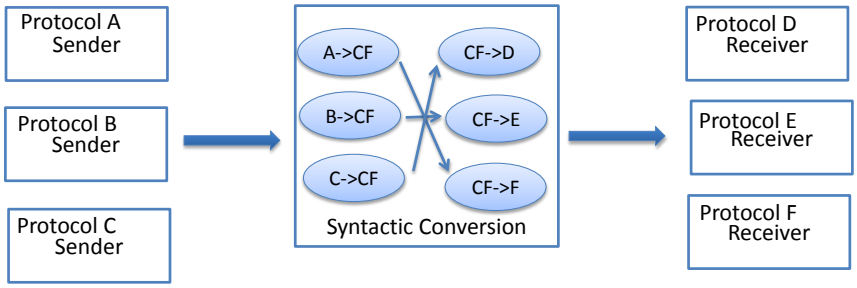

FIGURE 30 : INDIRECT CONVERSION FROM 3 PROTOCOLS TO 3PROTOCOLS

There exist several domains for data format. It is relatively easy to create a conversion inside of domain. It is getting harder to create a conversion between more different data format, for example JSON and ASN.1. In those cases the following figure (Figure 31) is beneficial. Each conversion function (in blue color) can be implemented in simple form and still less number of components are required even if extra conversion components (in red color) are required.

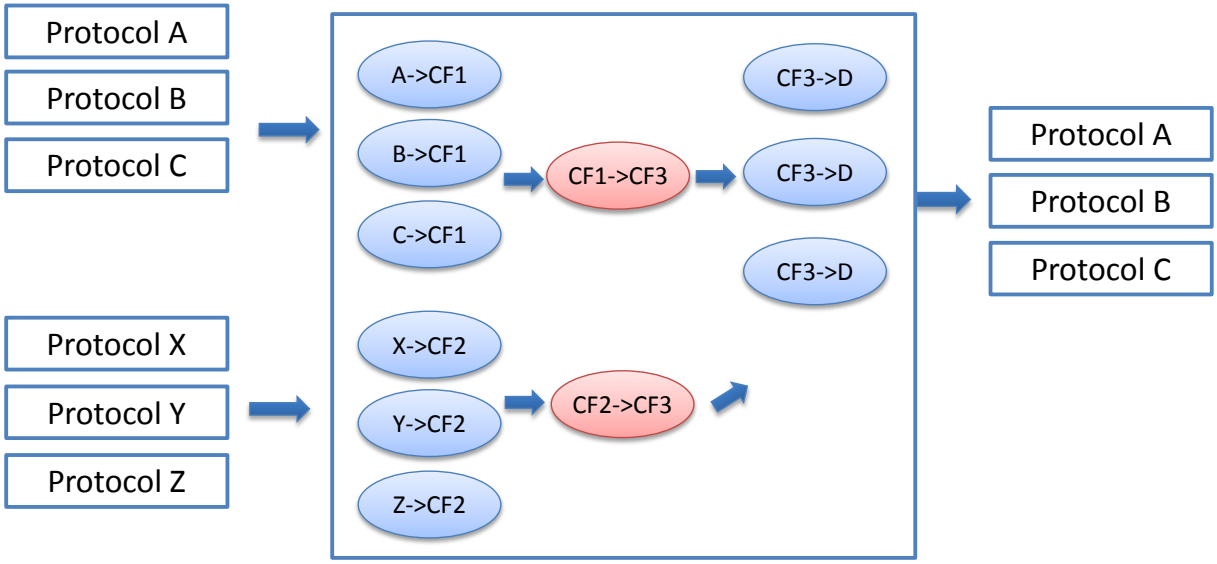

FIGURE 31 : INDIRECT CONVERSION USING MULTIPLE INTERMEDIATE FORMAT

Within this project we will define some of internal common data formats (ICDF). The purposes of the ICDF are:

- To reduce the number of components for conversions and increase reusability of the protocol conversion function.
- To support both inter process use and extra process use. For the former, small latency and computational efficiency are important. For the latter, it is important to pass information in a same time to process a chained processing.
- To provide intuitive format for developer in specific domain, rather than being abstract in order to cover all possible domains.

The concrete requirements for the ICDF should be determined in each domain of the data format, such as XML, JSON, TLV, ASN.1 and so on, because characteristics of data format are different depending on the domain.

We extracted requirements for ICDN in JSON domain as a first target.

- Must be represented in form of JSON.
- Must be also represented in a native object structure of target program language.
- Must accommodate multiple measurements and multiple observations.
- Must be translated to format used in external API, in systematic manner.
- Must be possible to pass the data to another function in the same process in efficient way, such as API call.
- Ways for quick manipulation and traversal of the structure must be provided.
- Should save resources by sharing metadata among multiple measurements and observations.

As a solution for the ICDF, we choose 'org.json' object library in native form for Java language. The JSON is represented in tree structure of objects and it is passed between conversion components. We assume that the JSON is described in the form reported in section 3.4.3.6 which is also used in SmartSantander.

Syntactic interoperability:

The syntactic interoperability processing consists in :

- the syntactic verification of received data (i.e. does the data respect the model/schema defined in its original format) ;
- potentially its interpretation, meaning the collecting of usefull data in its original format according to the one expected by the system ;
- potentially its transformation into the inner-system defined data format.

This syntactic interoperability processing, when finished, transmits treated data to the semantic verifier component which starts the semantic interoperability processing. Interpretation and transformation are both optional tasks in the particular case where received data are already in the same format that the inner-system defined one.

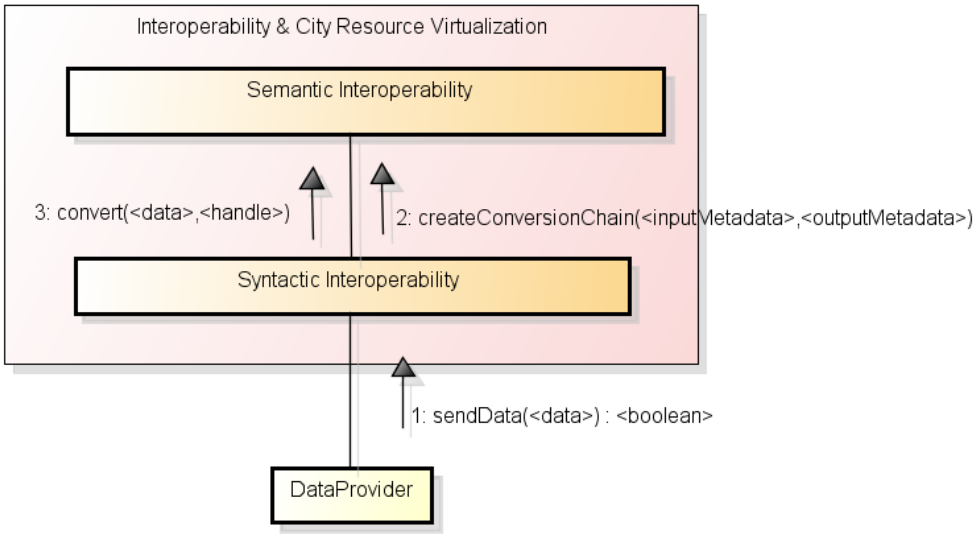

FIGURE 32: SYNTACTIC INTEROPERABILITY

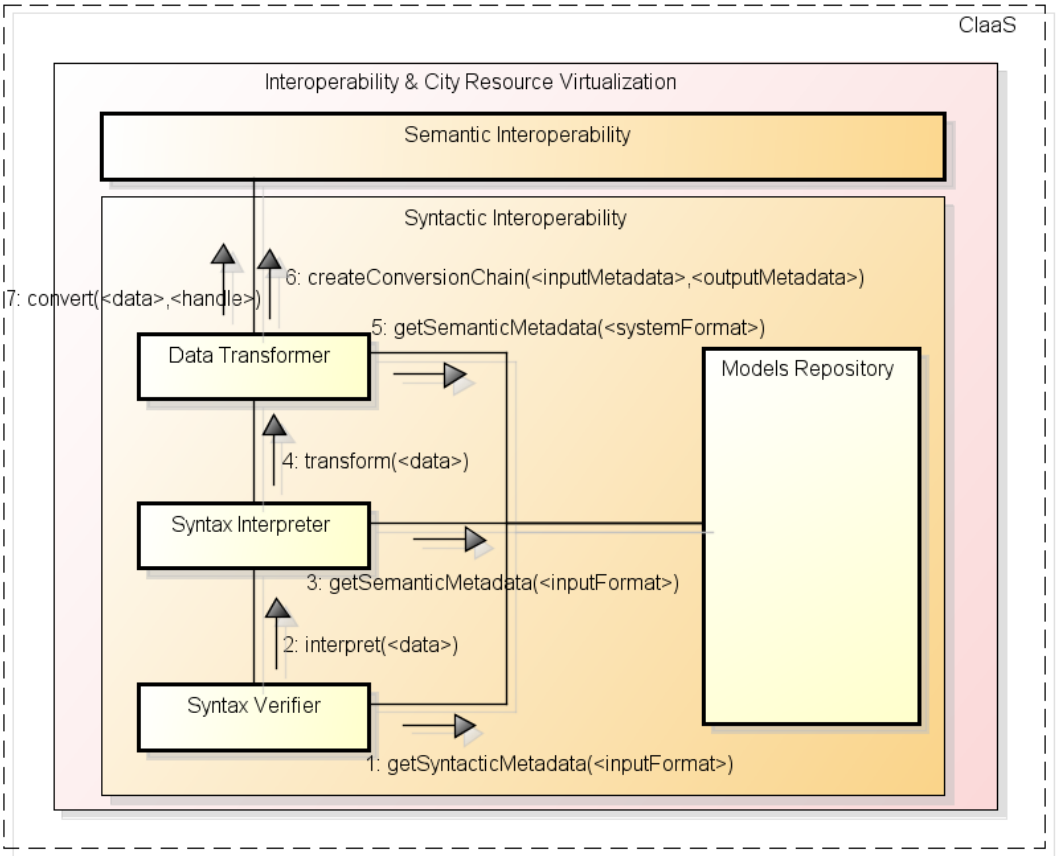

FIGURE 33: SYNTACTIC INTEROPERABILITY DETAILED

Semantic Interoperability:

Semantic interoperability block plays the role for converting data in the meanig. Typical conversions are label conversion of data, unit transformation, coordinate transformation and so on. In the ClouT project we have the following concept on semantic interoperability:

1) Componentize basic conversion components and reuse them as much as possible.

2) Prepare metadata describing the data the sending side is sending and the data which reciving side expects to receive.

3) By using the metadata, developing tool helps developers to integrate conversion components for realizing the required conversion. Possibly, fully automatic generation of conversion is also aimed.

For explaining detailed architecture in semantic interoperability, two communication diagrams are shown. First one (Figure 34) shows the generation phase of the conversion function, called conversion chain. The other one (Figure 35) shows the actual conversion phase of the data.

In the generation phase, an entity (shown as Binder) requests converter to form some conversion function. It passes metadata for the input and output data, they specify format and meaning. Here Binder can be a application developer who wants to connect one format to another. In the extreme case, this can be invoked by the application itself that wants to connect new type of sensors with conversion.

After the request above, 'converter' searches metadata repository and component repository and finds possible combinations of components to form the requested conversion. When multiple candidates are found, it selects the best combination. The converter returns a handle which represents the chain.

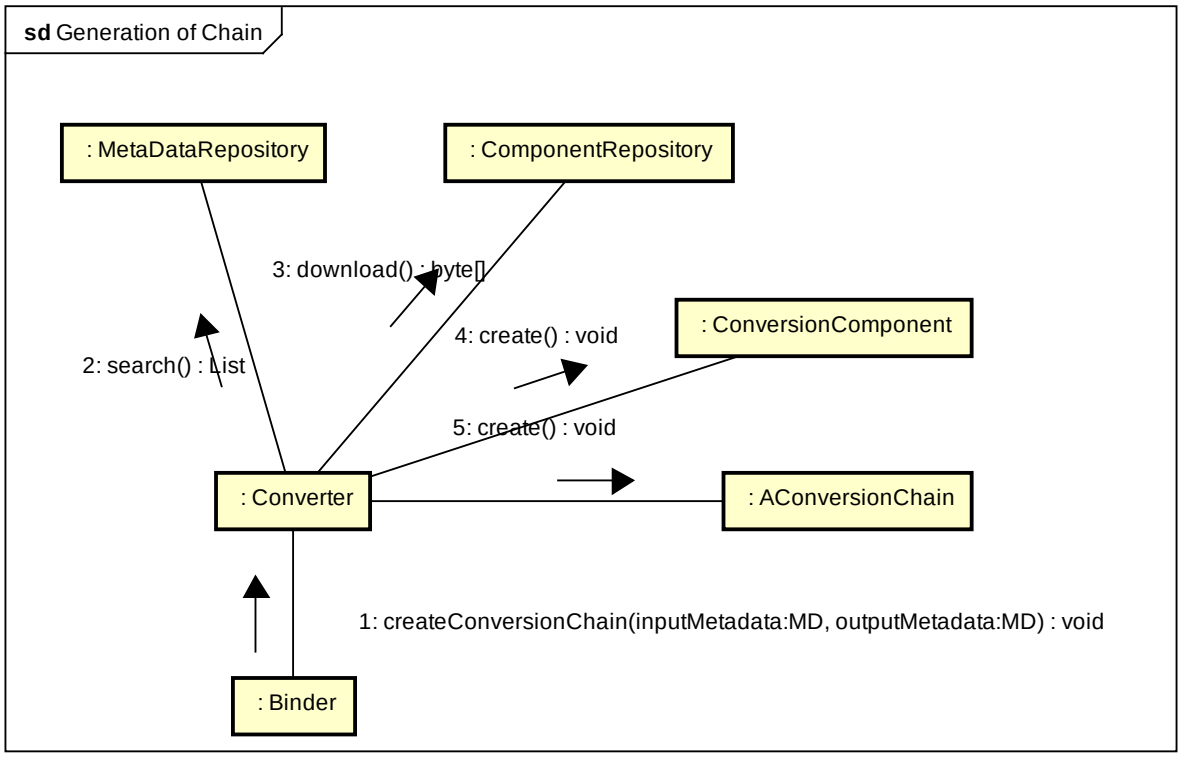

FIGURE 34: SEMANTIC INTEROPERABILITY - GENERATION

Figure 35 shows the actual conversion for the data. A client entity requests to the converter to convert by passing the original data and handle for the conversion chain. The converter searches and retrieves actual chain. It requests data conversion to each component step by step. Finally it gets the conversion result and returns it to the requested entity.

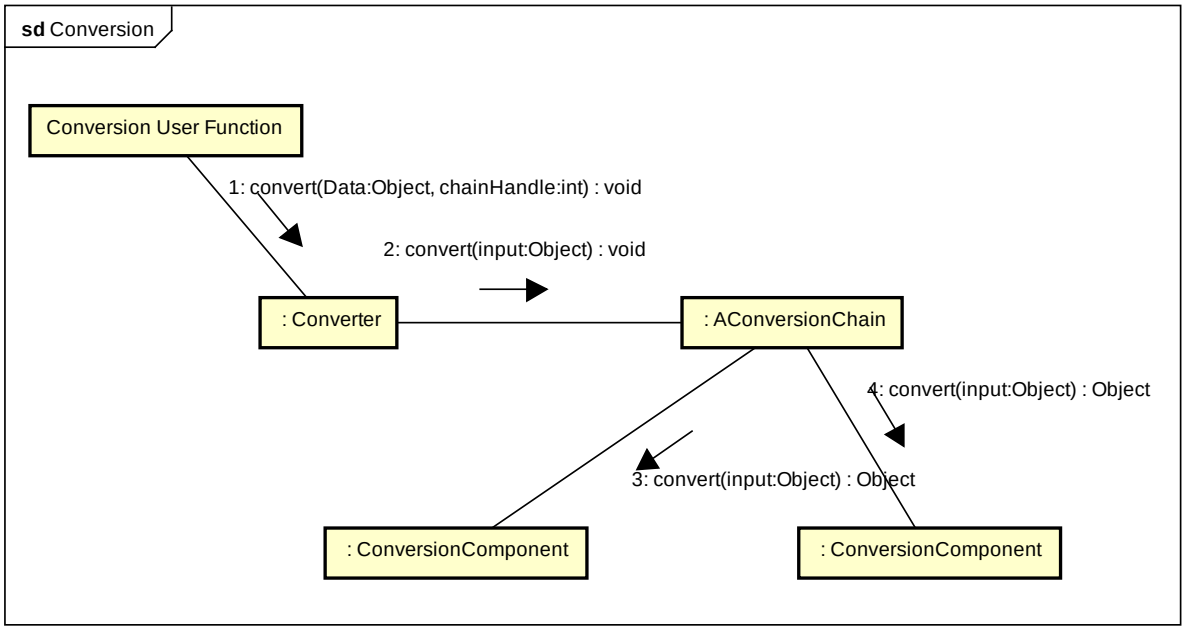

FIGURE 35: SEMANTIC INTEROPERABILITY - CONVERSION

Interoperability Specific Use Cases and Requirements

We derived and analyzed several use cases (table below) and extract more specific requirements on interoperability.

TABLE 19 : INTEROPERABILITY REQUIREMENTS

Use case name	Description	sub use cases	Interoperability Problem
Clean Slate	Developing new application on top of single sensor.		No interoperability problem
New App on Many Sensors	Developing new application on top of many types of sensors.		The cost of developing application gets higher.
New App on existing sensors	Importing Application to the environment with existing sensors.		In case that supported data type is different, the application needs to be modified. Or data converter needs to be provided.
New sensors under existing App.	Importing New sensors under existing application.	Replacement of the sensors by several reasons (discontinuation of the model, release of better model, new regulation apply)	In case that supported data type is different, the application needs to be modified, or data converter needs to be provided.
		Extension of geographical service area of the application. New type of sensors are supposed to be located at the extended area.	
		Extension of service domain of the application. Ex. The application has been designed in the traffic domain originally and used traffic monitoring sensors. But it newly starts supporting disaster preventing domain too, and needs to support fire alarms, and earthquake sensors.	

Interoperability specific requirements:

- Effectiveness: these Development need should be done in cost effective and timely manner.
- Estimation Capability: estimation of development amount should be possible. During development, interoperability support tools should be provided with an estimation of development amount.
- Testability: test should be possible in effective manner after these integrations.

5.2. Interoperability

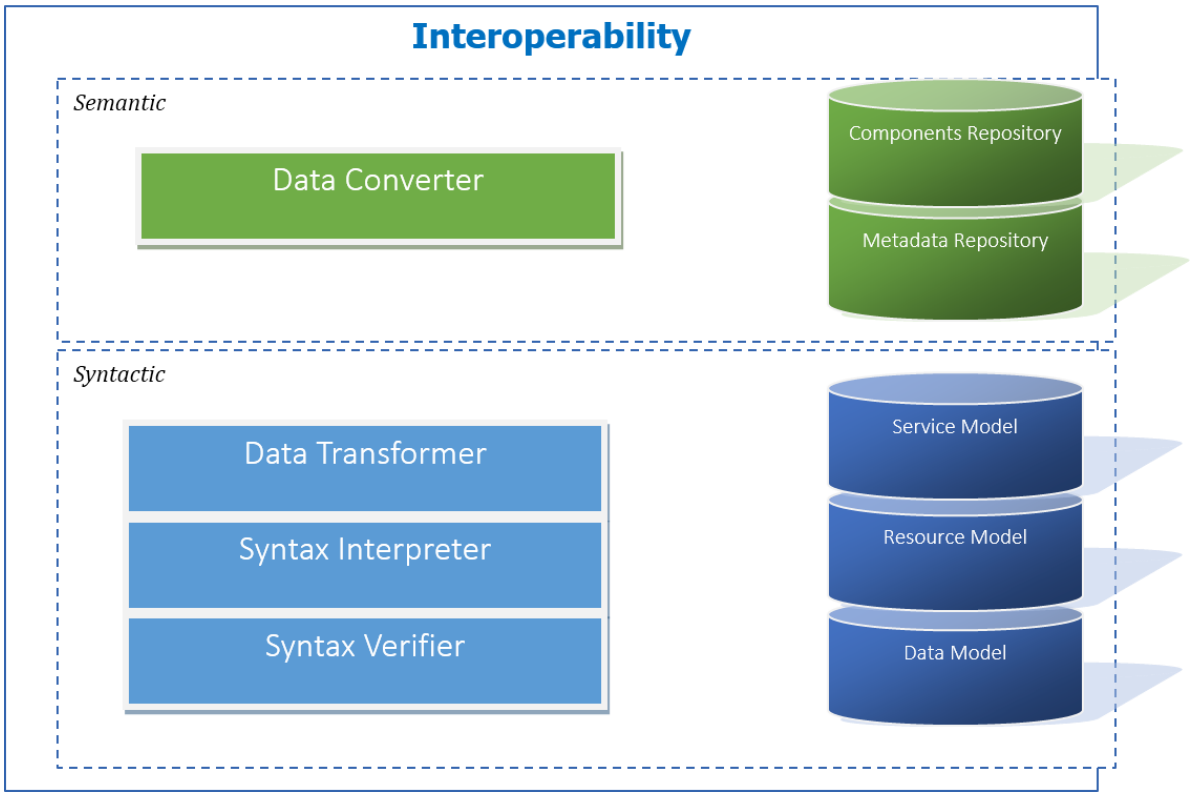

FIGURE 36 : INTEROPERABILITY ARCHITECTURE

5.2.1. Specification

5.2.1.1. Semantic Interoperability - Data Converter

Data Converter is an engine for data conversion and provides a data conversion capability by combining relatively simple conversion units, called conversion component.

Data converter provides following methods for initiating the creation of conversion capability.

TABLE 20 : DATA CONVERTER API

Modifier and Type	Method and Description
ConversionChain	**createConversionChain** (InputMetadata, OutputMetadata) This method creates a chain for the requested conversion specified with the parameters (InputMetadata and OutputMetadata) and returns the object represents the conversion chain. In case that there is no possible conversion path, it throws an exception.

By calling the createConversionChain method in the Data Converter, the Conversion Chain will be returned.

ConversionChain Object provides following methods.

TABLE 21 : CONVERSION CHAIN API

Modifier and Type	Method and Description
Data	**convertData** (ConversionChainID, InputData) This method converts the data with the specified conversion chain.
Void	**discard** () This method discards the conversion chain.

Relevant system requirements, as from [D1.3]: REQ_CIAAS_14, REQ_CIAAS_15, REQ_CIAAS_16.

5.2.1.2. Semantic Interoperability – Metadata Repository

Metadata Repository is provided on top of the Web Server. It provides following methods based on the concept of RESTful Service.

TABLE 22 : METADATA REPOSITORY API

Modifier and Type	Method and Description
List<**Metadata**>	**list** (filter) This method returns the metadata which matches the filter specified in the parameter. In case of null, all list of metadata will be returned.

Modifier and Type	Method and Description
Metadata	**get** (id) This method returns the metadata specified by the id.
List<**Metadata**>	**getBulk** (id) This method returns the set of metadata related the specified id.

