



Federating IoT and cloud infrastructures to provide scalable and interoperable Smart Cities applications, by introducing novel IoT virtualization technologies

EU Funding: H2020 Research and Innovation Action GA 814918; JP Funding: Ministry of Internal Affairs and Communications (MIC)

Deliverable 5.4

Pilot Integration - Second Release

Deliverable Type:	Report
Deliverable Number:	5.4
Contractual Date of Delivery to the EU:	30.07.2021
Actual Date of Delivery to the EU:	30.07.2021
Title of Deliverable:	Pilot Integration - Second Release
Work package contributing to the Deliverable:	WP5
Dissemination Level:	Public
Editor:	Antonio F. Skarmeta (OdinS), Kenichi Nakamura (PAN)
Author(s):	Juan A. Martinez, Juan A. Sanchez, Antonio Skarmeta (OdinS), Kenji Kanai (WAS), Andrea Detti, Giuseppe Tropea (CNIT), Kenichi Nakamura (PAN), Tetsuya Yokotani (KIT), Hiroaki Mukai (KIT), Gilles Orazi, Ahmed Abid (EGM), Bin Cheng (NEC)
Internal Reviewer(s):	Giuseppe Tropea (CNIT)
Abstract:	This deliverable describes the sec- ond release of the integration of the different pilots developed within the scope of Fed4IoT.
Keyword List:	Pilot; Integration; GitHub; Kuber- netes

Disclaimer

This document has been produced in the context of the EU-JP Fed4IoT project which is jointly funded by the European Commission (grant agreement n° 814918) and Ministry of Internal Affairs and Communications (MIC) from Japan. The document reflects only the author's view, European Commission and MIC are not responsible for any use that may be made of the information it contains

Table of Contents

Abbreviations	10
Fed4IoT Glossary	12
1 Introduction	14
1.1 Purpose of the Document	14
1.2 Executive Summary	14
1.3 Quality Review	15
1.4 Progress related to previous deliverable	15
2 Codebase Management and Integration	17
2.1 GitHub Strategy and Dockerized Components	17
2.2 Kubernetes Deployment	18
3 VirIoT Cross-border Platform	19
3.1 VirIoT Infrastructure	19
3.2 VirIoT Kubernetes services	20
3.3 VirIoT Deployment Guideline	22
4 Smart Parking Pilot	23
4.1 Description of the pilot	23
4.2 Description of the components to be instantiated	23
4.2.1 Root Data Domain	23
4.2.2 VirIoT Data Domain	24
4.2.3 Tenant Data Domain	26
4.3 Deployment strategy	27
4.3.1 Deploying VirIoT components	27
4.3.2 Deploying Pilot application	34
4.3.3 Edge-based Deployment with FogFlow	35
4.4 Data Model	37
5 Carpooling Pilot	38
5.1 Description of the pilot	38
5.1.1 Deployment Site	38
5.2 Description of the components to be instantiated	39
5.2.1 Root Data Domain	39
5.2.2 VirIoT Data Domain	41
5.2.3 Tenant Data Domain	42
5.3 Deployment strategy	42
5.3.1 Root data domain deployment	43
5.3.2 VirIoT data domain deployment	44

5.3.3	Tenant data domain deployment	44
5.4	Data Model	45
6	Cross-border Person Finder Pilot	46
6.1	Description of the pilot	46
6.2	Pilot Assumption	47
6.3	Description of the components to be instantiated	48
6.3.1	ThingVisor Variations	48
6.3.2	Attribute-based Authentication	53
6.3.3	Tenant Data Domain	54
6.4	Deployment strategy	54
6.5	Data model	56
7	Wildlife Monitoring Pilot	57
7.1	Description of the pilot	57
7.2	Description of the components to be instantiated	57
7.3	Deployment strategy	58
7.4	Data model	59
8	Modular Code for ThingVisors and vSilos	61
8.1	The <code>thingvisor_generic_module.py</code> python module	61
8.1.1	<code>initialize_vthing</code>	62
8.1.2	<code>params</code>	62
8.1.3	<code>publish_attributes_of_a_vthing</code>	63
8.1.4	<code>publish_actuation_response_message</code>	63
8.1.5	<code>upstream_entities</code> and <code>upstream_tv_http_service</code>	64
9	FaceRecognition ThingVisor	66
9.1	How it works	66
9.1.1	The Camera System	67
9.1.2	The CameraSensor ThingVisor	67
9.1.3	The FaceRecognition ThingVisor	69
9.1.4	The vSilo that has a “detector” vThing	72
9.2	How to run it	75
9.2.1	Running the CameraSensor ThingVisor	75
9.2.2	Running the Camera System	76
9.2.3	Running the FaceRecognition ThingVisor	76
9.2.4	Run a vSilo	77
10	Access Control Framework instantiation in VirIoT	78
10.1	Token-based Access Control of Actuators	78
10.2	DCapBAC Component functionalities	82
10.2.1	IdM-Keyrock	83
10.2.2	XACML framework PAP and PDP	84

10.2.3	Capability Manager	84
10.2.4	PEP-Proxy	84
10.3	DCapBAC Components operation	85
10.3.1	IdM-Keyrock	85
10.3.2	XACML framework PAP and PDP	86
10.3.3	Capability Manager	89
10.3.4	PEP-Proxy	91
10.3.5	Full integration view	91
10.4	Configuring and testing	91
10.4.1	IdM-Keyrock	93
10.4.2	XACML framework PAP and PDP	96
10.4.3	Capability Manager	97
10.4.4	PEP-Proxy	99
11	Conclusion	105
12	Annex: Data model updates	106
12.1	Carpooling pilot	106
	Bibliography	109

List of Figures

1	VirIoT Infrastructure for Pilots	19
2	VirIoT main components on Kubernetes	21
3	Smart Parking architecture	24
4	Smart Parking interactions VirIoT components	25
5	Parking Site functionality	25
6	Regulated Parking Zone functionality	26
7	Virtual Silo (orion-flavour) functionality	27
8	Smart Parking map-based GUI	28
9	Deployment strategy for Smart Parking pilot	29
10	Smart Parking Pilot components interactions	34
11	FogFlow-based deployment for smart parking	36
12	Demonstration of FogFlow-based Smart Parking	37
13	Example of entrance and exit images from a carpooling parking	38
14	Site where the deployment of the carpooling parking use case is deployed. The camera is located on a mat at 4.5m height on the black spot. From this point of view it is possible to see incoming and outgoing vehicles with a single camera.	39
15	Edge software architecture	40
16	Software modules of the carpooling pilot, blue show the root data domain, green the VirIoT data domain and yellow the tenant one.	41
17	The carpool dashboard	43
18	Concept image of Cross Border Person Finder pilot	47
19	Simple Relay CBPF ThingVisor	50
20	Monolithic CBPF ThingVisor	50
21	Service chaining CBPF ThingVisor	52
22	Protection of unauthorized access to ThingVisor	54
23	Deployment plan of Cross Border Person Finder pilot	56
24	Pilot system at Kanazawa Institute of Technology	57
25	GUI of the wildlife monitoring application	58
26	Data exchanges via Fed4IoT system	59
27	An example of NGSI-LD entity published by the thermometer Virtual Thing	60
28	FaceRecognition Architecture	67
29	NGSI-LD Entity representing context information of a lamp	80
30	Virtual Actuator workflow, QoS = 2	81
31	set-on actuation-command	82
32	DCapBAC Operation Model and Blockchain	83
33	IdM-Keyrock authentication	85
34	XACML authorisation request verdict	86
35	Capability Manager request	90
36	PEP-Proxy request	92
37	Full integration view	92

38	IdM-Keyrock - Login	94
39	IdM-Keyrock - Main page	94
40	IdM-Keyrock - Link User Management	95
41	IdM-Keyrock - User registration form	102
42	IdM-Keyrock - List of registered users	103
43	PAP - Main page	103
44	PAP - Attributes	104
45	PAP - Policies	104
46	Data Model used by the smart camera, the carpool pilot uses a subset of this data model	106

List of Tables

1	Abbreviations	11
2	Fed4IoT Dictionary	13
3	Version Control Table	16
4	Virtual Things for Wildlife Monitoring	60
5	Carpool use case entities	107

Abbreviations

Abbreviation	Definition
ADN	Application Dedicated Node
AE	Application Entity
AIMD	Additive Increase/Multiplicative Decrease
AKS	Azure Kubernetes Service
API	Application Programming Interface
ASM	Adaptive Semantic Module
ASN	Application Service Node
AWS	Amazon Web Services
CBPF	Cross-border Person Finder
CIM	Context Information Management
CSE	Common Services Entity
ETSI	European Telecommunications Standards Institute
FIB	Forwarding Information Base
GE	Generic Enabler
GDPR	General Data Protection Regulation
HTTP	HyperText Transfer Protocol
ICN	Information Centric Networks
ICT	Information and Communication Technologies
IN	Infrastructure Node
IP	Internet Protocol
ISG	Industry Specification Group
JSON	JavaScript Object Notation
MANO	MANagement and Network Orchestration
MMG	Morphing Mediation Gateway
MN	Middle Node
MQTT	Message Queue Telemetry Transport
NGSI	Next Generation Service Interfaces Architecture
NGSI-LD	Next Generation Service Interfaces Architecture - Linked Data
NSE	Network Service Entity
OMA	Open Mobile Alliance
PIT	Pending Interest Table
PPP	Public-Private Partnership
RDF	Resource Description Framework
RPZ	Regulated Parking Zone
REST	Representational State Transfer
SDK	Software Development Kit
TCP	Transmission Control Protocol
TM	Topology Master

TN	Task Name
TV	ThingVisor
UML	Unified Modeling Language
URI	Uniform Resource Identifier
VNF	Virtual Network Functions
vSilo	Virtual Silo
vThing	Virtual thing
WLAN	Wireless Local Area Network

Table 1: Abbreviations

Fed4IoT Glossary

Table 2 lists and describes terms relevant to this deliverable.

Term	Definition
FogFlow	An IoT edge computing framework that automatically orchestrates dynamic data processing flows over cloud- and edge-based infrastructures. Used for ThingVisor development
Information Centric Networking	New networking technology based on named contents rather than IP addresses. Used for ThingVisor development
IoT Broker	Software entity responsible for the distribution of IoT information. For instance, Mobius and Orion can be considered as Brokers of the oneM2M and FIWARE IoT platforms, respectively
Neutral-Format	IoT data representation format that can be easily translated to/from the different formats used by IoT Brokers
Real IoT System	IoT system formed by real things whose data is exposed through a Broker
System DataBase	Database for storing system information
ThingVisor	System entity that implements Virtual Things
VirIoT	Fed4IoT platform providing Virtual IoT systems, named Virtual Silos
Virtual Silo (new name for IoT slice in D2.1)	Isolated virtual IoT system formed by Virtual Things and a Broker
Virtual Silo Controller	Primary system entity working in a Virtual Silo
Virtual Silo Flavour	Virtual Silo type, e.g. "Mobius flavour" is related to a Virtual Silo that contains a Mobius broker, "MQTT flavour" refers to a Virtual Silo containing a MQTT broker, etc.
Virtual Thing (or vThing)	An emulation of a real thing that produces data obtained by processing/controlling data coming from real things
Tenant	User that accesses the Fed4IoT VirIoT platform to develop IoT applications through a vSilo
Root Data Domain	Set of sources providing IoT information to the VirIoT platform
Federated systems	External IoT systems that share information with VirIoT (through the System vSilo), forming a NGSI-LD global federated system
System vSilo	NGSI-LD vSilo used at system level to share information of vThings with external NGSI-LD federated systems

Table 2: Fed4IoT Dictionary

1 Introduction

1.1 Purpose of the Document

This deliverable reports the second iteration for pilot integration. It provides information about the strategy that Fed4IoT has followed in integrating the various components of the pilots into software packages that can be deployed in the respective target environments.

1.2 Executive Summary

This deliverable is the result of Task 5.2: Pilot Integration, of this project. As a result of this task, this deliverable describes the VirIoT infrastructure (design, current instance, and a guideline for its deployment) as a small lab scale experiment we initiated so that a further validation could be performed in the forthcoming Deliverable 5.5 Pilot Deployment, test execution and results analysis.

Within the scope of this task, a second iteration of the pilots defined in Fed4IoT is presented, focusing on how they have been set-up and integrated with the VirIoT platform.

More specifically, this deliverable covers the following aspects:

- 1) Codebase management and integration
- 2) Cross-border structure of the VirIoT testbed platform
- 3) Pilots and their integration with VirIoT platform
- 4) The instance of the Access Control Framework in VirIoT
- 5) ThingVisors and vSilos modularity
- 6) A detailed view of a FaceRecognition ThingVisor

Regarding the codebase management and integration strategy, we define two phases. The first one focuses on how we took advantage of the GitHub portal, coupling it with Docker technology, for seamlessly integrating code of the different components. The second one focuses on Kubernetes technology, in order to demonstrate easy handling of our components.

Additionally, we show that we have already instantiated our VirIoT platform. We describe the testbed structure and how we have leveraged this technology for better development of the platform through Azure services provided by Microsoft's Cloud and the Kubernetes technology. The testbed spans EU and Japan clusters.

Regarding the pilot integration, we define, for each use case, an overview description and its aim, we identify the components of the VirIoT platform to be instantiated for the pilot (ThingVisors and Virtual Silo) and how to deploy them, and we give details about pilots' data models changes, if any, from previous deliverables of the same Work Package, where we had started designing them.

We have included a section describing the modularity of ThingVisors and vSilos, also reporting on the design of a generic thing visor python module that fosters re-usability and quick prototyping of ThingVisors, as well as a section that provides a thorough view of the FaceRecognition ThingVisor: it presents the capability of sharing a sensor among multiple applications, as well as showing how this can be implemented through a chaining of ThingVisors.

Finally, we have also described the Access Control Framework instantiation in VirIoT. We provide a thorough view of the components comprising this framework, as well as how they (inter)operate.

1.3 Quality Review

The internal Reviewer for this deliverable is Giuseppe Tropea (CNIT).

1.4 Progress related to previous deliverable

This document presents a second release of the pilot integration in VirIoT. As a second iteration, we have introduced a number of improvements, as well as new sections in order to cover all the actions performed during Task 5.2. In this sense, compared with the first release, we have added three new sections:

- 8) Modular Code for ThingVisors and vSilos, describing the modularity of the code for developing ThingVisors and vSilos.
- 9) FaceRecognition ThingVisor, detailing how this ThingVisor uses an actuation workflow to implement its capability, also explaining how its design exploits many key features of the Fed4IoT architecture.
- 10) Access Control Framework Instantiation in VirIoT, explaining how the token-based and the Distributed Capability-Based Access Control technologies are instantiated and integrated into the VirIoT platform.

Additionally, the section about the cross-border aspects of VirIoT has been extended with a new subsection presenting a set of deployment guidelines for the various components of the platform.

Finally, each pilot's contribution has been improved, updating the description of the root data domain. Likewise, the deployment strategy has been extended, including new subsections for describing the instantiation of the VirIoT components for each pilot. It is also noteworthy that a new pilot for car pooling has been included in this document, as a substitute for the Illegal Waste Deposit Management Pilot. The Annex contains the details about the used data model.

Version Control Table			
V.	Purpose/Changes	Authors	Date
0.1	ToC	Juan Antonio Martinez, Antonio F. Skarmeta (OdinS)	13/05/2021
0.2	Smart Parking section	Juan A. Sanchez, Juan A. Martinez, Antonio F. Skarmeta (OdinS)	25/05/2021
0.3	VirIoT deployment guide-line	Andrea Detti, Giuseppe Tropea (CNIT)	25/05/2021
0.4	Access Control Framework instantiation in VirIoT	Juan Andrés Sánchez, Juan Antonio Martinez and Antonio Skarmeta (OdinS)	03/06/2021
0.5	Carpooling Pilot section	Gilles Orazi (EGM)	11/06/2021
0.6	Cross-border Person Finder Pilot	Hidenori Nakazato, Kenji Kanai (WAS)	11/06/2021
0.7	Wildlife Monitoring Pilot	Kenichi Nakamura (PAN), Tetsuya Yokotani (KIT), Hiroaki Mukai (KIT)	11/06/2021
0.8	FogFlow Contribution	Bin Cheng (NEC)	07/07/2021
0.9	FogFlow Contribution	Bin Cheng (NEC)	07/07/2021
0.10	Pilot updates	Juan Antonio Martinez (OdinS), Juan Andrés Sánchez (OdinS), Giuseppe Tropea (CNIT), Gilles Orazi (EGM), Kenichi Nakamura (PAN), Tetsuya Yokotani (KIT), Hiroaki Mukai (KIT), Hidenori Nakazato (WAS), Kenji Kanai (WAS)	18/07/2021
1.0	Quality review	Giuseppe Tropea (CNIT)	26/07/2021
1.1	Final review	Andrea Detti (CNIT)	27/07/2021

Table 3: Version Control Table

2 Codebase Management and Integration

VirIoT, the platform we have developed within the Fed4IoT project, comprises different components or enablers, which allow virtualising IoT information coming from different data providers. One of the goals is to have users and consumers access this information in a controlled and secure way, including mediated access to actuators.

Setting up a reliable strategy for quality control of the source code of such a platform, which comes from different partners and, given the capabilities of the platform to manage isolated containers, is implemented using different programming languages and programming patterns, is of paramount importance.

Deliverable D2.3 on the System Architecture provides the overall design of the various interfaces. In this Section of this deliverable we focus, instead, on issues of software integration, impacting how we manage code of the Master-Controller, SystemvSilo, ThingVisors, of the various flavours of vSilos, which are implemented both in Python and in node.js programming languages, and of the security enablers.

There are different strategies that could have been applied for a seamless codebase integration of the VirIoT components, ranging from a manual approach where code is stored in different code repositories, up to more advanced ones where we can also take advantage of the latest DevOps tools for an automated deployment and continuous integration.

During the course of the project we refined the management and integration of the codebases in two steps: first we took advantage of the GitHub portal, coupling it with Docker technology, for seamlessly integrating code of the different components. Secondly we focused on Kubernetes technology, in order to demonstrate easy handling and deploy of our components. We opted for using GitHub because of the familiarity that partners have with it, as well as the well-known capabilities that it provides as one of the most-known systems for managing code. Then we started using cloud providers for a coordinated deployment thanks to the Kubernetes technology. These two phases are explained in the following subsections.

2.1 GitHub Strategy and Dockerized Components

GitHub is a well-known platform that helps assuring quality when developing services and applications. Based on git repositories, GitHub is well-known world wide because of the mechanisms it provides to services and application owners to improve the quality of their development cycle, by allowing the community of developers to download, test, and even improve the quality of progressive development, in a controlled manner, thanks to the concept of *pull-request* by which they can request changes over the main branch, which must be verified by the maintainers of the code before being integrated.

This sort of tools is also a very good approach for collaborative development, as it is the case for the VirIoT platform, which has been developed in a collaborative fashion among all partners. The advantages offered by this solution make it ideal for distributing the platform to the public, and collecting feedback from the community.

Further, Docker is one of the technologies which has gained a great popularity because of its ease at configuring the running environment, defining what is going to be inside the containers, and also speeding up universal execution of the corresponding instances.

For the key components of our platform, we have written and deployed on GitHub, alongside the component's code, automated shell scripts that take care of the creation, building and updating of the corresponding Docker images, as well as pushing them to DockerHub.

This strategy has allowed us to obtain two important objectives:

- Assuring the quality of our codebase, fostering collaboration among partners and collaborative code writing.
- Provide an easy deployment cycle of our testbeds; of our open source platform.
- Give the opportunity to external developers to improve and give feedback about our open source platform, possibly supporting validation of it by the developers' community.

2.2 Kubernetes Deployment

Kubernetes allows for an orchestrated deployment in remote servers. In a second iteration, we have progressed by allowing our VirIoT platform to be deployed using this technology. In the following Sections, we see how we have materialized the VirIoT platform, using this technology, in two data centres and edge sites in different countries, starting to pave the ground for a system that can be exploited by the consortium.

3 VirIoT Cross-border Platform

The execution of the Fed4IoT pilots is supported by a cross-border deployment of VirIoT that is composed of Virtual Machines running Kubernetes deployed in Microsoft Azure data centers and edge sites. This section briefly describes related infrastructure, Kubernetes configuration and running VirIoT components (Kubernetes Pods).

3.1 VirIoT Infrastructure

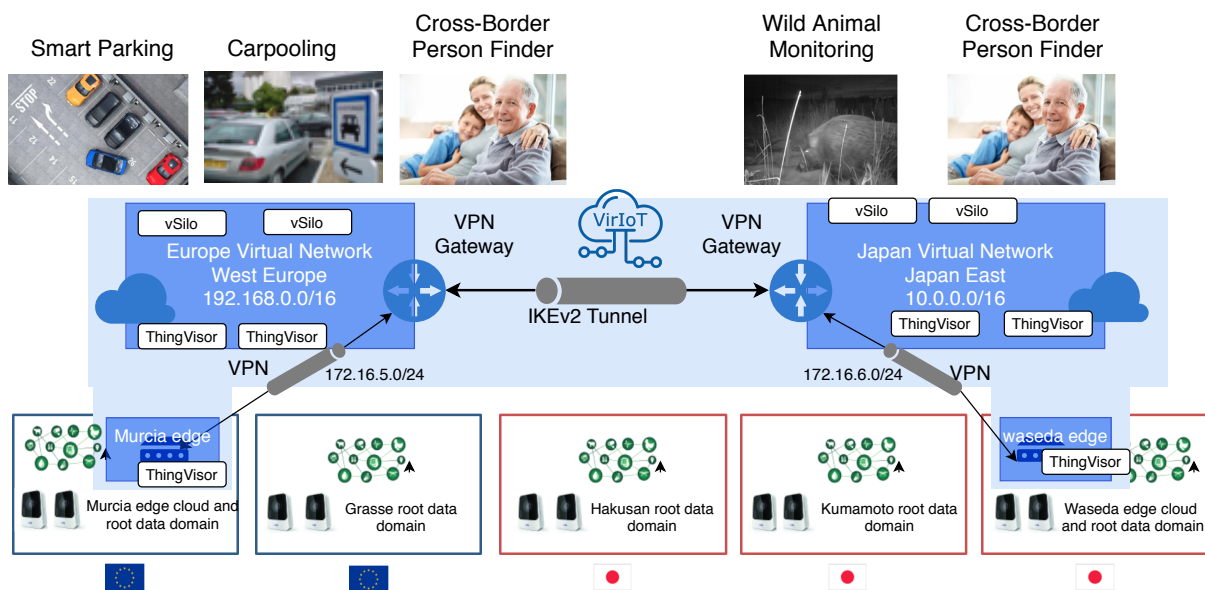


Figure 1: VirIoT Infrastructure for Pilots

Four pilot applications: smart parking, carpooling, cross-border person finder, and wildlife monitoring, are implemented using five root data domains: Murcia, Grasse, Hakusan, Kumamoto, and Waseda as shown in Figure 1¹.

Azure services of Microsoft's Cloud and some on-premise edge devices are used to implement the VirIoT infrastructure used by the four pilot applications that connects the five root data domains. About the physical location of the involved devices, we use 4 virtual machines (VMs) in Azure's West-Europe data center, 2 VMs in Azure's Japan-East data center, and two "edge" VMs located in the OdinS sites in Murcia (Spain) and Waseda University in Tokyo (Japan), respectively.

The cross-border interconnection among data centers and edge sites, as well as the relationship between pilot applications and pilot sites is visualized in Figure 1. Within each Azure data center, VMs are connected to each other by a virtual Ethernet network

¹Please note that the pilot application: Citizen Made IoT Applications is supported by the ThingVisor Factory and is now part of main components of Fed4IoT, to be shared among IoT applications. ThingVisor Factory is described in Deliverable 2.3 "System Architecture - Second Release" and Deliverable 3.2 "Cloud Oriented Services - Second Release."

that uses a unique /24 IP address class within the testbed. For the interconnection of these virtual networks, we used two Azure VPN gateways, one in EU and one in JP, which use IPsec/IKE VPN tunneling.

Extending the VirIoT platform to edge sites close to data or users has been implemented by connecting VMs running within edge sites through TLS "point-to-site" connections to Azure VPN gateways. For example, the Waseda edge site is made by a single VM connected with the JP VPN gateway via openVPN technology. Similarly, the edge site in Murcia has a VM connected to the EU VPN gateway. The EU and JP edge VMs belong to two different IP subnets (172.16.5.0/24 and 172.16.6.0/24) and the entire routing plane is controlled by the BGP protocol operating on the Azure VPN gateways.

The virtual machines are used to run ThingVisors, vSilos, as well as application programs in some of the pilot applications. ThingVisors can also run in the edge nodes. For example, some ThingVisors for cross-border person finder pilot will run on the edge nodes to save network bandwidth for image transferring. The vSilos for the smart parking pilot and carpooling pilot will run on the virtual machines in EU data center. The vSilos for the wild animal monitoring pilot will run on the virtual machines in JP data center. The vSilos for the cross-border person finder pilot will run in both data centers in order to confine person information in each region.

Virtual Machines are also used to run all control/networking service of VirIoT, i.e. Master Controller, MQTT and HTTP distribution systems, System Database, etc.

3.2 VirIoT Kubernetes services

In this section we explain the Kubernetes configuration for this testbed. Since we are leveraging Azure resources, they provide its customers with two main options for deploying a Kubernetes cluster:

- fully managed, ready-to-use Kubernetes service by Azure: Azure Kubernetes Service (AKS);
- configure your own Kubernetes cluster with resources provided by Azure (on-premise).

We have decided to go with the second options for different reasons: first of all, using Kubernetes as AKS means the cloud provider is hiding all the complexity from the user. Running it as an on-prem bare metal deployment, means you're on your own for managing these complexities – including persistent storage, load balancing, configmaps, services, availability, auto-scaling, networking, roll-back on faulty deployments, and more. This is translated into higher complexity, but indeed better understating of all the components running on the cluster, giving the ability to fine tune the requirements of the platform for all the contributors of the project.

Now we will dive into the main characteristics of the cluster. We have set up the Kubernetes cluster across all the available VMs in this manner: one machine is used as a Kubernetes master node and is located inside the Azure EU data center, while all the remaining ones, three in Europe, two in the Japan region, and other in edge sites are suited for being working nodes.

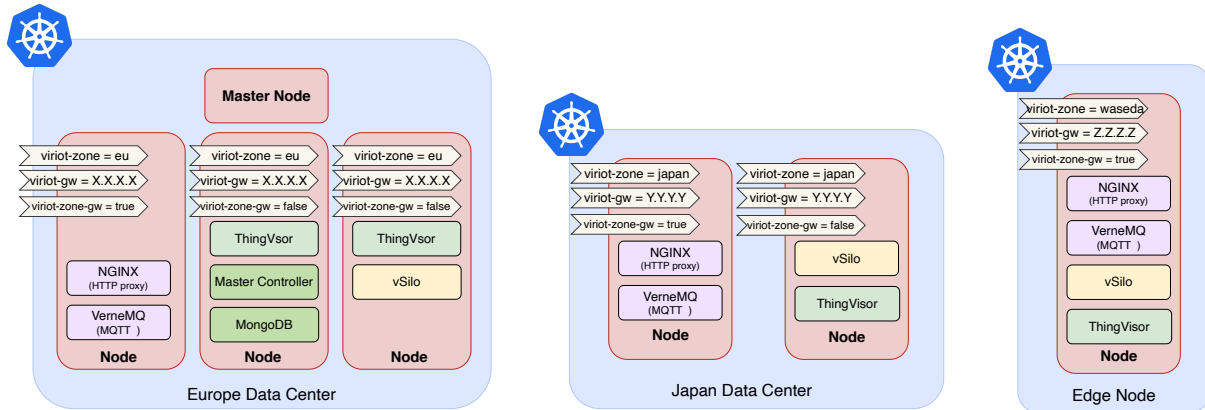


Figure 2: VirIoT main components on Kubernetes

To support edge computing functionality, we have exploited Kubernetes labels. Kubernetes labels enable users to map their own organizational structures onto system objects in a loosely coupled fashion, without requiring clients to store these mappings. By using labels on nodes, it is possible to constrain the running of a pod/container to specific nodes that matches the exact label value (*node-affinity*) or, on the contrary, that pod should avoid being allocated on a particular nodes with a different label (*pod-anti-affinity*). We are leveraging the aforementioned *node-affinity* to consider a Kubernetes distributed cluster formed by *zones* that are data centers and/or edge sites. Nodes of a zone must be labelled with “viriot-zone” and “viriot-gw” labels, as follows: the value of the “viriot-gw” key must contain a public IP address through which it is possible to access the data center or the edge node; the value of the “viriot-zone” is a unique name that identifies the zone. We have depicted our high-level cluster view in Figure 2, as we can see, we have used the “viriot-zone” and “viriot-gw” labels to differentiate nodes where to deploy VirIoT services.

