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 D2.3
System Architecture - Second Release

Deliverable Type:	Report
Deliverable Number:	D2.3
Contractual Date of Delivery to the EU:	30/06/2020
Actual Date of Delivery to the EU:	30/06/2020
Title of Deliverable:	System Architecture - Second Release
Work package contributing to the Deliverable:	WP2
Dissemination Level:	Public
Editor:	Andrea Detti (CNIT), Hidenori Nakazato (WAS)
Author(s):	Andrea Detti, Giuseppe Tropea, Ludovico Funari (CNIT); Juan A. Sanchez, Juan A. Martinez, Antonio F. Skarmeta (OdinS); Martin Bauer, Bin Cheng (NEC); Frank Le Gall (EGM); Hidenori Nakazato, Kenji Kanai (WAS); Kenichi Nakamura (PAN); Tetsuya Yokotani (KIT)
Internal Reviewer(s):	Nicola Blefari Melazzi (CNIT)
Abstract:	The deliverable reports the second release of Fed4IoT's system architecture
Keyword List:	IoT Virtualization, IoT Brokers, Actuators, NGSI-LD, ICN

Disclaimer

Table of Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Definition
ADN	Application Dedicated Node
AE	Application Entity
AIMD	Additive Increase/Multiplicative Decrease
API	Application Programming Interface
ASM	Adaptive Semantic Module
ASN	Application Service Node
AWS	Amazon Web Services
CIM	Context Information Management
CSE	Common Services Entity
DCapBAC	Distributed Capability-Based Access Control
ETSI	European Telecommunications Standards Institute
FIB	Forwarding Information Base
GE	Generic Enabler
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
JWT	JSON Web Token
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
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 trough 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	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 Glossary

1 Introduction

1.1 Deliverable Rationale

This deliverable reports the second version of the Fed4IoT system architecture. It is centred on our concept of an IoT Virtualizaiton Platform able to interoperate with different IoT standards and to offer IoT systems-as-a-service.

1.2 Quality review

The internal Reviewer responsible of this deliverable is Nicola Blefari Melazzi (CNIT).

Version Control Table			
V.	**Purpose/Changes**	**Authors**	**Date**
0.1	Table of content	Andrea Detti (CNIT), Hidenori Nakazato (WAS)	31/05/2020
0.2	Draft Version	Andrea Detti, Giuseppe Tropea (CNIT), Juan A. Sanchez, Juan A. Martinez, Antonio F. Skarmeta (OdinS), Hidenori Nakazato (WAS), Kenichi Nakamura (PAN), Tetsuya Yokotani (KIT)	23/06/2020
1.0	Quality review	Nicola Blefari Melazzi (CNIT)	28/06/2020
1.1	Final review	Andrea Detti (CNIT)	30/06/2020

Table 3: Version Control Table

1.3 Executive summary

1.3.1 Deliverable description

This deliverable describes the second version of the Fed4IoT system architecture. In section 2, we report some background information concerning IoT services offered by cloud providers, on IoT open-source brokers (with a focus to oneM2M, FiWARE and NGSI-LD solutions), and finally we report on related projects. In section 3, we introduce the concept of IoT virtualization that drives the project. In section 4 we describe the components of the architecture, their roles and relationships, in detail, including also security aspects and comparison with related works. In section 5, we describe two possible frameworks for the development of a fundamental system component, the ThingVisor, which actually carries out the "thing virtualization" process. This deliverable is not intended to cover all of the architectural details, but to provide a high-level view of the architecture and its potential. Fig. 1 relates this deliverable to other deliverables in this project that