Relevant system requirements, as from [D1.3]: REQ_CIIAS_2, REQ_CIAAS_11, REQ_CIAAS_14, REQ_CIAAS_15, REQ_CIAAS_21, REQ_CIAAS_23, REQ_CIAAS_26

5.2.1.3. Semantic Interoperability – Component Repository

Component repository is provided on top of the web server. It provides following methods based on the concept of RESTful Service.

TABLE 23 : COMPONENT REPOSITORY API

Modifier and Type	Method and Description
List<**ComponentDescription**>	**list** (InputFilter, OutputFilter) This method returns the list of component description which matches the filter specified in the parameter.
BinaryData	**get**(id) This method returns the component implementation.

Relevant system requirements, as from [D1.3]: REQ_CIAAS_14, REQ_CIAAS_15.

5.2.1.4. Syntactic Interoperability – Data Transformer

The *Data transformer* component is responsible for transforming information from one format to another. This component will get the data from the *Syntax interpreter*, transforms it into the desired data representation format, and provides it to the semantic interoperability level for potentially needed semantic transformation.

TABLE 24 : DATA TRANSFORMATION API

Modifier and Type	Method and Description
<Data>	**transform** (<Data>) Transforms the data passed as parameter to the inner-system format and returns the resulting transformed data
<Data>	**addSemantic** (<Data>) Adds semantic information to the data passed as parameter according to the expected of the inner-system format, the information provided by the original data format, and existing context aware data. According to the compliance of used "units" with expected ones, input and output metadata are created and transmitted to the Data Semantic Converter; If it is the case, data is passed through the semantic conversion chain before to be returned.

Relevant system requirements, as from [D1.3]: REQ_CIIAS_2, REQ_CIAAS_11, REQ_CIAAS_14, REQ_CIAAS_15, REQ_CIAAS_21, REQ_CIAAS_23, REQ_CIAAS_24, REQ_CIAAS_25, REQ_CIAAS_26

5.2.1.5. Syntactic Interoperability – Syntax Interpreter

The *Syntax interpreter* component is responsible for interpreting the received data and extracting the useful information from the content according to the models stored in the repository and the inner-system resource one. This component gets the data verified by the *Syntax verifier* component and sends it to the *Data transformation*.

TABLE 25 : SYNTAX INTERPRETER API

Modifier and Type	Method and Description
Void	**interpret** (<Data>) Interprets the data passed as parameter, meaning identifies meaningful data according to its original format, needs of the system and existing context aware information.
<Data>	**extract** (<Data>) Extracts and returns the identified meaningful data in the one passed as parameter

Relevant system requirements, as from [D1.3]: REQ_CIIAS_2, REQ_CIAAS_11, REQ_CIAAS_14, REQ_CIAAS_15, REQ_CIAAS_21, REQ_CIAAS_23, REQ_CIAAS_24, REQ_CIAAS_25, REQ_CIAAS_26

5.2.1.6. Syntactic Interoperability – Syntax Verifier

The *Syntax verifier* component is responsible for verifying the conformity of received data according to a recognized format. Allowed (i.e. verifiable) syntaxes are those which are defined in the models repository. This component is in interaction with the device abstraction layer components, in particular with the IoT protocol adapters and virtualization components to gather data in the format provided by those adapters, and it will check the syntax of the provided data. It will also be able to interact with the *Semantic Verifier* component in the case that interpretation and transformation of the data is not needed.

TABLE 26 : SYNTAX VERIFIER API

Modifier and Type	Method and Description
Boolean	**sendData** (<Data>) Initializes the syntax verification chain of the data passed as parameter and returns a boolean, whose value depends on the syntactic conformity of this data. Potentially, if no interpretation and no transformation are needed, this call is responsible for the transmission of data to the interoperable semantic processing.
Boolean	**checkSyntax** (<Data>) Checks whether the syntax of the data passed as parameter is conform to its recognized format, and returns a boolean whose value depends on this conformity

Relevant system requirements, as from [D1.3]: REQ_CIIAS_2, REQ_CIAAS_11, REQ_CIAAS_14, REQ_CIAAS_15, REQ_CIAAS_21, REQ_CIAAS_23, REQ_CIAAS_24, REQ_CIAAS_25, REQ_CIAAS_26

5.2.2. Implementation

5.2.2.1. Syntactic Interoperability

To provide interoperability the referential resource model is the one of the sensiNact gateway. In the sensiNact gateway the syntactic interoperability is managed at the bridge level, by a specific **Adapter** functional component.

A sensiNact bridge is decomposed as shown in the figure above:

- The **Protocol Stack** is the component which manages the communication with the user/device/platform;
- The **Bridge** is the component which is able to compose or to decompose the message (payload) transmitted by the way of the protocol stack;
- And finally as previously defined, the **Adapter** is the component ensuring the syntactic interoperability by validating the content of the message relayed by the bridge, by mapping transmitted data to the appropriate inner-system ones, and potentially by transforming those which need to be to feed the system.
-

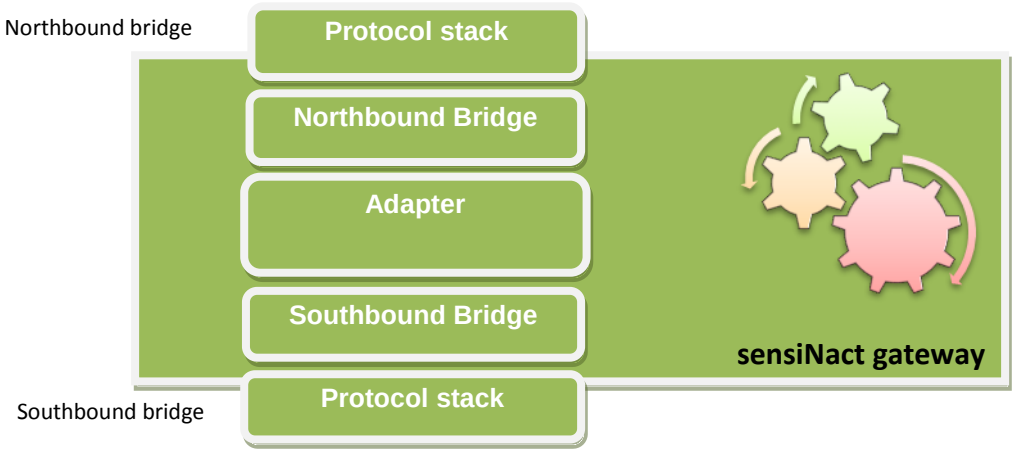

FIGURE 37: SENSINACT BRIDGE

The different syntactic interoperability steps that are the verification, the interpretation and the transformations imply developments depending on the protocol in use. For example, the XMPP Bridge uses a Keio's library which is mapping-objects oriented, allowing bypassing the verification step and making easier the translation into the inner-system format.

Others communication protocols used in the ClouT platform imply to define a a more complex transformation process : The Universal Service Description Language (USDL) described in the section 7.1.3.1 of this document is based on XML and allows specifying separately a service and an instance of it, leading to a loose coupling between definition and implementation. It is a rich format offering a wide range of predefined tags for properties, interface, preference, access and state definition. The choice of using JSON & JSON-SCHEMA

to format inner-system exchange data implies to define a transformation process that allows integrating devices using USDL in the system.

A type document allows specifying common properties for all inheriting instance of the described service, as well as accessible methods. For the particular case of the system registering of a device using USDL, the transformation of transmitted data will be made by the way of successive steps, the first being a JSON translation of the USDL associated type document framed by the JSON-Schema below:

```
{
 "id": "http://fr.cea.sensinact/usdl/resource#",
 "$schema": "http://json-schema.org/draft-04/schema#",
 "type": "object",
 "properties":{
  "id":{
   "type":"string",
   "description":"the resource's type identifier"
  },
  "name":{
   "type":"string",
   "description":"the resource's type name"
  },
  "name-lang":{
   "type":"string",
   "description":"the resource's type name used language"
  },
  "provider":{
   "type":"string",
   "description":"the resource's type provider"
  },
  "version":{
   "type":"string",
   "description":"the resource's type version"
  },
  "date":{
   "type":"number",
   "format":"utc-millisec",
   "description":"the resource's type definition timestamp"
  },
  "description":{
   "type":"string",
   "description":"the resource's type description"
  },
  "description-lang":{
   "type":"string",
   "description":"the resource's type description used language"
  },
  "interface":{
   "type":"array",
   "minItems": 0,
```

```
    "items": { "$ref" : "#/definitions/operation"},
    "description":"the resource's type defined operation(s)"
   },
  "map":{
     "type": "array",
     "items": {
     "type" : "object",
     "required" : ["type","resource"],
     "properties" : {
           "type" : {
              "type":"string",
              "format":"uri"
           },
          "resource":{
             "type":"string",
             "format":"uri"
           }
        }
     },
     "description":"mapping between the type model and the resource one"
   }
},
"definitions": {
   "operation" :{
      "type": "object",
      "required": [ "name", "sequence" ],
      "properties":{
           "name":{
             "type":"string",
             "description":"the operation's name"
           },
          "sequence":{
              "enum":["in","out","int-out"],
              "description" : "the operation's sequence type"
           },
           "description":{
              "type":"string",
              "description":"the operation's description"
           },
           "description-lang":{
              "type":"string",
              "description":"the operation's description used language"
          },
          "input":{
             "type": "object",
             "required": [ "type","pattern" ],
             "properties":{
             "type":{"type":"string"},
             "pattern":{"type":"string"}},
             "description":"the operation's input definition"
          },
         "output":{
```

```
            "type": "object",
            "required": [ "type","pattern" ],
            "properties":{
                    "type":{"type":"string"},
                    "pattern":{"type":"string"}},
                   "description":"the operation's output definition"
          },
          "consume":{
            "type":"string",
            "description":"the operation consumption"
          }
        }
      }
    }
}
```

It is important to notice that the map entry is in fact the list of uri couples allowing the mapping of a type object property into an inner-system resource one.

And so, according to this schema it is possible to convert this USDL type example:

```
<?xml version="1.0" encoding="UTF-8"?>
<type>
        <properties>
                <id>jp.ac.keio.sfc.ht.Device</id>
                <name lang="en">generic device</name>
                <provider>Jin Nakazawa, Keio University, ...</provider>
                <version>1.0.0</version>
                <date>112347192</date>
                <description lang="en">a generic device with a power
                control</description>
        </properties>
        <interface>
                <operation name="turn on" sequence="in-out">
                <description lang="en">turns the power on, and sends
                the result</description>
                <input>
                        <data type="mime">.*/.*</data>
                </input>
                <output>
                        <data type="mime">.*/.*</data>
                </output>
                <consume>energy</consume>
                </operation>
        </interface>
</type>
```

Into this JSON formatted representation:

```
{
  "$schema": " http://fr.cea.sensinact/usdl/resource#",
```

```
 "id" : "jp.ac.keio.sfc.ht.Device",
 "name" : "generic device",
 "name-lang" : "en",
 "provider" : "Jin Nakazawa, Keio University",
 "version" : "1.0.0",
 "date" : 112347192,
"description" : "a generic device with a power control",
"description-lang": "en",
"interface":[
   {
    "name" : "turn on",
    "sequence" : "int-out",
   "description" : "turns the power on, and sends the result",
   "description-lang": "en",
   "input" : {
       "type" : "mime",
       "pattern" : ".*/.*"
    },
   "output" : {
       "type" : "mime",
       "pattern" : ".*/.*"
    },
"consume": "energy"
   }
] ,
"map":[
   { "type" : "#/id", "resource" : "#/resourceID" },
   { "type" : "#/name", "resource" : "#/attributes/name" },
   { "type" : "#/name-lang", "resource" : "#/attributes/name/metadata/name-lang" },
   { "type" : "#/provider", "resource" : "#/attributes/provider" },
   { "type" : "#/version", "resource" : "#/attributes/version" },
   { "type" : "#/date", "resource" : "#/attributes/date" },
   { "type" : "#/description", "resource" : "#/attributes/description" },
   { "type" : "#/description-lang", "resource" :
"#/attributes/description/metadata/description-lang" },
   { "type" : "#/interface/$op/name", "resource" : "#/methods/turn-on/name" },
   { "type" : "#/interface/$op/sequence", "resource" : "#/methods/turn-on/type"},
   { "type" : "#/interface/$op/input", "resource" : "#/methods/turn-on/parameters" }
 ]
}
```

The transformation needs more information to be automatized:

- mapping between a type's sequence attribute and the type of the resource's method;
- as well as the mapping between type's input tag into parameters for the resource's method.

Most of the resource model can be completed without more information.

Finally, and in order to be compliant, with the other alternative languages shown in specification section, different adapters will be implemented to transform directly from the JSON-schema, or from the final USDL description.

5.2.2.2. Semantic Interoperability - Metadata

For the semantic interoperability, the first attempt from the project will be to adopt the QUDT vocabulary. Data converters will be created for data conversion chain by referring metadata for sending and receiving entities. Following shows a basic example on how this vocabulary could be used in the ClouT case. Under "data-type" keyword, there are semantic descriptions specifying measuring quantities and their units, using predefined ontology 'QUDT'. Regarding batteryCharge, there are no definitions in QUDT, so that we introduced new ontology under ClouT domain.

```
{
	"idntifier" : "*.urn",  // JSON path to identifier.
	"multi-sensors" : false,
	"multi-observations" : true,

	"observation" : "*.observations[*]",
	"timestamp" : "*.observations[*].timestamp",

	"observation-metadata": [	// relative to Observation Object
		"timestamp",
		"latitude",
		"longitude"
	],

	"measurements":"*.observations[*].measurements",//	JSON-PATH	to	each
measurement

	"data-value": "value",
	"value-syntax" : "string"

	"data-type":
		{
			"label-name": "type",
			"temperature":
"http://qudt.org/vocab/quantity#ThermodynamicTemperature",
			"batteryCharge":"http://clout-project.eu/data-
type#batteryCharge",
			"luminosity": "http://qudt.org/vocab/quantity#luminosFlux"
		}
	"data-unit": {
		"label-name" : "unit",
		"celsius":  "http://qudt.org/vocab/unit#DegreeCelsius",
		"percent": "http://qudt.org/vocab/unit#Percent",
		"lumen": "http://qudt.org/vocab/unity#lumen",
	}
}
```

The implementation of the semantic aspects will be more deeply considered at the 3rd year of the project, thus the third version of the CIaaS will contain more details on how ClouT will manage the semantic aspects.

5.3. City Entity Virtualisation

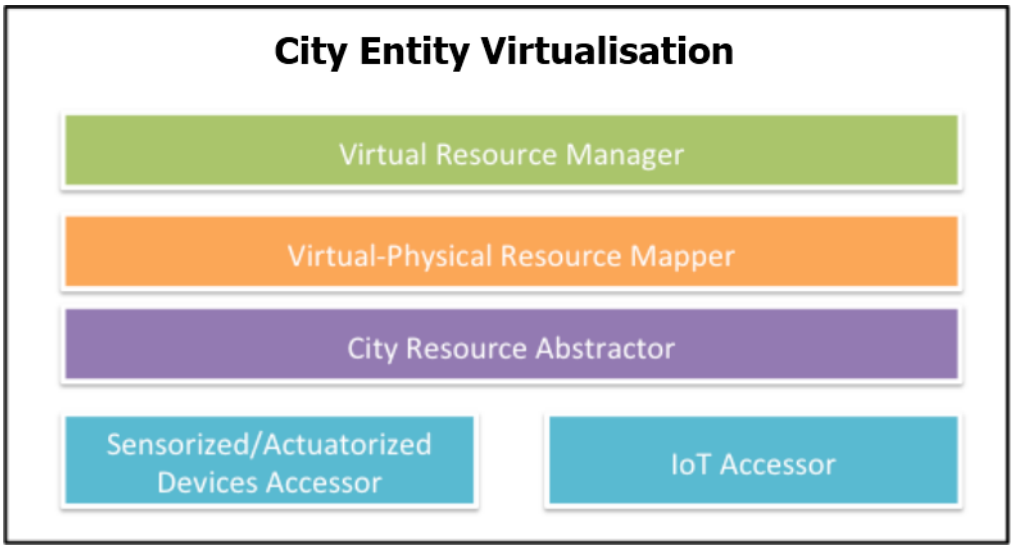

FIGURE 38: CITY ENTITY VITRUALISATION ARCHITECTURE

In Figure 38 we report a more detailed picture of the City Entity Virtualisation block, which is in charge of access to city resources. The architecture is logically split in four modules:

- Sensorized/Actuatorized Devices Accessor;
- IoT Accessor;
- City Resource Abstractor;
- Virtual-Physical Resource Mapper;
- Virtual Resource Manager;

5.3.1. Specification

The functionalities of the City Entity Virtualisation block are embedded in the gateway and cloud access deployments. We recall here the functions of the functional sub blocks as per the D1.3:

- The ***Sensorized/Actuatorized Device Accessor*** component is responsible for accessing sensorized/actuatorized devices according to requests from city resource abstractor. It directly communicates with Sensorisation & Actuatorisation layer.

- The ***IoT Accessor*** component is responsible for accessing IoT according to requests from city resource abstractor. It directly communicates with IoT Kernel layer.
- The ***City Resource Abstractor*** component is responsible for abstracting physical city resource as virtual city resources. The abstraction is achieved based on each city resources' property such as type, capability, location, etc.
- The ***Virtual-Physical Resource Mapper*** component is responsible for mapping between virtual city resources and city physical resources. Upper layer's applications which use virtual city resources do not have to care about actual physical resources for their application. This component provides appropriate physical city resources according to virtual city resources which are used in upper layer's applications.
- The ***Virtual Resource Manager*** component is responsible for managing virtual city resources. It manages both one-to-one mapped virtual resources which are abstracted through city resource abstractor, and user-defined virtual resources which are possible to be mapped to both single physical city resources and multiple physical city resources (device combination).

We report in the following section the two main implementations that cover aspects related to the virtualization task.

5.3.2. Implementation

5.3.2.1. SmartSantander virtualization module

The SmartSantander virtualization is considered as an extension (mainly at gateway level) of the basic capabilities offered by subjacent resources, offering four different types of virtualization:

- Resource combination: virtualized resource in a repository;
- Resource abstraction: new combined sensors that could be offered to other users (with the possibility of being also virtualized in the repository);
- Mobile sensor: Mobile phones or tablets can be virtualized, in this way citizens/users will behave as virtual sensors (with the possibility of being also virtualized in the repository);
- Sensor history: Information retrieved by both real and virtual resources, mining or aggregation over the historical data already stored.

Next figure shows the different modules used for carrying out virtualization.

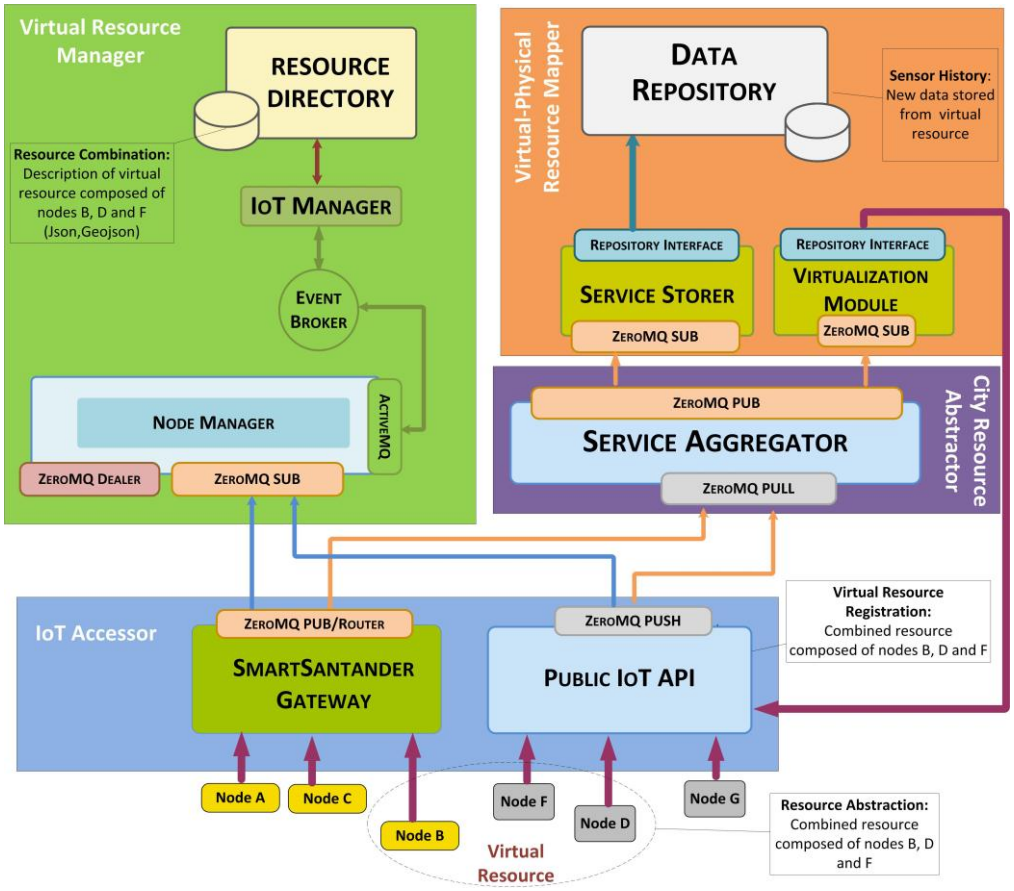

FIGURE 39 : VIRTUALISATION ARCHITECTURE

As it can be derived from the figure, the procedure to carry out the virtualization is as follows:

1. SmartSantander Gateway and Service Proxy are in charge of gathering information from SmartSantander deployment (through the corresponding supported interfaces), as well as external devices (accessing through the SmartSantander IoT API).

2. Service Aggregator will aggregate the data retrieved by the available resources, thus sending this data to the corresponding service storer, as well as to the virtualization module (when required).

3. Virtualization module is in charge of subscribing to the corresponding nodes that are composing the virtual device (in this case nodes B, D and F), thus aggregating the information (average, sum, correlation) from these devices accordingly. Additionally, the virtualization module also receives information from historical values provided by the data repository.

4. Once generated the virtual resource, this would inject data to the service proxy, thus accessing to the node manager, event broker and IoT manager in order to assign the corresponding unique identifier to this new virtual device, as well as updating the resource directory including the description of this resource.

5. Once registered, this new virtual resource will feed the service aggregator and, consequently the service storer, which will be in charge of storing the data associated to this virtual service into the data storage.

Considering this feedback procedure, the creation of new virtual resources and, consequently, new services associated to either these new resources or a combination of them, offers the possibility of a great portfolio of functionalities.

5.3.2.2. sensiNact virtualization module

In Figure 40**Erreur ! Source du renvoi introuvable.** we report the graphical interaction offered by the sensiNact gateway implementation, when deployed in the cloud.