As mentioned in D3.2, VirIoT platform needs some necessary elements:

- the Master-Controller
- the SystemDB
- the MQTT distribution system for context data and control messages
- the HTTP distribution system for large content (and more)

Master-Controller and SystemDB run in the Azure’s EU data center. The MQTT distribution system is made by a cluster of MQTT Brokers and we used VerneMQ software. The HTTP distribution system is made by a cluster of HTTP proxies and we used NGINX software. A single instance of a MQTT Broker and a HTTP proxy must run per data center / edge node and the selected node is the one of the zone that has the label “viriot-zone-gw=true”. In this way we created a topology based data distribution tree and optimized the data transfer between data centers / edge nodes, i.e. viriot-zones.

3.3 VirIoT Deployment Guideline

VirIoT is a microservice architecture therefore its services can be deployed in any node of the cluster. As deployment guidelines we recommend what follows:

- Master-Controller and System Database are the most critical services in the VirIoT control plane, so they should be deployed in a high reliability node. Interactions with them occur only during ThingVisors and vSilos configuration, network traffic is low, and network latency is not critical for performance.
- HTTP proxies and MQTT brokers are the services of a zone that interconnect the zone with other zones. Should be deployed in the highest reliable node of the zone.
- ThingVisors can process data from real sensors and/or activate real actuators. The vSilos send and receive data from tenant applications connected to them. Consequently, it is convenient to deploy ThingVisors in VirIoT zones near the real things they interact with and vSilos in VirIoT zones near the user applications they interact with. This reduces network traffic and latency.

4 Smart Parking Pilot

4.1 Description of the pilot

The Smart Parking pilot tries to deal with one of the most common issues of big cities, traffic congestion. One of the main causes which contributes to this problem is the number of vehicles wandering along the city, searching for a parking spot in a certain destination area. Smart Parking provides a solution that allows them to reduce the amount of time for this activity.

In this Smart Parking use case, the pilot is focused in Murcia, which is a city located in the south-east of Spain, and with a population of 450.000 citizens. This city has experienced a dramatic rise of accesses to the city centre in the last years, which provoke a considerable increase in traffic congestion. Day-by-day, commuters, tourists and families traveling by car collapse the city centre with cars intending to park at commerce, financial and historical areas.

The aim of this Smart Parking use case is taking advantage of Fed4IoT framework to provide a service that tracks the state of the parking spots, to provide the drivers with this information beforehand and, as a consequence, to get more fluid traffic in the centre of the city.

4.2 Description of the components to be instantiated

4.2.1 Root Data Domain

The Smart Parking solution provided by our Fed4IoT framework integrates the information coming from the FIWARE-based Mi-Murcia platform. Figure 3 presents the most relevant components of the envisioned platform according to this concrete use case.

Our Smart Parking use case receives two types of context sources:

- The availability of private parking sites in terms of unoccupied parking spots.
- The probability to be able to park in the Regulated Parking Zone (RPZ). obtained via a (Machine Learning) model, trained based on the logged history of daily expended tickets.

For the case of the parking sites, the sensors, deployed in each private parking site, send the number of actual unoccupied parking spots when a vehicle enters or exits from the parking site. Usually, an IoT gateway is required to transmit this information to an IoT platform too.

For the case of RPZ, since in Murcia city we count with old-fashion parking meters equipped with highly-constrained CPU, during the day they are completely dedicated to the ticket issuance task. These devices take advantage of the night time to perform the transmission of the activity of the whole day, providing detailed information regarding the expended tickets.

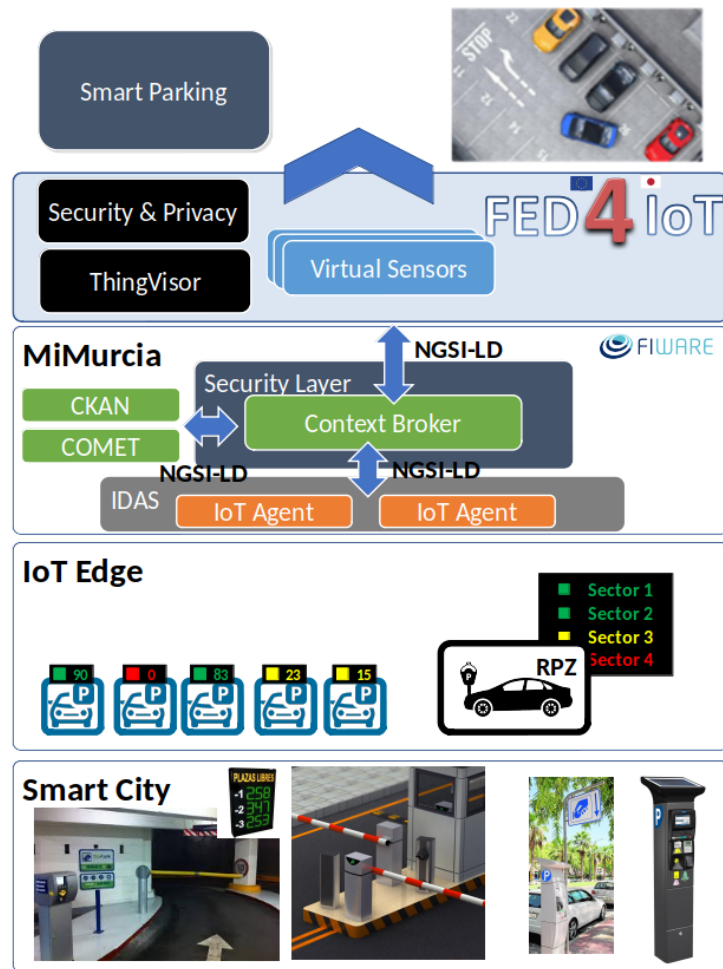


Figure 3: Smart Parking architecture

4.2.2 VirIoT Data Domain

Here we describe all components we need to instantiate in the VirIoT platform in order to have an operational Smart Parking pilot. These components are:

- **Parking Site ThingVisor**, which obtains the parking sites information coming from the FIWARE-based Mi-Murcia platform.
- **Regulated Parking Zone (RPZ) ThingVisor**, which obtains the RPZ information coming from the FIWARE-based Mi-Murcia platform.
- **Virtual Silo (orion-flavour)**, which receives the parking sites and RPZ information from the above ThingVisors and offers it to the Smart Parking pilot GUI.

Figure 4 shows the interactions between specific components of Smart Parking in VirIoT.

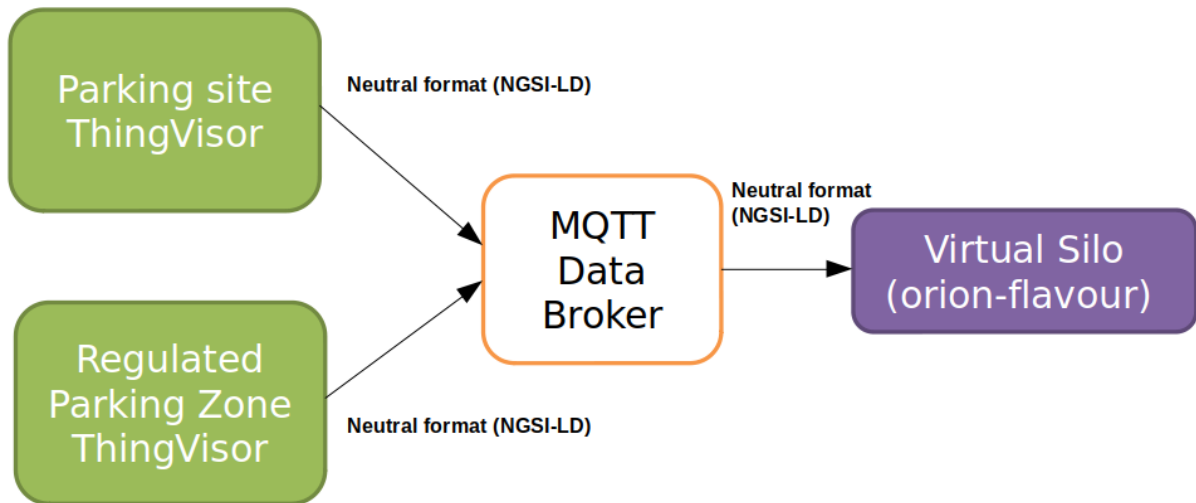


Figure 4: Smart Parking interactions VirIoT components

4.2.2.1 Parking Site ThingVisor

This ThingVisor component obtains parking sites information from the FIWARE-based Mi-Murcia platform. To do this, it subscribes to the entities of the platform which contain this specific information. So, when ThingVisor receives the notifications from the platform (NGSIV2 format), it processes its payload and produces a neutral format payload (NGSI-LD) which is sent to MQTT broker in a specific vThing topic.

Once the ThingVisor sends the NGSI-LD payload, the vSilos that are subscribed to the corresponding vThings will receive the information. Figure 5 shows this functionality by depicting the interactions commented above.

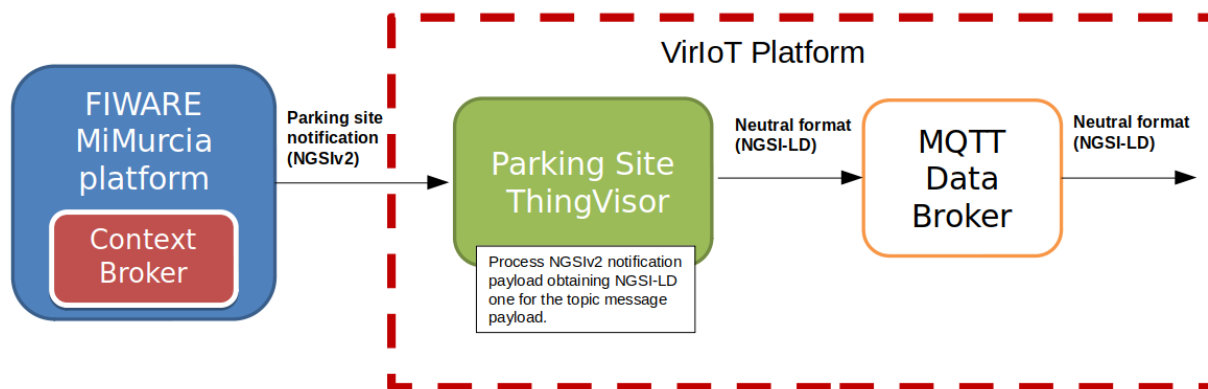


Figure 5: Parking Site functionality

4.2.2.2 Regulated Parking Zone (RPZ) ThingVisor

The functionality of this ThingVisor is the same as indicated in the previous **Parking Site ThingVisor**. The unique difference is that this ThingVisor subscribes to the FIWARE-based Mi-Murcia platform to obtain specific Regulated Parking Zones information as presented in Figure 6.

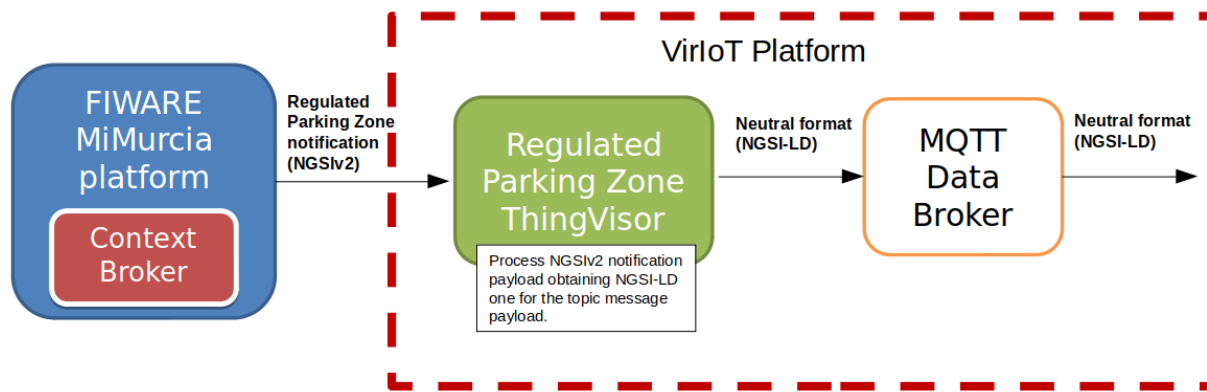


Figure 6: Regulated Parking Zone functionality

4.2.2.3 Virtual Silo

The vSilo component receives Neutral-Format data (NGSI-LD) through the platform's MQTT broker it is connected to. It receives whatever NGSI-LD payload was previously published by the ThingVisors to the platform's MQTT broker.

This component has a vSilo Controller which processes the Neutral-Format data and converts it to NGSIv2 data, which is stored in an Orion Context Broker, embedded in the vSilo, by issuing a request to the NGSIv2 API. This way, the vSilo offers to Smart Parking an NGSIv2 API to access its data, i.e., the information of parking sites and RPZ. To access data, NGSIv2 offers two options, either using the entity queries or through the subscription mechanism.

Figure 7 summarizes the functionality of this vSilo.

4.2.3 Tenant Data Domain

This solution will provide a GUI, as depicted in Figure 8, allowing the user to specify both the current location and the destination, as well as parking duration, the time when she will arrive and other user preferences (maximum desired cost, maximum desired distance from parking to destination, ...) by presenting a map-based web interface/App. Once the selection is made, our Smart Parking solution makes a complex reasoning to generate an informed recommendation about the best destination area where to park the vehicle.

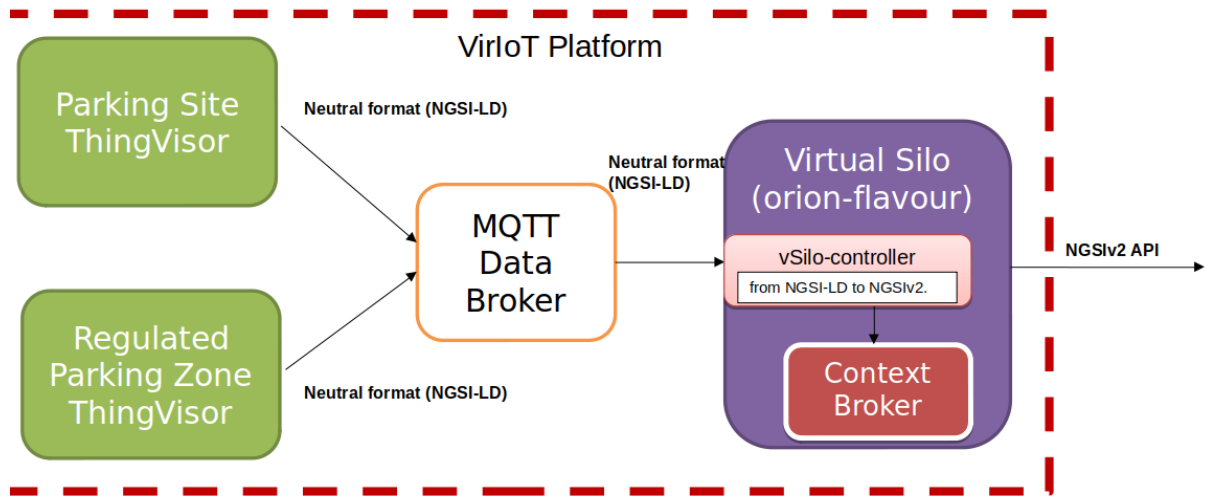


Figure 7: Virtual Silo (orion-flavour) functionality

4.3 Deployment strategy

This subsection details how the components of the Smart Parking pilot that were mentioned and detailed in previous Section 4.2 will be integrated into the VirIoT platform.

On the one hand, Smart parking ThingVisors and vSilo will be deployed in the edge node of EU by the Master-Controller. There are two methods to deploy components through the Master-Controller:

- using the Command Line Interface (VirIoT/CLI)
- using Master-Controller's API

Once ThingVisors are deployed, notifications received from the FIWARE-based Mi-Murcia platform are processed and sent to the vSilo, which then is able to offer the data through its NGSIv2 API of its local Orion Context Broker.

On the other hand, the Smart Parking pilot application will be deployed in a chosen environment (cloud or local one) and can obtain Smart Parking information by requesting the corresponding information directly to the vSilo Broker via the standard NGSIv2 API. This instance of the VirIoT platform is presented in Figure 9 where the specific deployment strategy is presented.

4.3.1 Deploying VirIoT components

As introduced in 4.3, there are two methods to deploy components through the Master-Controller. This section details how to deploy, in the EU node (default zone), the required ThingVisors and vSilo required by Smart Parking use case using both methods. We have assumed that the Master-Controller is already running in the VirIoT platform.

If we opt to use the VirIoT/CLI way, by default, the commands exposed are launched in the same environment (machine) of Master-Controller using therefore its loopback

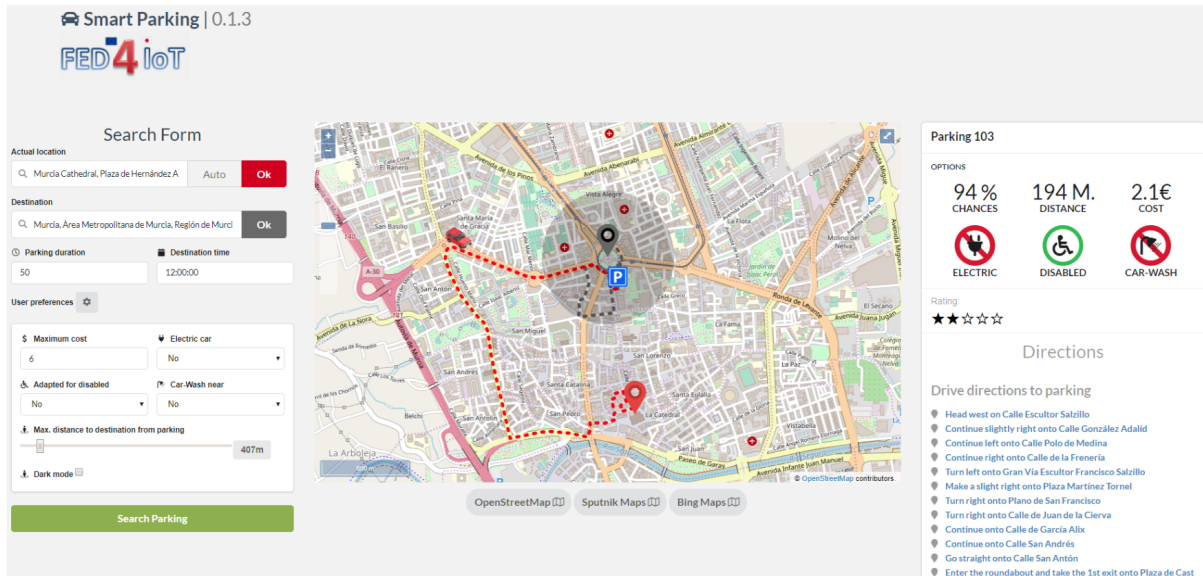


Figure 8: Smart Parking map-based GUI

interface for accessing its services e.g. using the URL <http://127.0.0.1:8090>. Anyway, we can run these commands from a remote machine replacing the URL with the public address where the VirIoT platform is running.

On the other hand, in case we are using the Master-Controller's API, by default, the exposed commands are launched from a remote machine and, in this sense, a public address is required. In this section, YAML files are used for deploying process and, in case using Master-Controller's API, *PublicAddressMC* corresponds with its public address.

Before try to deploy any component, first we need to login in Master-Controller. To see more details about the Login process review D3.2 section 2.1.3 Login. After performing this task, an access token is received which is required for subsequent requests when using the Master Controller's API.

Starting the deployment process, the following boxes show how to deploy Parking Site ThingVisor:

Using VirIoT/CLI - Parking Site ThingVisor

```
python3 f4i.py add-thingvisor -c http://127.0.0.1:8090 -n thingvisorid-
parkingsite -p '{"ocb_ip':'fiware-dev.inf.um.es', 'ocb_port':'1026'}"
-d "thingvisorid-parkingsite" -y "../yaml/thingVisor-murcia-
parkingsite.yaml" -z default
```

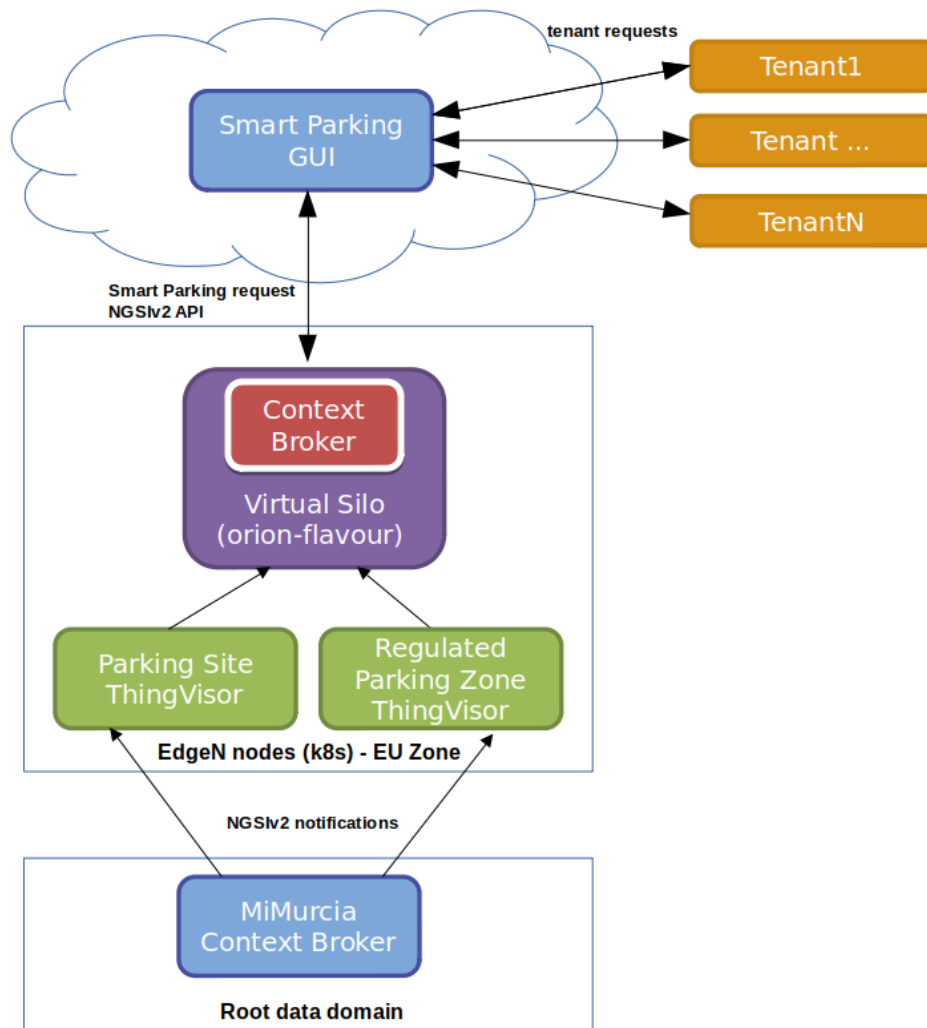



Figure 9: Deployment strategy for Smart Parking pilot

Using Master-Controller's API - Parking Site ThingVisor POST <http://{PublicAddressMC}/addThingVisor>

```
{
  "thingVisorID": "thingvisorid-parkingsite",
  "params": "{\"ocb_ip\":\"fiware-dev.inf.um.es\", \"ocb_port\": \"1026\"}",
  "description": "thingvisorid-parkingsite",
  "yamlFiles": "../yaml/thingVisor-murcia-parkingsite.yaml",
  "tvZone": "default"
}
```

HEADERS: _____

Content-Type: application/json
Accept: application/json
Authorization: Bearer {{token}}

The following boxes show how to deploy Regulated Parking Zones ThingVisor:

Using VirIoT/CLI - Regulated Parking Zones ThingVisor

```
python3 f4i.py add-thingvisor -c http://127.0.0.1:8090 -n thingvisorid-rpz -p '{"ocb_ip':'fiware-dev.inf.um.es', 'ocb_port':'1026'}" -d "thingvisorid-rpz" -y "../yaml/thingVisor-murcia-rpz.yaml" -z default
```

Using Master-Controller's API - Regulated Parking Zones ThingVisor POST `http://{PublicAddressMC}/addThingVisor`

```
{
  "thingVisorID": "thingvisorid-rpz",
  "params": "{\"ocb_ip\":\"fiware-dev.inf.um.es\", \"ocb_port\": \"1026\"}",
  "description": "thingvisorid-rpz",
  "yamlFiles": "../yaml/thingVisor-murcia-rpz.yaml",
  "tvZone": "default"
}
```

HEADERS: _____

Content-Type: application/json
Accept: application/json
Authorization: Bearer {{token}}

Before deploying the Virtual Silo, the orion-flavour must be added:

Using VirIoT/CLI - Add orion-flavour

```
python3 f4i.py add-flavour -c http://127.0.0.1:8090 -f orion-f -s "" -d "silo with a FIWARE Orion broker" -y "../yaml/flavours-orion.yaml" -d "silo with a FIWARE Orion Context Broker"
```

Using Master-Controller's API - Add orion-flavour

POST `http://{PublicAddressMC}/addFlavour`

```
{
  "flavourID": "orion-f",
  "flavourParams": "",
  "flavourDescription": "silo with a FIWARE Orion broker",
  "yamlFiles": "../yaml/flavours-orion.yaml"
}
```

HEADERS: _____

Content-Type: application/json
Accept: application/json
Authorization: Bearer {{token}}

Once orion-flavour is added, it is the turn for deploying Virtual Silo:

Using VirIoT/CLI - Virtual Silo (orion-flavour)

```
python3 f4i.py create-vsilo -c http://127.0.0.1:8090 -f orion-f -t smart
-s parking -z default
```

Using Master-Controller's API - Virtual Silo (orion-flavour)

POST `http://{PublicAddressMC}/siloCreate`

```
{
  "flavourID": "orion-f",
  "tenantID": "smart",
  "vSiloName": "parking",
  "vSiloZone": "default"
}
```

HEADERS: _____

Content-Type: application/json
Accept: application/json
Authorization: Bearer {{token}}

The following boxes show how to include data from Smart Parking ThingVisors. These ThingVisors, as mentioned in section 4.2.2.1 and section 4.2.2.2, obtain information from

the FIWARE-based Mi-Murcia platform related to Smart Parking use case. Section 4.4 shows an overview of the kind of entities obtained from them.