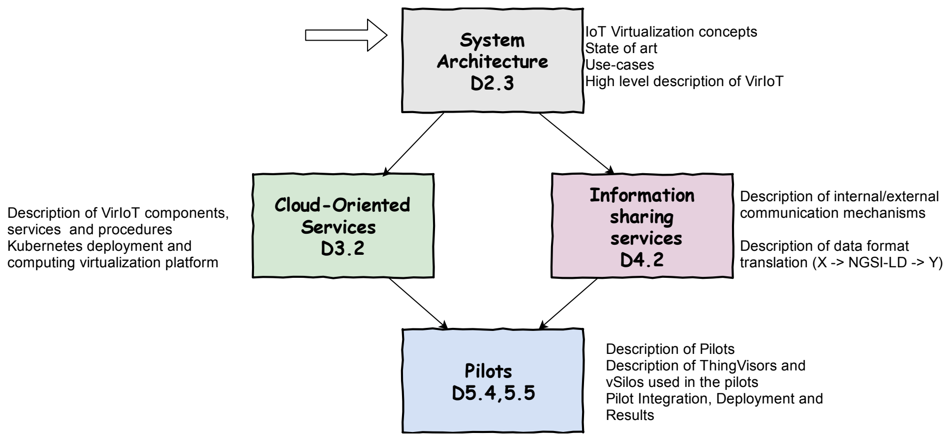

System Architecture D2.3
IoT Virtualization concepts
State of art
Use-cases
High level description of VirIoT
Cloud-Oriented Services D3.2
Description of VirIoT components, services and procedures
Kubernetes deployment and computing virtualization platform
Information sharing services D4.2
Description of internal/external communication mechanisms
Description of data format translation (X -> NGSI-LD -> Y)
Pilots D5.4,5.5
Description of Pilots
Description of ThingVisors and vSilos used in the pilots
Pilot Integration, Deployment and Results

Figure 1: Relationship among deliverable

have a relationship with architecture. Specifically, in D3.2 we provide details about the architecture components, procedures, messages, APIs, deployment, virtualization, etc. In D4.2 we provide details on the communication mechanisms and on the mapping between the various IoT standards we used within ThingVisors and Virtual Silos. Finally in D5.4 and D5.5 we describe how VirIoT was used in the pilot projects.

1.3.2 Summary of results

This deliverable presents the second version of the Fed4IoT system architecture, named VirIoT , which enables virtualization of IoT systems, and then enables information coming from those Virtual Things to be exposed by IoT Brokers, using several different standards and data models, including oneM2M, NGSIv2 and NGSI-LD.

Our goal is to decouple developers of IoT applications from providers of IoT infrastructures formed by physical things, sensors, actuators, data sources. VirIoT allows owners of IoT heterogeneous infrastructures to share them with many IoT application developers, which can simply rent the Virtual Things and the IoT Brokers their applications need. VirIoT can be useful for small stakeholders whose applications require large-scale IoT infrastructures, who are otherwise unable to handle the infrastructure deployment or ownership. VirIoT can also be useful for owners of private IoT infrastructures, in order to create isolated development environments where to run experimental services, before final deployment in the production system.

1.4 Major updates from version 1

- Revision and extended description of the concept of IoT Virtualization and use cases.
- Revision of the description of the architecture.

- Support of Actuators. Second version of the architecture supports the virtualization of actuators. The background sections has been consequently updated too.
- System vSilo. Second version of the architecture provides the federation with external NGSI-LD system through the System vSilo.
- Security aspects. Security aspects have been included in the second version of the architecture.
- ThingVisor Factories have been updated. A brand new VirIoT ThingVisor factory has been introduced.
- Removal of section describing NGSI-LD $\rightarrow$ X mapping used by vSilos because moved to D4.2 where X $\rightarrow$ NGSI-LD mapping used by ThingVisors will be described. This will lead to having a single deliverable that deals with interoperability between standards.

2 Background

2.1 IoT Cloud Services

Electronic industrial control devices deployed for tasks such as control of equipment for electricity generation in power plants, control of trains or automobiles, have been in use for years. Now that these devices are Internet-capable, an IoT revolution is starting to emerge, just like the Internet revolution emerged when computers in use for years, but in isolation, were interconnected thanks to the network.