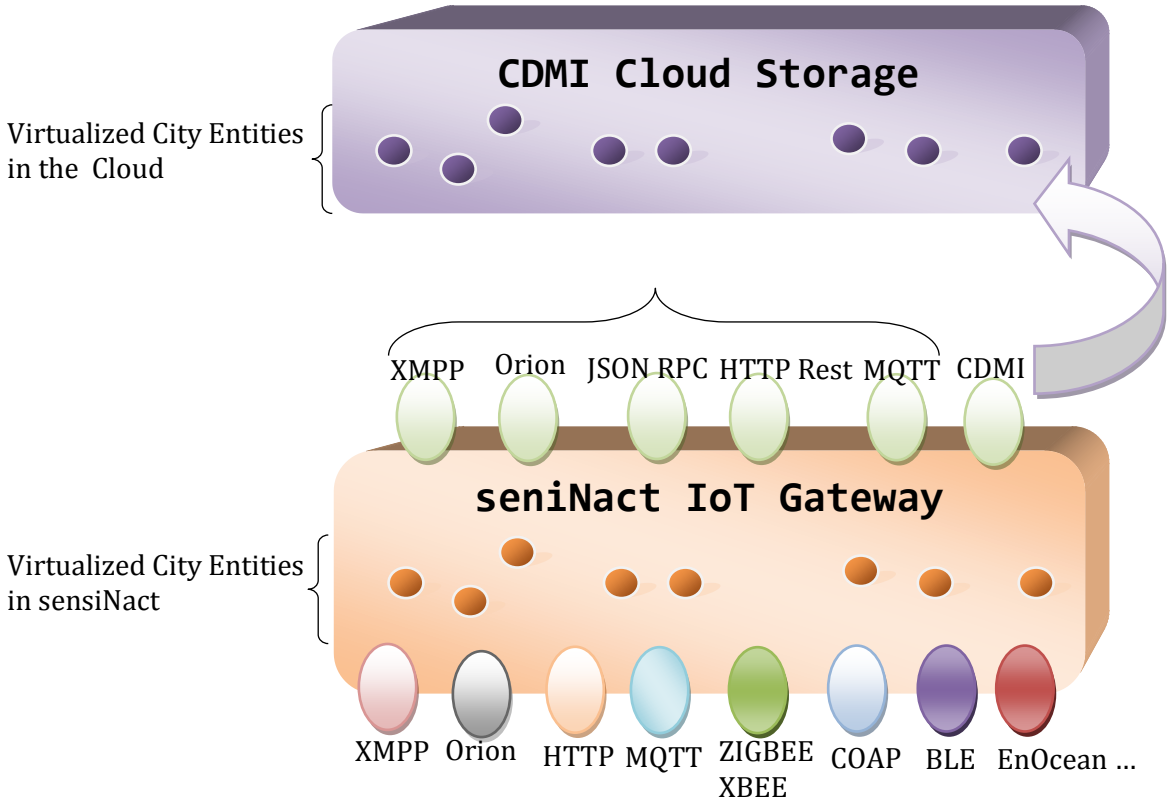

FIGURE 40 : SENSINACT INTERACTION WITH THE CLOUD

Further details on the Virtualization task as per the sensiNact implementation are currently given in section 7.3 (City Entity Management).

5.4. Opportunities/Threats Analysis

The following subsection provides 2 tables: 1 table clarifying the components that will be reused / extended from the state of the art and the components that will be developed in the project, and the 2nd table giving opportunities/threats analysis, for each component solution.

In Table 27 we report, for each functional block of Interoperable City Data, the reusable component that can be used to implement it.

TABLE 27 : [INTEROPERABILITY & CITY RESOURCE VIRTUALISATION] EXTERNAL COMPONENTS VS CLOUT DEVELOPED COMPONENTS

Functional block	External component to be reused or extended	Components being developed in the ClouT
Data Converter		ClouT is developing new converters (in Java and Javascript) to provide a semantic coherence between different data representations.
Data Transformer	JSON format for data serialization	ClouT is developing transformers between different data models used in the project (eg from USDL to ClouT model)
Syntax Interpreter		Mapping between the syntax of protocols in use and the inner-system one
Syntax Verifier	Syntax verifier is a sub-component of a bridge adapter (cf. sensiNact bridge [D4.2]) and depends on the protocol format (encoded array of bytes, raw string, xml, json, etc…), and on the device/system the sensiNact gateway is connected to. Some used libraries wrapped the communication and handle the verification (Keio's XMPP library); Some protocols do not need a verification (different from the one dedicated to the inner-system format) as allowing to freely format the transmitted message (MQTT);	Specific developments have been needed for some protocols (EnOcean reusing the open source protocol stack)

S/A Device Accessor	sensiNact gateway	Northbound bridge protocols dependent as a first level of request interpretation (authentication/validation/redirection)
IoT Accessor	OSGi registry and LDAP formatted requesting	Northbound bridge protocols dependent as a first level of request interpretation (authentication/validation/redirection)
City Resource Abstractor	SmartSantander Gateway and IoT API	Virtualization module
Virtual-Physical Resource Mapper	Resource Directory	Virtualisation module
Virtual Resource Manager	SmartSantander Resource Manager	Virtualisation module

In Table 28 we summarize opportunities and threats for the identified components.

TABLE 28: [INTEROPERABILITY & CITY RESOURCE VIRTUALISATION] OPPORTUNITIES VS THREATS ANALYSIS OF REUSABLE COMPONENTS

Solution	Corresponding ClouT components	Opportunities	Threats	Notes
Virtualisation module	City Entity Virtualisation	Create and register new virtual nodes covering new functionalities.	Could be difficult to dynamically manage these new virtual devices with the IoT resource manager.	To be adapted within the ClouT project.

5.5. Security Considerations

Since conversion function is composed with several conversion components, there is a risk of including malfunctioning components in the conversion function. If it happens, we can not trust on the correctness of data, and system crash may happen in worse case.

As a preventing method there are some approaches.

1. Authentication and Authorization: Only trusted member can have their account and submit the conversion component to the component repository.
2. Test and Validation: Someone execute test of the component and validates the component. It is also one approach to check source code of the component.
3. Robust Receiver code: It is important to ignore data if a certain subsystem receives data with illegal format.

In the case of SmartSantander gateway, currently virtualization would be restricted to network managers (authenticated against the platform).

6. COMPUTING AND STORAGE

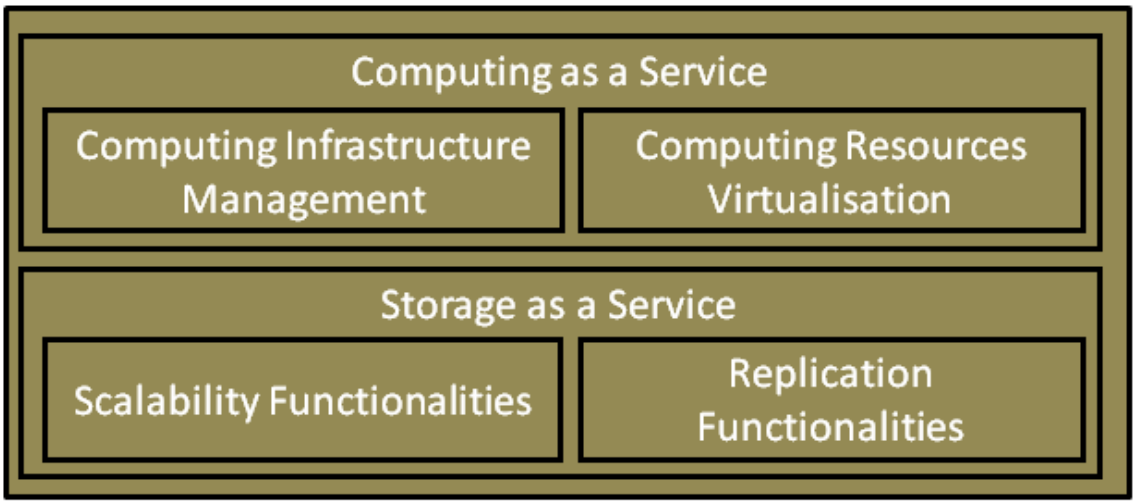

FIGURE 41 : COMPUTING AND STORAGE ARCHITECTURE

6.1. Storage as a Service

6.1.1. Architecture

ClouT Storage module is composed by three elements (Figure 42):

- CDMI Interfaces, provided by a web service deployed on Tomcat
- CDMI Service Gateway, which processes the requests and forwards the data to the Persistence Module (composed by OpenStack Swift and Hypertable)
- CDMI Storage, to store textual data, object data and metadata

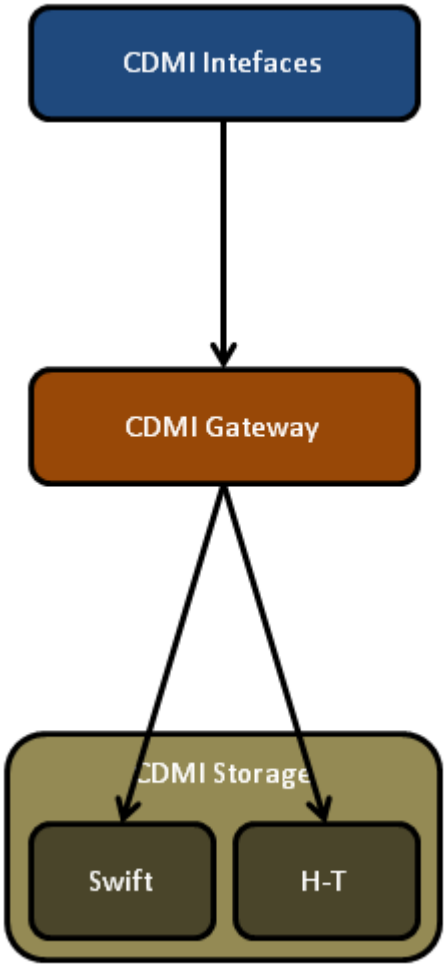

FIGURE 42 : ARCHITECTURE OF CLOUT STORAGE

CDMI Interfaces receives a message and, through a security filter implemented by ClouT security framework (Figure 44) check if the security policies are satisfied. Then the message is forwarded to CDMI Gateway which, according to the content type, takes decision about how the CDMI storage has to store the message. In particular, CDMI storage module is composed by two storages with different features. Metadata and textual data, such as temperatures, traffic levels or pressure are stored by Hypertable (H-T, in the picture), while data objects, such as images, files, e-mails, are stored by OpenStack Swift.

Figure 43 shows how the modules composing ClouT storage can be mapped in the Reference architecture. In particular CDMI interfaces are part of CPaaS layer and are described in D3.3, CDMI Gateway is part of City Infrastructure Management module and Swift and Hypertable are included in Computing and Storage block.

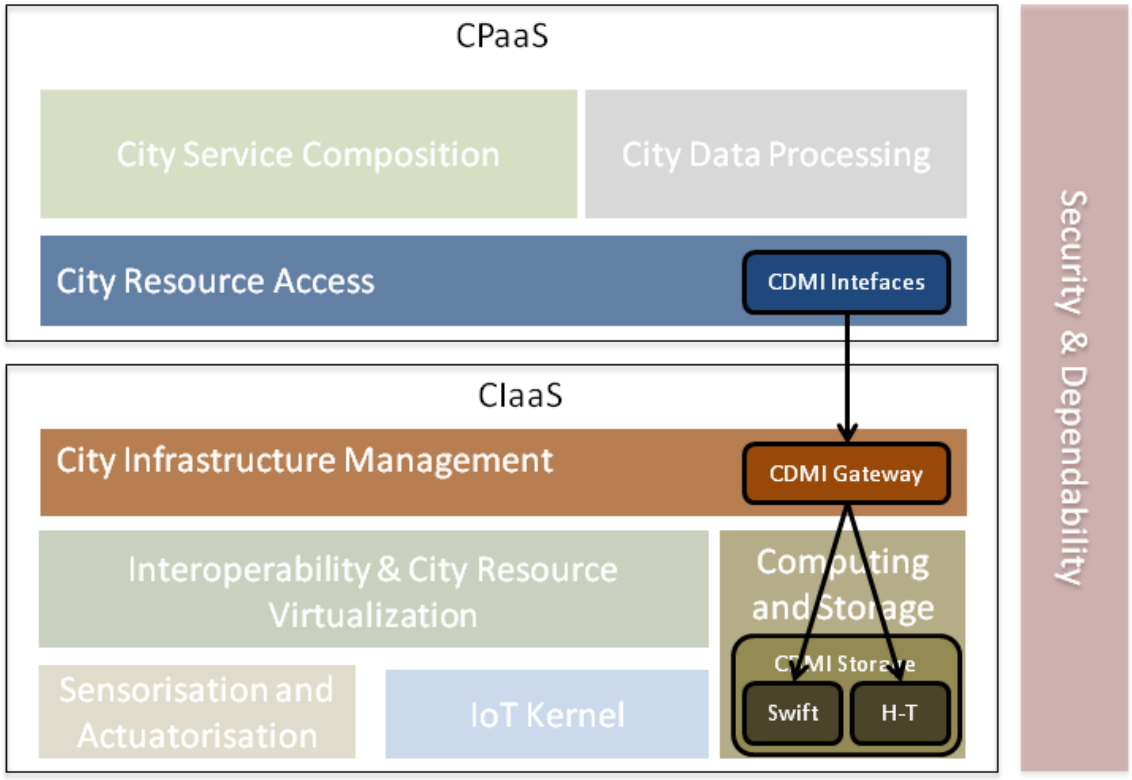

FIGURE 43 : CLOUT STORAGE IN THE REFERENCE ARCHITECTURE

In this section the specification and the implementation of CDMI Storage and CDMI Gateway will be provided: CDMI Interfaces, since are part of CPaaS layer, are described in D3.3.

6.1.2. Specification

An example of information flow is the following:

1. CDMI interfaces receives the CDMI PUT message containing data and credentials
2. The security framework checks the credentials and forwards the message to CDMI gateway
3. CDMI Gateway receives a REST PUT multipart message containing the file and metadata to be stored
4. CDMI gateway decodes the message and forwards the data object (file) to Swift and metadata to Hypertable
5. Swift and Hypertable store the assigned data

6. CDMI gateway responses 200 (OK) if the storage process is successfully complete, otherwise 401 (Not Succeed).

Swift plays a role in the flow only if a data object is included in the request: as data object is intended a large binary object (such as an Oracle BLOB), containing binary information. An image file, for instance, is a data object. ClouT storage can store information not containing data objects, for example a temperature: in this case only Hypertable plays a role in the process. Hypertable enables more efficient indexing and more consistent and optimized data organization, useful for textual data and metadata.

Figure 44 reports the communication diagram of ClouT Storage. ClouT Security Module, implemented by the Security Framework, acts as a filter between CDMI interfaces and CDMI Gateway checking the credentials on incoming requests and blocking unauthorized ones. This filter works only for requests coming from external applications (e.g. an application of CSaaS layer which needs to access a set of temperatures): security is not needed for internal request (e.g. temperatures coming from CIaaS layer to be stored).

ClouT CDMI Storage exposes APIs to external world through CDMI interfaces, which are considered as part of CPaaS layer. Details about these APIs are available on the deliverables about CPaaS layer. The current specification of ClouT does not provides direct access to the storage implementations, however, more information on storage management APIs can be found in Openstack documentation (http://developer.openstack.org/api-ref.html) and Hypertable Thrift APIs documentation (http://hypertable.com/documentation/developer_guide).

Relevant system requirements, as from [D1.3]: REQ_CIAAS_1, REQ_CIAAS_10, REQ_CIAAS_12, REQ_CIAAS_13, REQ_CIAAS_34.

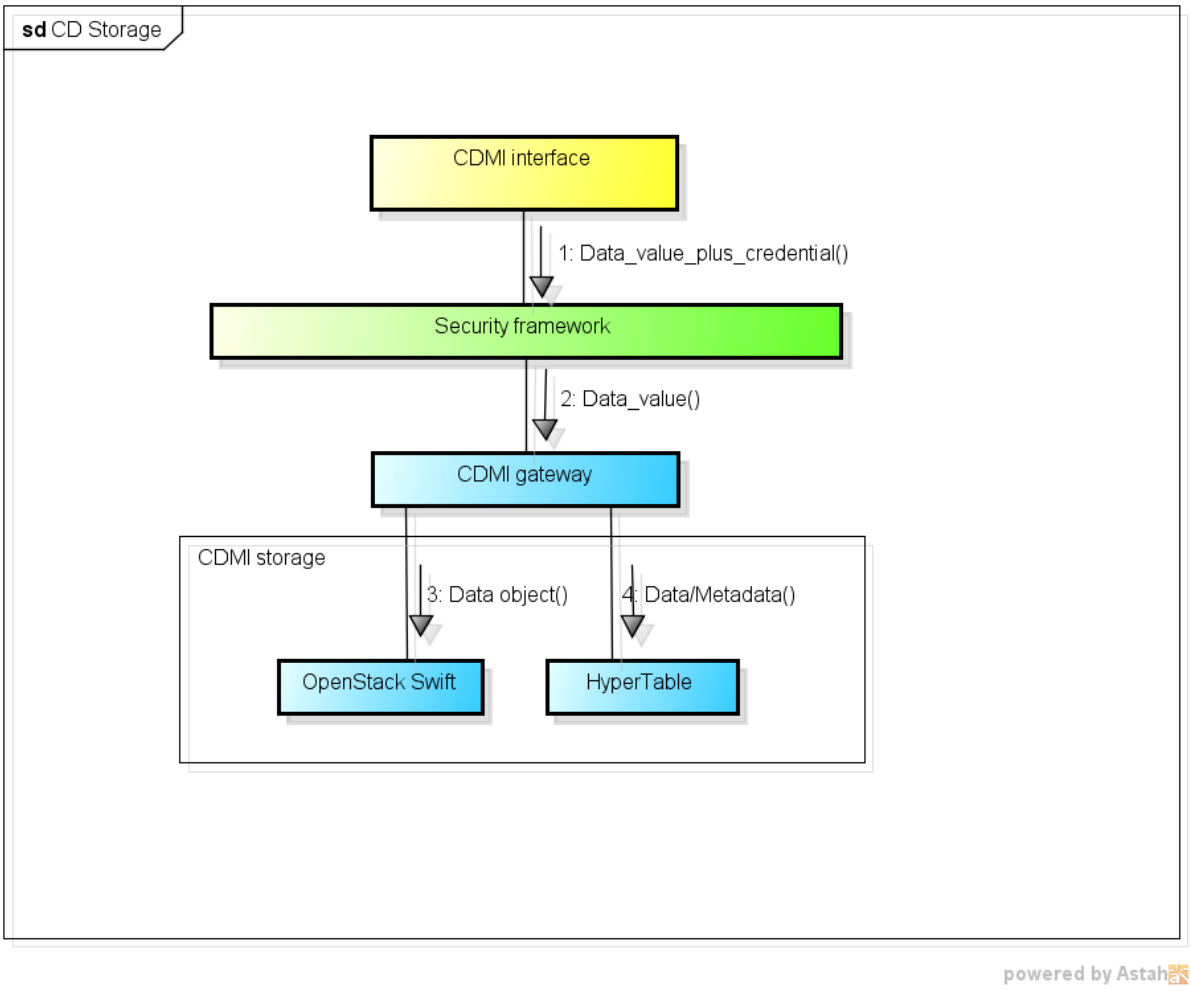

Figure 44 : COMMUNICATION DIAGRAM - STORAGE

6.1.3. Implementation

6.1.3.1. CDMI Gateway implementation

Figure 45 shows the activity diagram of CDMI Gateway. The module receives CDMI messages just after the security checks performing the following operations:

- Analysis of the headers, in particular Content-Type, that may be *multipart/mixed* or *application/cdmi-object*. The content type does not provide directly the nature of the data of the payload, but provides information on the structure of the message. Multipart/mixed messages are composed by a first part with a JSON object containing metadata and a second part with a data object in bytes; a content of cdmi-object type has only a single JSON object with data and metadata (the structure of the JSON object is described in D3.3).

- Multipart/Mixed messages are broken down into two parts: metadata cdmi object and data object. The first one is sent to Hypertable Database, the second one is sent to Swift
- Cdmi-object data messages are parsed for retrieving data and metadata: both are sent to Hypertable in different tables.

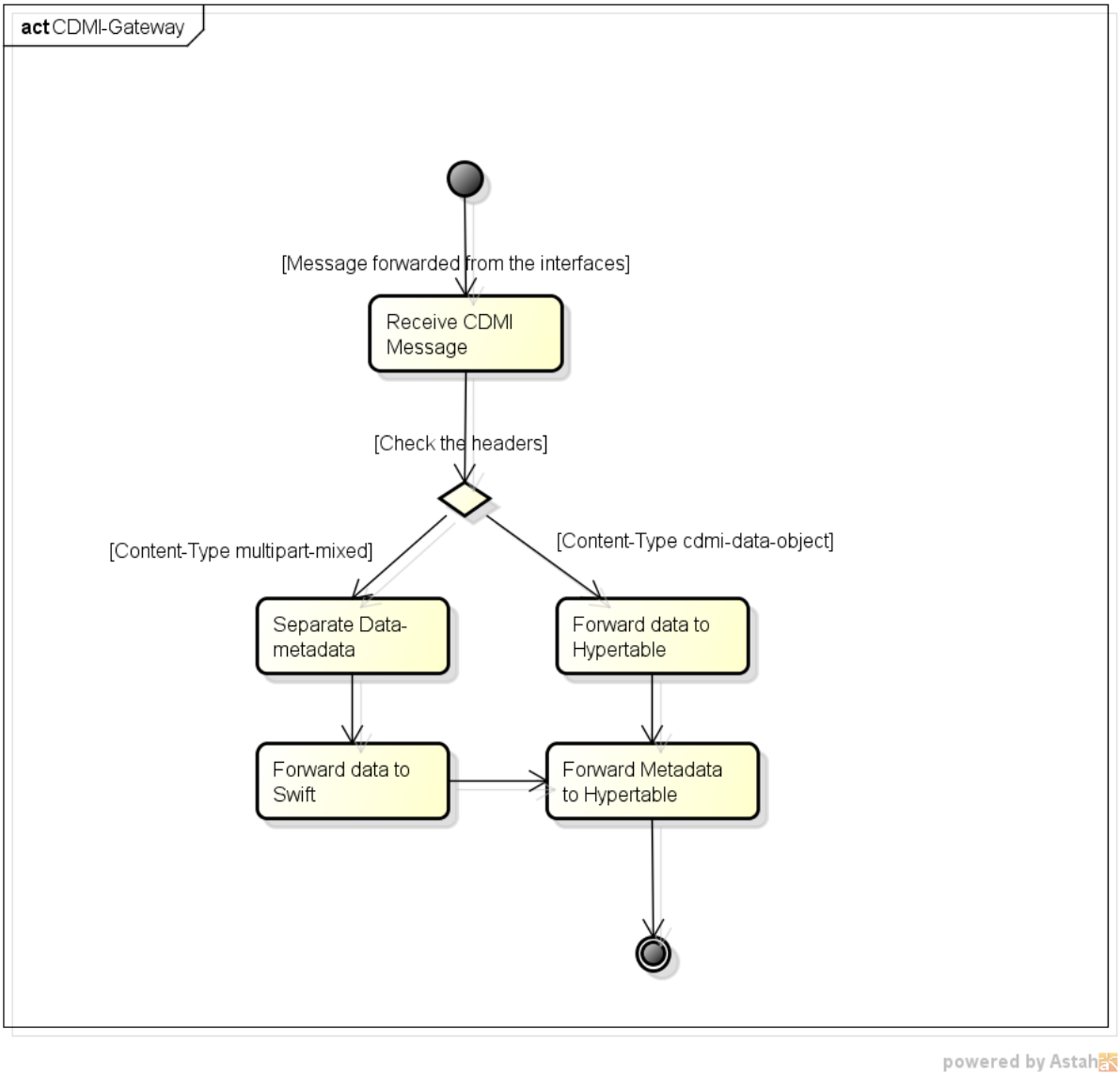

FIGURE 45 ACTIVITY DIAGRAM OF CDMI GATEWAY

6.1.3.2. CDMI Storage implementation

As described in the previous section, ClouT Storage is composed by two storage modules with different features:

- OpenStack Swift to store data objects
- Hypertable to store textual data and metadata.