Using VirIoT/CLI - Add vThings to Virtual Silo

```
python3 f4i.py add-vthing -c http://127.0.0.1:8090 -v thingvisorid-  
parkingsite/parkingsite -t smart -s parking;  
python3 f4i.py add-vthing -c http://127.0.0.1:8090 -v thingvisorid-  
parkingsite/policy -t smart -s parking;  
python3 f4i.py add-vthing -c http://127.0.0.1:8090 -v thingvisorid-rpz/  
parkingmeter -t smart -s parking;  
python3 f4i.py add-vthing -c http://127.0.0.1:8090 -v thingvisorid-rpz/  
policy -t smart -s parking;  
python3 f4i.py add-vthing -c http://127.0.0.1:8090 -v thingvisorid-rpz/  
sector -t smart -s parking;
```

Using Master-Controller's API - Add vThings to Virtual Silo

POST `http://{PublicAddressMC}/addVThing`

```
# Body request to add parking sites...
{
  "tenantID":"smart",
  "vThingID":"thingvisorid-parkingsite/parkingsite",
  "vSiloName":"parking"
}

# Body request to add parking site policies...
{
  "tenantID":"smart",
  "vThingID":"thingvisorid-parkingsite/policy",
  "vSiloName":"parking"
}

# Body request to add parking meters...
{
  "tenantID":"smart",
  "vThingID":"thingvisorid-rpz/parkingmeter",
  "vSiloName":"parking"
}

# Body request to add parking meter policies...
{
  "tenantID":"smart",
  "vThingID":"thingvisorid-rpz/policy",
  "vSiloName":"parking"
}

# Body request to add parking meter sectors...
{
  "tenantID":"smart",
  "vThingID":"thingvisorid-rpz/sector",
  "vSiloName":"parking"
}
```

HEADERS: _____

Content-Type: application/json

Accept: application/json

Authorization: Bearer {{token}}

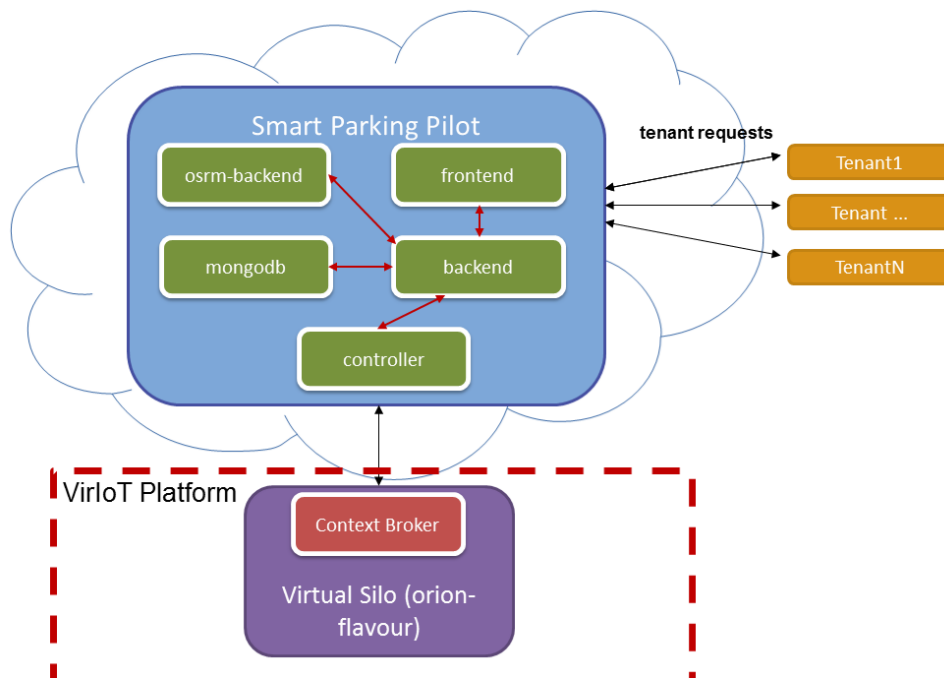


Figure 10: Smart Parking Pilot components interactions

4.3.2 Deploying Pilot application

As introduced in 4.3, the Smart Parking pilot application can be deployed in a chosen environment. The pre-requisites of this environment are docker and docker-compose working installations, since they are needed to deploy it.

Towards this goal, a docker-compose.yml file is available describing where the next components will be deployed:

- frontend, backend and mongodb: Main components of Smart Parking application. They offer the GUI, services and database to store the configuration.
- controller: Auxiliar component. This component is responsible for connecting to the Virtual Silo (orion-flavour) of Smart Parking's use case, in other words, when the user requires a parking spot recommendation, this component sends the corresponding geolocation request to the Virtual Silo to obtain parking place candidates and finally responds to the backend component with the parking's recommendation.
- osrm-backend: Auxiliar component. High performance routing engine written in C++14 designed to run on OpenStreetMap data. Used by the frontend component to show the route to follow from the current/origin location to the recommended parking spot.

Figure 10 shows Smart Parking pilot components interactions.

Although no further configuration is required in the docker-compose file, it is recommended to review the following environment variables of the controller component, so that they point out to the endpoint of Orion Context Broker (NGSIV2) of the vSilo:

- `pep_proxy_protocol`, `pep_proxy_host` and `pep_proxy_port`: Needed to obtain entities with contain a location attribute and can be filtered by it. In this sense, it considers the parking site and parking meter entity types.
- `ngsiv2_protocol`, `ngsiv2_host`, `ngsiv2_port`: Needed to obtain entities with contain a location attribute and can be filtered. In this sense, it considers the parking site and parking meter entity types.

Before deploying using docker-compose, a previous step is required in order to configure the osrm-backend container that will be deployed. Please launch the `get_OSRM_files.sh` script using the command line, being sure to launch when you are in the same folder of this file. The outcome of this script is a data folder creation that will be passed to osrm-backend container.

The next box shows the contents of the `get_OSRM_files.sh` file:

`get_OSRM_files.sh`

```
mkdir ./data
curl http://download.geofabrik.de/europe/spain-latest.osm.pbf -o ./data/
    spain-latest.osm.pbf
docker run --rm -t -v "${PWD}/data:/data" osrm/osrm-backend osrm-extract
    -p /opt/car.lua /data/spain-latest.osm.pbf
docker run --rm -t -v "${PWD}/data:/data" osrm/osrm-backend osrm-
    partition /data/spain-latest.osm
docker run --rm -t -v "${PWD}/data:/data" osrm/osrm-backend osrm-
    customize /data/spain-latest.osm
```

Finally, to deploy Smart Parking pilot application launch:

`deploy Smart Parking pilot`

```
docker-compose build;
docker-compose up -d;
```

Once every Smart Parking containers is running, the GUI is accessible at `http://localhost:81/login` using a web browser. Now, the functionality defined in section 4.2.3 is available.

4.3.3 Edge-based Deployment with FogFlow

In this advanced scenario we use FogFlow as the underlying ThingVisor factory to instantiate the Virtual Things required by the smart parking use case at the edge. This

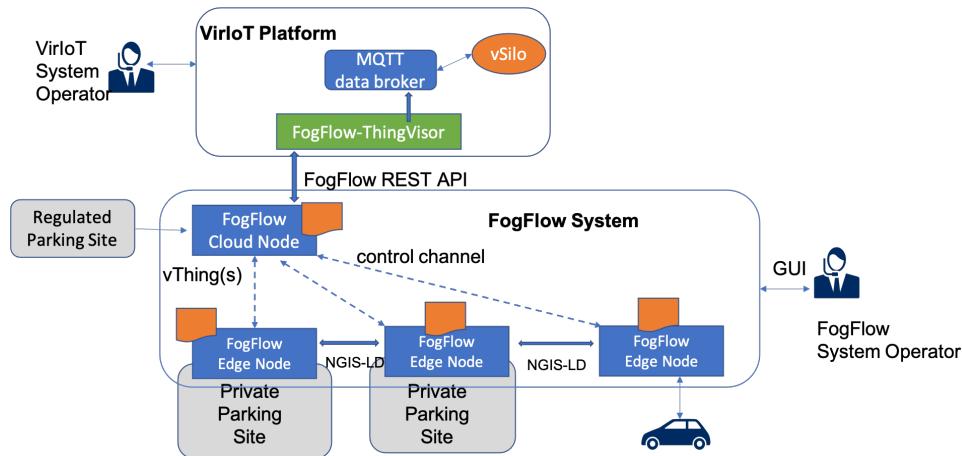


Figure 11: FogFlow-based deployment for smart parking

alternative implementation based on FogFlow can demonstrate the following benefit of creating and managing Virtual Things at the edge: 1) reducing the bandwidth consumption between physical devices and vSilos; 2) enabling and managing the direct communication between Virtual Things fully at the edge for fast response time; 3) reducing the concern on data privacy; 4) realizing various thing visors based on the same programming model in FogFlow.

Based on the intent-based programming model in FogFlow and its application template, we have realized three types of ThingVisors: 1) Private Parking Site, which presents the parking site managed by private companies; 2) Public Parking Site, which represents the regulated parking zones managed by the Murcia city government; 3) Car, which is to simulate a connected car used by a citizen. Each of these three ThingVisors is programmed as a FogFlow application, which defines the common logic of how to create and manage multiple instances of Virtual Things with the same type. Therefore, as an advanced ThingVisor factory, FogFlow can, not only simplify the programming of ThingVisors via the same programming model, but also ease and automate the management of Virtual Things associated with the same type of ThingVisor.

Figure 11 shows the deployment view of the FogFlow-based smart parking use case. The entire FogFlow consists of a centralized FogFlow cloud node and a set of FogFlow edge nodes, each of which is deployed locally at a private parking site. Three ThingVisor applications are registered to the FogFlow system via its web-based dashboard. After that, those ThingVisor applications can be enabled or disabled, either by FogFlow system operator via FogFlow dashboard or by VirIoT system operator via the implemented FogFlow-ThingVisor, which is a special VirIoT ThingVisor managed by the VirIoT master as a proxy to communicate with the FogFlow system, including fetching the information of Virtual Things from FogFlow and then making it available to vSilo via the VirIoT data broker.

Figure 12 shows the screenshot of this demo. We simulate a connected car and a number of parking sites, including both regulated parking sites and private parking sites.

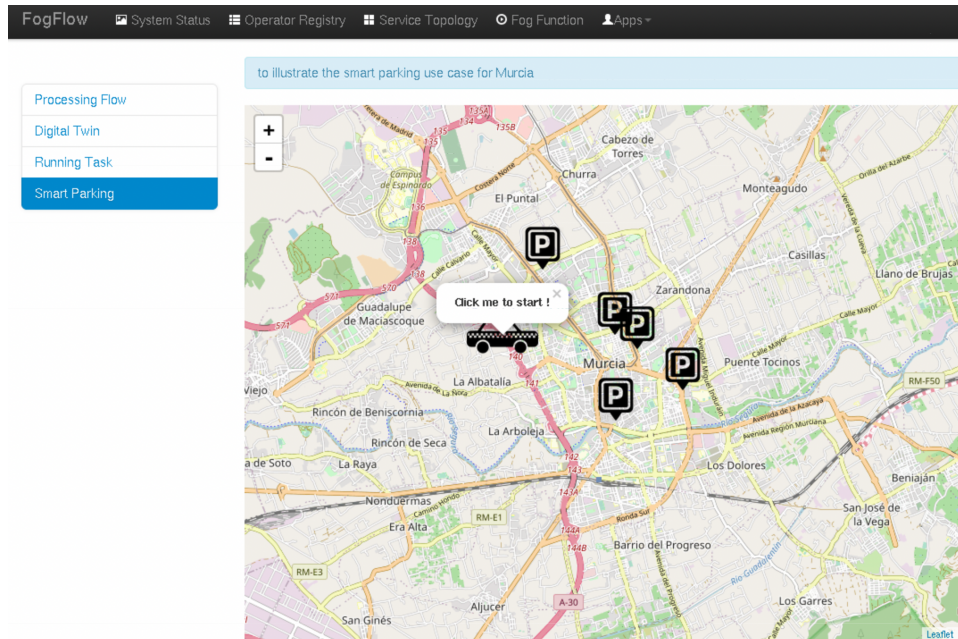


Figure 12: Demonstration of FogFlow-based Smart Parking

Each parking site is associated with a virtual thing, created by either Public Parking Site ThingVisor or Private Parking Site ThingVisor. Once the connected car joins the system from a nearby FogFlow edge, a virtual thing for the car will be created as well. The virtual of this connected car will communicate with the Virtual Things of the parking sites in the destination area and search for a parking site with free parking lots around the arrival time. The driver of the connected car will be informed with the recommended parking site.

4.4 Data Model

Data model used for Smart Parking is not updated from deliverable D5.2.

5 Carpooling Pilot

5.1 Description of the pilot

The idea behind this pilot is to use a camera associated with some deep learning machine vision algorithm to perform the monitoring of a given site, and to send events as they occur on the field. Our aim is to apply this scheme to the detection entries and exits of the cars in a carpooling parking to compute some statistics about its usage.

Carpooling parkings are created by local governments to encourage people to gather in a single car to drive in a common place. This is often used to commute. When they create such spaces, the authorities need to have an impact measurement, but this is difficult because the stay is free and there is thus no barrier which delivers tickets at the entrance.

In this pilot, the smart camera monitors the entrance and exit of each car and use its license plate number to uniquely identify it, so it is possible to compute statistics and monitor the usages of the parking. How many cars park in them each day? What is the average parking time? Which kind of cars are using it? Could we infer from "in and out" events how much CO_2 emissions are avoided by the carpoolers using this parking?

The camera sends anonymized detection events (not images) to the VirIoT infrastructure so the application dashboard can be updated in near real time while preserving private data of the parking users.



Figure 13: Example of entrance and exit images from a carpooling parking

5.1.1 Deployment Site

The deployment site is located in France, near the town of Perpignan, in a carpooling parking situated close to the highway. Its entrance and exit can be monitored using a single camera (see Figure 14).

The chosen hardware, to be deployed on site, is built around a Jetson Nano computer. This is a low power, small form factor computer for embedded edge computing built by Nvidia. It runs a quad-core CPU associated with a 472 GFlops GPU that is able to run deep learning algorithm, its maximum power consumption running at full GPU capacity is around 10W. However, this might be an issue especially during the summer and some software mitigation strategies are needed to avoid overheating.

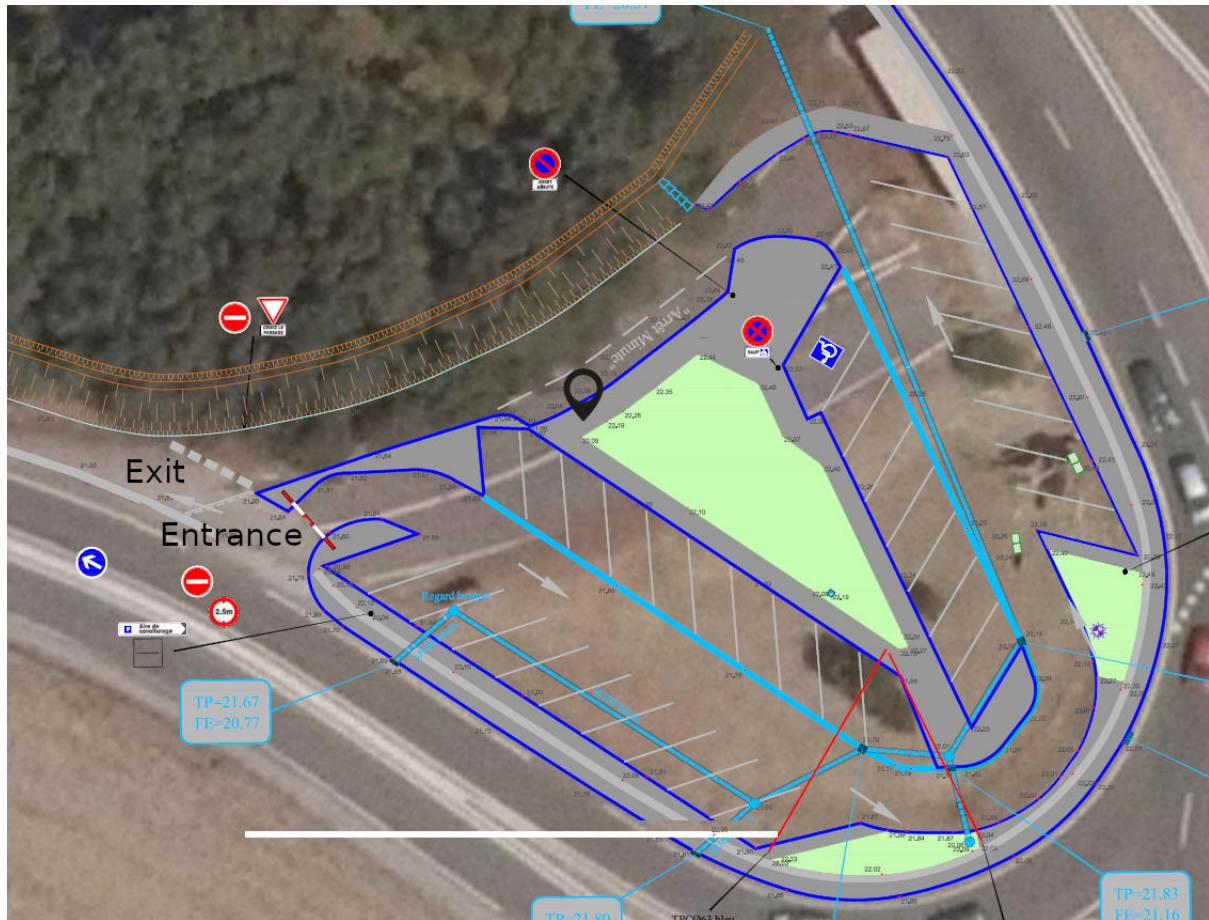


Figure 14: Site where the deployment of the carpooling parking use case is deployed. The camera is located on a mat at 4.5m height on the black spot. From this point of view it is possible to see incoming and outgoing vehicles with a single camera.

The camera has a 4MP sensor with IR illumination for night vision up to 50m. It also provide an optical zoom and pan/tilt abilities to tune the framing of the image. The whole system is powered by a 24V battery, 100m away, charged by the current taken from the lighting system of the highway during the night.

5.2 Description of the components to be instantiated

5.2.1 Root Data Domain

On the edge side, the software architecture is as depicted in Figure 15. The camera is providing an RTSP video stream which is passed through the processing pipeline with the following steps: motion detection, license plate detection, license plate reader and an event generator.

The motion detection is first used to get a rough selection of images that may contain interesting events. This is done using a motion detector based on classic computer vision

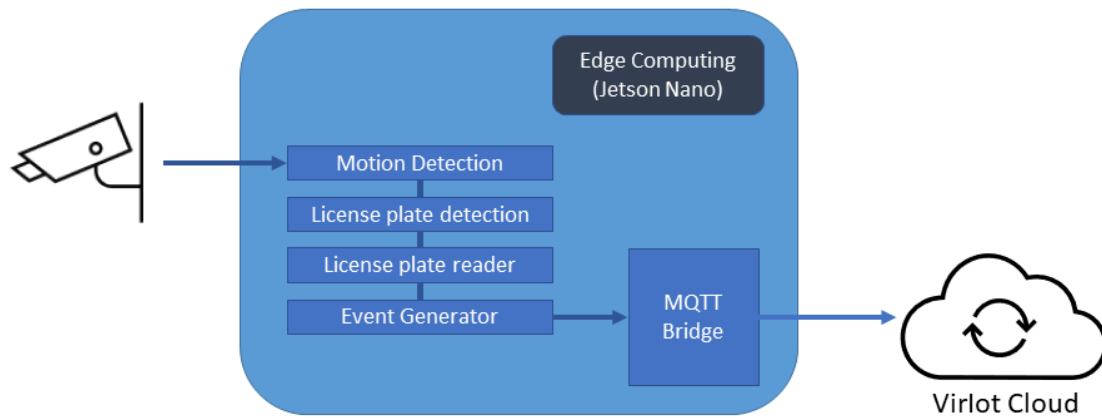


Figure 15: Edge software architecture

algorithms to produce a frame by frame estimate of the foreground characteristics (like number of pixels of the foreground, number and sizes of pixel clusters, ...). A simple machine learning model was trained to use them to determine if the frame may contain an event of interest. This relatively light processing step is implemented to avoid processing capabilities saturation by overfeeding the rest of the processing pipeline. This also helps a lot to lower the CPU/GPU temperature, which is an issue in this deployment since the Jetson Nano stands in a box on which the sun can hit very hard especially during the summer.

Then, a license plate detection step is done. It runs a Yolo-lite model trained using a transfer learning technique with some images taken from this deployment. This provides 1/ another filter for signal over noise reduction and 2/ the coordinates of the license plates seen in the image.

The result of this processing, as well as the whole raw frame is then passed to the license plate reader step which outputs the list of license plates in the image. If this list is not empty, the result is processed by the event generator that send the events via an MQTT bridge, connected to the MQTT broker of the root data domain infrastructure. The MQTT bridge is used to mitigate connectivity loss as it uses QOS of 1. This means that is the network connectivity is lost for a while, the events are stored in the local MQTT broker until it is restored. Then, the cached events are sent normally to the cloud and no one is lost.

The system has been deployed on the parking on 9th September of 2020. We faced various field-related issues like camera tuning (especially during the night), CPU/GPU overheating or even a spider web moving in the wind triggering fake motion detection.

Once the Edge processing is done, the message is received on a virtual computer in the cloud that does the following pre-processings: query a license plate public database to get some data of interest concerning the car (as vehicle type, engine type, CO_2 emissions)

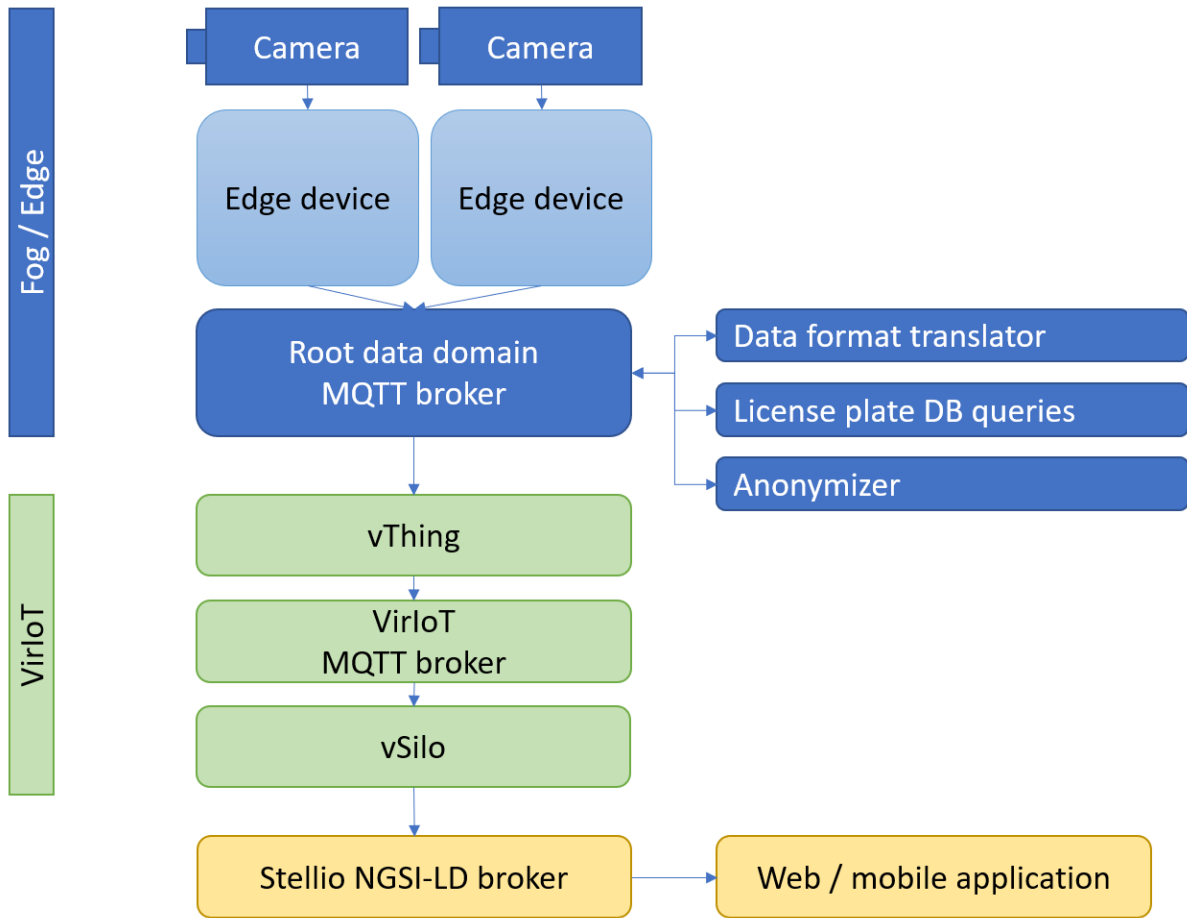


Figure 16: Software modules of the carpooling pilot, blue show the root data domain, green the VirIoT data domain and yellow the tenant one.

and then immediately pseudonymize the license plate number using a hash algorithm. The license plate is no more used nor stored in the following processes, ensuring users privacy.

5.2.2 VirIoT Data Domain

In order to have an operational carpooling pilot, the following VirIoT components must be instantiated:

- The car detector thingVisor, with one vThing per camera configured. It receives the entrance and exit events and some car characteristics from the root data domain services.
- One Virtual Silo (stellio-flavor), which receives events and car data and provide them to the application.

We first thought that the event data will be transmitted using a LoRaWAN network, we thus implemented a LoRaWAN thingVisor, connected to the MQTT broker of the

LoRa Chirpstack network server. This ThingVisor helps to connect incoming small pieces of data to their full NGSI-LD context. But it turns out that the LoRa network is not well suited to our needs, in particular it does not have enough bandwidth to send all the license plate detection events. We then switched to 4G connectivity, pushing data to the same MQTT broker as the one used by the LoRa server, using a compatible message format. This allows our LoRaWAN ThingVisor to be used for the integration of this smart camera setup into the Fed4IoT VirIoT platform.

The entry/exit event detection for a given car park is linked to a vThing in the virtual data domain. These vThings are managed by a single thingVisor, attached to a given root data domain. These vThings are notified of events by the root data domain services and provide the NGSI-LD context described by the use case data model (see 5.4) to the VirIoT MQTT broker.

The vSilo is registered to receive these events and the car data in the platform-neutral format. These data are directly sent to the Stellio broker which natively handles this format.

5.2.3 Tenant Data Domain

The tenant data domain is handled by an instance of the Stellio NGSI-LD broker, connected by a specialized vSilo. It first stores the entry and exit events in the same NGSI-LD format as the one shown before. It also runs specialized algorithms to associate entry and exit events to detect the passing of the cars in the parking and then compute the duration of each of them. These algorithms are implemented using various NGSI-LD queries through the NGSI-LD API implemented by Stellio.

The application dashboard is implemented using Grafana. To access the data, a specialized data source plugin was implemented. First, the NGSI-LD queries were directly called from within this plugin. But to build a dynamic dashboard, the number of queries and the size of the returned data made it very slow to refresh. We then decided to pre-process the dashboard data and store them in an intermediate MongoDB database. A dashboard data synchronization process is continuously running to update this database.

5.3 Deployment strategy

All the software components described for this use case are represented in the figure 16. The deployment of the carpooling pilot goes through the following steps:

- Hardware installation on site;
- Cloud services in the root data domain;
- Configure and start the ThingVisor to connect the root data domain to VirIoT;
- Start the Stellio-flavor vSilo;
- Start the Stellio instance and the carpooling application components : in/out association, dashboard database synchronization, Grafana and the carpooling datasource.