IoT applications require sophisticated coordination across connected objects, multiple clouds and networks, and the mobile front-ends. This is a complex endeavor, and developers do not want to do it from scratch. Hence cloud services for IoT are quickly emerging to facilitate IoT development, supported by providers that range from hardware vendors (Intel IoT platform, Bosch IoT Cloud) to system integrators (IBM Watson IoT) to the known ICT giants (Google Cloud IoT, AWS IoT, Microsoft Azure IoT).

All the above IoT cloud services operate on similar architectures. There is usually a Things layer, a (more or less explicit) Edge layer, a Cloud Layer and a Data layer. For instance, Microsoft Azure IoT has Things that generate data, Insights based on data generated, and Actions based on insights. Amazon AWS IoT has Device Software (both an OS for microcontrollers and the Greengrass Core to run on more powerful edge devices) and Control Services (Things Graphs, Analytics, Management) on the cloud. The Google Cloud IoT distinguishes between a Cloud IoT Edge plane, a Data Analytics in the cloud and a Data Usage plane.

The basic idea is to invite the user to bring her own set of sensors and actuators to their architecture, and they offer many functions on top, ranging from analytics to simplified device integration, from automated dashboards to improved security, from the scalability of billions of sensors/messages to flexible deployments. Accordingly, specific SDKs are provided to support application development.

2.1.1 Exemplary Case Study with AWS IoT

For instance, connecting a RaspberryPi-based device to the AWS IoT cloud is a matter of generating a pair of security keys through the graphical console, then registering the device in the same console and deploying the C SDK on the Raspberry, which then securely connects via MQTT to the AWS cloud. A Thing Shadow (the cloud counterpart of the device) is then available for UPDATE, GET or DELETE methods, via both MQTT or RESTful APIs.

It is interesting to study how, in the above scenario, the AWS IoT ecosystem (see figure 2) leverages this edge node to distribute computation. In our example, the RaspberryPi acts as edge, and by allowing the cloud ssh access to it, we are able to automate installing AWS IoT Greengrass on it, so as to seamlessly extend AWS to edge devices so they can act locally on the data they generate, while still using the cloud for management, analytics, and durable storage. The RaspberryPi is then able to run AWS Lambda functions, which we program and deploy through the unified dashboard or AWS CLI, and exploit it, for example, to keep device data in sync when going on/off line.