JSON objects are parsed and their data are forwarded to CDMI storage according to the rules described in section 6.1.3.1.

Open Stack Swift is a multi tenant, highly available, distributed object storage. It is designed to store big amount of data and exposes RESTful APIs. Swift is open source, distributed under Apache 2 license.

It can be used as a stand-alone storage or as part of a Cloud compute environment: it supports object replication for reliability. The addition and removal of nodes is easy and does not involve downtimes: this provides scalability, elasticity and fault tolerance.

Swift is a good solution to store unstructured data that could grow without bound. Furthermore Swift is the storage of OpenStack. OpenStack is a cloud operating system, able to control computing, storage, and networking resources. For this reason Swift is intrinsically linked to the Cloud environment providing Cloud computing services of ClouT.

Swift is by itself a complete object storage, able to store all types of files (images, e-mails, virtual machines...), providing high availability and reliability through replication and redundancy. Swift is also horizontally scalable, by easily adding new servers.

Apache **Hypertable** is a NoSQL database supporting scalability: as a NoSQL databases it represents data as tables of information with UTF-8 strings as keys. The key is the only primary key referencing data sets seen as byte string. Scalability is implemented by automatic data migration across multiple server, even if they are dynamically added. In other words Hypertable is able to detect that there are new servers and new capability available and will migrate ranges from an overloaded machine to the underloaded ones. The data migration also balances the load across the involved machines.

As a NoSQL database, Hypertable also supports some features provided by relational databases: in particular it represents data as tables of information with rows, uniquely identified cells and row keys. It also offers good performances also with big tables. In particular a comparison with HBase (http://hypertable.com/why_hypertable/hypertable_vs_hbase_2/) shows that Hypertable significantly outperformes HBase in several tests.

TABLE 29 : NUMBER OF QUERY PER SECOND AND LATENCY COMPARISON WITH A ZIPFIAN LOAD DISTRIBUTION

Dataset Size	Hypertable Queries/s	HBase Queries/s	Hypertable Latency (ms)	HBase Latency (ms)
0,5 TB	7901.02	4254.81	64.764	120.299
5TB	5842.37	3113.95	87.532	164.366

Table 29 shows the result of a comparison test between Hypertable and HBase with a *Zipfian* load distribution, that is a realistic workload model. The test has shown that Hypertable has better performances in term of numbers of Queries per second and latency. Other tests can be found on Hypertable portal.

The functional and non-functional considerations above led us to choose Hypertable as NoSQL database to host metadata and sensor data in ClouT.

6.1.3.3. SmartSantander data storage

Data retrieved from nodes composing SmartSantander platform is stored and retrieved, though the use of three FI-WARE generic enablers: DCA-IDAS Backend Device Management, Orion Context Broker and Cosmos Big Data Analysis. Next figure shows the interaction among these three components in order to store the corresponding information.

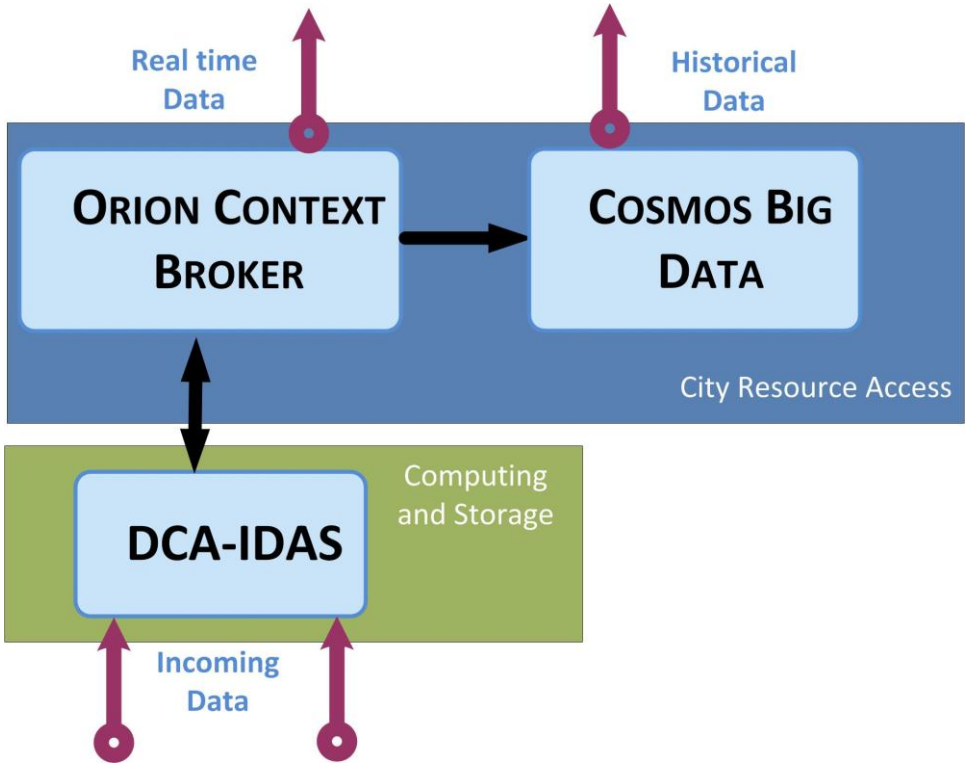

FIGURE 46 SMARTSANTANDER DATA STORAGE

As it can be observed in the figure, the data retrieved from the SmartSantander platform is stored in the DCA-IDAS module. This data is offered to external users using either Orion Context Broker or Cosmos Big Data. Orion provides an interface for extracting the last value associated to all the resources (both real and virtual), as well as feeds the Cosmos module with this information. In the Cosmos module historical values associated to all the available resources are stored, thus providing the corresponding access interface for an external user for requesting

them. In this sense, all this data could also be injected within CDMI storage using the corresponding interfaces indicated in the previous sections.

6.1.3.3.1. DCA-IDAS Backend Device Management

DCA-IDAS[1] is an implementation of the FI-WARE Backend Device Management GE that collects all observations of devices and translates them into NGSI events, available at the Orion Context Broker GE instance. This way, application developers are able to consume devices' observations and context data through the ContextBroker enabler. IDAS provides a Device Gateway in the form of a Device Communication API for devices (sensor/actuators/gateways) communication that currently supports SensorML and Lightweight SensorML protocols. This API is used by the SmartSantander Service Storer components to upload observations.

IDAS includes also its Admin REST API, provided to M2M application developers, that exposes the following functional blocks:

- Provision: Configuration of internal components.
- Data Model: Operations related to M2M entities (register/modify resources). Top level Object = Service. Part of this API is used by the SmartSantander Resource Configurator to register and update devices.
- Sensor Data: Operations to retrieve the last/historical measures (observations) of devices.
- Subscription: Configure apps to receive data published by devices.
- Command Requests: Configure apps to receive data published by devices and receive their output. Commands are formatted in XML.

Additionally, it provides a NGSI interface that exposes the following operations:

- QueryContext (NGSI 10, incoming message): Sent by an NGSI-enabled component, transformed to a query to DCA-IDAS database (last/historical measure).
- UpdateContext (NGSI 10, incoming message): Sent by an NGSI-enabled component. Information is parsed and transformed into a command request.
- RegisterContext (NGSI 9, outgoing message): When an SML RegisterSensor is received, a RegisterContextRequest is sent over NGSI9 (signaling).
- UpdateContext (NGSI 10, outgoing message): When an SML InsertObservation is received, an UpdateContextRequest is sent over NGSI 10 (data).

Regarding to the registration of new devices in the platform, it is carried out with the following command:

Registration message example
<rs><id href="1:1">urn:x-iot:smartsantander:u7jcfa:fixed:t93</id><id href="1:8">M04.urn:x-

[1] http://catalogue.fiware.org/enablers/backend-device-management-idas

```
iot:smartsantander:u7jcfa:fixed:t93</id>

 <param arcr="3:10" href="8:27" id="pgps" name="Location" role="3:15"><gps lat="43.46074"
lon="-3.80812"/></param>

<what href="urn:x-ogc:def:phenomenon:IDAS:1.0:temperature"/>

<what href="urn:x-ogc:def:phenomenon:IDAS:1.0:batteryCharge"/>

<data><quan uom="urn:x-ogc:def:uom:IDAS:1.0:celsius">0</quan></data>

<data><quan uom="urn:x-ogc:def:uom:IDAS:1.0:percent">0</quan></data>

</rs>
```

Both identifier and description of this new node will be generated by the resource naming and description modules, defined in D3.3. In this sense, the node provides temperature and battery sensing capabilities, being also included the location of the node.

In order to add a new measurement associated to this node, the procedure is shown next:

Observation message example

```
<io>

        <obs from="urn:x-iot:smartsantander:u7jcfa:fixed:t93">

                <stm>2014-03-26T03:10:45Z</stm>

                <what href="urn:x-ogc:def:phenomenon:IDAS:1.0:temperature"/>

                <param href="phenomenon:location">

                        <gps lat="43.46074" lon="-3.80812"/>

                </param>

                <param href="urn:x-ogc:def:phenomenon:IDAS:1.0:batteryCharge">

                        <quan uom="urn:x-ogc:def:uom:IDAS:1.0:percent">82.00</quan>

                </param>

                <data>

                        <quan uom="urn:x-ogc:def:uom:IDAS:1.0:celsius">9.61</quan>

                </data>

        </obs>

</io>
```

As it can be observed, it consists of an XML-based observation including values of location (same value as in registration as it is a fixed node), battery and temperature and the timestamp when measurement was taken.

6.1.3.3.2. Orion Context Broker

Orion Context Broker[2] presents an implementation of a context information broker with persistent storage that can play the role of two GEs within the FI-WARE platform.

- Pub/Sub Context Broker GE: Capacity to subscribe to different entities in order to get real time information published accordingly.
- Configuration Management GE: Capacity to register, update and remove context information associated to a determined entity.

In this sense, Orion Context Broker Handles information stored within DCA-IDAS, implementing both *OMA NGSI9 and OMA NGSI10* specifications. NGSI-9 deals with context information availability (registerContext, discoverContextAvailability, subscribeContextAvailability…), while NGSI-10 deals with context information itself (updateContext, queryContext…).

Through the use of the aforementioned NGSI-9 and NGSI-10 interfaces, there can be performed the following operations:

- *Register context producers*: Define the addition of an entity with a set of capabilities, i.e. a temperature sensor within a room
- *Update context information*: Update of the values associated to a determined entity, i.e. sending of temperature updates
- *Query entity context information*: This type of operations allow to get the set of attributes associates to an entity, i.e. sound, temperature, …; as well as the current values associated to this set of attributes.
- *Discover contest information*: It retrieves the current available entities with their attributes and the last values associated to each of these attributes.
- *Subscriptions/notifications*: Provide the ability to subscribe to context information when an event occurs (i.e., associates to a time interval, a change in the attribute value), receiving the subscribed application (specific IP and port), the corresponding asynchronous notification where appropriate.

All the specific functionality associated to this module is specifically detailed within D3.3, associated to the Context Management and City Data Access blocks.

6.1.3.3.3. Cosmos Big Data analysis

Cosmos[3] is an implementation of the Big Data GE, allowing the deployment of private computing clusters based on Hadoop ecosystem. Current version of Cosmos allows users to:

- I/O operations regarding Infinity, a persistent storage cluster based on HDFS.
- Creation, usage and deletion of private computing clusters based on MapReduce and SQL-like querying systems such as Hive or Pig.
- Manage the platform, in many aspects such as services, users, clusters, etc. from the Cosmos API or the Cosmos CLI.

For this purpose, they are used the following modules:

[2] http://catalogue.fiware.org/enablers/publishsubscribe-context-broker-orion-context-broker
[3] http://catalogue.fiware.org/enablers/bigdata-analysis-cosmos

- HDFS as its distributed file system.
- Hadoop core as its MapReduce engine.
- HiveQL or PIG for querying data.
- Oozie as remote MapReduce jobs and Hive launcher.

There exist a component called Cygnus in charge of receiving data from Orion and storing it in Cosmos HDFS file system, thus being gathered historical information of all the resources within the Cosmos module. In this sense, access to historical data could be carried out either by implementing a hive client and accessing using the corresponding HiveQL sentences or running a Hadoop MapReduce application. As an example, next it is shown a HiveQL sentence:

```
# SELECT * FROM opendata_smartcities_santander_smart WHERE
entity_name='urn:x-iot:smartsantander:u7jcfa:fixed:t93' ts BETWEEN
'2015-03-01T00:00:00' AND '2015-03-09T00:00:00';
```

In this case, they are being queried values associated to a specific node that were gathered between two defined timestamps.

6.2. Computing as a Service

6.2.1. Architecture

The *Computing as a Service* module provides computing capabilities with elasticity and flexibility.

The block is part of *Computing and Storage Module* and provides a reliable infrastructure layer and elastic resource virtualization. In particular, according to the needs of the upper layers, such as *City Data Processing* or *Applications*, the Computing as a Service layer optimizes the used resources reducing the costs for cities when less capabilities are requested.

The *Computing as a Service* paradigm represents one of the most important features to enable cities to shape their cost according to the actual use of resources. Figure 47 shows that the *Computing as a Service* block and implementation are totally included in Computing and Storage architectural module, i.e. it does not require any external feature or interface.

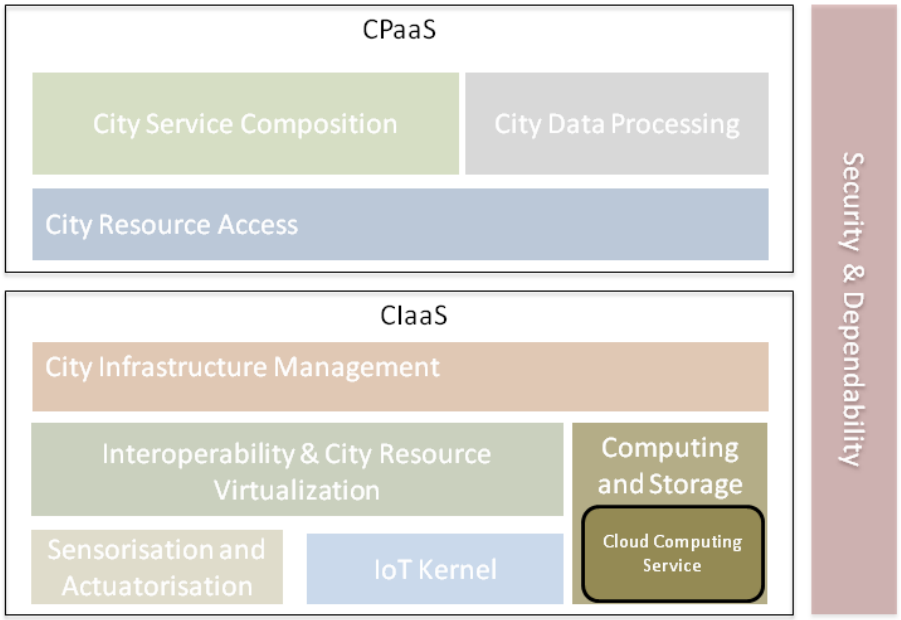

FIGURE 47 CLOUD COMPUTING SERVICE BLOCK

6.2.2. Specification

Computing as a Service role in ClouT architecture is assumed by OpenStack. OpenStack is a free and open source cloud operating System which provides several services, including Storage with Swift (section 6.1.3.2).

As Computing as a Service provider, OpenStack offers on-demand computing resources and manages a large set of Virtual Machines. Provided resources are accessible through APIs and horizontal scalability can be obtained by using standard hardware.

Figure 48 shows the whole deployment of ClouT Computing and Storage Module: it includes what has been described in section 6.1 about the storage and OpenStack computing modules.

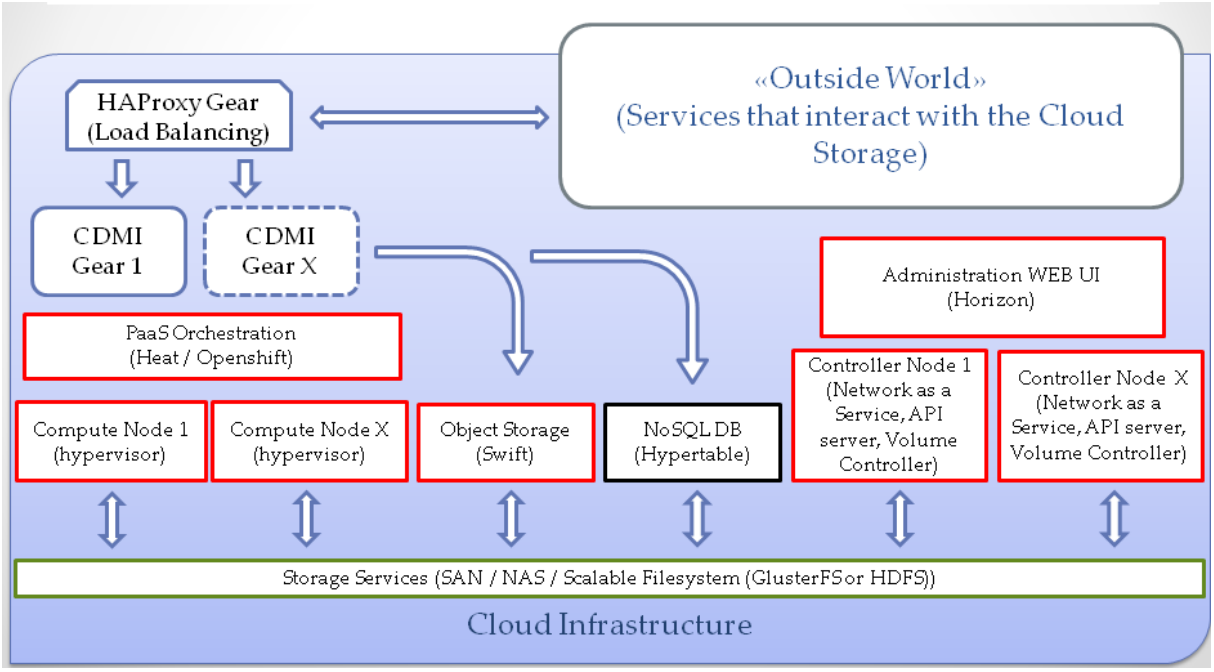

FIGURE 48 COMPUTING AND STORAGE DEPLOYMENT

Computing as a Service block does not expose public APIs to be accessed by clients external to ClouT architecture. It offers infrastructural services for providing the upper layers with computing power: everything is managed by the Administration WEB UI.

However OpenStack provides a set of API for developers, currently not used by ClouT: more information on these APIs are provided in http://developer.openstack.org/api-ref.html.

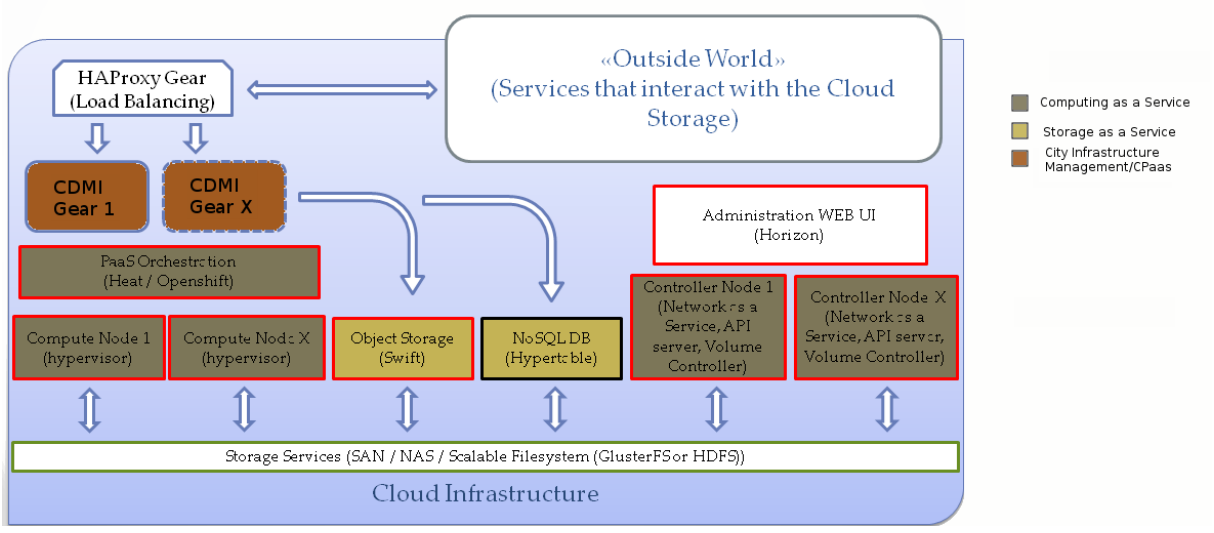

FIGURE 49 : COMPONENTS PROVIDING COMPUTING AND STORAGE FUNCTIONALITIES MAPPED ON THE ARCHITECTURE

Finally Figure 49 shows how the described implementation of computing and storage elements is mapped on the reference architecture. In particular the picture shows the elements described in the previous sections of this deliverable and what is described in deliverable 3.3, concerning CPaaS layer.

Relevant system requirements, as from [D1.3]: REQ_CIAAS_1, REQ_CIAAS_12, REQ_CIAAS_13, REQ_CIAAS_34, REQ_CIAAS_9

6.2.3. Implementation

Figure 48 shows the components of Computing and Storage block. The details of Swift and Hypertable modules are described in section 6.1.3.2: in this section more emphasis will be given to the modules concerning *Computing Infrastructure Management* and *Resources Virtualisation*.

ClouT's reference implementation is Openstack based first and foremost due to its scalability capabilities. Openstack is a modular software that is comprised of several distinct open source projects that are brought together as a complete Infrastructure as a Service solution. Due to its nature, any Openstack deployment may or may not include one or more specific modules; modules may even be deployed as standalone solutions. Figure 29 shows ClouT's reference implementation, but such infrastructure can be tailored to reflect specific municipality needs by adding or removing some of the elements that compose (or are external to) the reference implementation. In this paragraph only ClouT's reference architecture is going to be discussed.