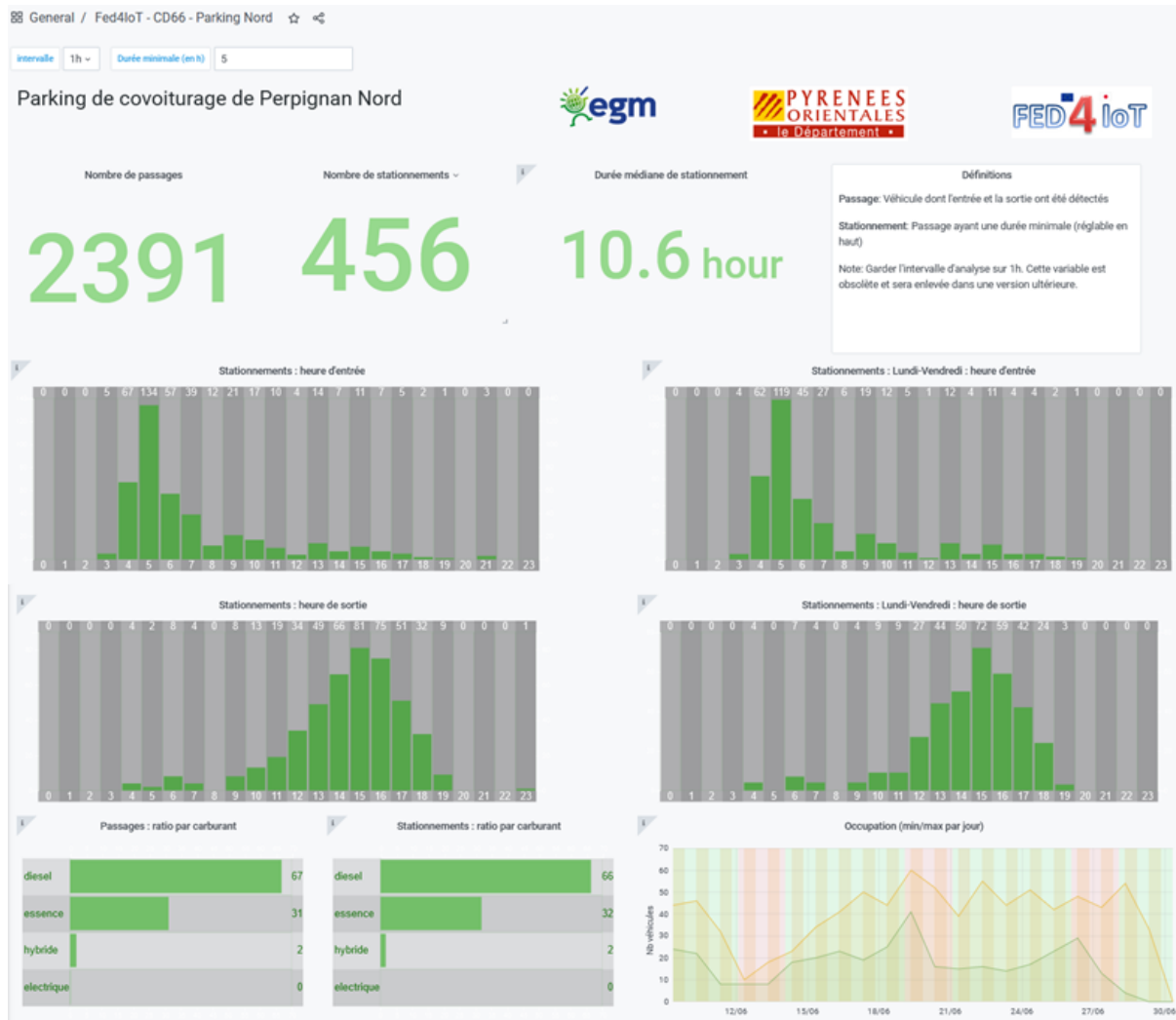


Figure 17: The carpool dashboard

5.3.1 Root data domain deployment

On the monitored sites the cameras and the edge processing capabilities are installed on the top of a mat. They are powered by a power management system delivering 5V for the Jetson Nano and 12V for the camera. The camera is configured with a fixed IP address and plugged to the ethernet port of the Jetson Nano. This latter device is connected to a 4G network with a USB stick. The internal MQTT bridge is connected to the root data domain MQTT broker. The various services on the Jetson Nano are managed using PM2 which is configured to launch them at startup. It also has a re-launch strategy to handle the potential software crashes.

On the root data domain, the cloud services are all hosted on a VM and managed with PM2. A first service listens to the MQTT messages coming from the edge. Once

the data are processed by the various services they are published on a specific topic of the same MQTT broker so they are available through subscription for the VirIoT thingVisors. These root data domain services provide:

- query the license plate database API and add the related data to the output message
- an anonymizer that computes a hash for the license plate

So, two types of messages are sent by the root data domain services:

- in/out events, containing the anonymized license plate and timestamp
- car data from license plate DB, containing the anonymized license plate and the relevant data (fuel type, CO2 emissions, ...)

5.3.2 VirIoT data domain deployment

There is one vThing instance for each smart camera installed on the field, deployed on the VirIoT cross-border platform, EU site. It is connected to the Root Data Domain MQTT broker, listening to messages transformed by the micro services described in the previous section.

To start the thingVisor, it is needed to provide:

- the MQTT broker address and port;
- the authentication informations on the MQTT broker;
- the base of the MQTT topics
- the list of cameras to be connected.

The thingVisor will create a vThing for each camera, each of these will be subscribed to the topic build from the base and the identifier of the camera.

Eventually, a vSilo embedding a Stellio NGSI-LD broker is instantiated in the VirIoT cross-border platform, EU site.

5.3.3 Tenant data domain deployment

The tenant data domain processes are all deployed on a specific VM. They are managed using PM2.

A first process queries at a regular pace the Stellio broker to try to associate new out events with a in event which occurred before. These queries involve multiple criteria and are done using the NGSI-LD specifications. This process thus only needs to connect to the Stellio broker of the vSilo. This information is given in the configuration file.

A second process synchronizes regularly the list of associated in/out events in a local database (for performance). The configuration of this process takes thus the connection information for the Stellio broker and also for the local data base.

A third one is a data source for Grafana, it is only connected to the local database. It is reachable by Grafana by the HTTP protocol and thus embeds a web server.

The GUI application is based on Grafana, it provides dashboards for the monitored sites (see 5.2.3). It is only connected to the data source described in the previous paragraph.

5.4 Data Model

Despite we have already provided a first version of the data model in D5.1, we have updated it as we are developing this pilot. So, we have taken the opportunity to bring all the new updates to this document. They can be found in the Annex at the end of this document.

6 Cross-border Person Finder Pilot

6.1 Description of the pilot

The Cross-border Person Finder (CBPF hereinafter) pilot is one possible application to demonstrate inter-operable capability of Fed4IoT's VirIoT platform. The pilot aims at virtually sharing the core functionality of CBPF application among surveillance cameras installed in multiple cities, and then at performing complex image processing at the edge, by complying to the requirements of data protection regulations, such as GDPR, avoiding to move personal data to the centralized cloud if not necessary.

Currently, according to the white paper on tourism in Japan [1], the number of international tourists is rapidly increasing, and tourism industry is grown remarkably. According to the white paper, the number of international visitors to Japan by air or sea is approximately 28.69 million in 2018, and this number is the second rank in Asia. As well as increasing of inbound tourists, outbound tourists (i.e., Japanese overseas travelers) are also increasing, and the numbers of them approximately 19 million in 2018. Those who inbound and outbound tourists tend to be elderly people or youth and not always able to speak local languages fluently. Therefore, their families are often worried about their safety, and demand for smart applications that can notify the tourist safety and monitor the tourist traces.

Based on the above background, the CBPF pilot provides an application able to notify the geographic location entered by a person, when authorized users, such as police officers, city hall staff and families, issue a request to try to locate the person. A conceptual representation of the CBPF pilot is shown in Figure 18. The CBPF application attempts to find the requested person from video feeds coming from surveillance cameras installed in multiple cities (e.g., EU and JP smart cities). In the pilot, in order to comply to the requirement of GDPR, the images captured by the surveillance cameras do not travel among cross-border countries (EU and JP). Thus, image processing is performed in edge devices (possibly in-camera) at EU or JP sites, and the geographic location information is extracted from the processed data.

In order to promote CBPF services, there are two security aspects we should consider. The first aspect is user consent when the CBPF application initiates service for a specific person, in order to ensure that the person agrees to provide information which is later necessary for the CBPF to operate (e.g., portrait of the person to be found). The second aspect is verification of essential workers providing CBPF services. It can be achieved with identity verification and attribute verification in case anonymity of workers is required. It should be noted that the verification of essential workers can be used for other applications (e.g., COVID-19) to verify essential workers such as doctors and essential service employees (e.g., grocery store, and pharmacy). Based on the user consent and the identity/attribute verification, CBPF application authorizes user to provide or receive information necessary for carrying-out the CBPF service.

In the pilot, we give a typical use case of CBPF application as a tourist monitoring tool for personal safety. However, the core functionalities of CBPF are to detect (and track)

requested persons and ensure security. Thus, we expect that the CBPF application, including individual components of the CBPF, can be adopted to various application demands, such as tracking traces of criminal persons and tracking virus infected persons (e.g., COVID-19).

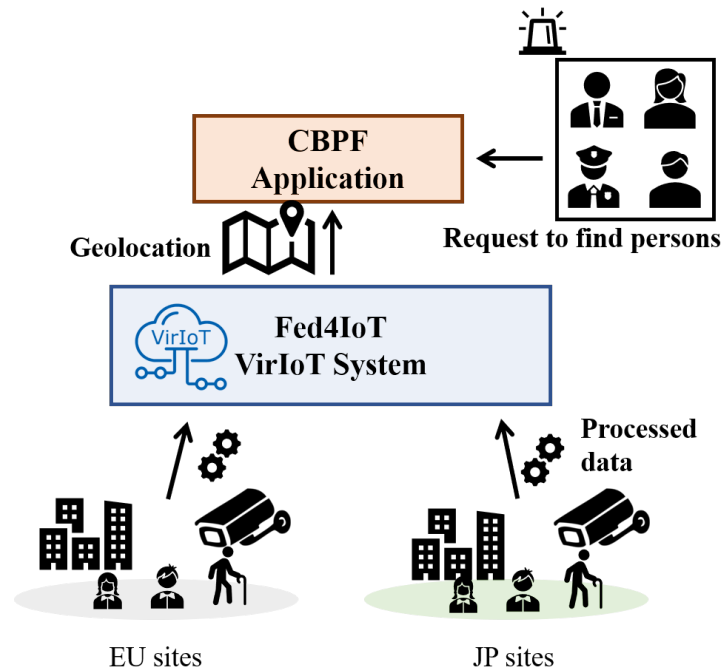


Figure 18: Concept image of Cross Border Person Finder pilot

6.2 Pilot Assumption

The CBPF pilots uses face matching technology to find a person. Recent face matching technologies use Artificial Intelligence technology (i.e., deep learning) and therefore EU Artificial Intelligence Act announced on 21st of April, 2021 should be considered. As written in "LAYING DOWN HARMONISED RULES ON ARTIFICIAL INTELLIGENCE (ARTIFICIAL INTELLIGENCE ACT) AND AMENDING CERTAIN UNION LEGISLATIVE ACTS" (19), the use of biometric matching by law enforcement is prohibited, but exceptions are described that the use is strictly necessary to achieve a substantial public interest, the importance of which outweighs the risks. This pilot aims to find a missing person which is the search for potential victims of crime, and also may be certain threats to the life or physical safety of natural persons.

From GDPR point of view, the following aspects are taken into account for the operation of the CPBF application.

- The owner of camera system authorizes the access from ThingVisor according to the service request from authorized user

- The authorized user is able to retrieve a face image if face matching result shows a certain level of score (likely the same person)
- The authorization of the user is based on attribute authentication (e.g. police officer) and therefore anonymous
- The user attributes and opt-in information (the owner of camera system authorized an access form the user) are managed by DLT (Distributed Ledger Technology) and each smart city has their own DLT node
- The user attributes and opt-in information are randomized and handled as a short-life token
- The data elements of short-life token are stored in DLT node as a record and record ID is used instead of user identifier to avoid a tracing
- The record is not always shared by all DLT nodes and the user can share the record with necessary smart city DLT nodes only
- The authentication response for user attributes and opt-in information is Boolean (ZKP:Zero Knowledge Proof approach)
- The record can be deleted from DLT node upon request

Thus, the CPBF application protects the privacy of both application user and subjects.

6.3 Description of the components to be instantiated

In this Section we describe the components, with a focus on a solution for the attribute-based authentication system.

6.3.1 ThingVisor Variations

First, we describe a design strategy for Cross-Border Person Finder (CBPF) ThingVisors. CBPF ThingVisors should be carefully designed by considering the GDPR requirements; Images or videos cannot be exchanged among EU clouds and JP clouds without user consent. Thus, in the pilot, we aim at having a privacy-preserving ThingVisor for detecting a specific person's face and some attributes corresponding to the person. This specific ThingVisor produces such context information as a Virtual Thing by converting the information to the NGSI-LD neutral-format.

To realize this, the CBPF ThingVisors require mainly four capabilities, namely face detection, feature extraction, feature matching and interaction with an authentication system. The design strategy of CBPF ThingVisor is concerned about which capabilities are deployed where. For example, are all services offered by a camera device (or a camera system) itself or only the privacy-preserving service (e.g., face detection) is done by the

camera and others are done by the edge and/or cloud servers?. In the following, we discuss four possible designs of the CBPF ThingVisors.

6.3.1.1 Option 1: Smart surveillance camera

6.3.1.1.1 Root Data Domain (Assumption)

This option assumes the surveillance camera involves face detection function and face recognition function and such functions can be activated via Camera Virtualization APIs as shown in Figure 19.

When the Camera Virtualization APIs are called by Relay-TV, the detection function and face recognition function are activated. The request message including reference data (i.e., a portrait of the missing person, or a target face) is sent to the smart surveillance camera and the smart surveillance camera verifies the access right of the user with Auth system. When the smart surveillance camera detects people in its sight, then it identifies the faces of the persons and compares them with the reference data. If the two images match, then the smart surveillance camera generates the data which contains temporary information, location information, and the URL of matched capture file.

6.3.1.1.2 ThingVisor

The user of CPBF application sends start/stop commands to Relay TV for activation/deactivation of the smart surveillance camera via vSilo. The payload of command includes a portrait of a missing person. The command also includes a short-time token to authenticate the user attribute and opt-in information of the user.

The Relay-TV activates the smart surveillance camera via Camera Virtualization APIs and also initiate the search of the missing person by sending a portrait of the missing person to the surveillance camera. The Relay-TV retrieves the data which contains temporary information, location information and the URL of matched capture file from the root data domain and sends it to the vSilo as a vThing via MQTT broker.

The user can identify the possible time and location of a missing person and also can retrieve a capture file to check whether the person shown in the capture file is the missing person or not.

6.3.1.2 Option 2: Monolithic ThingVisor deployed at the edge, with Person-Finding Capability

6.3.1.2.1 Root Data Domain (Assumption)

If the root data domain can only provide raw camera output images, we have another option in which all the manipulation of the images to find out the person in search is performed in a single ThingVisor, or a chain of ThingVisors, deployed at the edge. In this section, we discuss the former option where image manipulation is done by a single monolithic ThingVisor. The latter option is discussed in section 6.3.1.3.

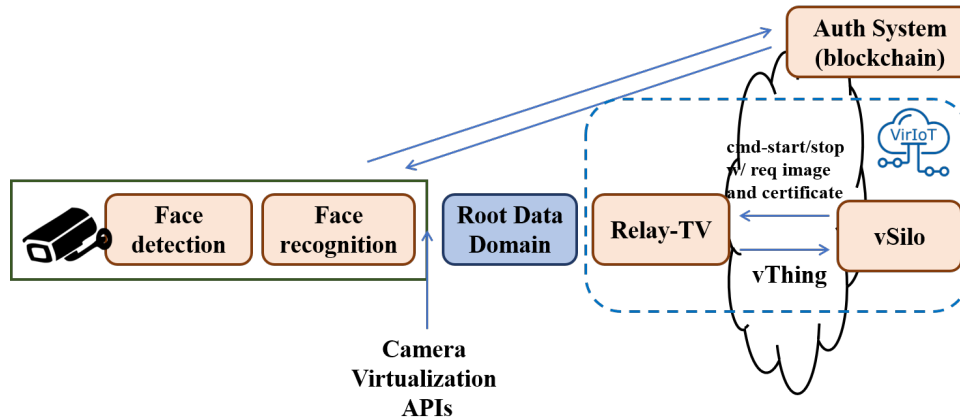


Figure 19: Simple Relay CBPF ThingVisor

The root data domain, in this case, contains the edge part of VirIoT. VirIoT is extended to the root data domain and the monolithic ThingVisor is housed within the edge part of VirIoT in the root data domain as shown in Figure 20.

The root data domain provides pictures taken by surveillance cameras through a REST interface and the ThingVisor accesses the pictures through the interface. The REST interface is assumed to provide the video stream taken by the surveillance camera. We assume the format of the video stream may vary.

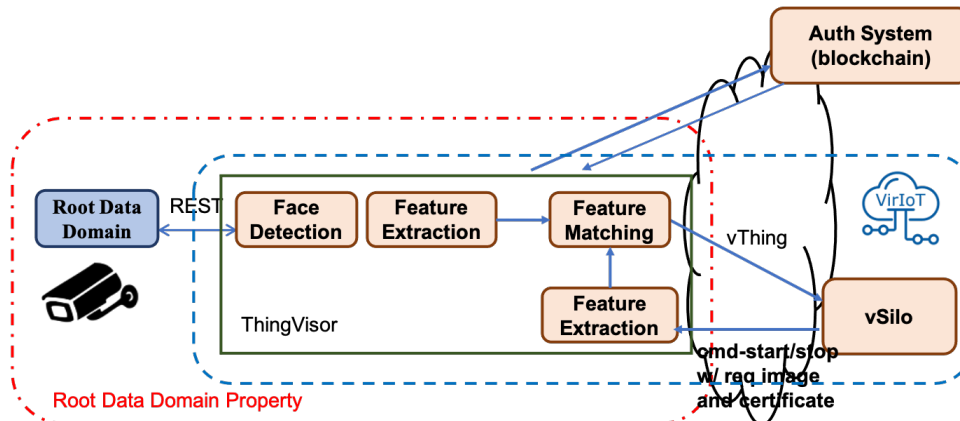


Figure 20: Monolithic CBPF ThingVisor

6.3.1.2.2 ThingVisor

The ThingVisor receives a face picture of the person to be found (target face) from a vSilo in an actuation command and tries to find the face in the pictures from the surveillance camera (camera pictures). The target face in the command is accompanied with an authorization code to authorize the use of the ThingVisor. The ThingVisor examines the authorization code through communications with the authentication system described in

Section 6.3.2. The interface between the root data domain where the surveillance camera is attached and the ThingVisor is REST. The ThingVisor uses HTTP to derive a series of pictures from the root data domain.

The ThingVisor has three types of function blocks: face detection, feature extraction, and feature matching. In this implementation, Face API of Microsoft Azure cloud is used for the feature extraction and feature matching.

The camera pictures received by the ThingVisor is first processed to find faces in the pictures by the face detection function. Then, features of the found faces are extracted by the feature extraction function. As for the target face, also the feature extraction function extracts features from the target face in the ThingVisor. The features from the camera pictures and ones from the target face are matched. If a match is found, the ThingVisor informs the match to the subscribing vSilo through VirIoT. VirIoT makes sure that the message is only delivered to the proper vSilo.

When the face picture of the person to be found is provided to the vSilo, the vSilo must receive a consent from the party requesting to find the person to transfer the face picture to ThingVisors which exist at the root data domains of surveillance cameras and also Microsoft Azure cloud Face API service in the region of the root data domain.

6.3.1.3 Option 3: Service chain ThingVisor deployed at the edge, with Person-Finding Capability

6.3.1.3.1 Root Data Domain (Assumption)

Similar to the previous one, in this option, the root data domain provides only raw images/video streams captured from surveillance cameras. VirIoT components, more precisely, ThingVisor will perform application-specific processing, like face detection, feature matching, etc.

Again, just like the previous scenario, in order to avoid exchanging privacy data, like face image, among EU and JP, we assume that VirIoT deploys a part of ThingVisor components physical proximity to the root data domain (e.g., an edge server installed at the pilot site).

The root data domain exposes the surveillance images/video streams using RESTful API, and ThingVisor can retrieve the data using the API.

6.3.1.3.2 ThingVisor

In this option, ThingVisor comprises several micro services, and the micro services are connected via the network, like service function chaining. Thanks to the micro service architecture, compared to the monolithic ThingVisor, the service chaining ThingVisor can improve operability. For instance, a developer of ThingVisor can select or update communication protocols and processing algorithms by design. In addition, the service chaining ThingVisor can also support flexible service deployment. For instance, privacy-concerned services, like face detection and feature extraction, are deployed to the edge server and other services are deployed to the cloud server.

In order to distribute the processing result as a vThing, we assume that data encapsulation and/or data format translation is carried on by a separate and dedicated ThingVisor, named "TV converter" as shown in Figure 21. TV converter formats the processing result as a vThing and streams to the vSilo.

Furthermore, TV converter has a capability to handle an actuation message from the vSilo. Once, TV converter receives an actuation message (e.g., start/stop command), it tries to verify the access right of the user with Auth System and activate the face recognition service.

We assume that service chaining ThingVisor is designed by using ThingVisor Factory.

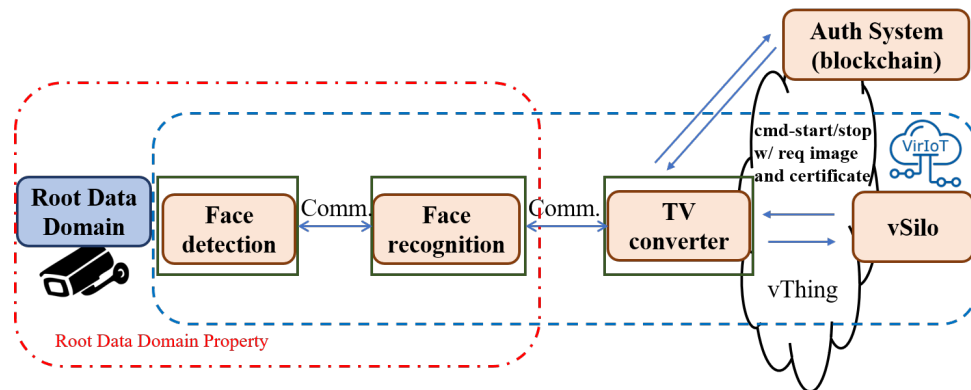


Figure 21: Service chaining CBPF ThingVisor

6.3.1.4 Option 4, Camera Sensor sharing: Chain of distinct ThingVisors with Face Recognition Capability

6.3.1.4.1 Root Data Domain (Assumption)

Similar to the previous two cases, in this option, the root data domain provides only raw images/video streams captured from surveillance cameras. ThingVisors will perform application-specific processing of face recognition. The difference is that an upstream ThingVisor is deployed in between the camera system providing raw images and the face recognition ThingVisor (see Figure 28). The goal of this intermediate CameraSensor ThingVisor is to demonstrate:

- **Sharing of a sensor among multiple applications:** only one copy of the stream of pictures travels from the real video camera to the virtual sensor within VirIoT. Multiple application can then import that "sensor" vThing, and the same video frames are efficiently delivered to them by the VirIoT Data Distribution System.
- **Chaining of different ThingVisors:** the application is not interested in the raw video frames, but in events such as "Andrea's face is detected in a picture". Thus, the raw video frames coming from the "sensor" vThing are processed by a downstream "detector" vThing. The application just imports the "detector" vThing.

The different ThingVisors can be deployed either both at the edge, to ensure privacy protection, or both at the cloud if privacy is less of a concern, and processing power is required for scaling-up of the face recognition jobs being shared by a large number of different tenants/applications.

6.3.1.4.2 ThingVisors

This scenario's primary focus is not the CBPF capabilities, but the sensor sharing and its architectural implications. This is why a separate and more detailed description is carried out in Section 9, to where the reader is redirected.

6.3.2 Attribute-based Authentication

To ensure the security of CBPF pilot, an authorized access to Virtual Things shall be protected to prevent unexpected use of information. This can be done by service authorization to ThingVisor.

There are three points to verify the access right to ThingVisor.

- ThingVisor itself has a function to verify the request message has an access permission to ThingVisor.
- vSilo has a function to verify that the request from CBPF application has an access permission to ThingVisor.
- CBPF application has a function to verify that the (end) user of CBPF application has an access right to ThingVisor

Figure 22 illustrates how an unauthorized access to ThingVisor is protected.

A token is used to identify user and/or user attribute. The user selects his/her identity and/or attribute to be authenticated and issue the token signed by holder device. The user sends a service request message to CBPF application attaching the signed token. CBPF application sends a request to authenticate his/her identity and/or attribute to Attribute-based authentication system and receives a response and verifies the user requesting the service is a right person.

Then the CBPF application sends a service authorization request to Attribute-based authentication system and then CBPF application receives an authorization code from Attribute-based authentication system.

Finally CBPF application will send a request to vSilo attaching the authorization code. vSilo can request a certificate (and public key, if vSilo does not have effective public key) corresponding to an authorization code. vSilo can verify the authorization code and it will allow the service if the verification of authorization is successful.

This verification function can be also implemented to ThingVisor.

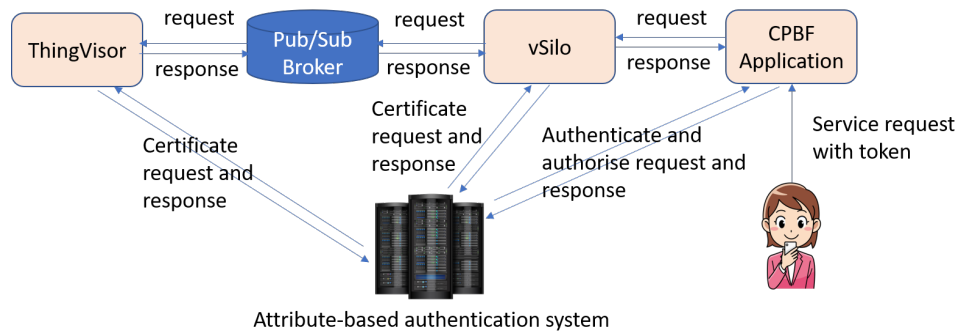


Figure 22: Protection of unauthorized access to ThingVisor

6.3.3 Tenant Data Domain

In order to retrieve vThings from the VirIoT, a tenant must instantiate a specific vSilo. As differed from the other pilots, the CBPF pilot requires an actuation capability in ThingVisor. In VirIoT, vSilo has an interface of actuation ThingVisor; vSilo receives the users' request messages from the application and transfers the request messages to the ThingVisor. A detailed mechanism of actuation ThingVisor is reported on Deliverable 3.2. In this pilot, vSilo handles start and stop commands for CBPF-ThingVisor. Once vSilo receives the start command from the application, vSilo transfers the start command message to the CBPF-ThingVisor, and the ThingVisor will activate the functions regarding a face recognition, such as feature extraction and feature matching functions. Similarly, once vSilo receives the stop command, vSilo transfers the message to the ThingVisor, and ThingVisor will deactivate its own task.

In addition to the actuation capability, vSilo also plays a role of a data broker just like the other pilots. In the CBPF pilot, vSilo stores vThing data which contains the geographical information where the request person is found. The example of data model has already described in Deliverable 5.1.