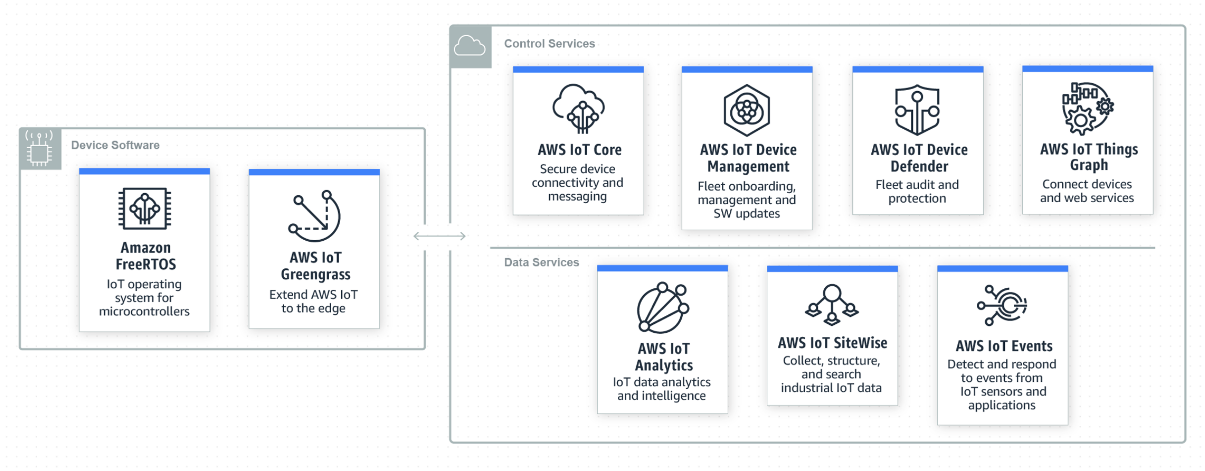

Device Software
Amazon FreeRTOS
IoT operating system for microcontrollers
AWS IoT Greengrass
Extend AWS IoT to the edge
Control Services
AWS IoT Core
Secure device connectivity and messaging
AWS IoT Device Management
Fleet onboarding, management and SW updates
AWS IoT Device Defender
Fleet audit and protection
AWS IoT Things Graph
Connect devices and web services
Data Services
AWS IoT Analytics
IoT data analytics and intelligence
AWS IoT SiteWise
Collect, structure, and search industrial IoT data
AWS IoT Events
Detect and respond to events from IoT sensors and applications

Figure 2: Amazon's AWS IoT ecosystem

Though all of the IoT cloud providers have similar levels of functionality and enterprise reliability, some peculiarities are worth noticing. For instance, AWS offers a customized version of FreeRTOS for incorporating low-power devices such as small microcontrollers within the AWS IoT ecosystem. Google IoT, on the other hand, has a major focus on machine learning and makes possible running TensorFlowLite over Linux and AndroidThings based edge devices/gateways.

2.1.2 Differences with Fed4IoT VirIoT platform

While the platforms mentioned above mainly offer cloud services to IoT devices of customers, greatly extending their potential, the Fed4IoT VirIoT system is instead focused on offering *things as-a-service*, by acquiring (control of) an ever-growing number of devices out there in the field, and by virtualising them to supply a scalable layer of horizontally share-able IoT resources to customers.

Although our architecture can make customers able to develop their applications in tightly isolated environments that may exploit thousands of diverse virtual things as if they were fully dedicated to the application, our focus is on **sharing** and **reusing** the same set of sensors for very diverse applications implemented by different tenants.

Further, the adoption of the flexible and standardized NGSI-LD data model, based on property graphs, as the internal format for all IoT data, allows integrating diverse information about the same object, or about linked objects, coming from data sources owned by heterogeneous and unrelated data sources.

The distinguishing feature of providing more than agnostic transport of IoT data from sensor to application (through edge and cloud), by being capable of data translation instead, and by managing strongly typed data pieces, permits aggregation of information, and novel applications to emerge.

Moreover, the virtual things rented by a customer/tenant can be, in turn, connected to upstream cloud service platforms as if they were real, un-shared, IoT devices speaking their native data formats. In this sense, the VirIoT services are complementary to most of the existing solutions and can interoperate with them in an extended IoT chain.

2.2 IoT Platforms and Brokers

IoT systems are composed of a set of interconnected devices handled by an IoT software platform, such as one of those previously presented. The platform is designed in such a way that it can connect to a vast number of IoT devices [1], and may rely on IoT *Brokers*, which are components exposing IoT information and services of the connected devices through a unified API and data model.

The design and development of the broker functionality are focused on efficiently managing a plethora of IoT use-cases, by employing both request/response and publish/-subscribe messaging patterns and by exposing a public API based on open and standard protocols. An IoT Broker stores information according to a specific data-model and exposes a secure API for publishing, fetching and discovering IoT data items, devices, and the likes. Additionally, by using a distributed approach, many brokers can be interconnected to scale out the system. E.g., an IoT platform can comprise a set of "edge" brokers connected to a core broker. Sensors publish data on edge brokers, while users submit data requests to the core broker, which in turn relay the request to specific edge brokers.

Besides proprietary cloud platforms, two different IoT platforms have gained much interest by both industry and academia, thanks to the endorsement received so far by standardization bodies and industry and their ultimate objective of providing interoperability with third-party systems. These two platforms are oneM2M [2, 3], and FIWARE [4]. Currently, another IoT platform is emerging, thanks to the effort made by the ETSI ISG CIM workgroup, namely NGSI-LD [5]. NGSI-LD is actually an evolution of NGSI specifications [6] used within FIWARE, it is based on JSON-LD (LD stands for Linked Data), which is now more powerful and flexible, allowing users not only to describe context entities but also to define relationships between them.