The *Controller Node* provides the basis for ClouT's cloud infrastructure. It includes all the modules required to run and manage an Openstack based cloud infrastructure, such as Nova management (Compute/Virtualization module), Keystone (Identity module), Neutron server (software-defined networking module), Glance (image service module), Cinder (block storage module), a MariaDB SQL database instance to store the internal cloud infrastructure configuration (e.g. service endpoints information), Heat (the Orchestration module) and RabbitMQ (the message broker of choice). A *Controller Node* may optionally include the Telemetry module which monitors resource usage. The gathered resource allocation information can be employed for billing purposes. It also includes the administration web U.I. module known as *Horizon*, which enables the management of the Openstack infrastructure by administrators and the management of *Projects*, administrator-defined pools of infrastructure resources, to enabled users. Moreover, administrators can manage the cloud infrastructure through the Controller Node via module-specific clients written in python that interact with each module's RESTful APIs. Multiple Controller Nodes may be instantiated for high availability purposes.

The *Compute Nodes* are mainly tasked with the virtualization of the available computing resources. Aside from the bare metal itself and a minimal Linux O.S. distribution, each *Compute Node* is typically composed of instances of the agents of the Compute module (Nova) running on top of the hypervisor of choice. Supported hypervisors are KVM, QEMU, Xen, Hyper-V and VMWare, KVM having the bigger number of features supported by Openstack, therefore being the optimal choice. Software-defined networks are available through the Neutron agent. Other than making the software-defined networks and routers available to the virtual machines running on the *Compute Node*, it is worth noting that Neutron can be configured to provide network traffic isolation, therefore securing network traffic in multi-tenants environments.

Optionally *Compute Nodes* may include a Telemetry module agent to gather in-depth information circa resource usage on that specific node.

Storage Services include both block storage available through a storage area network, network attached storage and storage provided via scalable filesystems, e.g. GlusterFS, Hadoop's DFS, Ceph, depending on the municipality specific requisites and possibilities. ClouT architecture manages its Storage as a Service offering via *Storage Nodes*. *Storage Nodes* offer data redundancy and scalability via a multiple node deployment. *Storage Nodes* can be deployed on commodity hardware, possibly cutting costs related to storage by exploiting pre-owned on-premises hardware. In ClouT a *Storage Node* can be a Swift node (Object Storage) which, aside from ClouT's specific application, may also employed to store isos required to install an O.S. or run "ephemeral" virtual machines as a Glance module backend or other data. On the other hand a ClouT Storage Node can be a Hadoop DFS cluster which hosts Hypertable data. Block storage access is managed via Cinder module, which supports several backend technologies including iSCSI, Fibre Channel, NFS.

PaaS Orchestration is achieved through two main components: the Heat module, which provides service orchestration and Openshift, which provides the actual Platform as a Service functionalities. Heat is capable of deploying applications on an existing Openstack infrastructure by using ClouFormation compatible templates which describe the required Openstack resources that are necessary to deploy the application itself. Heat is the tool of choice to deploy instances of Openshift in ClouT's reference implementation. Openshift is a Platform as a Service solution which hosts cloud applications in a dynamically scalable environment. Openshift is architecturally composed of a broker and several nodes. Each node is broken into *gears.* Each *gear* represents a set of available resources. When a cloud application is deployed, a minimum and maximum number of *gears* are assigned to it. When the application consumes more than the minimum resources allocated due to an increasing workload, Openshift assigns more *gears* to it, therefore scaling it out dynamically. When scaling out, a High Availability Proxy is instantiated, effectively balancing the load between the various instances of the cloud application. Once the application workload decreases, for e.g. due to a decreasing number of requests, Openshift decreases the number of assigned *gears*, effectively shrinking the allocated resources, scaling them *in*. Another interesting feature is the use of *cartridges*. In Openshift, a *cartridge* can be a framework, a database, a web server, a connector and so on. *Cartrdiges* help ClouT developers by making a development/deployment environment easy to provision. In ClouT's reference implementation, an Openshift instance with an adequate number of nodes hosts instances of the CDMI compliant interface that exposes the Cloud Storage solution used to gather sensor data and expose it to the other reference architecture components.

6.3. Opportunities/Threats Analysis

In Table 30 we report, for each functional block of Computing And Storage, the reusable component that can be used to implement it.

TABLE 30: [COMPUTING AND STORAGE] EXTERNAL COMPONENTS VS CLOUT DEVELOPED COMPONENTS

Functional block	External component to be reused or extended	Components being developed in the ClouT
Computing Infrastructure Management	Openstack	Openstack is a modular software providing infrastructure management, computing and resources virtualization and storage
Computing Resources Virtualisation	Openstack	Openstack is a modular software providing infrastructure management, computing and resources virtualization and storage
Scalability Functionalities	Openstack Swift, HDFS, Openshift, Hypertable	Openstack Swift, which will be employed to store larger objects will support CDMI through CDMI Gateway and CDMI Interfaces; Hypertable provides scalabiliy functionalities for textual data
Replication Functionalities	Openstack Swift, HDFS, Hypertable	Openstack Swift, which will be employed to store larger objects will support CDMI through CDMI Gateway and CDMI Interfaces; Hypertable provides replication functionalities for textual data
Data storage	IDAS	Development of the mechanisms for including data retrieved from ClouT platform in this block.
Real time data request	Orion Context Broker	Subscription for the corresponding data generated by the entities/resources in the platform.
Historical data request	Cosmos Big Data	Request of historical data associated to a corresponding device, attribute and/or time period.

In Table 31 we summarize opportunities and threats for these platforms.

TABLE 31: [COMPUTING AND STORAGE] OPPORTUNITIES VS THREATS ANALYSIS OF REUSABLE COMPONENTS

Solution	Corresponding ClouT components	Opportunities	Threats	Notes
Openstack	All the components in Computing and Storage Block	Widely used open source cloud operative system that enable the control of large pools of cloud resources	A typical on-premises deployment of Openstack requires a minimum range of 6-10 physical servers.	Other IaaS solutions were evaluated, especially open sourced, notably oVirt/RHEV. However Openstack, with its modular approach, appears to be the most flexible and best suited solution.
Hypertable	Scalability Functionalities, Replication Functionalities	Hypertable is a NoSQL database providing scalability and good performances. It is a good solution to efficiently store sensor data	Nothing specific	
IDAS, Orion and Cosmos (GEs from FIWARE)	Data storage and access	FIWARE is a widely used platform offering a powerful set of APIs that ease the development	Nothing specific.	It is important to reuse components and results from other EU projects.

		of Smart Applications in multiple vertical sectors		

6.4. Security Considerations

A high level of security is required for all the operations related to Computing and Storage. At Computing level, a certain level of Authentication and Authorization is required to check the identities allowed or denied to manage and to access the virtual machines or set of services related to the virtual machines. At Storage level, data access should be checked with authentication and authorization and privacy should be ensured by data encryption.

The implementation elements chosen as components of ClouT Computing and Storage module provide several security features. In particular Openstack, including also Swift and other technologies related to infrastructure services and management, tries to address several different security issues. The specific security solutions proposed by Openstack team can be found in http://docs.openstack.org/security-guide/content/introduction-to-openstack.html.

In particular, for the Computing module supports instance isolation, secure communication between sub-components, and resiliency of public-facing API endpoints. The Storage module (Swift) provides a high degree of resiliency through data replication and can handle petabytes of data.

Openstack also provides Keystone, a shared Identity Management Service providing token based Authentication and Authorization.

Hypertable by default does not support encryption or strong authentication: these features should be added at application level.

Service Oriented Authentication, Authorization and Accounting (SOA3) is a security framework providing authentication and authorization (and accounting) in a service oriented fashion and storing identities in an LDAP directory. In Deliverable 1.3 it is proposed as reusable component and in ClouT will be integrated at CDMI Interface level for providing security at application level for all the Storage System. SOA3 can intercept the web service calls, check the credentials and block the unauthenticated clients or unauthorized operations (Figure 44). Since SOA3 is Service Oriented, it can also be integrated with other ClouT modules providing security for the architectural elements that still lack this feature.

In order to request data from both Orion and Cosmos GEs in order to inject them within CDMI storage, firstly it would be needed to create an account against the FIWARE platform, being provided with the corresponding user and password credentials. In the case of Cosmos, these credentials are enough for query for historical data stored within it, but Orion instance implements OAuth authentication, so it is needed to include a valid X-Auth-Token HTTP header in the requests sent to Orion. For getting this specific token it is just needed to send a POST sentence asking for credentials, specifying the specific user and password in the payload. Then, the system retrieved a temporal key that should be renewed each certain time.

7. CITY INFRASTRUCTURE MANAGEMENT

City Infrastructure Management functional block aims to provide an access to City *Services* and *Resources* by maintaining the bindings between logical entities and physical, sensorised and virtualized devices. It offers a capabilities search feature in the form of *Services*

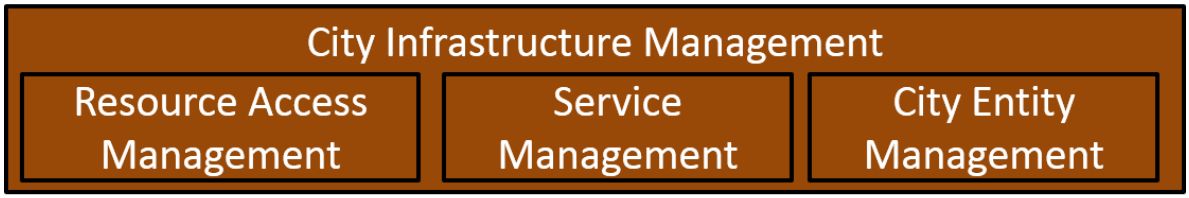

FIGURE 50: CITY INFRASTRUCTURE MANAGEMENT

The ***City Infrastructure Management*** is composed of three main components:

- ***Service Management*** is in charge of the abstraction of City *Resources* as services.
- ***Resource Access Management*** is in charge of the communication with the physical, virtualised or sensorised devices.
- ***City Entity Management*** is in charge of keeping track of the mapping among Entities and the IoT, Sensorised/Actuatorised Devices, while actual sensor data is stored in the Storage as a Service block.

In Figure 51 we recall the CIaaS domain model with regards to the Resources / Entity / Services definition as per the ClouT concept. We can summarize the roles of these components as follows:

- **Entity** is a representation of any physical or virtual entity providing data streams or actions;
- A **Service** virtually represents a City Infrastructure Entity and exports means to access to Virtualized City Resources exposed by the services;
- A **Resource**: a resource of a city which is represented by a city infrastructure service. It is classified to Data Resource and Action Resource.

To be more concrete, we can give some examples taking them both from the IoT and cloud perspective:

- **Entity**:
  - (IoT) IoT devices, web applications**,**
  - (Cloud) Physical or Virtual machines
- **Service**:
  - (IoT) Light service, Temperature Service, Presence Service
  - (Cloud) computational service
- **Resource**:
  - (IoT) Virtual City Temperature (data resource), Virtual City Light (action resource)

- (Cloud) current load of the machine (data resource), sleep action of the machine (action resource)

So, using the IoT domain terminology, we can say that:

a thermometer (Entity) provides the Temperature (Data Resource) by a method getTemperature (Service)

or that:

a semaphore (Entity) switches the light to red (Action Resource) by a method setLightAccordingToTraffic (Service)

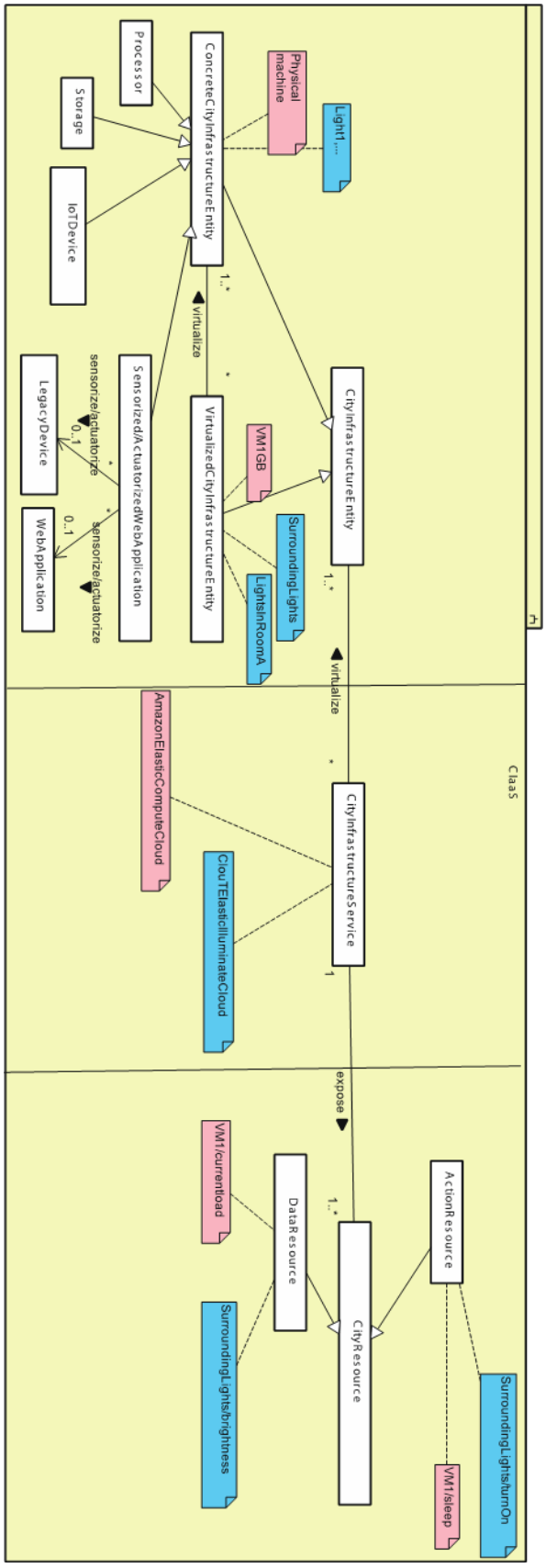

FIGURE 51 : CIAAS DOMAIN MODEL

7.1. Service Management

7.1.1. Architecture

FIGURE 52 : SERVICE MANAGEMENT

The Service Management block is composed of three functional components:

- The **Service Search** component provides mechanisms for searching and reusing the available services provided by the CIaaS to the CPaaS layer through the City Resource Access block.
- The **Service Description** component provides the representation of available services exposed by the City Entities, by means of widely adopted standards.
- The **Service (auto)Discovery** component is in charge of discovering the services exposed by the City Entities and bind them if necessary to other services. The Discovery phase can leverage different protocols and tools depending on the nature of the involved Entity.

7.1.2. Specification

7.1.2.1. Service Search

Adoption of service-oriented approach in organizations translates into cost-saving mechanisms derived from the reuse of existing services, being a service registry one of the fundamental pieces of service-oriented architecture (SOA), in order to achieve this reuse. Service registry refers to a place in which providers are able to include information about their offered services, which can be searched by potential clients. In this sense, the capacity for reusing these services is highly dependable on the ability to publish and describe the functionality offered by the services to potential consumers (clients). In this sense, UDDI and WSDL, as well as JSDL are standards that are in charge of addressing the description of both services and providers, as well as the way services are consumed.

This functional block stores profiles of city resource entities, and allows applications to find a city resource giving a query. The query is evaluated against the stored profiles in either semantic way with support of "Service Semantic Search" functional block or regular expression. This block additionally can provide continuous query support to enable applications to be updated on city resources availability.

TABLE 32 : SERVICE SEARCH API

Modifier and Type	Method and Description
List<**Service**>	**searchServices** (ServiceQuery query) Search corresponded services according to query.
Boolean	**registerServices** (Profile type, Profile entity) Registers a new service

Relevant system requirements, as from [D1.3]: REQ_CIAAS_4, REQ_CIAAS_20, REQ_CIAAS_28, REQ_CIAAS_29, REQ_CIAAS_30, REQ_CIAAS_31.

7.1.2.2. Service Description

Regarding to the service description, there exist different alternatives; all of them intended to describe the main capabilities of the service, as well as the methods offered by this service.

WSDL (Web Services Description Language)[4] allows describing the services registered with UDDI[5], including all technical information needed to access the service, including its operations with parameters and results. Additionally, it offers abstract service description, offering the so called bindings, as service elements that include all available implementations of the abstract interface at endpoints. UDDI/WSDL usually works in combination with SOAP.

In order to complement UDDI/WSDL, both WSO2[6] and JSDL (JSON Service Description Language)[7] present different approaches. On the one hand, WSO2 registry supports several usage scenarios. On the one hand, it can be embedded in any Java application that needs a registry and repository to store resources with Web 2.0-style features for commenting, tagging and rating those resources. Second, it offers a REST-style Web API that allows the WSO2 registry to be used remotely from any application from any language, including Java, PHP, C++, and JavaScript. On the other hand, JSDL (JSON Service Description Language) is a JSON-based interface description language that is used for describing the functionality offered by a (JSON) HTTP web service. This project contains the JSDL specification and gives the tools to automatically generate JSDL definitions and service client code for various programming languages. Using these generators the developer can concentrate on developing business code rather than creating boiler plate code for each service operation.

[4] http://www.w3.org/TR/wsdl20/
[5] http://uddi.org/pubs/uddi_v3.htm
[6] http://wso2.com/about/news/registry-1-0-release/
[7] http://jsdl.codeplex.com/

The aim of this block is to describe entities related to pervasive computing, including objects that provide digital services, spaces that contain the objects, and humans who manage and/or use them. The major functionality includes:

- enabling the description in XML-based human/machine readable documents with a support of language called Universal Service Description Language (USDL),
- describing both type-specific and entity-specific information,
- enabling applications to find entities matching a query.

The Service Description task defines a XML-based language specification, and it is supposed to provide developers with an authoring tool for assembling profiles of city resources. The specification allows developers:

1. to define types of city resources
2. to assemble profiles of city resource entities.

The separate description of types and entities is similar to the ideas of classes and instances in object-oriented programming. Type documents describe abstract functions of corresponding entities, and do not contain any platform-specific information. Entity documents, on the other hand, describe how the functions defined in a type are provided by a specific device. USDL allows types to inherit other types. Types are similar to Java interface definition in that they do not describe any implementation-specific issues, such as communication protocols and device identifiers used in services.

Profiles of city resource entities are stored by this functional block, allowing applications to find a city resource giving a query. The query is evaluated against the stored profiles in either semantic way with support of "Service Semantic Search" functional block or regular expression. This block additionally can provide continuous query support to enable applications to be updated on city resources availability.

TABLE 33 : SERVICE DESCRIPTION API

Modifier and Type	Method and Description
Profile	**assembleProfiles** (Profile prof1, Profile prof2) Assemble different profiles and return the profile.
Boolean	**validateProfile** (Profile profile) Checks validity of a given profile

Relevant system requirements, as from [D1.3]: REQ_CIAAS_4, REQ_CIAAS_20, REQ_CIAAS_27, REQ_CIAAS_28, REQ_CIAAS_29, REQ_CIAAS_30, REQ_CIAAS_31.

7.1.2.3. Service Discovery

This module is in charge of starting the Service Discovery phase and keeping the Service (*CityInfrastructureServices*) list up to date in case of changes of the devices availability. In order to make an efficient search of the services exposed by the different city entities, it is needed to have them stored in a suitable platform. In this sense, UDDI (Universal Description, Discovery and Integration) provides an infrastructure that supports the description, publication, and discovery of service providers; the services that they offer; and the technical details for accessing those services. The UDDI operation consists of information entities (UDDI data) that organize in a data model, being stored within a UDDI service registry. UDDI allows searching for different entries, as well as publishing, updating and deleting information related to each of the services registry. With a similar approach, platforms like Visual Web Service that started off as an online tool offering the capacity to study the interface provided by a web service, has currently evolved towards one of the largest web service repositories of publicly available web services. New web services can be easily added to this repository, just entering the WSDL URL and visualizing it.

The *CityInfrastructureService* abstraction (as defined in the ClouT domain model – ref. [D1.3]), that exposes the *CityInfrastructureEntities* capabilities, is used to access the actual Resources belonging to the IoT/Legacy Devices or Sensorised data sources.

TABLE 34 : SERVICE DISCOVERY API

Modifier and Type	Method and Description
Void	**registerService** (Service service) Register a new service upon discovery
Subscription	**subscribeToServiceArrival** (Listener callback, Type servicetype) Allows notification of subscribers about arrival of new services of type serviceType.
Subscription	**subscribeToServiceDeparture** (Listener callback, int serviceId) Allows notification of subscribers of departure of the service of the given Id.
Void	**updateDirectory** (Service service) Updates the Service Directory with the information about a service that was previously registered.

Relevant system requirements, as from [D1.3]: REQ_CIAAS_4, REQ_CIAAS_20, REQ_CIAAS_27, REQ_CIAAS_28, REQ_CIAAS_29, REQ_CIAAS_30, REQ_CIAAS_31, REQ_CIAAS_32.

7.1.3. Implementation

7.1.3.1. Service Description

USDL – UNIFIED SERVICE DESCRIPTION LANGUAGE

Additionally to the different languages indicated in specification sections, Universal Service Description Language (also called as USDL, like Unified Service Description Language) will provide a common representation of services, thus setting the basis for knowing the functionality of devices, and interoperability between certain devices across communication platforms. USDL includes two classes of documents: type documents and entity documents. Type documents describe platform-independent capability of services. On the other hand, entity documents describe platform-specific details on how a service implements a type. These classes separate platform-independent/-dependent concerns, so it is useful for various service description/lookup. USDL allows types to inherit other types. For example, the networked television is a sub-type of the television, and the television in turn is a sub-type of the device (as shown in next figure with an example of the definition of the television and the device). These types are similar to Java interface definition in that they do not describe any implementation-specific issues, such as communication protocols and device identifiers used in services.

```
<?xml version="1.0" encoding="UTF-8"?>
<type>
    <properties>
        <id>jp.ac.keio.sfc.ht.Device</id>
        <name lang="en">generic device</name>
        <provider>Jin Nakazawa, Keio University, ...</provider
        <version>1.0.0</version>
        <date>112347192</date>
        <description lang="en">
            a generic device with a power control
        </description>
    </properties>

    <interface>
        <operation name="turn on" sequence="in-out">
            <description lang="en">
                turns the power on, and sends the result
            </description>
            <input>
                <data type="mime">.*/.*</data>
            </input>
            <output>
                <data type="mime">.*/.*</data>
            </output>
            <consume>energy</consume>
        </operation>
      *** snip ***
    </interface>
</type>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<type>
  <properties>
    <id>jp.ac.keio.sfc.ht.TV</id>
    <name lang="en">television</name>
    <inherit>jp.ac.keio.sfc.ht.Device</inh
    <provider>Jin Nakazawa, Keio University</provider>
    <version>1.0.0</version>
    <date>128736912837</date>
    <description lang="en">TV type</description>
  </properties>

  <interface>
    *** snip ***
    <operation name="next channel" sequence="in-out">
      <name lang="en">select channel</name>
      <description lang="en">selects the ...</description>
      <input>
        <data type="mime">.*/.*</data>
        <description lang="en">select the ...</description>
      </input>
      <output>
        <data type="mime">.*/.*</data>
        <description lang="en">the result ...</description>
      </output>
      <consume>energy</consume>
      <provide>light</provide>
      <provide>sound</provide>
    </operation>
    *** snip ***
```

inheritance

FIGURE 53: EXAMPLE OF SERVICE DESCRIPTION WITH USDL (LEFT DESCRIPTION FOR GENERIC DEVICE, AND RIGHT DESCRIPTION FOR TELEVISION)

THE SENSINACT SERVICE AND RESOURCE MODEL

It allows exposing the resources provided by an individual service. The latter, characterized by a service identifier, represents a concrete physical device (e.g., a temperature sensor, a TV, a motion detector, a parking space detector) or a logical entity not directly bound to any device (e.g. a weather service, parking space service, etc.). Each sensiNact service exposes resources and could use resources provided by other services. The sensiNact gateway is Java OSGi based and uses JSON formatted data. The resource model is a hierarchical five-tiered tree: A Device owns Services which in turn own Resources, which hold Attributes, on which apply Metadata.