In the application side, we will retrieve vThing from vSilo using HTTP GET method and visualize the result on the web browser.

6.4 Deployment strategy

Since the CBPF pilot's objective is to demonstrate interoperability of Fed4IoT VirIoT system, the pilot will be deployed at multiple domains. Candidate sites are Murcia and Grasse as EU sites and Kumamoto, Hakusan and Tokyo as JP sites. A plan for deploying the pilot is shown in Figure 23. As described in Section 8.2, the pilot needs to comply the requirement of GDPR. In other words, the surveillance camera images are prohibited to be exchanged among EU and JP. Therefore as shown in the figure, we can select four deployment patterns: a root data domain can call the camera virtualization APIs or not, and CBPF-ThingVisor are deployed on the cloud server or the edge server.

First, we assume that the root data domain can call the camera virtualization APIs.

In this case, the privacy-concerned processing, face detection and face recognition, are done by the camera system itself, so the root data domain handles the privacy-preserving data. Therefore, we deploy an option 1 CBPF-ThingVisor (see Section 6.3.1.1) in the edge or cloud server. In the demonstration, we will plan to adopt this option in both EU and JP sites where the edge service is not available. We will implement the smart surveillance camera system in the Jetson Nano, and Relay CBPF-ThingVisor is deployed in the cloud.

Second, we assume that the root data domain has no capability to call the camera virtualization APIs. In this case, the root data domain provides the raw images/video streams, and ThingVisor performs face detection and face recognition operations. Thus, we plan to deploy an option 2 monolithic CBPF ThingVisor at the edge (see Section 6.3.1.2) or an option 3 service chain CBPF plus a TV converter at the edge (see Section 6.3.1.3), on the only edge server at Tokyo site (Waseda University), thus ensuring privacy-preserving operation. We will implement the CBPF-ThingVisor in the barebone PC which joins the VirIoT system as a kubernetes edge worker node.

Unlike ThingVisor, vSilo and CBPF application will be deployed in the cloud. This is because the CBPF application will be accessed by various users, such as police officers, city hall staffs and families, and vSilo needs to broker Virtual Things produced at multiple sites. As shown in Figure 23, as well as vSilo and CBPF application, attribute-based authentication system is also deployed in the cloud in order to provide secure operations among ThingVisor and CBPF application. A deployment possibility that is Not Good for privacy concerns is also shown (red dotted box), but not implemented in the CBPF pilot, where raw images would travel from the Local environment to the Cloud environment.

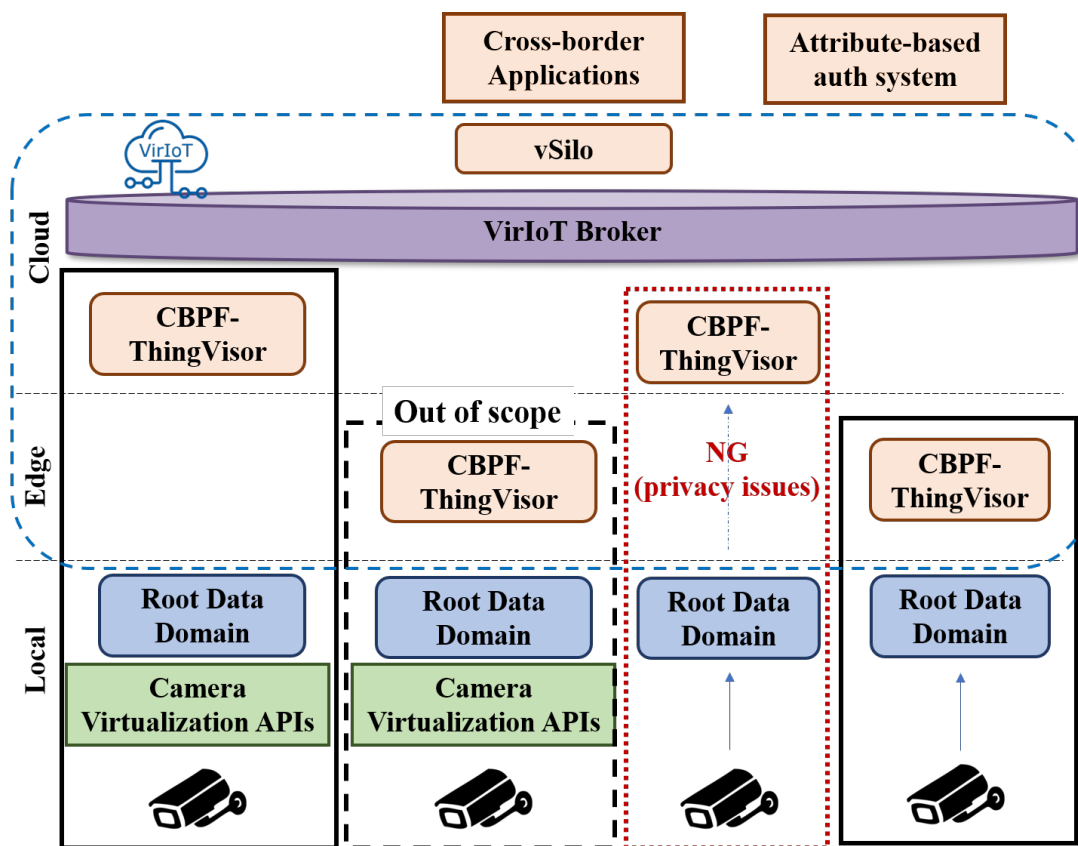


Figure 23: Deployment plan of Cross Border Person Finder pilot

6.5 Data model

Data model used for Cross-border Person Finder is not updated from deliverable D5.1.

7 Wildlife Monitoring Pilot

7.1 Description of the pilot

In rural areas, damage to agricultural products by wildlife is serious. It is effective to use IoT technology to collect information, such as captured and approaching animals, for countermeasures against wildlife damage, but the cost required for sensor device installation and application development is a problem. In local cities, budgets and manpower are not enough, those factors prevent the introduction of IoT technology. To solve this problem, sensor device installers and application software developers can be connected to the Fed4IoT virtual IoT environment (the VirIoT platform), which may enable reuse/repurposing of sensors and reduces the time and cost required for IoT system development. For verification purposes, a wildlife monitoring pilot system will be deployed in Hakusan City, Ishikawa Prefecture, Japan, where wildlife damages are serious. In addition, a sensor device will be installed for purposes other than animal damage control. By making these exist as Virtual Thing in the VirIoT platform, it will be possible to develop applications other than animal damage control applications, such as environment monitoring.

7.2 Description of the components to be instantiated

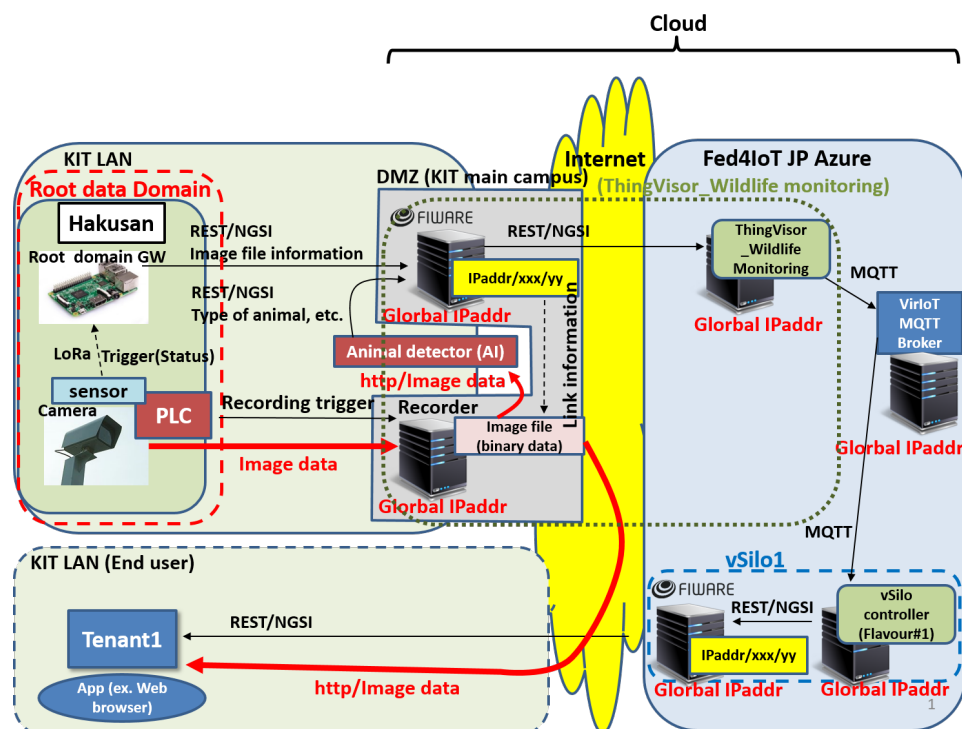


Figure 24: Pilot system at Kanazawa Institute of Technology

Figure 24 shows the configuration of the test environment for the pilot system at

Kanazawa Institute of Technology. Cameras and sensors are installed in the field to collect information necessary for animal damage control, such as animal types and images, and environmental information such as temperature and humidity. The application displays the detected animal position, environmental information, and the like. FIWARE is used as a data utilization platform.

Figure 25 shows the GUI of the wildlife monitoring application. The service provider provides the Web based application of the wildlife monitoring. With the application software, geographical position of the sensor devices are indicated in GUI. Information such as captured animal type, number of animals and URL of animal image files are indicated as detailed information of each sensor. Environmental information of devices, e.g. temperature, humidity, rainfall and illuminance around the devices are also indicated.

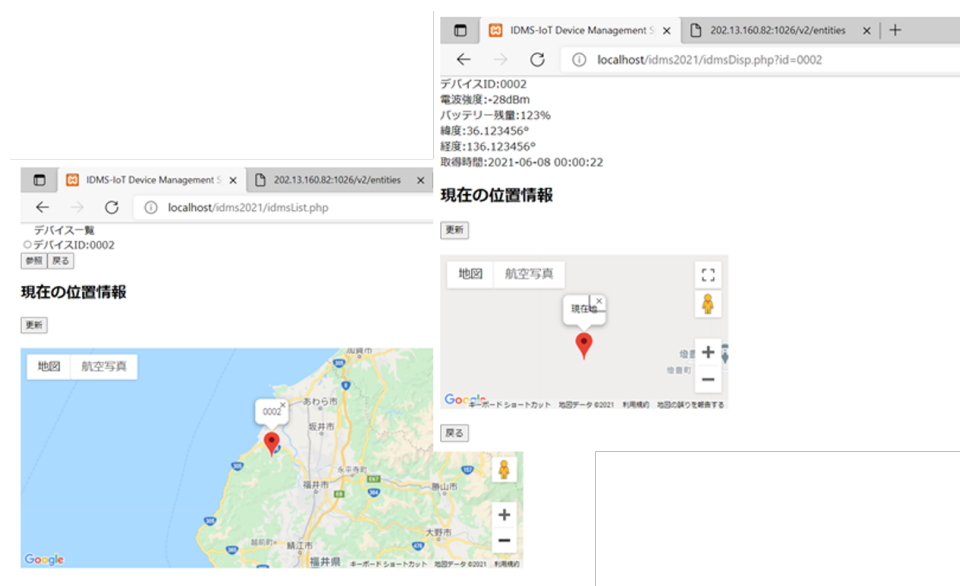


Figure 25: GUI of the wildlife monitoring application

As shown in Figure 26, collected data from actual devices exists as Virtual Things in the VirIoT system, and it can be used by various application software developers as needed. The VirIoT system creates a vSilo to collect the data needed by application software developers from various Virtual Things. Since the data format is interchangeable (e.g. oneM2M, NGSI, etc.) in VirIoT, assets can be used all over the world, depending on the end-applications data formats.

7.3 Deployment strategy

The pilot system will be deployed around Kanazawa Institute of Technology Hakusan campus at Hakusan City, Ishikawa Prefecture, Japan. Physical devices such as cameras, sensors and the root domain gateway will be installed outdoors, in the field. The recorder of the cameras, some servers for related software running and animal detector (Jetson Nano) will be installed in a KIT office. The recorder and the servers are accessible

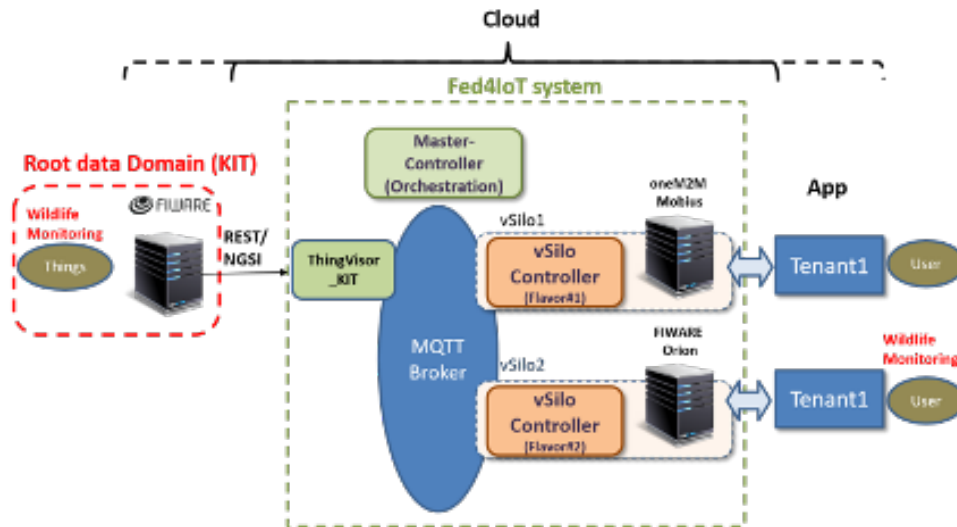


Figure 26: Data exchanges via Fed4IoT system

from the Internet. Necessary ThingVisor and vSilos will be deployed in the JP site of the VirIoT cross-border platform, exploiting the local MQTT Broker of the cluster for achieving low latency.

7.4 Data model

Table 4 lists the Virtual Things in VirIoT, based on the Wildlife Monitoring use case and its Real Things, as we are going to virtualize them to be flexibly reused in different vSilos. The Table also reports the structure of the attributes that will be internally exposed as NGSI-LD properties of the corresponding Entities, conveyed from the ThingVisors to the vSilos.

Figure 27 shows an example of the NGSI-LD Entity associated with the Thermometer Virtual Thing. Other Virtual Things have a similar structure.

Virtual Things	Attributes	Description
Thermometer	Location	Location of the device
	Temperature	Sensed data
Hygrometer	Location	Location of the device
	Humidity	Sensed data
Rain gauge	Location	Location of the device
	Rainfall	Sensed data
Illuminometer	Location	Location of the device
	Illuminance	Sensed data
Animal detector	Location	Location of the device
	Animal is present	Sensed data
	Animal type	Results of judgement from photo by Jetson-nano
Camera	Location	Location of the device
	Photo data of animal	URL of the image file
	Camera type	Type of camera
	Resolution	Provisioned value

Table 4: Virtual Things for Wildlife Monitoring

```

1 {
2   "id": "urn:ngsi-ld:KIT:Thermometer01",
3   "type": "Thermometer",
4   "location": {
5     "type": "GeoProperty",
6     "value": {
7       "type": "Point",
8       "coordinates": [36.5313, 136.6285]
9     }
10  },
11  "temperature": {
12    "type": "Property",
13    "observedAt": "2020-05-12 16:02:56.343000",
14    "value": "1",
15    "unitCode": "CEL"
16  }
17 }

```

Figure 27: An example of NGSI-LD entity published by the thermometer Virtual Thing

8 Modular Code for ThingVisors and vSilos

VirIoT is implemented as a microservices architecture. Each component of the platform runs in a separate container, and as long as the component correctly implements the VirIoT interfaces, developers are free to implement it using their preferred programming languages, frameworks and technologies. For instance, two ThingVisors can be developed in two different languages: currently, our NGSiv2/FIWARE ThingVisors are written in JavaScript/NodeJS, while the others are written in Python. The advantages of this approach are evident, and it has allowed each research team collaborating in Fed4IoT to exploit its own software-writing expertise to the fullest extent, resulting in quick prototypes.

On the other hand, it is easy to recognize that ThingVisors, and vSilos similarly, share a bulk of common functionality (e.g. for connecting to the MQTT Data Distribution, for processing the commands of the control plane, for implementing the vThing's data structures, etc...), which needs to be re-implemented in each different programming language. But within the boundaries of the same programming language, it is advisable that such common functionality is collected in a common (e.g. python) module that all ThingVisors can import. The main advantage is that developers wanting to expand the portfolio of ThingVisors, can focus on the functionality, specific to their ThingVisors, and integrate with VirIoT via the common module, out-of-the-box. This is crucial for ease of adoption of the platform from external developers, but plays an important role also for quickly bootstrapping the Fed4IoT internal teams themselves, when a new vSilo or ThingVisors is implemented.

For vSilos, we have created a set of bulk python functions that can be easily re-used in each vSilo.

For ThingVisors, which are the components that external developers are more likely to contribute, so as to implement an ever growing number of different vThings, we have created an importable python module that takes care of all the routine workflows. This way, external developers can focus on implementing the distinguishing functionality of the ThingVisor, only.

8.1 The `thingVisor_generic_module.py` python module

The `thingVisor_generic_module.py` python module is the Fed4IoT generic modular code to import in custom ThingVisors, to facilitate developing them. It offers the API described in the following sections. We assume that the user's python code imports the module as follows:

```
import thingVisor_generic_module as thingvisor
```

8.1.1 initialize_vthing

The most important interface is normally used at the beginning of the user code, in order to create one (or more) vThing in the ThingVisor. In order to create the vThing, the following parameters are needed: **name**, **type**, **description**, **array of commands**.

- The **name** is used, together with the name assigned to the ThingVisor when it is added to VirIoT via CLI, in order to form the vThingID that uniquely identifies the vThing inside the platform.
- The **type** is the NGSI-LD type assigned to NGSI-LD Entities produced by the vThing as neutral-format entities representing the stream of context information generated by this sensor/virtual thing.
- The **description** is a descriptive string that appears when all vThings are listed via VirIoT's CLI.
- The **array of commands** is an array of strings representing all commands supported by the vThing in case it is an actuator.

The following snippet of python code is an example creating a vThing named "detector", that produces entities of type "FaceRecognitionEvent" and supports two actuation commands: "startjob" and "deletejob".

```
thingvisor.initialize_vthing(  
    "detector",  
    "FaceRecognitionEvent",  
    "faceRecognition virtual thing",  
    ["startjob","deletejob"]  
)
```

If the ThingVisor has more than one vThing, subsequent `initialize_vthing` calls can be executed.

8.1.2 params

The `thingvisor_generic_module.py` module defines and initializes a `thingvisor.params` python dictionary automatically. All parameters that are entered (via the `-p` option) in the CLI command that creates the ThingVisor, are automatically available, using the parameter name as a key in the dictionary.

For instance, the following code snippet would check whether a specific parameter was given at CLI creation time, and if not, assigns a default value.

```
if 'fps' in thingvisor.params:  
    print("parsed fps parameter: " + str(thingvisor.params['fps']))  
else:  
    thingvisor.params['fps'] = 2  
    print("defaulting fps parameter: " + str(thingvisor.params['fps']))
```

Moreover, the `thingVisor_generic_module.py` module takes care of intercepting any `update-thingvisor` command issued via CLI, automatically updating in the `params` dictionary all parameters present in the update command, to the new values.

8.1.3 `publish_attributes_of_a_vthing`

This interface is the one normally invoked when the `vThing`, acting as a sensor, is producing a new value. The newly created context information is structured as neutral-format NGSI-LD Entities. This API creates a NGSI-LD entity with `id` and `type` extracted from the `vthingindex` and with properties (or relationships) coming from the given `attributes` list. Each item of the list is a python dictionary, as follows:

```
{
    attributename:STRING,
    attributevalue:WHATEVER,
    isrelationship:BOOL (optional, default False)
}
```

An example usage follows of a camera sensor `vThing` named "sensor" that produces random frame identifiers, e.g. entities with just one property, named "frameIdentifier".

```
attributes = [
    {"attributename":"frameIdentifier", "attributevalue":"xs34Gf"}
]
thingvisor.publish_attributes_of_a_vthing("sensor", attributes)
```

8.1.4 `publish_actuation_response_message`

This interface creates a message in response to feedback coming from an actuator, which is to be communicated back to the `vSilo` that originated the actuation command. In case the unicast `cmd-nuri` of the destination `vSilo` was not set in the received actuation command, then a fallback broadcast topic is used, so that data is sent to all subscribers of this `vThing`. The following parameters are needed: `cmd_name`, `cmd_info`, `id_LD`, `payload`, `type_of_message`.

- The `cmd_name` is the name of the original actuation command that triggered the response being published.
- The `cmd_info` is the original actuation command request itself, which contains the `cmd_value`, `cmd_nuri`, `cmd_qos`, `cmd_id` and `cmd_token`, as described in section 10.1.
- The `id_LD` is the NGSI-LD identifier of the neutral-format entities produced by the `vThing`.
- The `payload` is a generic object representing the result of the actuation, to be communicated back to the `vSilo` that originated the actuation command request.

- The `type_of_message` can be one of "status" (for actuation that continuously needs to report changing results, i.e. QoS=2) or "result" (for actuation that is reporting its final result, i.e. QoS=1).

An example usage follows. Please notice that `cmd_name`, `cmd_info`, and `id_LD` don't need to be constructed by hand, because the generic module passes them around automatically, whenever a callback associated to an actuation command is invoked.

```
thingvisor.publish_actuation_response_message(  
    cmd_name,  
    cmd_info,  
    id_LD,  
    "The temperature has now reached 30 degrees",  
    "status"  
)
```

8.1.5 upstream_entities and upstream_tv_http_service

The `thingvisor_generic_module.py` module is designed so as to easily allow ThingVisors to “chain” to a specifiable upstream vThing: upon chaining, it subscribes to Entities coming from the upstream vThing, making them easily available.

The name of the upstream vThing to chain to is a configurable parameter, named `upstream_vthingid`. By default, at startup, if no upstream vThing of a ThingVisor is specified via CLI, the module waits for an update command to specify it.

The name of the upstream vThing to chain to can be specified at creation time, when the ThingVisor is added to VirIoT, as in the following example, where a ThingVisor is added using the yaml that specifies a FaceRecognition, the name “facerecognition-tv” is given to it, the `upstream_vthingid` parameter is set to `camerasensor-tv/sensor` (as it represents a vThing identifier, which in VirIoT is composed of ThingVisorName/vThing-Name), and additionally, another custom parameter, named `fps`, is set to 12.

```
f4i.py add-thingvisor -y ../yaml/thingvisor-faceRecognition.yaml  
-n "facerecognition-tv" -d "recognizes faces"  
-p '{"fps":12, "upstream_vthingid":"camerasensor-tv/sensor"}' -z default
```

Alternatively, by using the `update-thingvisor VirIoT` command on a running instance of the TV, the `upstream_vthingid` parameter can be changed in real-time, as follows:

```
f4i.py update-thingvisor -n facerecognition-tv  
-p '{"upstream_vthingid":"camerasensor-tv/sensor"}'
```

Once the chain is successfully established, the `upstream_entities` array is used as follows, for instance to access the value of the upstream property named “frameIdentifier”:

```
thingvisor.upstream_entities[0]["frameIdentifier"]["value"]
```

It is possible to access the HTTP data streams available at the upstream vThing, as well, if it offers any, by using the automatically created `thingvisor.upstream_tv_http_service` variable, as follows:

```
# pick second element of the split
upstream_vthing = thingvisor.params['upstream_vthingid'].split('/',1)[1]
upstream_url = "/" + upstream_vthing + "/WHATEVER"
final_url = "http://" + thingvisor.upstream_tv_http_service + upstream_url
```

9 FaceRecognition ThingVisor

This ThingVisor allows to do face recognition using a camera system, and to virtualize the camera system as a single face recognition device. It is an updated version of the FaceRecognition ThingVisor described in deliverable D3.2, section 3.3.11. This updated version was developed to demonstrate the following functionality:

- **Sharing of a sensor among multiple applications:** only one copy of the stream of pictures travels from the real video camera to the virtual sensor within VirIoT. Multiple application can then import that "sensor" vThing, and the same video frames are efficiently delivered to them by the VirIoT Data Distribution System.
- **Chaining of ThingVisors:** the application is not interested in the raw video frames, but in events such as "Andrea's face is detected in a picture". Thus, the raw video frames coming from the "sensor" vThing are processed by a downstream "detector" vThing. The application just imports the "detector" vThing.

9.1 How it works

Figure 28 shows the face recognition architecture, comprising the Camera System, which includes a CSI-based (Camera Serial Interface) video Camera, the CameraSensor ThingVisor, which implements the "sensor" vThing, and the FaceRecognition ThingVisor, which implements the "detector" vThing.

Overall, the Camera System sends to the CameraSensor TV every new video frame it captures from the Camera. The CameraSensor TV buffers the video frames and gives them unique identifiers. Whenever the FaceRecognition TV is ready to process a new video frame, it gets it by name, asking it to the CameraSensor TV. The FaceRecognition TV processes the frame by comparing it to a target picture of a person. If a match is found, an event is sent from the FaceRecognition TV to a vSilo (that hosts the IoT and HTTP Brokers and talks to an external Application).

Users do not interact directly with the FaceRecognition ThingVisor. The whole process is driven via the User's vSilo, instead, as usual in VirIoT. Users POST target faces (to be matched) to the HTTP Broker running inside the vSilo. Moreover, they can act (using the usual VirIoT's actuation workflow) upon the face recognition process by starting (or deleting) a specific job recognition process, identifying it by a unique identifier of the job and the name of a target person (for example "123456/Andrea" in Figure 28).

The purpose of having the CameraSensor ThingVisor in between the Camera System and the FaceRecognition is that, in principle, several different downstream ThingVisors, performing diverse tasks such as face recognition, object recognition, motion detection, and the likes, can attach to the upstream CameraSensor TV. They will act as downstream processors, each fetching a copy of the same video frame for different purposes. This avoids each processor fetching a separate copy of the current video frame from the Camera System, thus avoiding redundant network traffic from the Root Data Domain (where the Camera System lives) into VirIoT. Moreover, VirIoT's HTTP Data Distribution System is

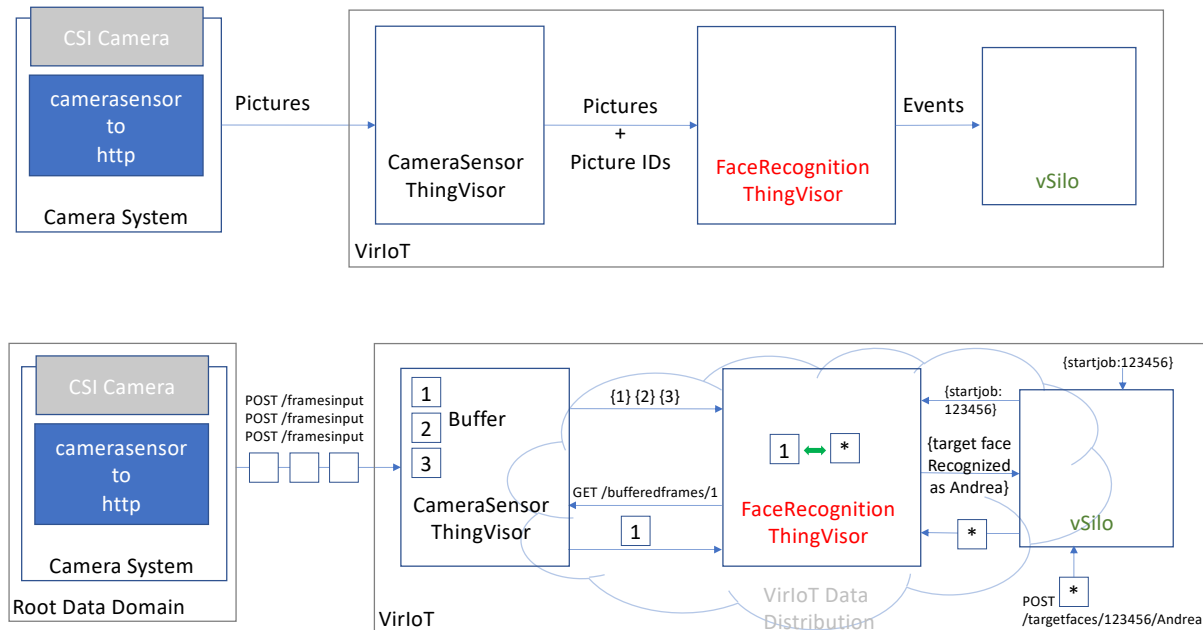


Figure 28: FaceRecognition Architecture

operating in between the upstream CameraSensor TV and the downstream ThingVisors, transparently caching all HTTP requests and responses, so that ThingVisors (and vSilos) that are requesting the same picture (by id) from the CameraSensor TV, will efficiently get it from the closest proxy.