In what follows we introduce oneM2M and FIWARE, while in section 2.2.3 we present NGSI-LD, whose specification processes are strongly supported by the Fed4IoT project.

2.2.1 oneM2M

The oneM2M platform is a global standard initiative supported by eight ICT standard development organizations spread all over the world [3], six industry fora and more than 200 members.

oneM2M provides functionality for managing IoT devices and their information. The functionality forms the so-called Common Service Layer, where things are represented including their semantics, and API for discovery, data subscription/notification, etc.

The oneM2M architecture consists of *nodes*, which could be Application Dedicated Nodes (ADNs), Application Service Nodes (ASNs), Middle Nodes (MNs), and Infrastructure Node (INs). Each node can comprise three different entities: Network Service

Entity (NSE), Common Service Entity (CSE) and Application Entity (AE). An AE is actually the application specific software that generates or consumes IoT data. A CSE is the entity offering the functionality of the Common Service Layer, e.g. it stores data comings from AEs, supports their discovery and registering, exposes pub/sub API, manages access policies, etc. An NSE is the entity providing data transport. The standard defines a number of different bindings to transport protocols, including HTTP, MQTT, CoAP, and Websockets.

AEs run in ADNs or ASNs and interact with the platform through a Common Service Entity (CSE), running in either in an MN (MN-CSE) or in an IN (IN-CSE). The CSE API supports data publishing, authentication, information discovering and subscriptions, to name a few. A CSE can operate as stand-alone or in a hierarchy formed by a central infrastructure CSE (IN-CSE) and peripheral Middle Nodes CSE (MN-CSEs).

For interaction among AEs and CSE, oneM2M defines three reference points. The Mca reference point for interactions between AEs and CSEs, the Mcc for interactions between different CSEs of the same provider and the Mcc' which is for the interaction between the infrastructure CSEs of different providers.

Within a CSE, oneM2M [3] represents IoT resources in a hierarchy whose main elements are Application Entities (AEs), Containers and Content Instances (the actual data items), as shown in Figure 3. Every IoT device or IoT application is an AE represented by an AE element of the resource tree. The AE element contains Containers that store Content Instances, i.e., the actual IoT data items. For instance, a sensor can be a source of content instances; an actuator can be a consumer of content instances, which represent its status (e.g., on/off); an application logic can fetch Content Instances from different Containers, make some reasoning on top of them and publish a new state information in a Container where the actuator is registered to. Relevant oneM2M resources including AEs, containers and content instances can also be annotated with semantic information about the resources and the contained data.

oneM2M has also faced the problem of representing information from different vendors. For instance, for a light switch we can have a different set of values: "On/Off"; "1/0" or "True/False". In the scope of Home Appliances, they have defined an Information Model (document TS-0023) providing a unified basis for the oneM2M system.

Currently, many CSE implementations exist, including Mobius [7], OpenMTC, Eclipse OM2M, etc. For our purposes, the CSE can be considered as a oneM2M Broker.

2.2.2 FIWARE

FIWARE [8] has been developed as the core platform of the Future Internet PPP funded by the European Commission between 2011 and 2016. During this time a FIWARE open source community and ecosystem has been created, whose coordination has since been taken over by the FIWARE Foundation. FIWARE provides a catalog for sharing open-source platform components, called Generic Enablers (GEs) that are intended to make the development of smart applications easier.