To describe one element of this tree there is no restriction about how many sub-elements it can contain. The description of a resource and its value (result of an access method execution) are distinct from one to the other. The choice of this separation is to lighten the work of components whose work is to process the result of an access method execution, by avoiding the reification of big high level data structures to only extract the content of one (or two) attribute(s).

As only the resources are the containers of information, those which target the device are grouped in a specific service which is the administration one (« AdminService » prefixed). Those resources can be one specifying the location, or the vendor of the device, or any other data that are common to all provided services (and so resources). Formally, a device is a JSON object containing an array of services. The list of the services a device provides can be summarized or detailed. If it is summarized only the name of the services are part of the description (otherwise each service is completely described).

Example of the description of the device whose identifier is « fake-1234 »:

```
{
"serial-number" : "fake-1234",
"services" : [
{ "ID" : "AdminService_f1To4"},
{ "ID" : "temperature_f1To4"}
]
}
```

A service description gathers resources, and it references the unique identifier of the device holding it. It represents the entry point to access to resources through the OSGi context. The list of the resources a service provides can be summarized or detailed. If it is summarized only the name and the type of the resource are part of the description (otherwise each resource is completely described).

Example of the description of the administration service of a device whose identifier is « fake-1234 »:

```
{
"ID" : "AdminService_f1To4",
"properties" : [
{"device.serial-number" : "fake-1234"}
],
"resources" : [
{"name" : "location", "type" : "property"},
{"name" : "owner", "type" : "property"},
{"name" : "vendor" "type" : "property"},
{"name" : "SLEEP" , "type" :"action"}
]
```

```
}
```

Finally the resource description owns the following properties:

- the data structures are mainly nestings of triplets : name, type and value.
- the type of the resource itself can be: property, variable, sensor, or action
- the type key of a 'name-type-value' data structure (embedded in the resource description) can have a primitive as value (byte, short, int, long, double, char, boolean, [string]) or the canonical name of the java class used to reify it in the gateway.
- for each resource access method signatures are also described in a JSON array. Some of them can be shortcuts to other ones: a GET method without parameter is a shortcut to the GET method whose unique parameter "attributeName" has for value "value", for example. A parameter of an access method can be completed with the constraints which apply on it (« min », « max », « fixed », regular expression « pattern », « enumeration » of allowed values) or the JSON schema of the expected JSON object from which to reify the appropriate Java object in the gateway.
- at least two metadata exist for each attribute: the "hidden" one defining whether the attribute has to be specified in the description of the resource, and the "modifiable" one defining whether the value of the attribute can be modified by the client. By default, the « hidden » attribute is not visible in the description (if the attribute is visible that's mean that this metadata value is set to false, and if it is set to true the client is, at the end, not aware of that)
- a metadata specified as « dynamic » will be added to the result of an access method execution
- timestamps are « epoch » formatted (number of seconds since 1970 January the first); to avoid the reification of high level objects to make calculations (that are at least as easy with this format). High level programming languages handle this format. It is also possible to multiply it by 1000 if handling of milliseconds is needed (what is done natively by java for example)

Example of a "temperature" resource description, holding only one « value » attribute (in fact, name and type are also attributes, but as constants they appear as simple key-value pairs of the JSON object):

```
{
"name" : "temperature" ,
"type" : "sensor",
"attributes" : [
{
"name" : "value",
"type" : "int"
"metadata" : [
{
"name" : "modifiable",
```

```
"type" : "boolean",
"value": false,
"dynamic": false
},
{
"name" : "timestamp",
"type" : "long",
"value": 1418541626,
"dynamic": true
},
{
"name" : "description",
"type" : "string",
"value": "temperature measure",
"dynamic": false
},
{
"name" : "unit",
"type" : "string",
"value": "celsius degree",
"dynamic": false
}
]
}
],
"accessMethods" : [
{
"name" : "GET"
"parameters": []
},
{
"name" : "GET"
"parameters": [
{
"name" : "attributeName",
"type" : "string"
}
]
},
{
"name" : "SUBSCRIBE"
"parameters": [
{
"name" : "listener",
"type" : "object",
"schema-id" : "http://fr.cea.sensinact/subscription/listener",
"description": "parameter value example: '{\"callback\":\"<uri>\"}' "
},
{
"name" : "condition",
"type" : "object",
"schema-id" : "http://fr.cea.sensinact/subscription/condition" ,
"description": " parameter value example : '{ \"condition\" : \"less\", \"value\" : \"5\" }'
},
{
"name" : "lifetime",
"type" : "long"
}
]
},
{
"name" : "SUBSCRIBE"
```

```
"parameters": [
{
"name" : "listener",
"type" : "object",
"schema-id" : "http://fr.cea.sensinact/subscription/listener",
"description": "parameter value example: '{\"callback\":\"<uri>\"}' "
},
]
},
{
"name" : "UNSUBSCRIBE"
"parameters": [
{
"name" : "subscriptionID",
"type" : "string"
}
]
}
]
}
```

The location of a device (service, resource) is frequently a needed context information. By default a device always contains one (its administration service in fact), and a link to it is created in all services it provides. If needed, a link to this resource could also be created as an attribute of all resources (mainly if this location is supposed to change frequently and so to avoid to require the complete device description to update the information). Its content is not restricted (as it is the case for the others) and can so contain attributes defining longitude, latitude, altitude, a friendly name or whatever is needed to specify it (for now we are using « <latitude>,<longitude> » formatted string as value)

7.1.3.2. Service Search/Discovery

In order to perform the search and discovery of the corresponding services, two additional modules should be implemented, the service search engine and the service discovery engine. The interaction among these modules and other needed modules is shown in next figure:

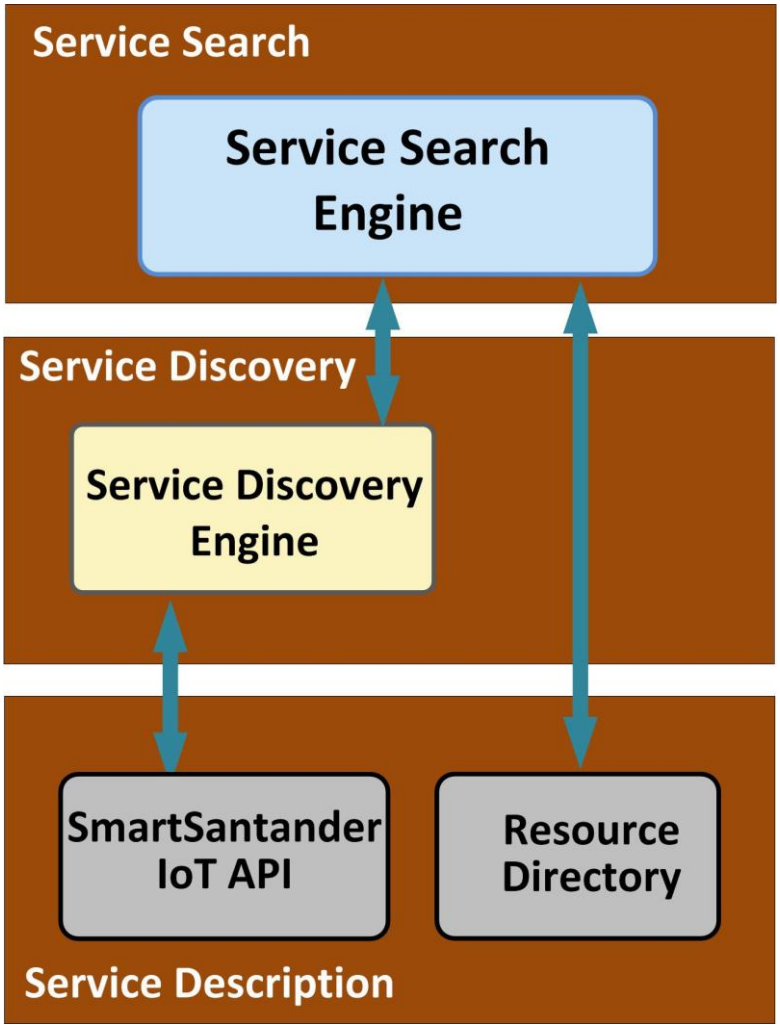

FIGURE 54: EXAMPLE OF SERVICE SEARCH/DISCOVERY

As defined from Figure 54, the service discovery/search would be as follows:

1. The service search engine receives a request for searching for a service, thus formatting accordingly the query command in order to access in a suitable way to the service discovery engine.

2. The service discovery engine receives the request from the service search engine thus accessing to the both the resource directory and the SmartSantander API. The first one will retrieve the information regarding to the resource/s associated to this service or set of services. On the other hand, the IoT API is in charge of retrieving the corresponding methods offered by the service to interacting with the corresponding resource.

3. Both SmartSantander IoT API and Resource Directory will maintain available resources and access methods accordingly updated.

7.2. Resource Access Management

7.2.1. Architecture

FIGURE 55 : RESOURCE ACCESS MANAGEMENT

The Resource Access Management functional block is composed of three components:

- Historical access: it is the component in charge of providing historical access facilities to the upper layers (i.e. it provides the capability of asking data for a given period of time).
- On-Demand Access: it is the block in charge of the Request/Response interaction with a City Entity. This kind of interaction is typically synchronous as the client/requestor waits for the response carrying the latest value or status of the City Entity.
- Event Based Access: it is the block in charge the Push (or "Observe") type of interaction with a City Entity. In this type of interaction (typically asynchronous) the client/consumer is notified about the value/status of an entity on event basis: this can be a change on the value/status (optionally above a certain percentage or absolute value) or a periodic timeout.

7.2.2. Specification

7.2.2.1. Historical Access

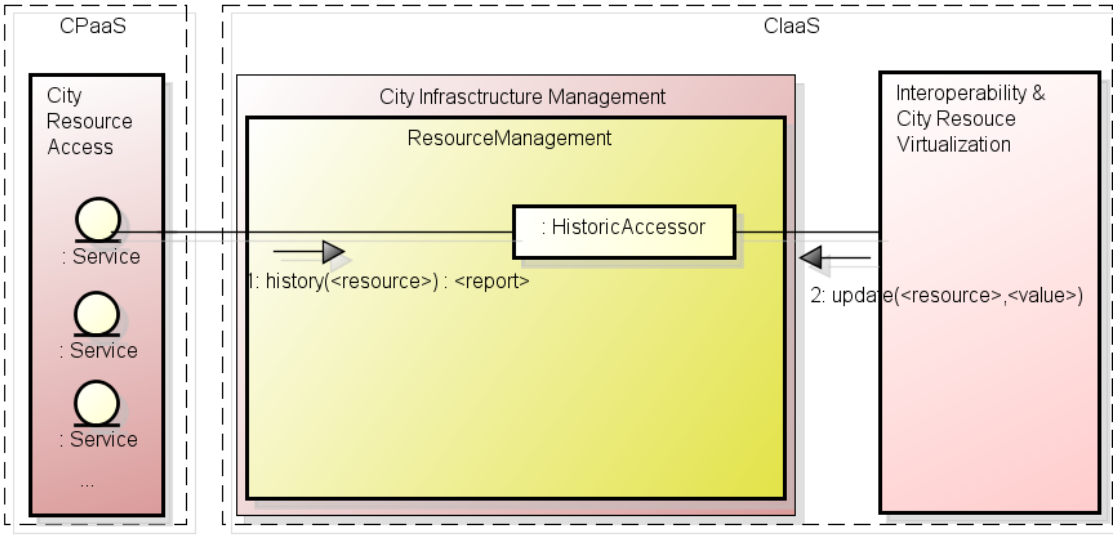

FIGURE 56 : HISTORICAL ACCESS COMMUNICATION DIAGRAM

Historical Access component mainly interacts with services provided to the **City Resource Access** block, as requirers and recipients of a targeted virtualized resource updates history; Both **Event Based** and **On-Demand** Access components feed this one when notified about a value change of a defined resource

TABLE 35 : HISTORICAL ACCESS API

Modifier and Type	Method and Description
<report>	**history** (<resource>[,<from>[, <to>]]) Asks for a report of a resource changes history potentially for a defined time delay • The "resource" parameter is the identifier of the targeted resource • The optional "from" parameter defines the date time from which to start the report • The optional "to" parameter defines the date time ending the report The functionality returns an history "report" object for the specified resource
Void	**update** (<resource>,<value>) records an update of the specified resource

Modifier and Type	Method and Description
	• The “resource” parameter is the identifier of the targeted resource • The “value” parameter is the updated value

Relevant system requirements, as from [D1.3]: REQ_CIAAS_18, REQ_CIAAS_2, REQ_CIAAS_27, REQ_CIAAS_28, REQ_CIAAS_29, REQ_CIAAS_30, REQ_CIAAS_32, REQ_CIAAS_4.

7.2.2.2. On Demand Access

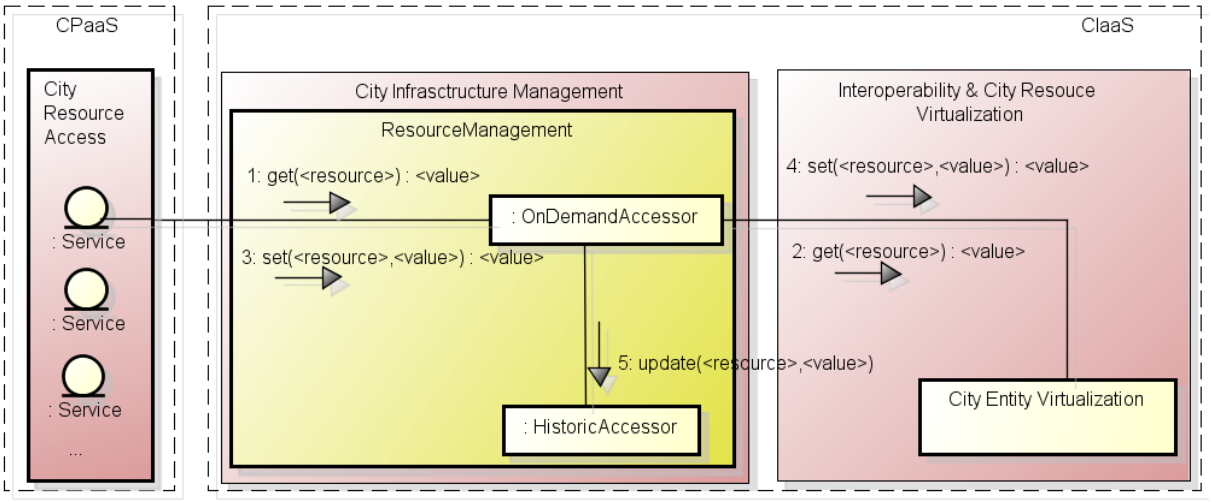

FIGURE 57 : ON-DEMAND ACCESSCOMMUNICATION DIAGRAM

On-Demand Access component mainly interacts with:

- the services provided to the **City Resource Access** block as requirers and recipients of a targeted virtualized resource current "value" setting or getting;
- the **Historic Access** component to store updates ;
- the **City Entity Virtualization** to which requests (get or set) are transmitted;

TABLE 36 : ON DEMAND ACCESS API

Modifier and Type	Method and Description

Modifier and Type	Method and Description
<value>	**get** (<resource>) Asks for the current "value" of the specified resource • The “resource” parameter is the identifier of the targeted resource The functionality returns the current value of the targeted resource
<value>	**set** (<resource>, <value>) Defines the "value " of the specified resource • The “resource” parameter is the identifier of the targeted resource • The “value” parameter defines the value to set to the specified resource The functionality returns the updated value of the specified resource

Relevant system requirements, as from [D1.3]: REQ_CIAAS_18, REQ_CIAAS_2, REQ_CIAAS_27, REQ_CIAAS_28, REQ_CIAAS_29, REQ_CIAAS_30, REQ_CIAAS_32, REQ_CIAAS_4.

7.2.2.3. Event Based Access

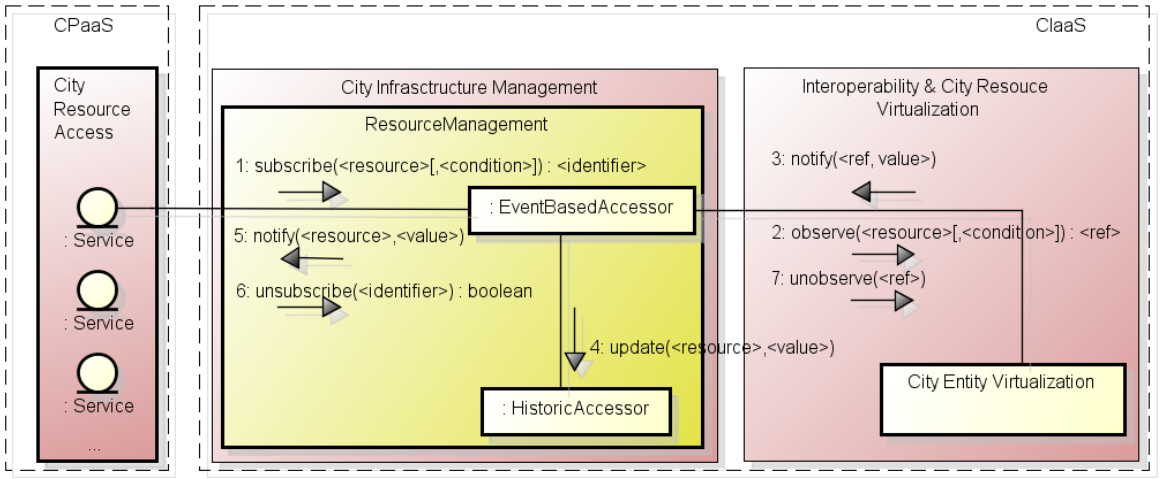

FIGURE 58: EVENT BASED ACCESS COMMUNICATION DIAGRAM

Event Based Access component mainly interacts with:

- the services provided to the **City Resource Access** block as requirers and recipients of a targeted virtualized resource update notifications;
- the **Historic Access** component to store updates ;
- the **City Entity Virtualization** to require and to parameterize the update notifications coming from the virtualized resources ;

TABLE 37 : EVENT BASED ACCESS API

Modifier and Type	Method and Description
<identifier>	**subscribe** (<resource>[, <condition>]) Asks for update notifications. • The "resource" parameter is the identifier of the targeted resource • The optional "condition" parameter allow to parameterize the update notifications The functionality returns an identifier object of this subscription request
boolean	**unsubscribe** (<identifier>) Asks for the end of the update notifications. • The "identifier" parameter is the identifier of the

Modifier and Type	Method and Description
	previously made subscription request to stop The functionality returns a boolean value defining whether the unsubscription has been made properly or not
Void	**notify** (<ref>, <value>) The Event Based Access component is informed about an update of an observed virtualized resource. • The "ref" parameter identifies the concerned updated resource • The "value" parameter is the updated value of the resource.

Relevant system requirements, as from [D1.3]: REQ_CIAAS_18, REQ_CIAAS_2, REQ_CIAAS_27, REQ_CIAAS_28, REQ_CIAAS_29, REQ_CIAAS_30, REQ_CIAAS_32, REQ_CIAAS_4.

7.2.3. Implementation

7.2.3.1. CEA Gateway Solution (sensiNact)

Here we will illustrate the implementation of the sub-blocks composing the City Infrastructure Management main block, as per the solution offered by the IoT Gateway developed by CEA. The Historical Access is not handled by this solution, as the platform can take benefit of the cloud storage to which this kind of access will be delegated

7.2.3.1.1. On-Demand Access

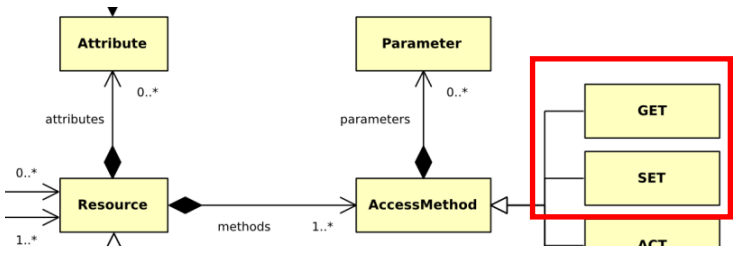

FIGURE 59: SENSINACT GATEWAY RESOURCE MODEL - GET / SET METHODS DETAIL

As it is described above in this document, among access methods provided to interact with a resource, GET and SET (and ACT) ones provide a synchronous read/write access to the value of a resource (or to one of its attribute).

For example, the array below presents a GET access method structure:

Signature:	GET http://host:port/sensinact/devices/{deviceID}/services/{serviceID}/resources/{resourceName}/GET
Path Parameters:	{deviceID}: the id of the device to be consulted {serviceID}: the id of the service to be consulted {resourceName}: the name of the resource to be get
Data Parameters:	NONE
Request Example:	http://193.48.18.248:8080/sensinact/devices/SmartPlug_0/services/PowerService_SmartPlug_0/resources/status/GET

Response Example:	{
	"name": "status",
	"type": "int",
	"value": 0
	}

7.2.3.1.2. Event Based Access

The sensiNact gateway also provides an event based access by the way of the SUBSCRIBE method. This method allows to make the subscription conditional: more, less, equals or different from a predefined value, deviation (relative or absolute) from an initial value, etc. The UNSUBSCRIBE method, as it allows to ask for the end of notification

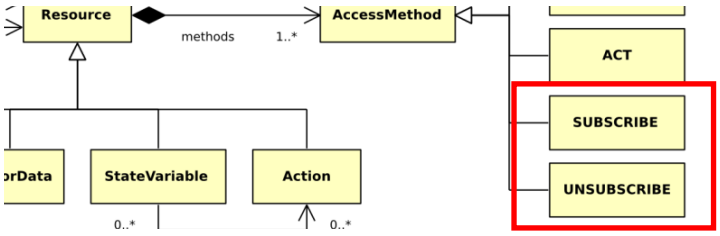

FIGURE 60: SENSINACT GATEWAY RESOURCE MODEL - SUBSCRIBE / UNSUBSCRIBE METHODS DETAIL

Signature:	POST http://host:port/sensinact/devices/{deviceID}/services/{serviceID}/resources/{resourceID}/SUBSCRIBE
Path Parameters:	{deviceID} : the id of device to be consulted

	{serviceID}: the id of the service to be consulted {resourceName}: the name of the resource to be invoked

Data Parameters:	{
	callback : "<callback-url>"
	}

Request Example:	http://193.48.18.248:8080/sensinact/devices/WSPT_XBEE_4565 /services/for_WSPT_XBEE_4565/resources/for/SUBSCRIBE This is a subscription to an xbee force resource

Data Parameters: Example	{
	callback : "http://43.74.12.03:8111"
	}

Response Example:	An Id (text value) used for notification. Thus, if the client subscribe for many resources, the notification will enclose the id that will be used by the client to know which resource is concerned by the notification. Knowing that multiple devices could expose resources with same name.
	