More specifically:

9.1.1 The Camera System

- Connects to a CSI (Camera Serial Interface) Camera. The current implementation uses CV2 to capture video from a camera attached to a NVIDIA Jetson Nano board.
- Undistorts, compresses to jpeg, and scales down each video frame.
- Sends each new video frame to the CameraSensor TV via an HTTP POST request to the TV's `/framesinput` API.

The Camera System is currently implemented as a python script responsible for compression and HTTP communication, which is called `camerasensor-to-http.py`. It, in turn, imports a python module, which is responsible for video capture, and is called `camera-mod.py`. Both can be found in the `jetbot_scripts` folder of the CameraSensor TV.

9.1.2 The CameraSensor ThingVisor

The CameraSensor ThingVisor is organized in four main components.

9.1.2.1 REST interface to receive frames

It offers a REST interface to receive video frames, via HTTP POST. The interface is called `/framesinput` and accepts multipart POST requests composed of two parts:

- a part named `file` that is a jpeg file representing the current video frame
- a part named `json` that is a json file representing the timestamp when the video frame was captured, in the following form: `{"observedAt":STRING}`

This REST API is intended for access from the outside of VirIoT, i.e. from the Root Data Domain, where the Camera System lives. The REST API is available at internal ip port 5000, so if, for instance, `<CAMERASENSORTV_PUBLIC_IP>` is the public ip address to reach the CameraSensor TV and is the external port mapped onto internal port 5000, the following `echo` and `curl` command sequence is an example to POST a new video frame:

```
echo {"observedAt": "\"02-02-2021 14:34\""} > metadata.json

curl -F "file=@currentframe.jpg" -F "json=@metadata.json"
http://<CAMERASENSORTV_PUBLIC_IP>:<PORT_MAPPED_TO_5000>/framesinput
```

9.1.2.2 Video Buffer

It buffers a certain (configurable) amount of video frames, FIFO style, and it gives unique identifiers to them, upon arrival of each new frame. It buffers the jpeg compressed pictures in memory.

The size of the buffer is a configurable parameter of the ThingVisor, named `bufferSize`. The default size of the video buffer is 20. It can be specified at creation time, when the TV is added to VirIoT, as in the following example, where a TV is added using the yaml that specifies a CameraSensor TV, the name “camerasensor-tv” is given to it, and the `bufferSize` parameter is set to 30.

```
f4i.py add-thingvisor -y ../yaml/thingvisor-cameraSensor-http.yaml
-n camerasensor-tv -d "camera frames via http" -p '{"bufferSize":30}'
-z default
```

Alternatively, by using the `update-thingvisor` VirIoT command on a running instance of the TV, the `bufferSize` parameter can be changed in real-time, as follows:

```
f4i.py update-thingvisor -n camerasensor-tv -p '{"bufferSize":40}'
```


9.1.2.3 "sensor" vThing

It offers a single vThing, named "sensor". This vThing, upon arrival of each new frame, emits an event representing context information about the received video frame, in the form of a NGSI-LD Entity containing the picture's identifier.

The NGSI-LD Entity emitted at each frame arrival is represented in NGSI-LD "neutral format" (assuming the ThingVisor's name is "camerasensor-tv") by entity of type "NewFrameEvent". It has just one Property, named "frameIdentifier". The information about the timestamp of the video frame is dropped, as of now, since it is not needed for face recognition purposes. Here follows an example:

```
1 {  
2   "id" : "urn:ngsi-ld:camerasensor-tv:sensor",  
3   "type" : "NewFrameEvent",  
4   "frameIdentifier" : {  
5     "type" : "Property",  
6     "value" : "1623229264110-0"  
7   }  
8 }
```

9.1.2.4 REST interface to fetch frames

It offers a REST interface to fetch a specific frame by its identifier, via HTTP GET. The interface is called `/bufferedframes/<frameidentifier>` and accepts GET HTTP requests. It gives back data with mime-type "image/jpeg". The following `curl` command is an example to GET video frame by its id:

```
curl --output videoframe.jpg  
http://<CAMERASENSORTV_PUBLIC_IP>:<PORT_MAPPED_TO_5000>/bufferedframes  
/1623229264110-0
```

9.1.3 The FaceRecognition ThingVisor

It is designed so that it "chains" to an upstream "sensor" vThing implemented by a CameraSensor ThingVisor: upon chaining, it subscribes to Entities coming from the upstream CameraSensor's "sensor" vThing. Such Entities convey the identifiers of a stream of video frames buffered by the CameraSensor TV. FaceRecognition GETs the new frames at its convenience (thus operating at its own framerate, usually different than the framerate the video frames are produced by the Camera System).

The rate that FaceRecognition uses to get frames from CameraSensor TV is a configurable parameter of the TV, named `fps`. Its default value is 2.

The name of the upstream vThing to chain to is a configurable parameter as well, named `upstream_vthingid`. By default, at startup, if no upstream vThing of a CameraSensor TV is specified, the FaceRecognition TV sits idle, waiting for an update command to specify it.

Both parameters can be specified at creation time, when the TV is added to VirIoT, as in the following example, where a TV is added using the yaml that specifies a FaceRecognition, the name “facerecognition-tv” is given to it, the `upstream_vthingid` parameter is set to `camerasensor-tv/sensor` (as it represents a vThing identifier, which in VirIoT is composed of ThingVisorName/vThingName), and the `fps` parameter is set to 12.

```
f4i.py add-thingvisor -y ../yaml/thingvisor-faceRecognition.yaml
-n "facerecognition-tv" -d "recognizes faces"
-p '{"fps":12, "upstream_vthingid":"camerasensor-tv/sensor"}' -z default
```

Alternatively, by using the `update-thingvisor` VirIoT command on a running instance of the TV, both parameters (either one-by-one or together) can be changed in real-time, as follows:

```
f4i.py update-thingvisor -n facerecognition-tv
-p '{"upstream_vthingid":"camerasensor-tv/sensor"}'

f4i.py update-thingvisor -n facerecognition-tv -p '{"fps":6}'
```

9.1.3.1 “detector” vThing

It offers a single vThing, named “detector”. The “detector” vThing does not produce any information (in the form of NGSI-LD Entities) on its own. Thus, it is not a sensor, rather an actuator. Specifically, it represents an actuator that is activated by users via `startjob` commands, that need to have a QoS of 2, meaning the command does not terminate immediately with a given result, but the status of the command is going to be continuously updated by the actuator.

Users give an identifier to the job, and then they issue the `startjob` command to the actuator. Once the job is started via its command, the “detector” updates the command status whenever a matching face is detected. The updated `cmd-status` embeds links to the video frame that matched (`recognized-uri`), to the original picture of the face (`original-uri`), as well as the corresponding job’s name (`job`) and the name of the person depicted in the original picture (`name`).

In parallel and asynchronously to starting jobs, users have to POST target pictures of faces they want to be recognized (see the `/targetfaces` API below). Such target pictures are POSTed under a given identifier that has to match the identifier of a corresponding job, and a person’s name (e.g. “Andrea”) has to be specified, additionally, to further

tag the matches when they occur. Pictures can be POSTed without a corresponding job being started yet; a job can be started without targets. As soon as both a job and target pictures, with the same identifier, are present in the “detector”, it starts sending back the `cmd-status` updates to the `startjob` command status receiver queue of every vSilo that has the “detector” vThing.

Several jobs can be started (giving different identifiers to them), and the status queue will receive a stream of updates (each possibly overwriting the previous update, depending on the specific IoT Broker that receives the updates).

More specifically, the “detector” vThing implemented by the FaceRecognition TV offers two commands:

- `startjob` command;
- `deletejob` command.

An example JSON object to send to the `startjob` command, that starts a job, giving “123456” identifier to it, is:

```
1 {
2   "cmd-id": "xaxaxa",
3   "cmd-qos": 2,
4   "cmd-value": {"job": "123456"}
5 }
```

The `deletejob` command is used to remove all pictures for a given job name, and to stop the corresponding recognition process. An example follows, that stops the above job:

```
1 {
2   "cmd-id": "fgfgfg",
3   "cmd-value": {"job": "123456"}
4 }
```

The “detector” vThing, being an actuator only, is represented at the ThingVisor by an NGSI-LD Entity having just one property, i.e. the default `commands` property that all VirIoT actuators have, that lists all commands available at the actuator. Its id is “urn:ngsi-ld:facerecognition-tv:detector” (assuming the ThingVisor is named “facerecognition-tv”), and its type is “FaceRecognitionEvent”, as follows:

```
1 {
2   "id": "urn:ngsi-ld:facerecognition-tv:detector",
3   "type": "FaceRecognitionEvent",
4   "commands": {
5     "value": [ "startjob", "deletejob" ],
6     "type": "Property"
7   },
8 }
```

Additionally, the "detector" vThing:

- Offers a REST interface to insert target pictures under a job identifier, additionally tagging them with a person name, via HTTP POST, and to retrieve information about them, via HTTP GET. The interface is called:

`/targetfaces/<jobidentifier>/<personname>.`

This REST API is available at internal ip port 5000, but it is NOT intended for access from the outside of VirIoT, i.e. Users do not directly POST target pictures to this FaceRecognition TV's `/targetfaces` endpoint at port 5000. The API is proxied by the vSilos that have the "detector" vThing, instead, because Users endpoints to VirIoT are vSilos, not ThingVisors.

- Offers a REST interface to fetch pictures (both target and recognized) via their identifier, via HTTP GET. Similar to the above, this is accessed through the vSilo. The interface is called:

`/media/<pictureidentifier>.`

9.1.4 The vSilo that has a "detector" vThing

Target pictures of faces to be matched against the incoming video frames are POSTed by Users to the vSilo's HTTP Broker. The HTTP Broker running inside the vSilo acts as a proxy to the HTTP REST interfaces offered by the FaceRecognition ThingVisor, that are not intended for direct access from Users. Thus the vSilo's HTTP Broker acts as the only entry point for Users (and Applications) to the face recognition process.

From the User's perspective, this is the typical workflow to operate the face recognition process. In the following, we assume an NGSI-LD flavor vSilo is used.

- 1) Add the "detector" to the vSilo:

If the FaceDetector TV is called "facerecognition-tv", the NGSI-LD silo is called "silongsildorionld1-eu" and the User is called "tenant1", then the command is:

```
f4i.py add-vthing -v facerecognition-tv/detector -t tenant1
-s silongsildorionld1-eu
```

- 2) Check the received NGSI-LD Entity and its capabilities:

Assuming the NGSI-LD Broker runs on internal port 1026, which is mapped to external, the command will be:

```
curl http://<VSIL0_PUBLIC_IP>:<PORT_MAPPED_TO_1026>/ngsi-ld/v1/entities/
urn:ngsi-ld:facerecognition-tv:detector | jq
```

Resulting in the following NGSI-LD Entity, where we see the list of available commands, and the empty `commands`, `command-status` and `command-result` properties created in the Broker to implement the actuation. We also see the `generatedByVThing` property that keeps track of the vThing that created the Entity (not relevant to this workflow).

```
1 {
2   "id": "urn:ngsi-ld:facerecognition-tv:detector",
3   "type": "FaceRecognitionEvent",
4   "commands": {
5     "value": [ "startjob", "deletejob" ],
6     "type": "Property"
7   },
8   "generatedByVThing": {
9     "value": "facerecognition-tv/detector",
10    "type": "Property"
11  },
12  "startjob": {
13    "value": {},
14    "type": "Property"
15  },
16  "startjob-status": {
17    "value": {},
18    "type": "Property"
19  },
20  "startjob-result": {
21    "value": {},
22    "type": "Property"
23  },
24  "deletejob": {
25    "value": {},
26    "type": "Property"
27  },
28  "deletejob-status": {
29    "value": {},
30    "type": "Property"
31  },
32  "deletejob-result": {
33    "value": {},
34    "type": "Property"
35  }
36 }
```

3) Send one (or more) target pictures of faces to be recognized, assigning persons' names to them and a job identifier:

Pictures are POSTed to vSilo's HTTP proxy running on internal port 80 (mapped to external). It is important to notice that the User addresses the "detector" vThing directly, without knowing the details where the vThing is currently deployed within VirIoT distributed microservices architecture. The HTTP Data Distribution will take care of efficiently routing the HTTP request.

The 123456 job identifier serves the purpose of a "secret link" able to protect and isolate the various jobs that different Users of different vSilos are sending to the FaceRecognition's "detector" in parallel.

```
curl -X POST -F "pic=@bio.jpg" http://<VSIL0_PUBLIC_IP>:<
  PORT_MAPPED_TO_80>/vstream/facerecognition-tv/detector/targetfaces
  /123456/Andrea
```

4) Send a `startjob` command to the detector with the job identifier we want to start: This is accomplished, in case of NGSI-LD Broker, by updating the value of the `startjob` NGSI-LD property (by a PATCH call at the `/attrs/startjob` endpoint of the Broker's API).

```
curl -X PATCH http://<VSIL0_PUBLIC_IP>:<PORT_MAPPED_TO_1026>/ngsi-ld/v1/
  entities/urn:ngsi-ld:facerecognition-tv:detector/attrs/startjob
  -d '{"value":{"cmd-id":"xaxaxa","cmd-qos":2,"cmd-value":{"job
    ":"123456"}}}'
  -H "Content-Type: application/json"
```

5) Check for periodic updates of the `startjob-status` property:

Here follows an example snapshot of the NGSI-LD Entity representing the “detector” right after it has detected a match for Andrea’s face. The original picture was POSTed under job “123456” and tagged as “Andrea”. It is downloadable, through vSilo’s HTTP proxy, at `/media/60ba4b6a5faca138c398b3d4`. The video frame that matched is available at `/media/60ba4b7b5faca138c398b3e8`.

```
1 {
2   "id": "urn:ngsi-ld:facerecognition-tv:detector",
3   "type": "FaceRecognitionEvent",
4   "commands": {
5     "value": [ "startjob", "deletejob" ],
6     "type": "Property"
7   },
8   "startjob": {
9     "value": {
10      "cmd-id": "xaxaxa",
11      "cmd-qos": 2,
12      "cmd-value": {
13        "job": "123456"
14      }
15    },
16    "type": "Property"
17  },
18  "startjob-status": {
19    "value": {
20      "cmd-id": "xaxaxa",
21      "cmd-nuri": "viriote://vSilo/tenant1_silongsildorionld1-eu/data_in",
22      "cmd-qos": 2,
23      "cmd-value": {
24        "job": "123456"
25      },
26      "cmd-status": {
27        "job": "123456",
28        "name": "Andrea",
29        "original-uri": "/media/60ba4b6a5faca138c398b3d4",
30        "recognized-uri": "/media/60ba4b7b5faca138c398b3e8"
```

```
31     }
32   },
33   "type": "Property"
34 },
35 "startjob-result": {
36   "value": {},
37   "type": "Property"
38 },
39 "deletejob": {
40   "value": {},
41   "type": "Property"
42 },
43 "deletejob-status": {
44   "value": {},
45   "type": "Property"
46 },
47 "deletejob-result": {
48   "value": {},
49   "type": "Property"
50 }
51 }
```

6) Download the matching picture from the `/media` API.

9.2 How to run it

As explained above, the FaceRecognition ThingVisor gets video frames from an upstream CameraSensor ThingVisor. Hence the first thing to do is to add a CameraSensor TV to VirIoT

9.2.1 Running the CameraSensor ThingVisor

```
f4i.py add-thingvisor -y ../yaml/thingvisor-cameraSensor-http.yaml
-n camerasensor-tv -d "camera frames via http"
-p '{"bufferSize":30}' -z default
```

Also, please run a `set-endpoint` VirIoT command on the “sensor” vThing of the CameraSensor TV, to make the HTTP Data Distribution able to proxy the `/framesinput` and `/bufferedframes` APIs by simply using the name of the vThing. This way, the FaceRecognition TV (and other ThingVisors as well) can GET video frames using the CameraSensor TV’s service name, globally within VirIoT, exploiting efficient caching and multicast distribution of the video frames.

```
f4i.py set-vthing-endpoint -v camerasensor-tv/sensor
-e http://127.0.0.1:5000
```

9.2.2 Running the Camera System

The next step is to run the camera system, which sends video frames to the CameraSensor TV's HTTP interface. Thus, we need to know public IP address to access the services running inside the VirIoT cluster and, most important, the external port that maps to the internal port 5000 of the CameraSensor TV. This can be accomplished by inspecting the thingvisor via a VirIoT `inspect-thingvisor` command:

```
f4i.py inspect-thingvisor -v camerasensor-tv | grep 5000/tcp
```

The Camera System that we have developed for testing purposes runs on a Jetson Nano. It can be found in the `jetbot_scripts` folder of the CameraSensor ThingVisor and is called `camerasensor-to-http.py`. It, in turn, imports a python module, which is responsible for video capture, and is called `camera.mod.py`. Both can be found in the `jetbot_scripts` folder of the CameraSensor TV.

Some pre-requisites are needed:

```
pip3 install requests
pip3 install traitlets
pip3 install opencv-python
```

Assuming that is the output of the `inspect-thingvisor` VirIoT command above, it is executed simply as follows, i.e. by specifying the base URL of the CameraSensor TV's API:

```
python3 ./camerasensor-to-http.py
http://<CAMERASENSORTV_PUBLIC_IP>:<PORT_MAPPED_TO_5000>
```

9.2.3 Running the FaceRecognition ThingVisor

Is is now possible to run the FaceRecognition ThingVisor, specifying the name of its upstream CameraSensor TV, which we had created above:

```
f4i.py add-thingvisor -y ../yaml/thingVisor-faceRecognition.yaml
-n "facerecognition-tv" -d "recognizes faces"
-p '{"fps":4, "upstream_vthingid":"camerasensor-tv/sensor"}'
-z default
```

Also, please run a `set-endpoint` VirIoT command on the “detector” vThing of the FaceRecognition TV, to make the HTTP Data Distribution able to proxy the `/media` and `/targetfaces` APIs.


```
f4i.py set-vthing-endpoint -v facerecognition-tv/detector  
-e http://127.0.0.1:5000
```

9.2.4 Run a vSilo

At this point, Users can run their favourite vSilo and add the “detector” vThing to it, as explained above, to control the face recognition process.

10 Access Control Framework instantiation in VirIoT

In previous Deliverables (D2.3 - System Architecture Second Release, and D4.2 - Smart City information sharing services - second release), we have explained VirIoT's architecture, and how security and privacy are provided. Two different access control technologies have been implemented, a more coarse grained one based on JSON Web Tokens (JWT), and a more fine grained one, which implements the Distributed Capability-Based Access Control DCapBAC approach.

In Section 10.1 we show how the token-based approach is used to control access by vSilos (and consequently applications) to virtualized actuators, acting at the level of the actuation commands offered by the vThings inside ThingVisors.

In sections 10.2 to 10.4 we show how the DCapBAC was integrated in VirIoT as a security plugin that interacts directly with the Master Controller, presenting all the relevant interactions.

10.1 Token-based Access Control of Actuators

While physical objects can usually be manipulated by just one person at a time, vThings can much more easily offer different partial functionality in parallel, to different (or the same) applications.

For instance, a virtualized door lock may offer both a "door-open" command and a "permanently-door-block" command: the "door-open" functionality is available to every application that has been granted the key, while the "permanently-door-block" command is only available to specific applications (for instance police permanently blocking the door from remote).

Access control to the "door-open" capability is resource-centric: the possibility to open the door must be granted to all applications that possess the key, regardless their identity.

Conversely, only personnel positively identified as police has the possibility to operate the "permanently-door-block" command: access control is identity-based in this case.

One application at a time acquires exclusive access to the resource to be actuated: for instance the "students application" has exclusive access to the "door-open" command of the laboratory during the day, while the "lab security" application has exclusive access at night time. Both applications can be overridden by the "police application" at any time, because that application possesses a different token, allowing execution of the "permanently-door-block" command.

Other unorthodox access control policies are location or time based. Consider for instance a kiosk issuing progressive tickets that are subsequently used to grant access to a time consuming experience, such as a 10 minutes test flight of a drone from a remote control application running inside a vSilo. In this case access is conditional to the time the ticket was acquired (or possibly even the physical presence of the application's users at the kiosk).

This idea of granting "the possibility to operate" different parts of the virtualized actuators, based on flexible policies, is nicely implemented by the concept of a token-based access control, whereas tokens are released by vThings to applications.

We have explored this concept by associating, inside ThingVisors, one or more tokens to each command supported by the ThingVisor's vThings. Tokens are then distributed to the vSilos that need to operate the command. The vSilo must subsequently show the token to the ThingVisor in order to gain access to that command.

Our token-based approach has the following characteristics:

- ThingVisors can be stateless to a greater extent, because the application, not the ThingVisor, keeps track of tokens: the functions that an application chooses, or is authorized, to access, are proved by possession of the token;
- it does not require ThingVisors to identify applications, in order to evaluate the outcome of access control, i.e. to grant or deny access. ThingVisors do not need to assume that applications are known beforehand in an access control list. Access can be granted to anonymous applications, or applications identified by pseudonyms;
- tokens are a flexible access control method that can, with the help of dedicated rules, support many different, and generic, access control policies, even unorthodox ones;
- it fosters collaboration among applications, because tokens can be traded in order to access other applications's capabilities. Since tokens only relate to a limited set of commands, they are less risky to disclose or broadcast to others than, for instance a password. As an example, the "lab security" application that can "door-open" the laboratory to the security personnel, may lend its token to another application that has permission to operate a camera for face recognition purposes; in this way, after recognizing a face, it can also open the door to let the person enter the lab. The two applications now collaborate without the need for developers to foresee and implement any access control at the ThingVisor, specific for this collaboration use case: the security policy is implemented outside of the vThings, thanks to the trading of tokens.

From the examples above, we see that different virtualized actuators may need different access-control policies: applications can send actuation-commands, but those commands may or may not be accepted by the ThingVisor depending on the tenant's priority and the current state of the vThing. By embedding authorization tokens within commands we provide to VirIoT's applications the means to carry access-control policy decisions, but leave the implementation of the specific policy within each ThingVisor, and thus in the hands of the developer of the vThing. This makes VirIoT flexible enough to include actuators with unforeseen policies.

The policy for obtaining the token(s) that grant authorization to subsequently execute an actuation command, or more broadly to control a vThing, is vThing-dependent, since the information needed to obtain the authorization depends on the specific use case's

```
1 {"id": "urn:ngsi-ld:lamp:light1",  
2  "type": "lamp",  
3  "on": {  
4    "type": "Property",  
5    "value": false},  
6  "commands": {  
7    "type": "Property",  
8    "value": ["set-on", "token-req"]},  
9  "@context": ["https://uri.etsi.org/..."]}
```

Figure 29: NGSI-LD Entity representing context information of a lamp

security. But we have designed a dedicated field of command requests, named `cmd-token`, to carry the token once obtained, which proves that the application issuing the command is authorized.

Let us now explain in details how we have implemented actuation and management of authorization tokens, through a generic example involving a simple “virtual lamp” vThing, that is a Virtual Actuator because it supports two actuation commands, specifically `set-on` and `token-req`.

Figure 29 shows the context data associated to the virtual lamp, which includes two Properties. The first Property is named `on`; its value is the status of the lamp (false means lamp off). The second Property is VirIoT’s standard property in case the vThing is a Virtual Actuator, i.e. `commands`; its value is the list of possible actuation-commands offered by the Virtual Actuator. In this case, this list is composed of `set-on` and `token-req` commands, that can be used to configure the `on` status of the lamp and to request an authorization token, respectively.

Figure 30 shows the interaction between tenants and the virtual lamp, resulting in the twinned real lamp changing its state from off to on. We have two tenants who have the virtual lamp in their vSilos, whose initial state is off (`on=false`). For each actuation-command, a vSilo exposes in its IoT Broker three *actuation-pipes* used by the tenant to inject the actuation-command and receive feedback messages².

To turn on the lamp, the tenant of vSilo1 obtains a authorization token to execute the command as discussed later on (steps 1,2) and then injects into the `set-on` actuation pipe the actuation-command shown in Figure 31. The command is immediately received by the IoT Controller and then transferred to the ThingVisor that implements the virtual lamp (step 3). When the virtual lamp receives the message, the actuation-command is accepted and the virtual lamp starts the actuation process of the real lamp, using the proprietary API the lamp provides.

When the actuation-command is accepted, a *status* feedback is sent to vSilo1 only, to inform the tenant that the actuation is in progress (step 4). The feedback message is received by the IoT Controller and relayed to the `set-on-status` actuation-pipe of the IoT Broker, from which it can be observed by the tenant.

²The implementation of an actuation-pipe depends on the IoT Broker. For example: in oneM2M, an actuation-pipe is implemented by a oneM2M container; in NGSIv2/NGSI-LD, an actuation pipe is implemented by an Entity. The IoT Controller is made aware of the actuation-pipes to be created via the `commands` Property.