One of the most significant platform components is the Context Broker, the entity

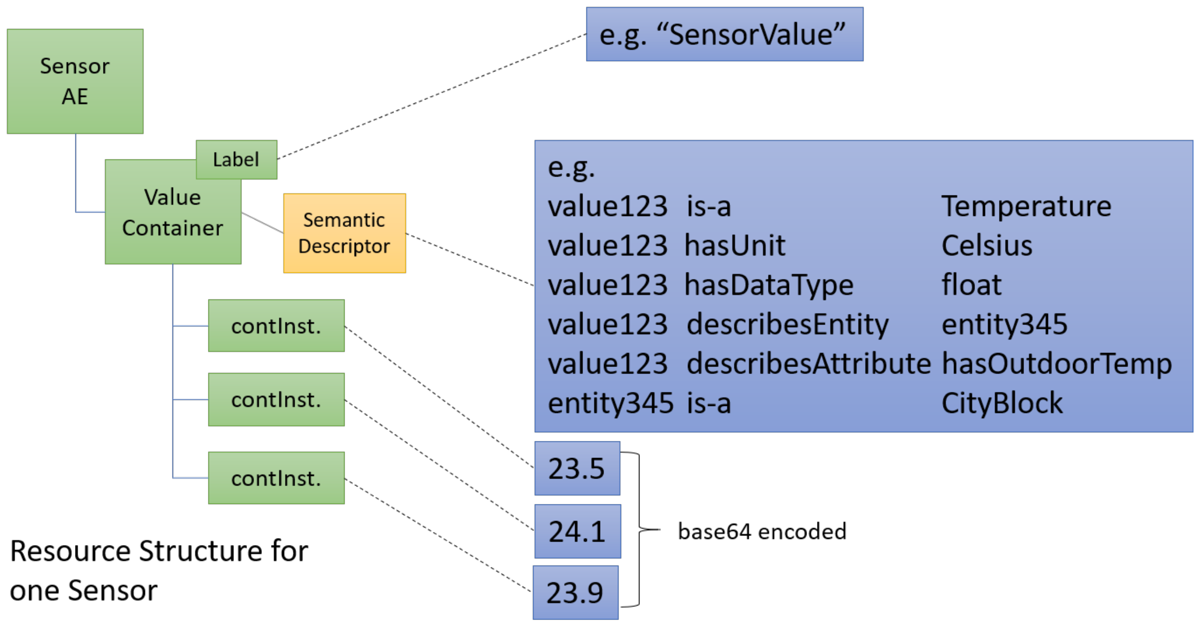


Figure 3: oneM2M resrouce tree

responsible for the distribution of the information. So far, there are two implementations: Orion Context Broker[1] and Aeron IoT Broker[2]. Both implementations provide a publish/subscribe messaging pattern, as well, as a method to query the stored context information. They adopted Next Generation Service Interfaces (NGSI) REST API, a technology standardised at Open Mobile Alliance (OMA) [6, 9]. Additionally, thanks to the work of ETSI ISG CIM workgroup [5], NGSI has evolved into NGSI-LD (based on JSON-LD) allowing for a richer representation of information. Since NGSI-LD is of paramount importance for this project we have dedicated a whole section, Section 2.2.3, to thoroughly describe it. NEC has recently implemented its Scorpio NGSI-LD Broker[3] and made it available as an Incubated FIWARE GE, replacing the Aeron IoT Broker. In parallel, there exists an extension to ORION called ORION-LD[4] that also adopts NGSI-LD.

Focusing on the IoT domain, and the corresponding integration of IoT devices into the platform, FIWARE uses a component called IDAS, which is a backend for device management. This component makes use of IoT Agents for translating the information coming from IoT lightweight protocols, such as MQTT or CoAP, among others, to the NGSI representation. Figure 4 shows the interactions between these components.