	The gateway will post the notification at this address:
	Post callback+"/sensinact/{id}"
	
	Example :
	http://43.74.12.03:8111/sensinact/ for123523645636735635213452
	Content :
	{
	"name" : "force",
	"type" : "java.lang.Float",
	"value" : 10.5
	}

Signature:	POST http://host:port/sensinact/devices/{deviceID}/services/{serviceID}/resources/{resourceID}/UNSUBSCRIBE
Path Parameters:	{{deviceID} : the id of device to be consulted {serviceID}: the id of the service to be consulted {resourceName}: the name of the resource to be invoked

Data Parameters:	{
	subscriptionId: "<usid>"
	}

Request Example:	http://193.48.18.248:8080/sensinact/devices/WSPT_XBEE_4565 /services/for_WSPT_XBEE_4565/for/UNSUBSCRIBE

Data Parameters: Example	{
	subscriptionId : "for123523645636735635213452"
	}

Response Example:	Status.*SUCCESS_CREATED*

7.3. City Entity Management

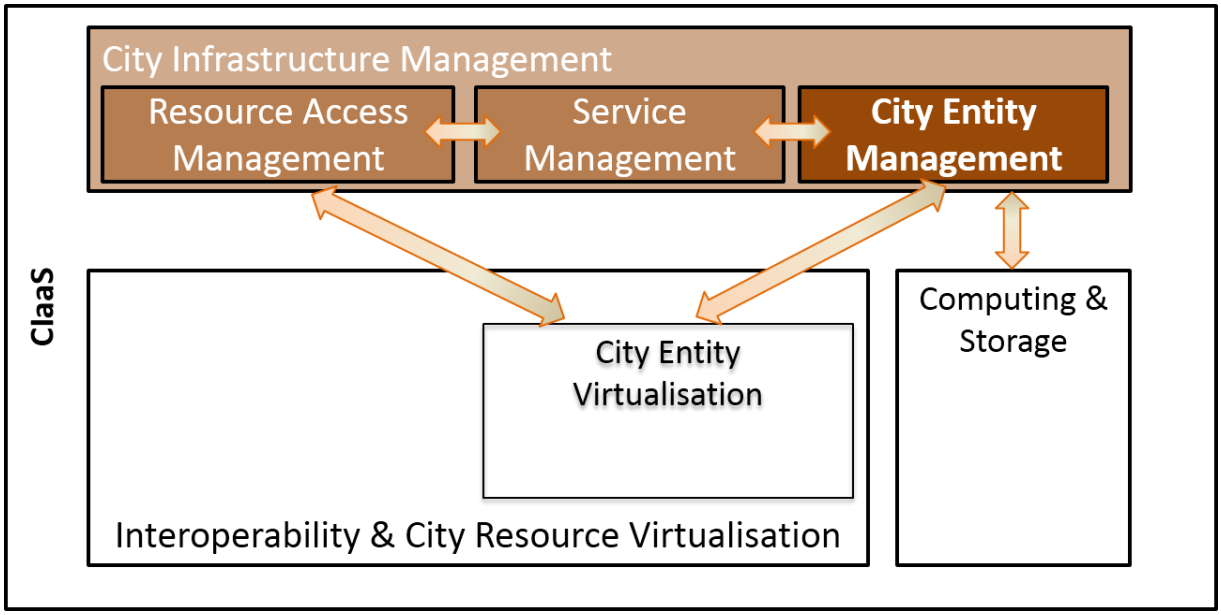

FIGURE 61 : CITY ENTITY MANAGEMENT

The **City Entity Management** functional component offers an abstract and homogeneous way of representing and accessing different entities, meaning the ClouT platform's logical counterparts of IoT or sensorised/actuatorised devices, as well as web-service/web-application available on the internet. The managed entities are the ones providing the services, themselves managed by the way of the **Service Access Management** functional block, and providing the resources accessible from the **Resource Access Management**.

7.3.1. City Entity

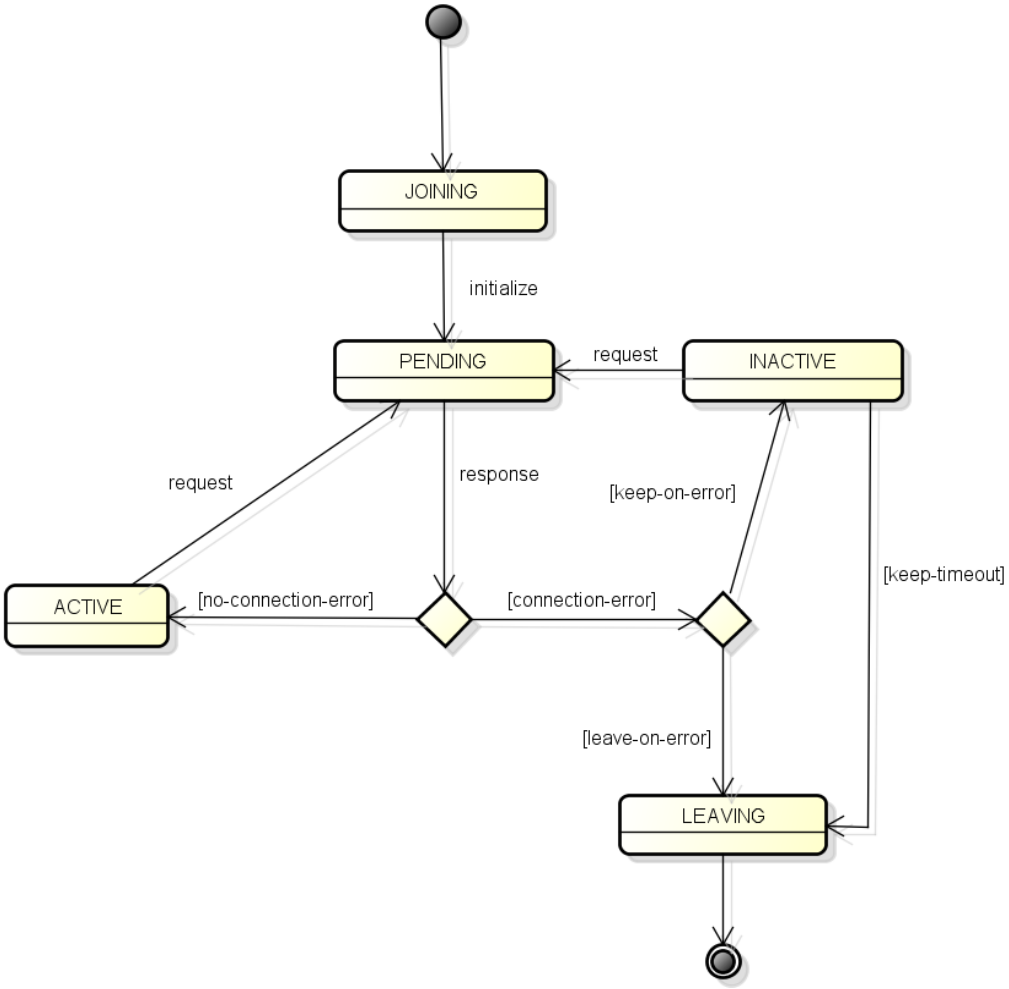

FIGURE 62: CITY ENTITY LIFECYCLE

The City Entity lifecycle is shown in the figure above. It is defined by five possible states:

- **JOINING:** it is the initial state of any entity, until inner-system initialization is done. After local data structures have been completed, the lifecycle city entity passes to the PENDING state, for finalizing its instantiation by collecting remote information if needed;
- **PENDING:** it is the state of an entity trying to connect to its remote (physical or not) counterpart.
- **ACTIVE:** it is the state of an entity for which the last connection has been made without error;
- **INACTIVE:** it is the state of an entity for which the last connection has not been realized properly;

- **LEAVING:** it is the state of an entity leaving the system, freeing previously preempted local resources;

The **City Entity Access API** allows to configure the behavior of an entity when a connection error occurred: It is possible to define if the entity has to be destroyed by the system (LEAVING) or to be kept (INACTIVE), and in this last case for how long.

TABLE 38: ENTITY ACCESS API

Modifier and Type	Method and Description
<location>	**getLocation** () Returns the targeted entity location. This location is a data structure containing at least GPS location information for a physical device counterpart, or an URI for a web service/application one.
<identifier>	**getIdentifier** () Returns the unique identifier data structure of the targeted entity.
Service	**getService** (<service-identifier>) Returns the service whose identifier is passed as parameter of this functionality.
<service-identifiers>	**getServices** () Returns the list of identifiers of the services of the targeted entity.
<status>	**getStatus** () Returns the lifecycle « status » of the targeted entity.
Boolean	**attach** (<status>, <behavior>) This functionality allows to define a « behavior » data structure which is activated if the lifecycle « status » passed as parameter is reached by the targeted entity. Returns a Boolean defining whether the trigger attachment has been realized properly.

Relevant system requirements, as from [D1.3]: REQ_CIAAS_18, REQ_CIAAS_20, REQ_CIAAS_28, REQ_CIAAS_31, REQ_CIAAS_32.

7.3.1.1. Specification

The **City Entity Management** provides a service search feature used by the **City Service Discovery**.

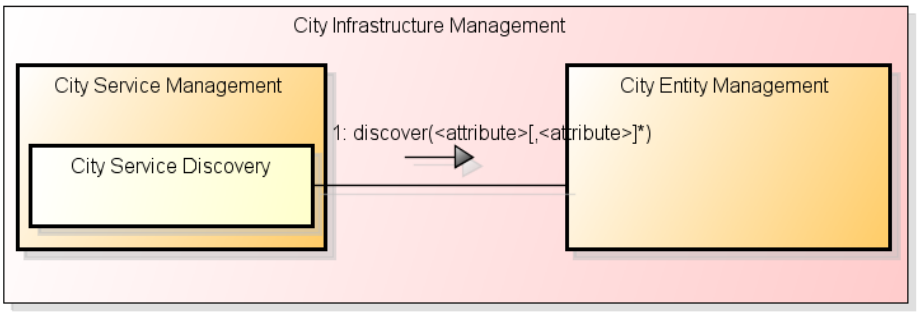

FIGURE 63: CITY INFRASTRUCTURE MANAGEMENT - DISCOVER

The **City Entity Management** functional component is responsible for provisioning the **City Service Description** one with the descriptions of discovered entities. If a created entity contains a service whose description is compliant with a previous search made by the **City Service Discovery** component, the description of this service is automatically registered

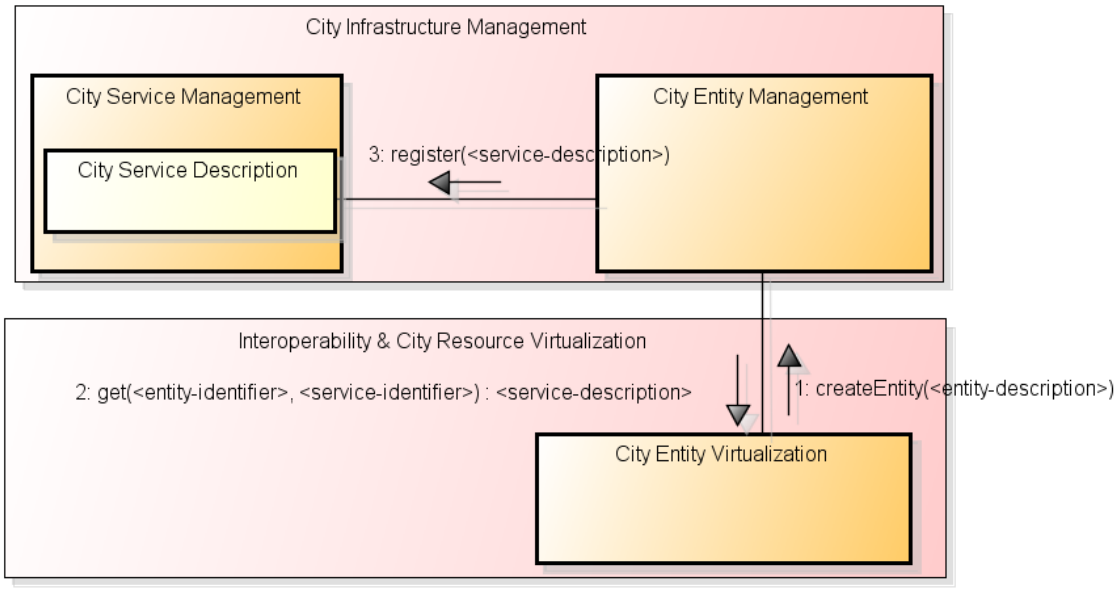

FIGURE 64: CITY INFRASTRUCTURE MANAGEMENT - REGISTER

TABLE 39: ENTITY MANAGEMENT API

Modifier and Type	Method and Description
Void	**createEntity** (<entity-description>) Creates the city entity whose description is passed as parameter.
Void	**discover** (<attribute>[,<attribute>]*) Registers a service search. The description of the searched service is created using the variable list of attribute data structures.

Relevant system requirements, as from [D1.3]: REQ_CIAAS_18, REQ_CIAAS_20, REQ_CIAAS_28, REQ_CIAAS_31, REQ_CIAAS_32.

7.3.1.2. Implementation

7.3.1.2.1. CEA Gateway solution (sensiNact)

Entity creation:

The concepts of Device and SmartService from the Resource Model of sensiNact (previously presented) define any entity which is able to provide a Service and are so jointly close from the ClouT's City Entity concept. Device and SmartService concepts will become more simply ServiceProviders in the new sensiNact version. The discovery process of a new entity is dependent on the protocol in use; but from the initial information of a ServiceProvider appearance sensiNact defines an instantiation flow control allowing to reify a complete instance of the Resource model for the discovered entity. In this process an extension point allows to complete the instantiation flow control to adapt to a protocol or a service provider specificities.

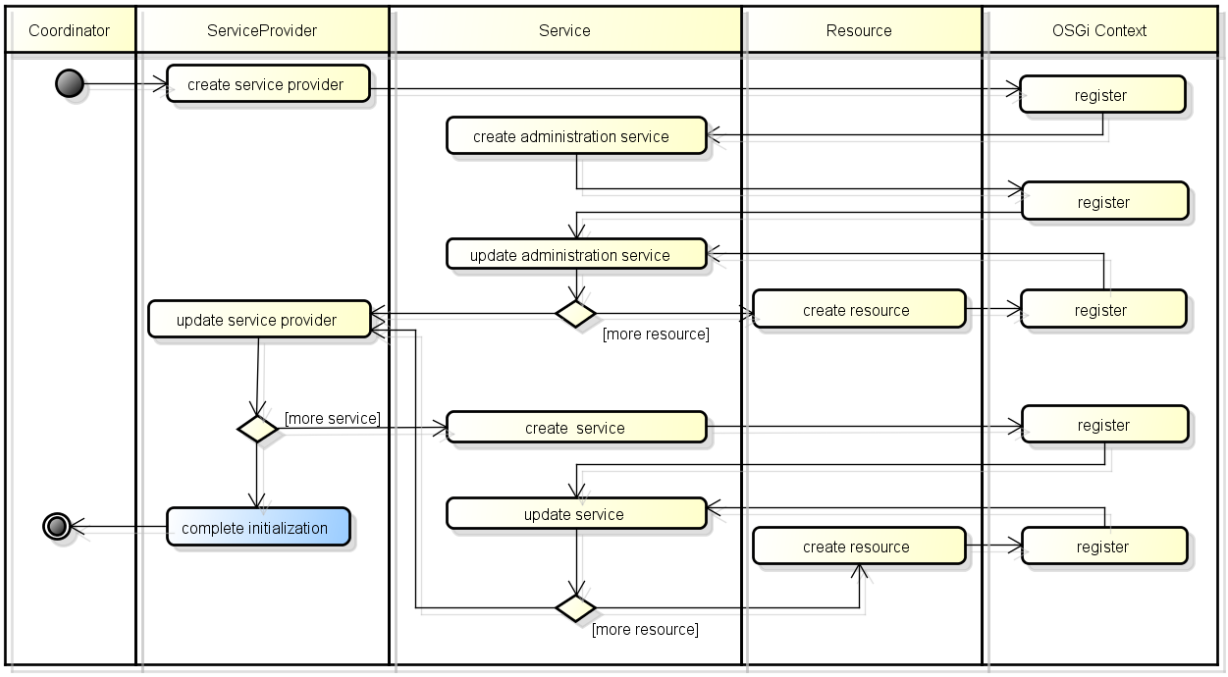

FIGURE 65: SENSINACT SERVICEPROVIDER INSTANTIATION FLOW CONTROL

The service discovery feature is in fact covered by two other functional components:

- The CDMI Cloud Storage for the discovery through multiple sensiNact gateways, as each of those gateways logs to it every appearance and disappearance of ServiceProviders and their associated Services.
- On each sensiNact gateway, retrieving a service by the way of a set of properties is ensures by the OSGi Registry that has been presented in the IoT Kernel description (section 3.4.1) of this document.

7.4. Opportunities/Threats Analysis

The following subsection provides 2 tables: 1 table clarifying the components that will be reused / extended from the state of the art and the components that will be developed in the project, and the 2nd table giving opportunities/threats analysis, for each component solution.

In Table 40 we report, for each functional block of City Infrastructure Management, the reusable component that can be used to implement it.

TABLE 40: [CITY INFRASTRUCTURE MANAGEMENT] EXTERNAL COMPONENTS VS CLOUT DEVELOPED COMPONENTS

Functional block	External component to be reused or extended	Components being developed in the ClouT

Service Search	UDDI/WSDL/JSDL, sensiNact GW, Service Search Engine	Reusable components will be used in combination to describe offered services and to provide to the user a usable interface
Service Description	Universal Service Description Language (USDL), Unified SDL, WSDL, JSDL	Will be adapted and extended to fit ClouT model. Comparison among different approaches
Service Discovery	sensiNact GW, Service Discovery Engine, SmartSantander IoT API	Extended to support interaction with cloud components and with new IoT devices
Historical Access	-	Functionalities covered using the Storage functional block
On Demand Access	sensiNact GW, Resource Directory	Extended to support interaction with cloud components and with new IoT devices
Event Based Access	sensiNact GW	Extended to support interaction with cloud components and with new IoT devices
City Entity Managemet	sensiNact GW, CDMI, OSGi, SmartSantander IoT Resource Manager	sensiNact gateway is extended to support interaction with cloud components (and CDMI in particular)

In Table 41 we summarize opportunities and threats for these platforms.

TABLE 41 : [CITY INFRASTRUCTURE MANAGEMENT] OPPORTUNITIES VS THREATS ANALYSIS OF REUSABLE COMPONENTS

Solution	Corresponding ClouT components	Opportunities	Threats	Notes
sensiNact Gateway	City Infrastructure Management block	Scalable IoT gateway solution, open source, developed by one member of ClouT consortium. Addresses some of the functionalities	To be verified how it will be maintained and the licenses policy that will be adopted	We believe it is a plus to reuse components from other EU funded projects, granting a sort of synergy to these

		that we need and can be easily extended for the missing ones.		activities
SmartSantander IoT Resource Manager, SmartSantander IoT API	City Entity Management	Functionalities for managing and keeping resources, as well as access methods updated.	Identify services and associated description to be included.	It is a plus to reuse components from other EU funded projects.
Service Search and Service Discovery Engine	Service Search and Discovery	Access to methods offer by the services to easily exploit functionalities offered by them.	Complicated to establish search/discovery patterns.	To be adapted in the context of ClouT project.
UDDI/WSDL/JSDL	Service Search	Standards to be used to describe the service capabilities	None specific	
USDL	Service Description	Good standard starting point for Service description	Not complete for our purposes	Will be adapted in the context of ClouT project
OSGi	City Entity Management	Widely adopted application container for Java developments that takes care of static/dynamic bindings and offers service storage capability.	Does not fit non Java based deployments	

7.5. Security considerations

As previously mentioned a City Entity is mapped to a ServiceProvider object in the sensiNact gateway. It is so at the top level of the hierarchical resource model. The security considerations specified in the section 3.6 of this document imply that adding a new ServiceProvider lead the definition of the minimum required *UserProfile* for each data structure and AccessMethod. By default if the owner of a ServiceProvider does not specify any security information, the default security policy is applied:

TABLE 42 : USER PROFILES AND RELATED POLICIES

UserProfile	Policy
Owner	All access granted
Administrator	All access granted except security policy
Authenticated	Read, write and act
Anonymous	Read only
Unauthorized	Forbidden

By the same way, introducing a new ServiceProvider can imply to define the map between identities, data structures and *UserProfile.*

Regarding to service semantic search, security credentials are similar to those shown in the IoT API, thus generating API keys in a random way and assigning to different users depending on their needs and access privileges.

8. CONCLUSIONS

This deliverable illustrates the second specification of the ClouT CIaaS layer, along with details on the ongoing work of specification and integration.

Following the outcomes of the deliverable D1.3 "*Final requirements and Reference Architecture*", in this document the role of each identified component has been analyzed more in details;

- For each block an *architectural* description is given by means of communication diagrams;
- The *specification* of the API implemented by the various blocks is given with the description of the interfaces exposed by the different blocks;

- The *implementation* details are reported, aggregating them per solution/development. In these sections we identified which of the ClouT reference architecture's functionalities are covered by the various implementations, and we give details on which components were reused and how they have been modified or extended.

In this second version of the document, we mainly dealt with the implementations proposed and developed by the various partners, and to make them successfully interoperate following the identified final reference architecture. Brief sections on the security aspects have been added for each chapter.

The outcome of this document is intended to serve as an input to D4.2 "*Middle report of city application developments and in-lab evaluation and field trials*".

Higher level functionalities, such as service search facilities and security aspects, which are not yet completely addressed in this specification, will be covered in the final version of this document (D2.4).

In Figure 66 we report items that are mainly developed in the ClouT project, i.e. what we have identified as the "ClouT product" in [D5.5], with a mapping to the reference architecture.