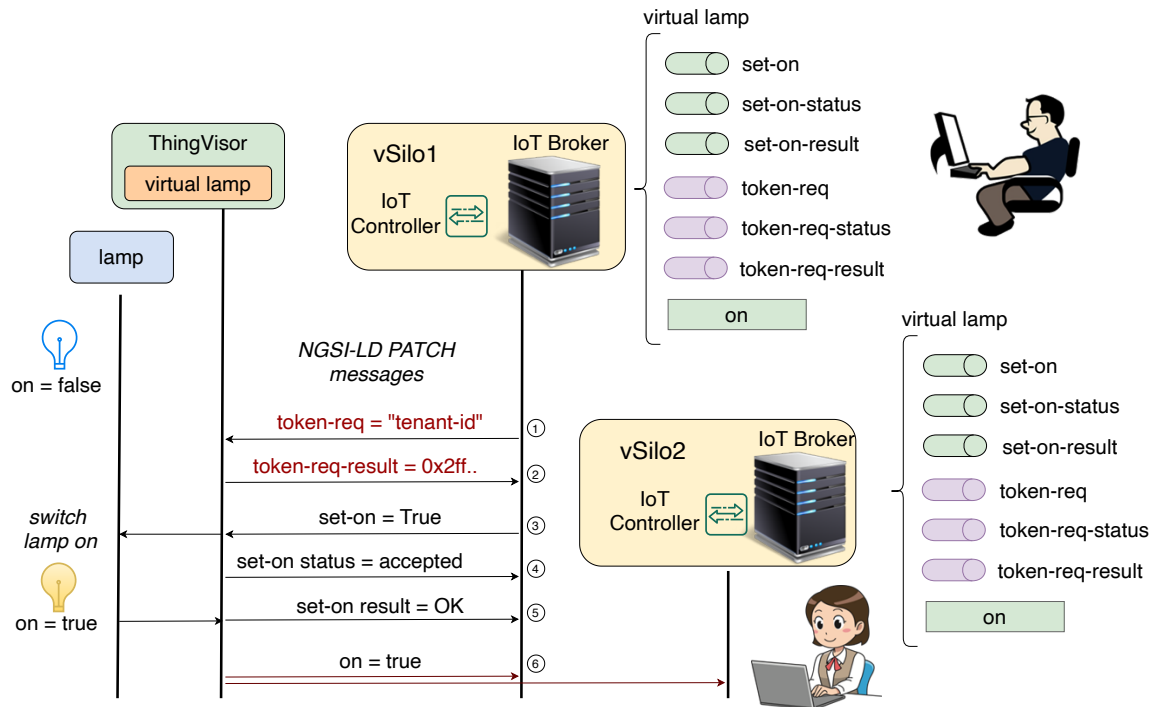


Figure 30: Virtual Actuator workflow, QoS = 2

At the end of the actuation, when the lamp is turned on, a *result* feedback is sent to vSilo1 only, to inform the tenant that the actuation is complete (step 5). This feedback message is relayed from the IoT Controller to the **set-on-result** actuation-pipe of the IoT Broker. Also, since the context Property **on** is changed, as a result of the actuation, from **False** to **True**, a context update message is sent to all vSilos, as previously explained (step 6).

Actuation-commands and status/result feedback are simple messages, and for mere format consistency, they are encoded by NGSI-LD PATCH messages, as it is shown, for the actuation-command, in Figure 31. The message represents the **set-on** actuation-command as a NGSI-LD Property whose value is a JSON object containing the following attributes:

- **cmd-value** contains the arguments of the command;
- **cmd-id** is a unique id of the command;
- **cmd-qos** is a concept of “actuation QoS” we introduced to differentiate the types of feedback messages sent by a Virtual Actuator: 0 = no feedback; 1 = result message at the end of the actuation; 2 = one or more status messages during the actuation and a result message at the end of the actuation³;

³QoS=0 is useful for use cases where actuation-commands are issued at such a high rate that waiting for feedback is useless. QoS=1 is useful for fast and reliable actuation. QoS=2 is useful for a long

```

1 {"id": "urn:ngsi-ld:lamp:light1",
2  "type": "lamp",
3  "set-on" : {
4    "type": "Property",
5    "value": {
6      "cmd-value": true,
7      "cmd-id": "123456",
8      "cmd-qos": "1",
9      "cmd-token": "0x23456",
10     "cmd-nuri" : "viriot:/vSilo/tenant1_vSilo1/data_in" }}}

```

Figure 31: **set-on** actuation-command

- **cmd-token** is the authorization token;
- **cmd-nuri** is the notification URI where to send actuation feedback and, by default, it is the **data_in** topic of the vSilo (see Deliverable D4.2), so that only the requesting vSilo will receive feedback messages.

Status/result feedback messages are identical to the actuation-command they refer to, but have the additional **cmd-status/cmd-result** keys.

Figure 30 shows a possible example of the procedure to obtain and use the authorization token. The virtual lamp exposes a specific actuation-command **token-req**, which is used by a tenant to request a token to be used in any subsequent actuation-command (e.g., **set-on**) in the **cmd-token** field (step 1). Information about the tenant, for example the tenant-id, is included in the **cmd-value** field of **token-req**. When the ThingVisor receives the **token-req**, the access-control policy is used and eventually a token is sent back with the **token-req-result** message (step 2).

Thus, VirIoT uses the same approach as HTTP, where the authorization header is the means to carry result of an access-control policy, but without using the HTTP protocol.

10.2 DCapBAC Component functionalities

The Distributed Capability-Based Access Control (DCapBAC) technology decouples the traditional approach provided by the XACML framework into two phases. The authorisation request phase, which is provided by this framework, and the access phase. Once the authorisation is granted, the Capability Manager component issues authorisations tokens which must be included in subsequent requests to the Master-Controller using its proper API. A PEP_Proxy captures these requests, validates the authorisation tokens, and in case of a positive validation, acts as a mere intermediary between the User and the Master Controller forwarding forth and back the corresponding messages.

The authentication is also considered in this scenario thanks to the Identity Manager, to be more specific we are using the FIWARE GE Keyrock.

lasting actuation that requires feedback during execution. For example, for the virtual face detector, an actuation-command **set-face-feature** is used to send the parameters of the face to be detected, QoS is set to 2, and status messages are sent to the requesting vSilo each time the face is detected. We used QoS=2 for the lamp example for completeness, but QoS=1 is more appropriate for this use case.

Figure 32 shows DCapBAC and Blockchain technologies, where access control process is decoupled in two phases:

- 1st operation to receive authorisation. A token is issued
- 2nd operation access to the resource previously validating the token.

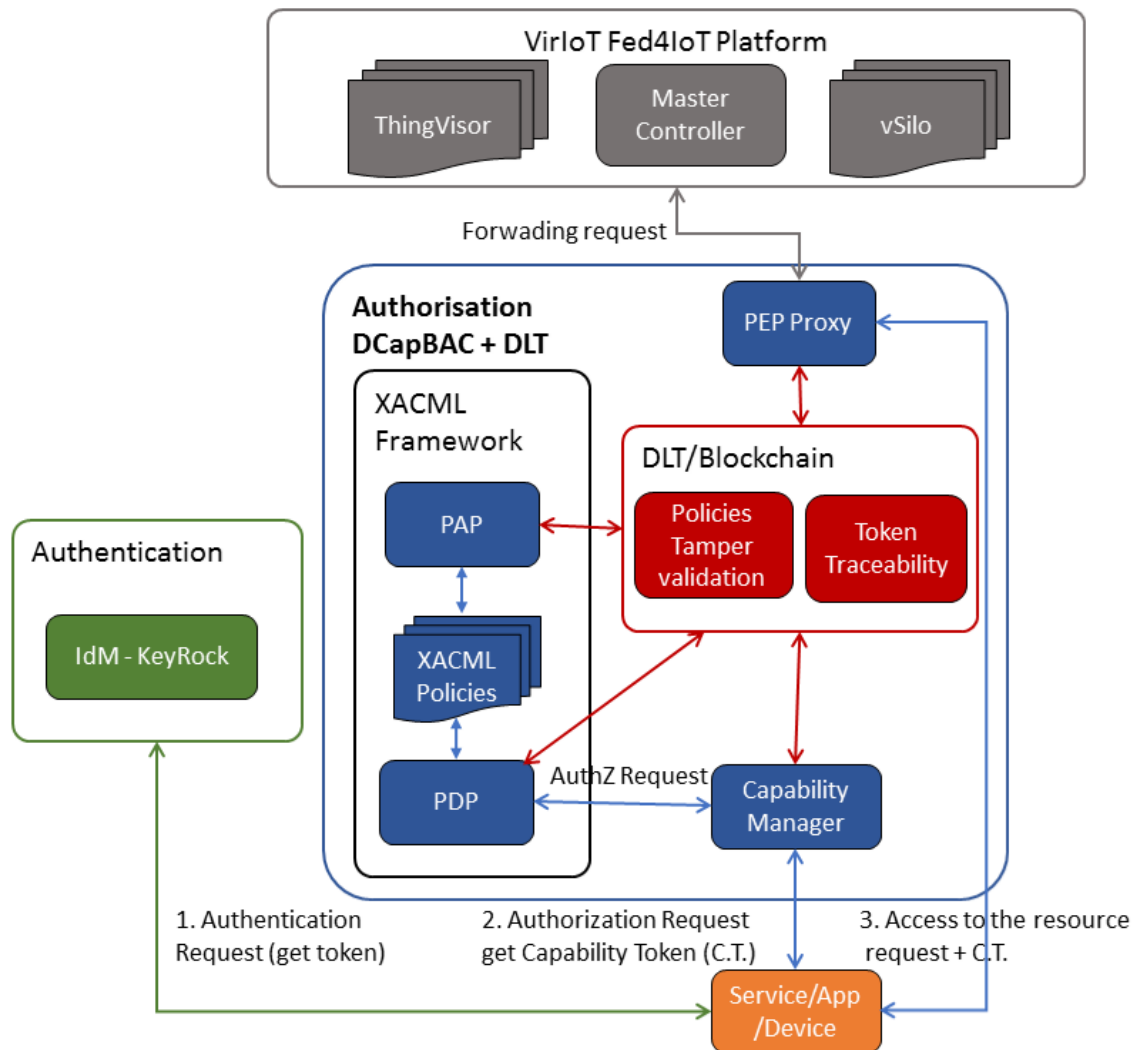


Figure 32: DCapBAC Operation Model and Blockchain

10.2.1 IdM-Keyrock

Keyrock⁴ is a FIWARE GE responsible for the Identity Management. It implements an OAuth2 API for managing the identities in its repository, as well as for authentication

⁴FIWARE GE Keyrock: <https://fiware-idm.readthedocs.io/en/latest/index.html>

purposes. Once the authentication is granted, an authentication token is issued by the Identity Manager, and it should be included in the next call to the Capability Manager. The authentication enforcement is usually carried out by a PEP_PROXY which after receiving the request, with the corresponding authentication token within, validates this latter by contacting the Identity Manager.

Nevertheless, in our scenario, the authentication enforcement is carried out by the Capability Manager, as we will later see.

10.2.2 XACML framework PAP and PDP

This element corresponds to the implementation of the XACML framework. It comprises:

- a Policy Administration Point (PAP) which is responsible for managing the XACML authorisation policies through a GUI. These XACML policies are defined according to a triplet (subject, resource, action).
- a Policy Decision Point (PDP), responsible for matching the authorisation requests with the XACML policies defined by the PAP, and issuing positive/negative verdicts accordingly.

10.2.3 Capability Manager

The Capability Manager is the contact point for services and users that intend to access the resources offered by the Master-Controller's API. In this sense, it provides a REST API for receiving authorisation queries. These must include the authentication token already issued by the Identity Manager. So, once an authorisation query is received, it firstly interacts with the Identity Manager to validate the authentication token. Secondly, it tailors this to a XACML authorisation request, and forwards it to the XACML-PDP. Finally, after receiving a positive verdict from this component, it issues the Capability Token (authorisation token) which will be later used in future queries to the Master Controller.

Regarding DCapBAC scenario, Capability Manager covers the first phase of the access control process (receive authorisation).

10.2.4 PEP-Proxy

This PEP-Proxy is responsible for enforcing the authorisation in the access phase. Therefore, this component is responsible for receiving the queries aimed at the Master Controller, which must include the corresponding 'Capability Token', validates them, and forward the requests to the Master-Controller, as well as the responses coming from the Master Controller back to the requester.

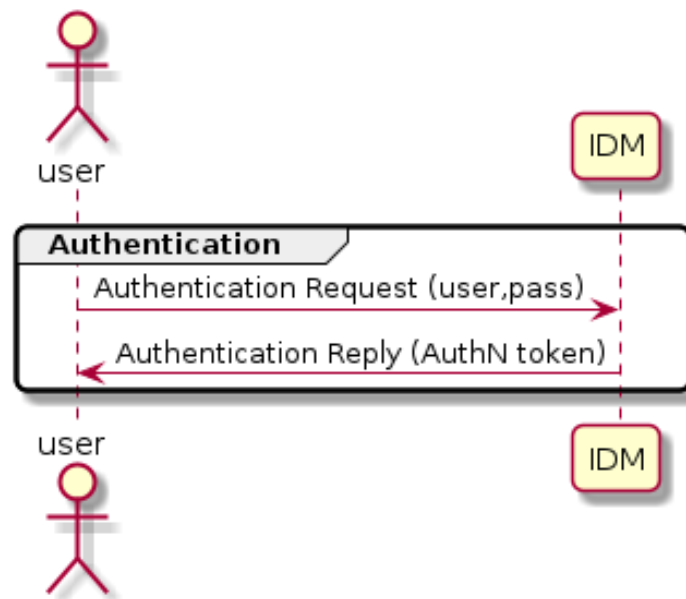


Figure 33: IdM-Keyrock authentication

10.3 DCapBAC Components operation

This section explains the interaction between all the security components and the integration with Master-Controller component. In this sense, it details the corresponding API of each component and the sequence of the resulting accesses.

10.3.1 IdM-Keyrock

IdM-Keyrock provides a REST API for receiving authentication queries. When an authentication request is received, the component recovers user credentials from the JSON body request and returns an IdM token if it's the case. This token will be required by authorisation process (through Capability Manager). Figure 33 shows the sequence diagram of authentication request.

Resources can be managed through the API (e.g. Users, applications and organizations). One of the main uses of IdM-Keyrock is to allow developers to add identity management (authentication and authorization) to their applications based on FIWARE identity. This is possible thanks to OAuth2 protocol. Further information can be found in the Keyrock Apiary⁵

IdM-Keyrock allows defining user attributes, and too, assign them to organizations or application roles, etc... This information is one of the pieces that the XACML framework will require to define access control policies.

⁵Keyrock Apiary:<https://keyrock.docs.apiary.io>

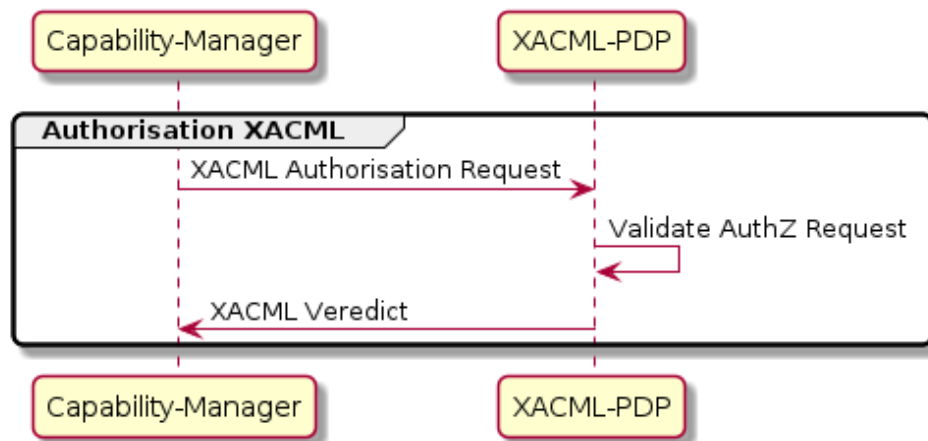


Figure 34: XACML authorisation request verdict

10.3.2 XACML framework PAP and PDP

XACML-PAP is a GUI for managing XACML policies (configuration), it's not interfering in obtaining authorisation requests verdict.

XACML-PDP offers an endpoint which returns the verdict when an authorisation request is received from Capability Manager, it recovers from the body:

- the subject and subject's type of the resource's request. These fields reference the IdM-Keyrock stored information. In this sense, it can be a user attribute (user name, email, etc...), a specific organization, etc...
- the resource: endpoint (protocol+IP+PORT) + path of the resource's request
- the action: method of the resource's request ("POST", "GET", "PATCH", "DELETE")

With this information, XACML-PDP accesses the XACML policies for validating authorisation requests and obtain if the subject can access a resource and can perform the action over the resource (verdict). Figure 34 shows the sequence diagram of this interaction request.

The next box shows the format of a PDP obtain verdict request, only it is needed to replace 'subjectType', 'subject', 'resource' and 'action' with the correct values, related with XACML policies defined previously in XAMCL-PAP.

POST http://{XACML-IP}:{XACML-Port}/XACMLServletPDP/

```
<Request xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os">
  <Subject SubjectCategory="urn:oasis:names:tc:xacml:1.0:subject-
    category:access-subject">
    <Attribute AttributeId=subjectType DataType="http://www.w3.org
      /2001/XMLSchema#string">
      <AttributeValue>subject</AttributeValue>
    </Attribute>
  </Subject>

  <Resource>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:
      resource-id" DataType="http://www.w3.org/2001/XMLSchema#string
      ">
      <AttributeValue>resource</AttributeValue>
    </Attribute>
  </Resource>

  <Action>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-
      id" DataType="http://www.w3.org/2001/XMLSchema#string">
      <AttributeValue>action</AttributeValue>
    </Attribute>
  </Action>

  <Environment/>
</Request>
```

HEADERS: _____

Content-Type: text/plain

Regarding the response, there are three possible verdicts (Permit, NotApplicable, Deny) all of them with 200 - OK status code response. The following boxes show the format of the different responses.

RESPONSE: HTTP/1.1 200 OK; Verdict = Permit

```
<Response>
  <Result ResourceID="resource">
    <Decision>Permit</Decision>
    <Status>
      <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
    <Obligations>
      <Obligation ObligationId="liveTime" FulfillOn="Permit">
      </Obligation>
    </Obligations>
  </Result>
</Response>
```

RESPONSE: HTTP/1.1 200 OK; Verdict = NotApplicable

```
<Response>
  <Result ResourceID="resource">
    <Decision>NotApplicable</Decision>
    <Status>
      <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
  </Result>
</Response>
```

RESPONSE: HTTP/1.1 200 OK; Verdict = Deny

```
<Response>
  <Result ResourceID="resource">
    <Decision>Deny</Decision>
    <Status>
      <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
  </Result>
</Response>
```

10.3.3 Capability Manager

This component provides a REST API for receiving authorisation queries, which are tailored and forwarded to the XACML PDP for a verdict.

When an authorisation request is received by Capability Manager, it recovers from the JSON body:

- an authentication token which proceeds from the authentication phase (access to IdM-Keyrock).
- an endpoint of the resource's request (protocol+IP+PORT). In DCapBAC scenario, it corresponds with PEP-Proxy component.
- the action/method of the resource's request ("POST", "GET", "PATCH", "DELETE").
- the path of the resource's request.

With this information, Capability Manager:

- Access to authentication component (IdM-Keyrock) to validate authentication token.
- Access to XACML framework for validating authorisation requests (through PDP) and obtain if the subject can access a resource and can perform the action over the resource (verdict).
- If a positive verdict is received, finally, the Capability Manager issues an authorisation token called Capability Token which is a signed JSON document that contains all the required information for the authorisation, such as the resource to be accessed, the action to be performed, and also a time interval during the Capability Token is valid. This token will be required to access to the resource (through PEP-Proxy).

Figure 35 shows the sequence diagram of full first phase of DCapBAC access control process.

The next box shows the format of the request, only it is needed to replace 'authToken', 'device', 'action' and 'resource' with the correct values.

POST https://{CapMan-IP}:{CapMan-Port}:3040

```
{
  "token": "authToken",
  "de": "device",
  "ac": "action",
  "re": "resource"
}
```

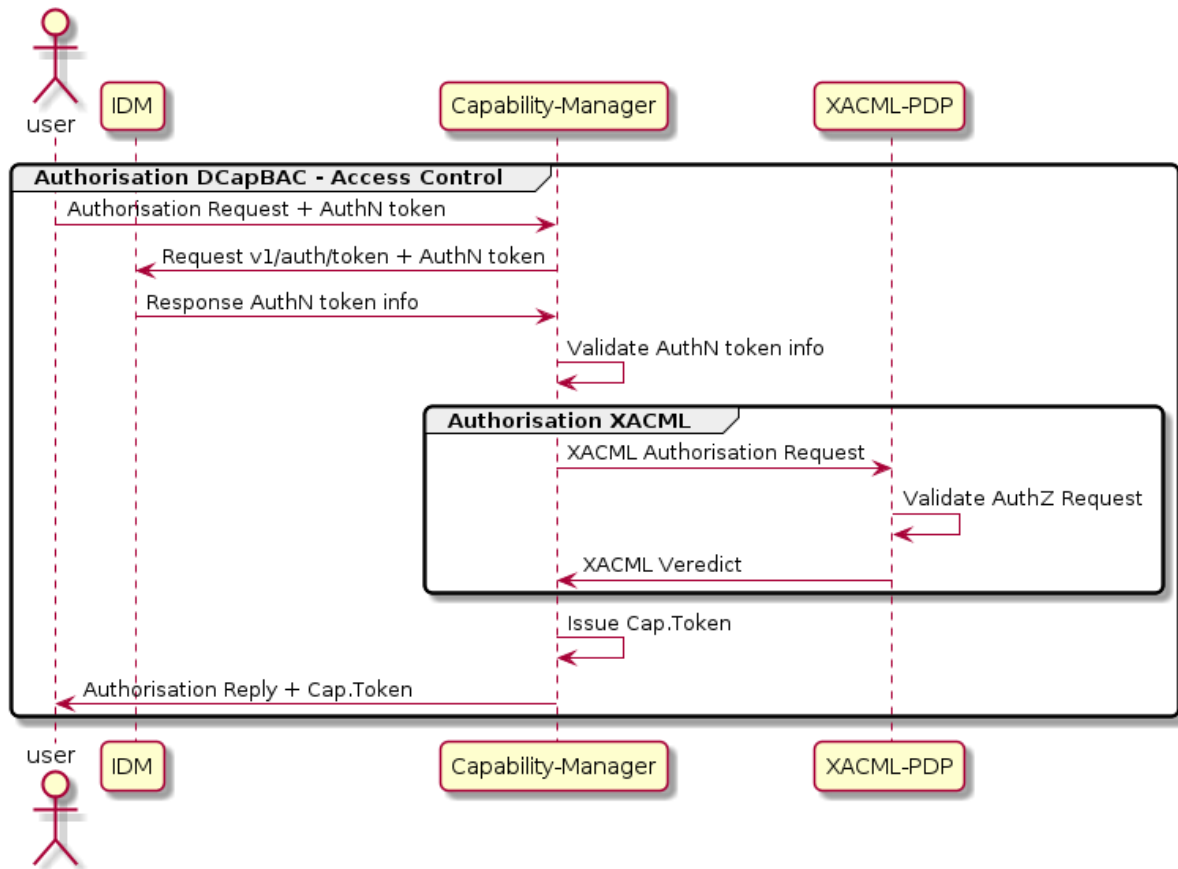


Figure 35: Capability Manager request

HEADERS: _____

Content-Type: application/json

Regarding the response, there are three possible responses:

- 200-OK, with Capability Token in response body.

RESPONSE: HTTPS/1.1 200 OK

"{Capability token}"

- 401-Unauthorized.

RESPONSE: HTTPS/1.1 401 Unauthorized

```
{"error": {"message": "Auth Token has expired", "code": 401, "title": "Unauthorized"}}
```

- 500-Internal Server Error.

RESPONSE: HTTPS/1.1 500 Internal Server Error

```
Can't generate capability token
```

10.3.4 PEP-Proxy

This component receives queries aimed to access to a resource, queries contain a ‘Capability Token’. The PEP-Proxy validates this token, and in case the evaluation is positive, it forwards requests to the specific endpoint’s API.

When an access resource request is received by PEP-Proxy:

- recovers the ‘x-auth-token’ header (‘Capability Token’).
- validates ‘Capability Token’.
- If ‘Capability Token’ validation is successful, PEP-Proxy forwards the message and sends responses back to the requester.

Figure 36 shows the sequence diagram of PEP-Proxy request (second phase of DCap-BAC access control process).

The PEP-Proxy component supports multiple ‘REST APIs’. It offers the same API as the component one where it forwards requests.

10.3.5 Full integration view

Figure 37 shows the full integration sequence diagram of security components with our platform and more exactly over the Master-Controller component, explained in the previous subsections.

10.4 Configuring and testing

This section explains the configuration requirements of security components in order to test them and interact with Master-Controller’s API.

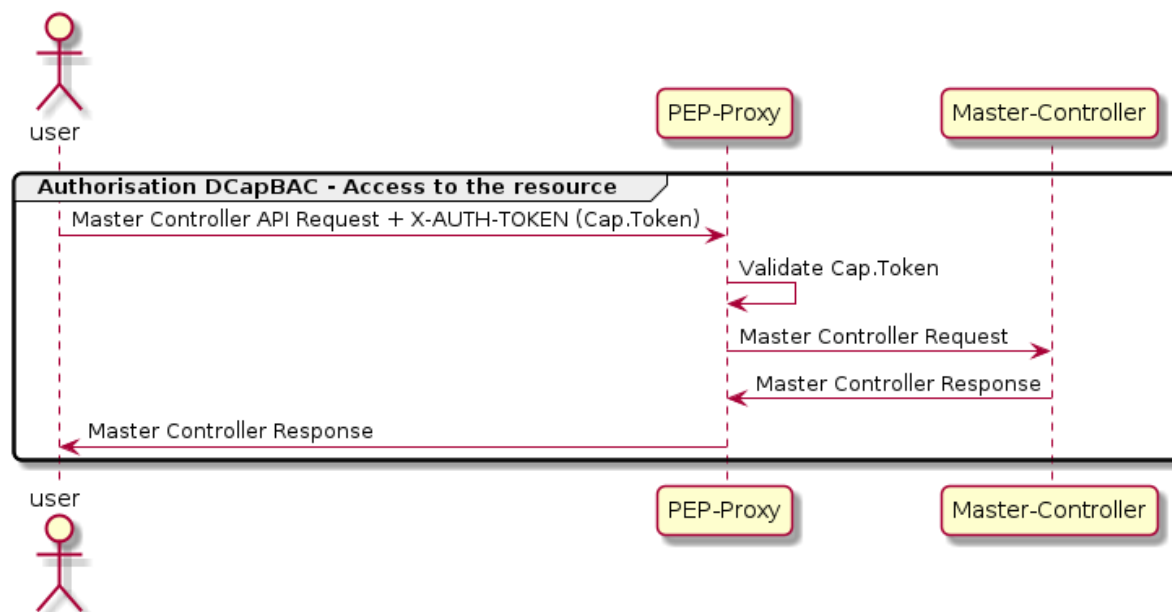


Figure 36: PEP-Proxy request

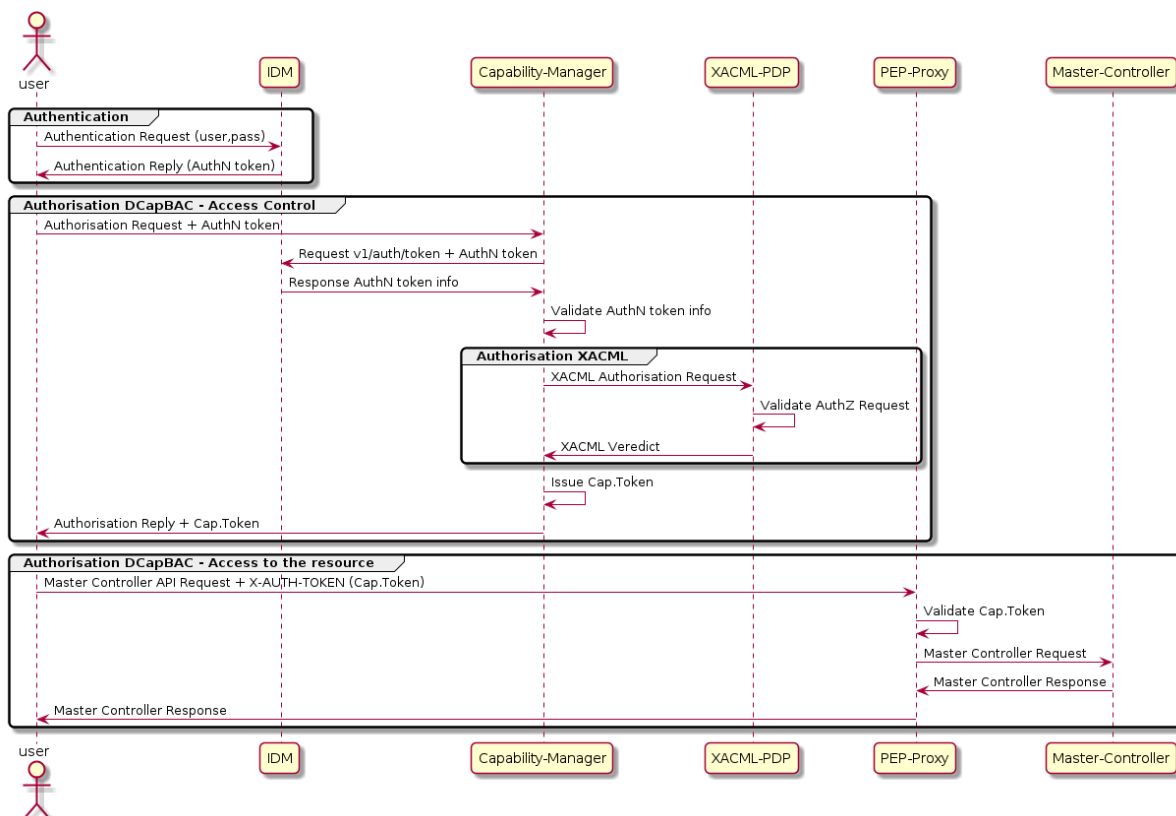


Figure 37: Full integration view

Security components allow be deployed using Docker or Kubernetes technologies and, this deployment can be launched in the same environment than Master-Controller component or not, anyway, with the goal of do not complicate the architecture unnecessarily, this section understands that the environment is the same one.

Regarding the configuration, two aspects will considered, on the one hand, the necessary configuration for the deployment of the components will be seen, on the other hand, the actions to be carried out on them, once deployed, will be shown to define access control.

10.4.1 IdM-Keyrock

As mentioned, Keyrock⁶ is a FIWARE GE responsible for the Identity Management. Keyrock installation documentation⁷ describes two ways of installing, in this sense:

- Host installation.
- Docker installation.

Anyway, **Kubernetes** installation could be done too.

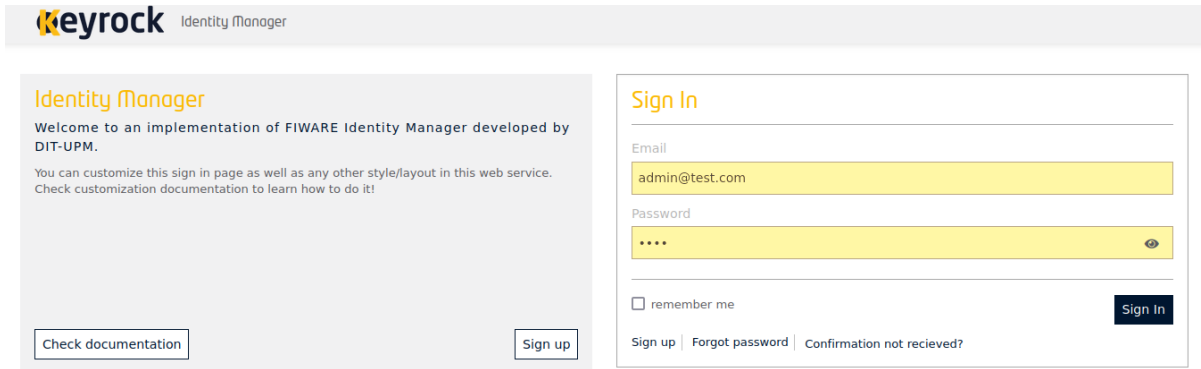
Before deploying this component, the IdM-Keyrock environment variables must be considered, the official documentation⁸ of this component details all the environment variables are supported by the component but, with the aim to simplify the process will be considered only these ones:

- `IDM_DB_HOST`: Name of the host where is running the database.
- `IDM_HOST`: Name of the host where is running IdM-Keyrock.
- `IDM_PORT`: Port where IdM-Keyrock will be running.
- `IDM_DB_PASS`: Password to authenticate IdM-Keyrock to perform actions against the database.
- `IDM_DB_USER`: Password to authenticate IdM-Keyrock to perform actions against the database.
- `IDM_ADMIN_USER`: Username of admin default user in IdM-Keyrock.
- `IDM_ADMIN_EMAIL`: Email of admin default user in IdM-Keyrock.
- `IDM_ADMIN_PASS`: Password of admin default user in IdM-Keyrock.
- `IDM_HTTPS_ENABLED`: Enable IdM-Keyrock to listen on HTTPS.

⁶FIWARE GE Keyrock: <https://fiware-idm.readthedocs.io/en/latest/index.html>

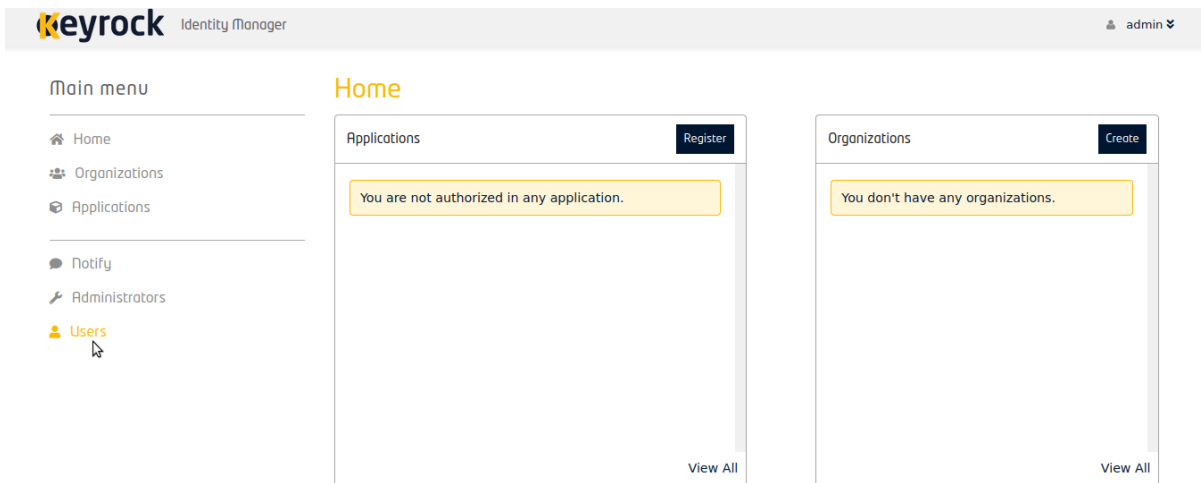
⁷Keyrock, installation: https://fiware-idm.readthedocs.io/en/latest/installation_and_administration_guide/introduction/index.html

⁸Keyrock, Environment variables: https://fiware-idm.readthedocs.io/en/latest/installation_and_administration_guide/environment_variables/index.html



The image shows the login page of the Keyrock Identity Manager. The header includes the Keyrock logo and the text 'Identity Manager'. The main content area is divided into two sections. The left section, titled 'Identity Manager', contains a welcome message: 'Welcome to an implementation of FIWARE Identity Manager developed by DIT-UPM. You can customize this sign in page as well as any other style/layout in this web service. Check customization documentation to learn how to do it!'. Below this message are two buttons: 'Check documentation' and 'Sign up'. The right section, titled 'Sign In', contains a form with fields for 'Email' (with the value 'admin@test.com') and 'Password' (with masked characters '....'). There is a 'remember me' checkbox and a 'Sign In' button. At the bottom of the 'Sign In' section are links for 'Sign up', 'Forgot password', and 'Confirmation not recieved?'.

Figure 38: IdM-Keyrock - Login



The image shows the main page of the Keyrock Identity Manager. The header includes the Keyrock logo, the text 'Identity Manager', and a user profile icon labeled 'admin'. The main content area is divided into three sections. The left section, titled 'Main menu', contains a list of links: 'Home', 'Organizations', 'Applications', 'Notify', 'Administrators', and 'Users' (which is highlighted with a mouse cursor). The middle section, titled 'Home', contains two panels. The first panel, titled 'Applications', has a 'Register' button and a message: 'You are not authorized in any application.' with a 'View All' link at the bottom. The second panel, titled 'Organizations', has a 'Create' button and a message: 'You don't have any organizations.' with a 'View All' link at the bottom.

Figure 39: IdM-Keyrock - Main page

- `IDM_HTTPS_PORT`: Port where IdM-Keyrock will listen if HTTPS is enable.

Once the component is running, the first step is to register users in this authentication component. So, we have to access the IdM-Keyrock (for instance: 'https://IdM-IP:IdM-Port' from a web browser using administrator credentials defined in the deployment process. Figure 38 shows the GUI of IdM-Keyrock.

After accessing as administrator, the registration of users is carried out in the Users management section. Figures 39, 40, 41 and 42 show how to do it.

So, for authentication purposes, the user sends his credentials using the IdM API to obtain the authentication token. The correct values for the IP address and port of the IdM must be defined, for instance 'localhost:443'.

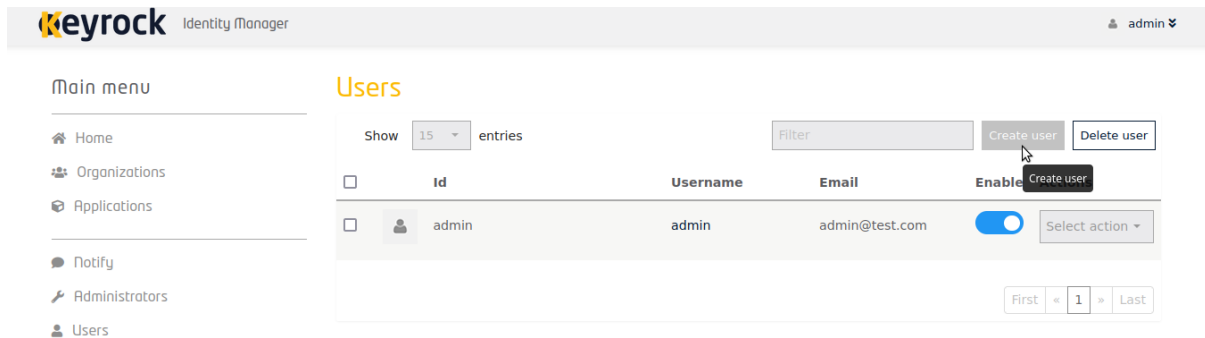


Figure 40: IdM-Keyrock - Link User Management

POST `https://{IdM-IP}:{IdM-Port}/v1/auth/token`

```
{
  "name": "jasanchez@odins.es",
  "password": "1234"
}
```

HEADERS: _____

Content-Type: application/json

If the answer from the IdM is positive, the next body response contains both (201-Created) and the authentication token contained in the 'X-Subject-Token' header response. For instance with the following value '907ad546-3df4-4738-8b56-2a1dec486476'.

RESPONSE: HTTPS/1.1 201 Created

```
{
  "token": {
    "methods": [
      "password"
    ],
    "expires_at": "2021-06-23T12:02:17.676Z"
  },
  "idm_authorization_config": {
    "level": "basic",
    "authzforce": false
  }
}
```

HEADERS: _____

Content-Type: application/json
X-Subject-Token: 907ad546-3df4-4738-8b56-2a1dec486476

10.4.2 XACML framework PAP and PDP

Before launching this component, the configuration to deploy the component must be reviewed. Optionally, this component allows enabling the integration with a traceability component, for instance Blockchain. Towards this end, the following environment variables must be defined:

- **BlockChain_integration:** To enable integration with a traceability component. Admitted values: 0-No integration, 1-With integration
- **BlockChain_configuration:** If the traceability component integration is enabled, this variable allows to define if configuration of the traceability endpoint is defined in a configuration file or in environment variables. Admitted values: 0-Uses configuration from blockchain.conf file, 1-Uses configuration from environment variables.
- **BlockChain_protocol:** Protocol of the traceability component.
- **BlockChain_domain:** Domain included in traceability component.
- **BlockChain_IP:** Name of the host where the traceability component is running.
- **BlockChain_port:** Port where the traceability component is running.
- **BlockChain_get_resource:** GET resource to obtain a domain information included in traceability component.
- **BlockChain_post_resource:** POST resource to register a domain in the traceability component.
- **BlockChain_update_resource:** POST resource to update a domain in the traceability component.

Once the component is running, it is the time of define the access control policies. They are required so that we could properly restrict the access to the operations which can be performed over the Master Controller. So, firstly, using the PAP GUI accessing to its endpoint (for instance: `http://{XACML-IP}:{XACML-Port}/XACML-WebPAP-2`) from a web browser. Figure 43 shows the main page of PAP component.

First, click the “Manage Attributes” button to define the resources, actions and subjects. In the subject’s case, specify the Usernames you defined in IdM-Keyrock. To save click “Save All Attributes” and “Back”. In the resources box, the format of the elements is the next one ‘PEP-Proxy endpoint + Master-Controller’s API resource’. For instance,

we can see ‘https://localhost:1040’ as the PEP-Proxy endpoint and different resources of the Master-Controller component (‘/addthingVisor’, ‘listThingVisors’, etc...). Figure 44 shows the management of attributes.

Finally, click the “Manage Policies” button to define the policies. Here, attributes defined previously are showed. On this page, define Policies and, into them, rules. Each rule can link resources, actions and subjects and establish if this combination has a “Permit” or “Deny” verdict. Figure 45 shows the management of policies.

Regarding XACML-PDP configuration there is nothing to do with this component, anyway, launching this request to know if it is running, must obtain a 200 - OK status code response. The next box shows the format of the request, only it is needed to replace ‘XACML-IP’ and ‘XACML-Port’ with the correct values, for instance ‘localhost:8080’.

GET http://{XACML-IP}:{XACML-Port}/XACMLServletPDP

(empty request body)

HEADERS: _____

empty request headers

RESPONSE: HTTP/1.1 200 OK

(Empty response body)

10.4.3 Capability Manager

Before launching this component, the configuration to deploy the component must be reviewed. If must configure endpoints of IdM-Keyrock and PDP. Optionally, this component allows, too, enable the integration with a traceability component, for instance Blockchain. The next environment variables must be defined:

- keyrock_protocol: Protocol of the IdM-Keyrock.
- keyrock_host: Name of the host where is running the IdM-Keyrock component.
- keyrock_port: Port where the IdM-Keyrock component is running.
- keyrock_admin_email: Email of admin default user in IdM-Keyrock.
- keyrock_admin_pass: Password of admin default user in IdM-Keyrock.

- `blockchain_usevalidation`: To enable integration with a traceability component. Admitted values: 0-No integration, 1-With integration.
- `blockchain_protocol`: Protocol of the traceability component.
- `blockchain_host`: Name of the host where is running the traceability component.
- `blockchain_port`: Port where the traceability component is running.
- `PDP_URL`: URL offered by PDP to obtain verdict.

Additionally, create certificates to support HTTPS connections with validation, considering them in the deployment process chosen.

Once the component is running, launching the following request, in order to know if it's enabled, must result in a 200 - OK status code response. The box shows the format of the request: it is needed to replace 'CapMan-IP' and 'CapMan-Port' with the correct values, for instance 'localhost:3040'.

GET https://{CapMan-IP}:{CapMan-Port}/

(empty request body)

HEADERS: _____

empty request headers

RESPONSE: HTTPS/1.1 200 OK

(Empty response body)

Following the current example, one can send the following request:

POST https://localhost:3040

```
{
  "token": "907ad546-3df4-4738-8b56-2a1dec486476",
  "ac": "GET",
  "de": "https://localhost:1040",
  "re": "/listThingVisors"
}
```

HEADERS: _____

Content-Type: application/json

Obtaining 200-OK response with the following Capability Token:

RESPONSE: HTTPS/1.1 200 OK

```
{"id": "va6t2r5qet9p85tt3sf73ihcje", "ii": 1624449195,
"is": "capabilitymanager@odins.es", "su": "jasanchez@odins.es",
"de": "https://localhost:1040",
"si": "MEYCIQC7vXKpBaLd3N0jw5Sn1BLvVDGtLfeZQn0db3Ub9ZSInQIhAPY7CTDNp
ZVf8kL00U7tRGEuFjXNKsxpWLvCs1NG4m0+",
"ar": [{"ac": "GET", "re": "/listThingVisors"}],
"nb": 1624450195, "na": 1624460195}
```

10.4.4 PEP-Proxy

Before launching this component, the configuration to deploy the component must be reviewed. If needed, do configure endpoints of IdM-Keyrock and PDP. Optionally, this component allows, too, to enable the integration with a traceability component, for instance Blockchain. The following environment variables must be defined:

- target_protocol: Protocol of the target component endpoints (Master-Controller).
- target_host: Name of the host where is running the target component (Master-Controller).
- target_port: Port where the target component is running (Master-Controller).
- target_API: Type of API of target component (Master-Controller). In our case "Fed4IoTMC".
- blockchain_usevalidation: To enable integration with a traceability component. Admitted values: 0-No integration, 1-With integration.
- blockchain_protocol: Protocol of the traceability component.
- blockchain_host: Name of the host where is running the traceability component.
- blockchain_port: Port where the traceability component is running.
- PEP_ENDPOINT: PEP-Proxy Public address in format `https://<PEP-IP>:<PEP-PORT>`.
- fed4iotmc_protocol: Protocol of the Master-Controller component endpoints (Master-Controller).

- `fed4iotmc_host`: Name of the host where is running the Master-Controller component.
- `fed4iotmc_port`: Port where the Master-Controller component is running.
- `fed4iotmc_authz_testpath`: : Test path of the Master-Controller component API, in order to test if PEP-Proxy has access to Master-Controller API endpoints. For instance: `"/listFlavours"`.
- `fed4iotmc_login_path`: Login path of Master-Controller API. In or case: `"/login"`.
- `fed4iotmc_login_userID`: User of Master-Controller component. It is required an user that has access to all endpoints that are offered to be user through PEP-Proxy.
- `fed4iotmc_login_password`: Password of user of Master-Controller component.

Additionally, create certificates to support HTTPS connections with validation, considering them in the deployment process chosen.

Once the component is running, launching this request in order to know if it's enabled, must result in a 200 - OK status code response. The following box shows the format of the request: it is needed to replace 'PEP-Proxy-IP' and 'PEP-Proxy-Port' with the correct values, for instance 'localhost:1040'.

GET https://{PEP-Proxy-IP}:{PEP-Proxy-Port}/

(empty request body)

HEADERS: _____

empty request headers

RESPONSE: HTTPS/1.1 200 OK

(Empty response body)

Following the current example, one can send the following request:

GET https://localhost:1040/listThingVisors

(empty request body)

HEADERS: _____

Accept: application/json
x-auth-token: {"id":"va6t2r5qet9p85tt3sf73ihcje","ii":1624449195,
"is":"capabilitymanager@odins.es","su":"jasanchez@odins.es",
"de":"https://localhost:1040",
"si":"MEYCIQC7vXKpBaLd3N0jw5Sn1BLvVDGtLfeZQn0db3Ub9ZSInQIhAP
Y7CTDNpZVf8kL00U7tRGEuFjXNKsxpWlvCs1NG4m0+",
"ar":[{"ac":"GET","re":"/listThingVisors"}]},
"nb": 1624450195, "na": 1624460195}

Obtaining the response from Master-Controller's API through PEP-Proxy.

Create user

Username

Juan Andres

Email

jasanchez@odins.es

Password

.....

Password (again)

.....

Description

Master-Controller user

Website

☐ Send an email to the user

☒ Enable

Create

Figure 41: IdM-Keyrock - User registration form



Keyrock Identity Manager admin ▼

Main menu

- Home
- Organizations
- Applications
- Notify
- Administrators
- Users

Users

Show 15 entries Filter Create user Delete user

<input type="checkbox"/>	Id	Username	Email	Enable	Actions
<input type="checkbox"/>	 admin	admin	admin@test.com	<input checked="" type="checkbox"/>	Select action ▼
<input type="checkbox"/>	 d2ff3828-351a-42df-be24-da0402219b17	Juan Andres	jasanchez@odins.es	<input checked="" type="checkbox"/>	Select action ▼

First < 1 > Last

Figure 42: IdM-Keyrock - List of registered users

Policy Administration Point

Administration

Manage Policies

Manage Attributes

Configuration

PAP Configuration

Exit

Figure 43: PAP - Main page

Policy Administration Point

Attributes Management		
Resources	Action	Subjects
<pre>https://localhost:1040/addThingVisor <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id https://localhost:1040/listThingVisors <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id https://localhost:1040/deleteThingVisor <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id https://localhost:1040/addFlavour <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id https://localhost:1040/listFlavours <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id https://localhost:1040/deleteFlavour <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id https://localhost:1040/siloCreate <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id https://localhost:1040/listVirtualSilos <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id https://localhost:1040/siloDestroy <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id https://localhost:1040/addVThing <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id https://localhost:1040/deleteVThing <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id</pre>	<pre>GET <()> urn:oasis:names:tc:xacml:1.0:action:action-id POST <()> urn:oasis:names:tc:xacml:1.0:action:action-id PUT <()> urn:oasis:names:tc:xacml:1.0:action:action-id DELETE <()> urn:oasis:names:tc:xacml:1.0:action:action-id PATCH <()> urn:oasis:names:tc:xacml:1.0:action:action-id</pre>	<pre>jasanchez@odins.es <()> urn:ietf:params:scim:schemas:core:2.0:email</pre>
<input type="button" value="New Resource"/> <input type="button" value="Delete Resource"/>	<input type="button" value="New Action"/> <input type="button" value="Delete Action"/>	<input type="button" value="New Subject"/> <input type="button" value="Delete Subject"/> <input type="button" value="Save All Attributes"/>

Figure 44: PAP - Attributes

Policy Administration Point

Policies Management			
Policies	Resources	Subjects	Actions
DemoPolicy <input type="button" value="New"/> <input type="button" value="Del"/> <input type="button" value="Rename"/>	<pre>https://localhost:1040/deleteVThing <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id https://localhost:1040/addThingVisor <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id <input checked="" type="checkbox"/> https://localhost:1040/listThingVisors <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id <input type="checkbox"/> https://localhost:1040/deleteFlavour <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id https://localhost:1040/siloCreate <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id https://localhost:1040/siloDestroy <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id <input type="checkbox"/> https://localhost:1040/deleteThingVisor <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id <input type="checkbox"/> https://localhost:1040/addFlavour <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id <input checked="" type="checkbox"/> https://localhost:1040/listVirtualSilos <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id <input checked="" type="checkbox"/> https://localhost:1040/listFlavours <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id <input type="checkbox"/> https://localhost:1040/addVThing <()> urn:oasis:names:tc:xacml:1.0:resource:resource-id</pre> <div style="text-align: right;"><input type="checkbox"/> All</div>	<pre><input checked="" type="checkbox"/> jasanchez@odins.es <()> urn:ietf:params:scim:schemas:core:2.0:email</pre> <div style="text-align: right;"><input type="checkbox"/> All</div>	<pre><input type="checkbox"/> PUT <()> urn:oasis:names:tc:xacml:1.0:action:action-id <input type="checkbox"/> POST <()> urn:oasis:names:tc:xacml:1.0:action:action-id <input type="checkbox"/> PATCH <()> urn:oasis:names:tc:xacml:1.0:action:action-id <input type="checkbox"/> DELETE <()> urn:oasis:names:tc:xacml:1.0:action:action-id <input checked="" type="checkbox"/> GET <()> urn:oasis:names:tc:xacml:1.0:action:action-id</pre> <div style="text-align: right;"><input type="checkbox"/> All</div>
Rules RuleAdmin RuleUser <input type="button" value="New"/> <input type="button" value="Del"/> <input type="button" value="Apply"/>	<div> Rule CA urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:first-applicable <div style="float: right;">Rule Permit</div> </div>		

Figure 45: PAP - Policies

11 Conclusion

This deliverable reflects the work carried out during Task 5.2: "Pilot Integration", which we started to describe in Deliverable D5.2 Pilot Integration - First Release, in its first phase as the name reads, and which has now been completed in this second release.

This document extends, therefore, the previous deliverable's content, providing a more complete vision of the VirIoT cross-border platform, with more details related to its deployment. We explain in greater detail the work carried out during this task, regarding pilot integration in the platform, now updating previous information, as well as including new sections that better explain how to integrate platform components and pilot components.

We take advantage of this deliverable to explain how we have designed a generic module, offering a bulk of common code, API and functions in python, to build custom ThingVisors quickly and easily, fostering adoption of the platform through the github repository. Another important contribution of this deliverable, connected with the above, is the FaceRecognition ThingVisor section where a new ThingVisor with actuation capabilities for face recognition has been presented, based on said modular approach.

We also show how VirIoT can be integrated with FogFlow as a ThingVisor factory for a dynamic orchestration of edge and cloud processing tasks.

Finally, a very detailed explanation of our access control framework, that can leverage both a token-based approach (for actuators) and a powerful, finer-grained DCapBAC technology, concludes this document: we show how to practically instantiate it in our testbed and the various interactions among all relevant platform components.

Carpool pilot	
Entity Name	Entity Type
Site	Site
SmartCamera	Device
Vehicle	Vehicle
EntryEvent	EntryEvent
ExitEvent	ExitEvent

Table 5: Carpool use case entities

The camera is sending in and out events which are stored as, respectively EntryEvent and ExitEvent entities. The offline algorithm that detects the passing of a car updates their refExit and refEntry relationships (respectively).

An example of such a pair of EntryEvent and ExitEvent is shown in the following listing:

Listing 1: Example of the Vehicle Stand Entity

```

1 {
2   "id": "urn:ngsi-ld:EntryEvent:01",
3   "type": "EntryEvent",
4   "refVehicle" : "urn:ngsi-ld:Vehicle:d2738e13ec69c8662818f943653af98a",
5   "refExit" : "urn:ngsi-ld:ExitEvent:05"
6 }
7
8 {
9   "id": "urn:ngsi-ld:ExitEvent:05",
10  "type": "ExitEvent",
11  "refVehicle" : "urn:ngsi-ld:Vehicle:d2738e13ec69c8662818f943653af98a",
12  "refEntry" : "urn:ngsi-ld:EntryEvent:01"
13 }
```

Using on the Registration Plate number of a Vehicle, it is possible to query national data bases for more details about the Vehicle. These details are stored as properties of the entity which follows the model of Vehicle in <https://schema.org/Vehicle>. An example of such an entity is shown in the following listing:

Listing 2: Example of the Vehicle Stand Entity

```

1 {
2   "id": "urn:ngsi-ld:Vehicle:d2738e13ec69c8662818f943653af98a",
3   "type": "Vehicle",
```

```
4    "fuelType": {
5        "type": "Property",
6        "value": "DIESEL"
7    },
8    "brand": {
9        "type": "Property",
10       "value": "ALFA ROMEO"
11    },
12    "model": {
13        "type": "Property",
14        "value": "MITO"
15    },
16    "vehicleIdentifier": {
17        "type": "Property",
18        "value": "8608f81f137c38655abbd6c9cf7b54f39fa5a00f05
19              aafb7a7c2c791f3191d219"
20    },
21    "carrosserie": {
22        "type": "Property",
23        "value": "CI"
24    },
25    "co2": {
26        "type": "Property",
27        "value": "90"
28    },
29    "pfisc": {
30        "type": "Property",
31        "value": "4"
32    },
33    "vtype": {
34        "type": "Property",
35        "value": "VP"
36    },
37    "@context": [
38        "https://raw.githubusercontent.com/easy-global-market/
39          ngsild-api-data-models/master/smartCameraUseCase/
40          jsonld-contexts/smartCamera-context.jsonld",
41        "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.
42          jsonld"
43    ]
44 }
```


References

- [1] Japan Tourism Agency: White Paper on Tourism in Japan, 2019. [Online]. Available: <https://www.mlit.go.jp/kankocho/en/siryou/content/001312296.pdf>