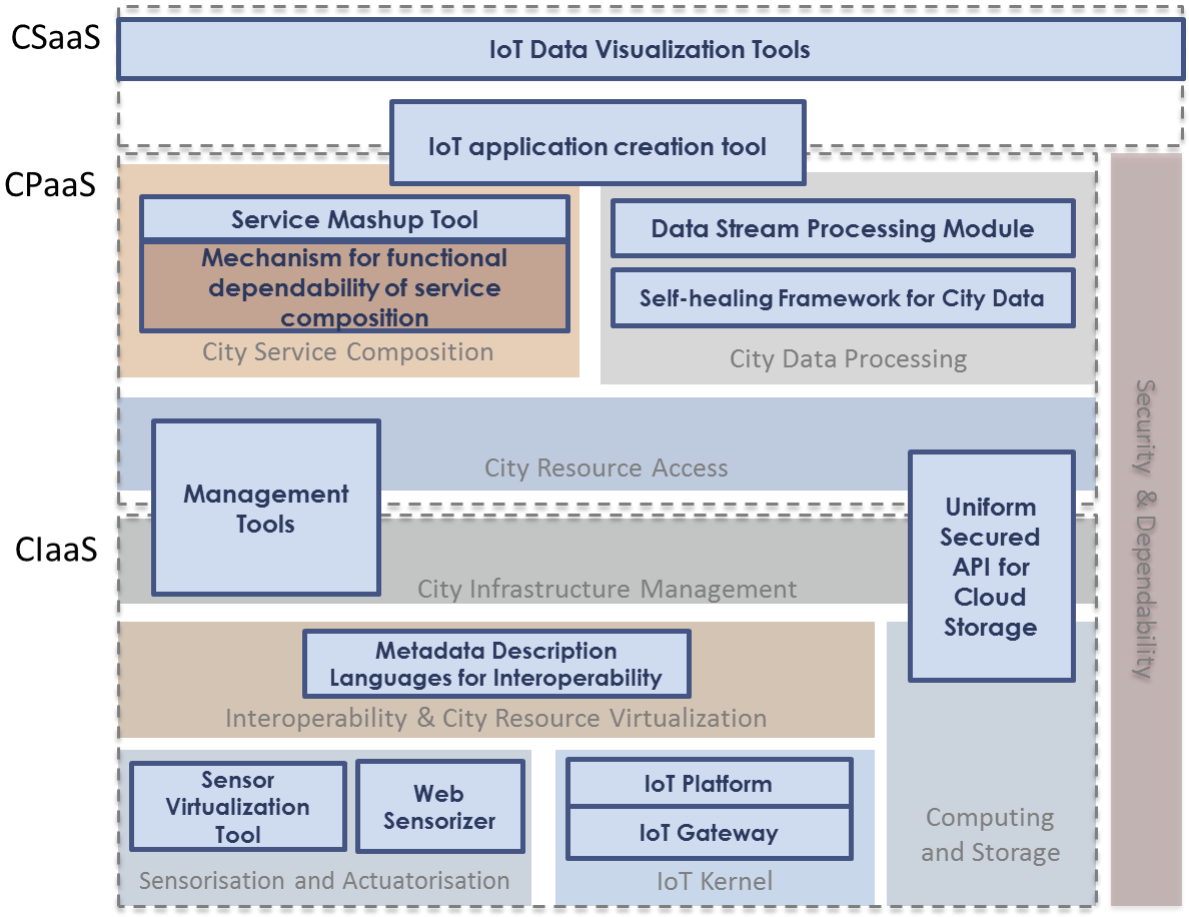

FIGURE 66 : CLOUT PRODUCT MAPPED TO THE CLOUT ARCHITECTURE

REFERENCES

[DoW] FP7 EU Research Project "*Cloud of Things for empowering the citizen clout in smart cities* ", Grant agreement no: 608641, Version date: 2013-03-26 - Annex I - "Description of Work".

[D1.3] ClouT project deliverable - "*Final requirements and Reference Architecture*" - 30/09/2014.

[D5.5] ClouT project deliverable - *"First progress on Business Models"*

[IPSO] http://www.ipso-alliance.org/smart-object-guidelines

[OMA-LWM2M] http://technical.openmobilealliance.org/Technical/technical-information/release-program/current-releases/oma-lightweightm2m-v1-0

[CORE-INT] http://tools.ietf.org/html/draft-shelby-core-interfaces-03.

[UDDI_WSDL] UDDI/WSDL comparison. http://msdn.microsoft.com/es-es/architecture/aa699419.aspx.

[RFC-2141] http://www.ietf.org/rfc/rfc2141.txt

[RFC-3406] http://www.ietf.org/rfc/rfc3406.txt

APPENDIX

APPENDIX A – OMA LWM2M

A graphical representation of the OMA LWM2M Object and Resource model (taken from the specification document [OMA-LWM2M]) is reported in Figure 67.

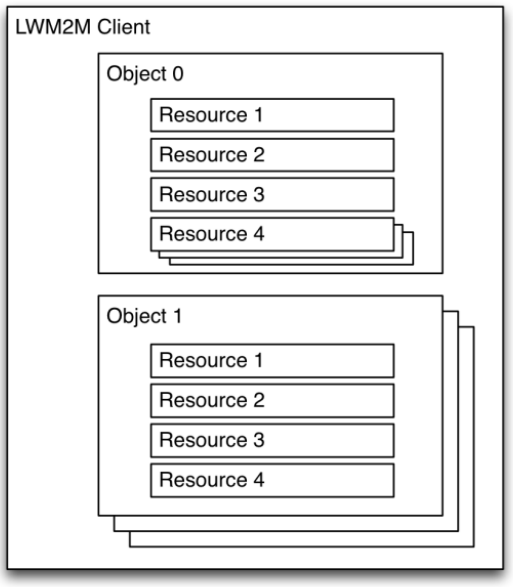

FIGURE 67 : OMA LWM2M CLIENT OBJECT AND RESOURCE MODEL

The OMA LWM2M specification is based on a Client / Server communication model that is illustrated in Figure 68.

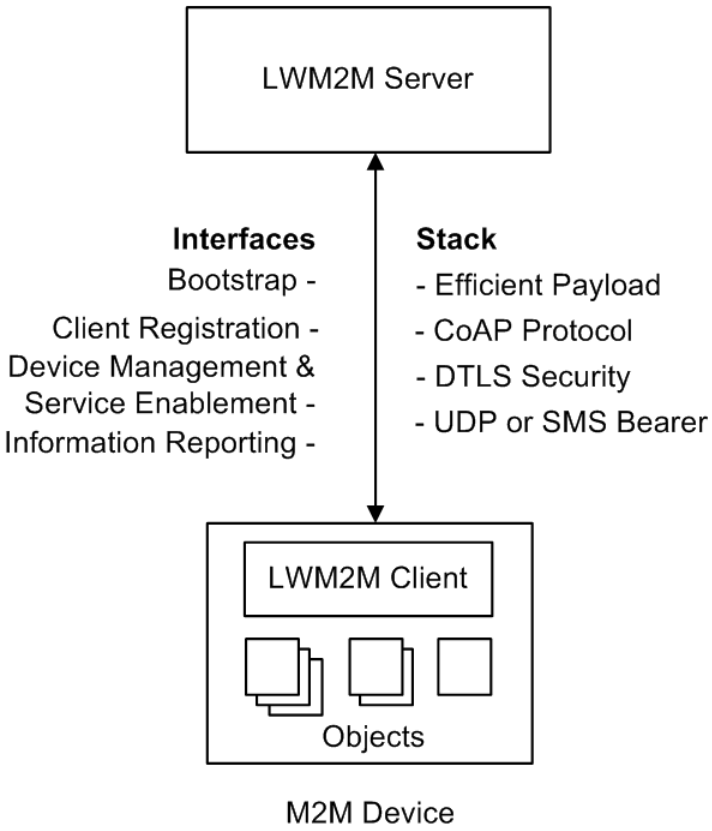

FIGURE 68 : LWM2M COMMUNICATION MODEL

SPECIFICATION OF OBJECTS BASED ON OMA LWM2M MODEL

Four types of interface are defined by the standard, and these can be classified as uplink or downlink operations:

1) Bootstrap
2) Client Registration,
3) Device Management and Service Enablement,
4) Information Reporting.

In this section we are in particular interested in the way the Sensor Resources are represented and accessed, according to the IPSO Smart Objects specification and using the mechanisms provided by the underlying CoAP protocol. These functionalities are covered by:

- the Device Management and Service Enablement interface that includes the operations reported in Figure …: all these operations are of downlink type.
- The Information Reporting Interface: used to instantiate/cancel an “observe” type of interaction

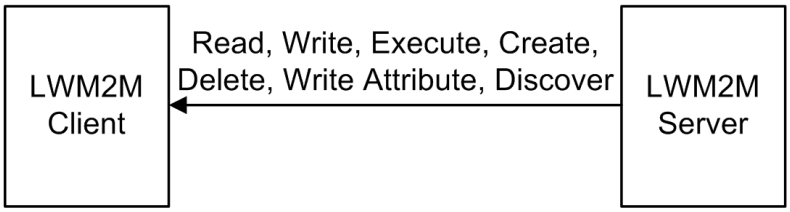

FIGURE 69 : DEVICE MANAGEMENT AND SERVICE ENABLEMENT INTERFACE

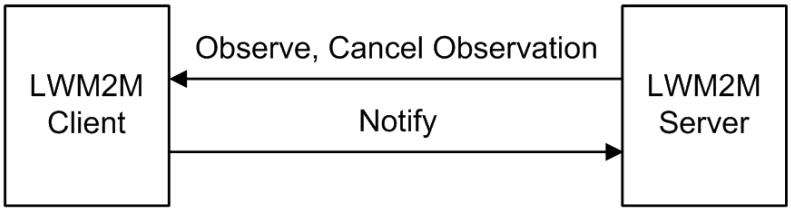

FIGURE 70 : INFORMATION REPORTING INTERFACE

These whole set of OMALWM2M interfaces implements the functionalities that in the ClouT Reference Architecture are covered by:

- IoT Device Wrapping
- Device Discovery
- Device Directory (this functionality implemented in the context of the LWM2M Server, using the Client Registration interface)
- Device Manager (some of the functionalities are covered by the Bootstrap interface as well)

In the following figures we report some example of flow for both the interfaces that build the generic API of a sensor network based on OMA LWM2M standard:

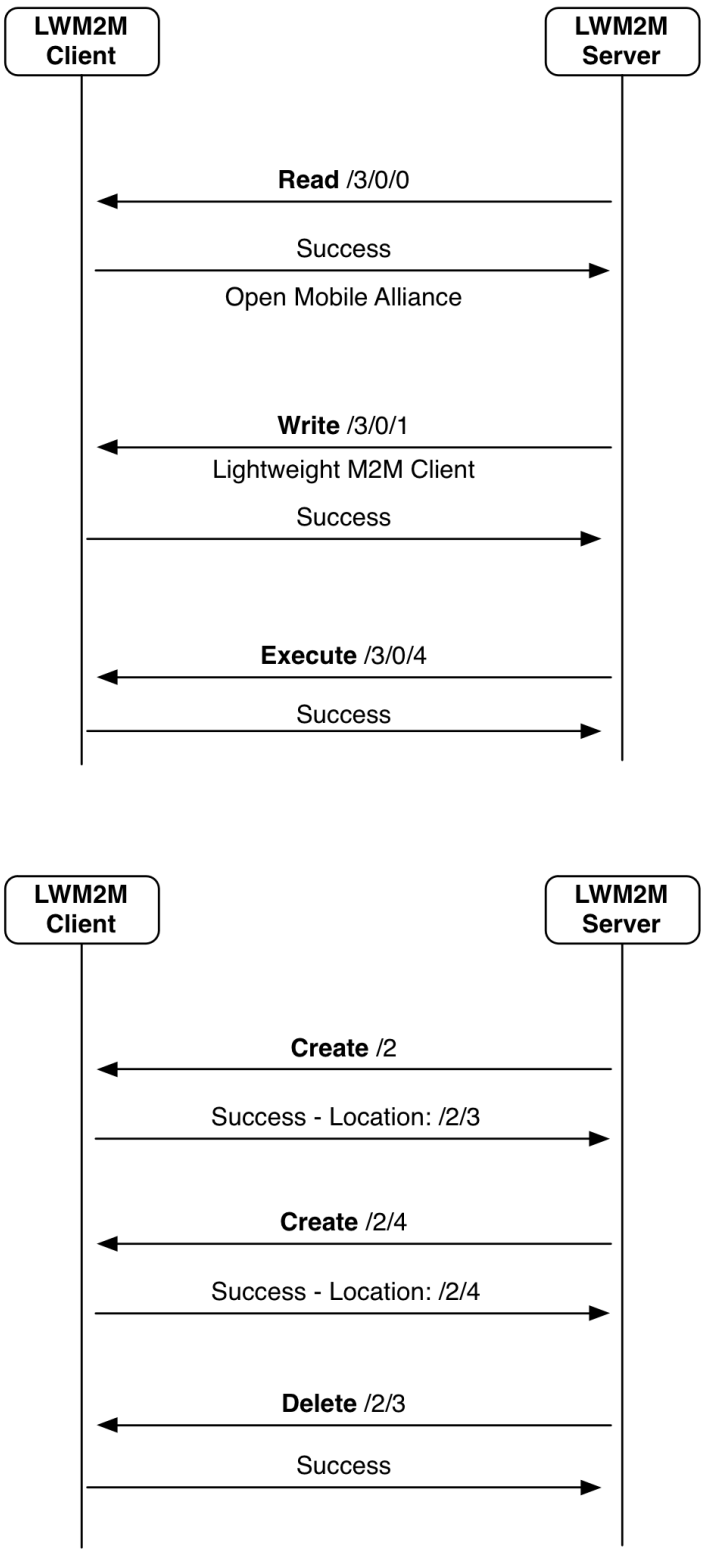

FIGURE 71 : DEVICE MANAGEMENT AND SERVICE ENABLEMENT FLOW

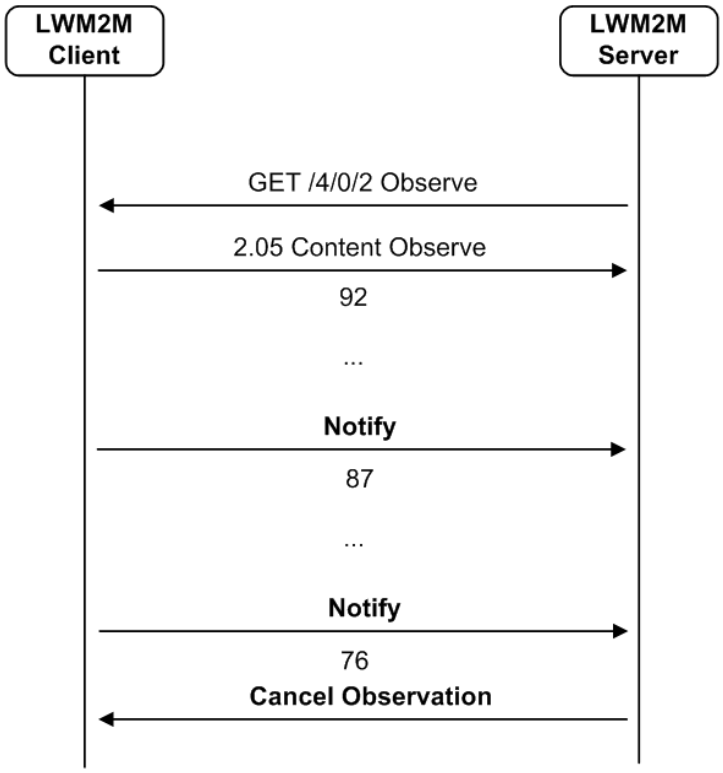

FIGURE 72 : INFORMATION REPORTING FLOW

Being the OMA LWM2M standard based on top of the CoRE CoAP application protocol, the above mentioned generic API are mapped to the GET, PUT, POST, DELETE verbs of CoAP: the mapping of relevant API are reported in Table 43 and Table 44.

TABLE 43 : DEVICE MANAGEMENT AND SERVICE ENABLEMENT OPERATION TO METHOD MAPPING

Operation	CoAP Method	Path	Success	Failure
Read	GET	/{Object ID}/{Object Instance ID}/{Resource ID}	2.05 Content	4.01 Unauthorized, 4.04 Not Found, 4.05 Method Not Allowed
Discover	GET Accept: application/link-format	/{Object ID}/{Object Instance ID}/{Resource ID}	2.05 Content	4.04 Not Found, 4.01 Unauthorized, 4.05 Method Not Allowed
Write	PUT / POST	/{Object ID}/{Object Instance ID}/{Resource ID}	2.04 Changed	4.00 Bad Request, 4.04 Not Found, 4.01

				Unauthorized, 4.05 Method Not Allowed
Write Attributes	PUT	/{Object ID}/{Object Instance ID}/{Resource ID}?pmin={minimum period}&pmax={maximum period}&gt={greater than}&lt={less than}&st={step} &cancel	2.04 Changed	4.00 Bad Request, 4.04 Not Found, 4.01 Unauthorized, 4.05 Method Not Allowed
Execute	POST	/{Object ID}/{Object Instance ID}/{Resource ID}	2.04 Changed	4.00 Bad Request, 4.01 Unauthorized, 4.04 Not Found, 4.05 Method Not Allowed
Create	POST	/{Object ID}/{Object Instance ID}	2.01 Created	4.00 Bad Request, 4.01 Unauthorized, 4.04 Not Found, 4.05 Method Not Allowed
Delete	DELETE	/{Object ID}/{Object Instance ID}	2.02 Deleted	4.01 Unauthorized, 4.04 Not Found, 4.05 Method Not Allowed

TABLE 44 : INFORMATION REPORTING OPERATION TO METHOD MAPPING

Operation	CoAP Method	Path	Success	Failure
Observe	GET with Observe option	/{Object ID}/{Object Instance ID}/{Resource ID}	2.05 Content with Observe option	4.04 Not Found, 4.05 Method Not Allowed

Cancel Observation	Reset message			
Notify	Asynchronous Response		2.04 Changed	

IPSO SMART OBJECTS IN CLOUT DEVICES

The IPSO Smart Objects "Starter Pack" is a basic set of defined (and registered) Smart Objects based on a specified collection of reusable resources. This has to be intended just as a starting collection of Objects, which can then be extended.

The basic list of IPSO Smart Objects is reported in Table 45.

TABLE 45 : IPSO SMART OBJECTS AND ASSIGNED NUMBERS

Object	Object ID	Multiple instance?
IPSO Digital Input	3200	Yes
IPSO Digital Output	3201	Yes
IPSO Analogue Input	3202	Yes
IPSO Analogue Output	3203	Yes
IPSO Generic Sensor	3300	Yes
IPSO Illuminance Sensor	3301	Yes
IPSO Presence Sensor	3302	Yes
IPSO Temperature Sensor	3303	Yes
IPSO Humidity Sensor	3304	Yes
IPSO Power Measurement	3305	Yes
IPSO Actuation	3306	Yes
IPSO Set Point	3308	Yes
IPSO Load Control	3310	Yes
IPSO Light Control	3311	Yes
IPSO Power Control	3312	Yes
IPSO Accelerometer	3313	Yes
IPSO Magnetometer	3314	Yes
IPSO Barometer	3315	Yes

The complete list of defined reusable resources can be found in IPSO Smart Objects specification document, in Table 46 some of the most common/useful resources are reported:

TABLE 46 : IPSO REUSABLE RESOURCES

Resource Name	Resource ID	Access Type	Type	Range or Enumeration	Units	Descriptions
Min Measured Value	5601	R	Float	Same as Measured Value	Same as Measured Value	The minimum value measured by the sensor since power ON or reset
Max Measured	5602	R	Float	Same as Measured Value	Same as Measured	The maximum value measured by the sensor

Value					Value	since power ON or reset
Min Range Value	5603	R	Float	Same as Measured Value	Same as Measured Value	The minimum value that can be measured by the sensor
Max Range Value	5604	R	Float	Same as Measured Value	Same as Measured Value	The maximum value that can be measured by the sensor
Sensor Value	5700	R	Float		Defined by "Units" resource.	Last or Current Measured Value from the Sensor
Sensor Units	5701	R	String			Measurement Units Definition e.g. "Cel" for Temperature in Celsius
On/Off	5850	R, W	Boolean			This resource represents an on/off actuator, which can be controlled, the setting of which is a Boolean value (1,0) where 1 is on and 0 is off.
On time	5852	R, W	Integer		s	The time in seconds that the device has been turned on. Writing a value of 0 resets the counter.

Basic objects can then be linked and aggregated to describe more complex objects.

We can report here as an example how an IPSO Temperature Sensors is represented:

TABLE 47 : IPSO TEMPERATURE

Object	Object ID	Object URN	Multiple Instances?	Description
IPSO Temperature	3303	urn:oma:lwm2m:ext:3303	Yes	Temperature sensor, example units = Cel

TABLE 48 : RESOURCES OF IPSO TEMPERATURE OBJECT

Resource Name	Resource ID	Access Type	Multiple Instances?	Mandatory	Type	Range or Enumeration	Units	Descriptions
Sensor Value	5700	R	No	Mandatory	Float			Last or Current Measured Value from the Sensor
Units	5701	R	No	Optional	String			Measurement Units

								Definition e.g. “Cel” for Temperature in Celsius.
Min Measured Value	5601	R	No	Optional	Float	Same as Measured Value	Same as Measured Value	The minimum value measured by the sensor since power ON or reset
Max Measured Value	5602	R	No	Optional	Float	Same as Measured Value	Same as Measured Value	The maximum value measured by the sensor since power ON or reset
Min Range Value	5603	R	No	Optional	Float	Same as Measured Value	Same as Measured Value	The minimum value that can be measured by the sensor
Max Range Value	5604	R	No	Optional	Float	Same as Measured Value	Same as Measured Value	The maximum value that can be measured by the sensor
Reset Min and Max Measured Values	5605	E	No	Optional	Opaque			Reset the Min and Max Measured Values to Current Value

As it can be expected, the only mandatory resource is the one reporting the actual sensor value.

Four possible data format are supported by the LWM2M standard: Text, Opaque, TLV and JSON.

The binary TLV (Type-Length-Value) format is the only mandatory one and thus is the first one adopted in our implementation of the IPSO Smart Objects for the ClouT project.

TLV is used to represent a singular value/an array of values by means of a compact binary representation. The overhead per value varies from 2 to 5 bytes depending on the type of

Identifier and length of the value itself. The maximum size of an Object Instance or Resource in this format is 16.7 MB.

This data format has a Media Type of application/vnd.oma.lwm2m+tlv

The format is an array of the following byte sequence, where each array entry represents an Object Instance, Resource, or Resource Instance:

TABLE 49 : TLV ENCODING

Field	Format and Length	Description
Type	8-bits masked field: 0bxxxxxxxx (MSB is the bit following 0b) Bit numbering is 0 for the LSB to 7 for the MSB	Bits 7-6: Indicates the type of Identifier. 00= Object Instance in which case the Value contains one or more Resource TLVs 01= Resource Instance with Value for use within a multiple Resource TLV 10= multiple Resource, in which case the Value contains one or more Resource Instance TLVs 11= Resource with Value
		Bit 5: Indicates the Length of the Identifier. 0=The Identifier field of this TLV is 8 bits long 1=The Identifier field of this TLV is 16 bits long
		Bit 4-3: Indicates the type of Length. 00=No length field, the value immediately follows the Identifier field in is of the length indicated by Bits 2-0 of this field 01 = The Length field is 8-bits and Bits 2-0 MUST be ignored 10 = The Length field is 16-

		bits and Bits 2-0 MUST be ignored 11 = The Length field is 24-bits and Bits 2-0 MUST be ignored
		Bits 2-0: A 3-bit unsigned integer indicating the Length of the Value.
Identifier	8-bit or 16-bit unsigned integer as indicated by the Type field.	The Object Instance, Resource, or Resource Instance ID as indicated by the Type field.
Length	0-24-bit unsigned integer as indicated by the Type field.	The Length of the following field in bytes.
Value	Sequence of bytes of Length	Value of the tag. The format of the value depends on the Resource's data type.

The support of JSON based representation will be evaluated in a subsequent part of the work.