To accomplish the NGSI representation of the IoT devices' information in FIWARE's platform, IoT devices must be provisioned using IDAS' API. This provisioning implies, on the one hand, that an entity is generated inside the Context Broker which is representing

[1]https://fiware-orion.readthedocs.io/en/master/
[2]https://github.com/Aeronbroker/Aeron
[3]https://github.com/ScorpioBroker/ScorpioBroker
[4]https://github.com/Fiware/context.Orion-LD

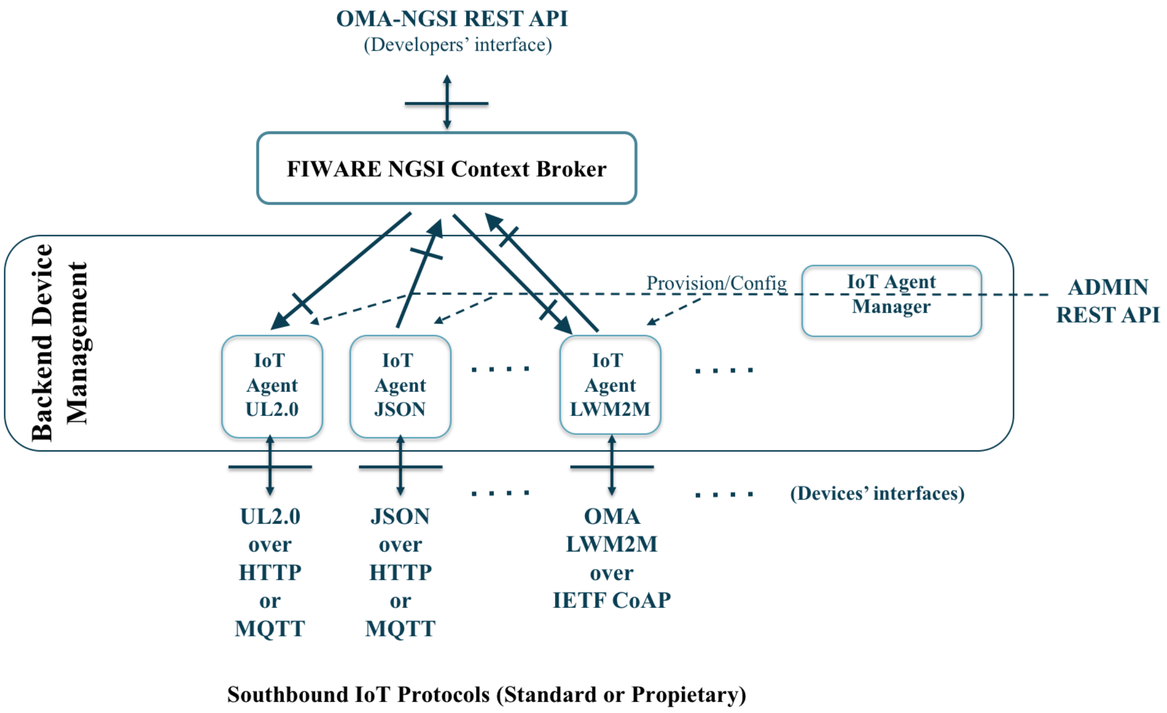


Figure 4: FIWARE-Diagram

the corresponding IoT device. On the other hand, the IoT Agent, which is being used, has been configured so that when receiving the information from IoT lightweight protocols, it knows where/what it must update in the Context Broker.

To provision IoT devices' sensors, it's appropriate to use URNs (for instance borrowing their structure from the novel NGSI-LD specification) when creating identifiers for the corresponding entities, since it's easier to understand meaningful names when defining data attributes. These mappings can be defined in the provisioning device request.

The types of measurement attributes that can be provisioned are:

- `attributes`: they are active readings from the device
- `lazy attributes`: they are only sent on request. The IoT Agent will inform the device to return the measurement
- `static attributes`: they are, as the name suggests, static data about the device (such as relationships) passed on to the Context Broker.

Provisioning IoT devices' actuators, is similar to provisioning a sensor. This time the provisioning request must define an array of commands, which defines the list of each of the commands that can be possibly invoked.

For instance, Figure 5 shows how to provision, using a curl request, an IoT device which contains measurement attributes and supports commands to actuators.

```
curl -iX POST 'http://<IoTAgent-Host>:4041/iot/devices' \
  -H 'Content-Type: application/json' \
  -H 'fiware-service: demo' \
  -H 'fiware-servicepath: /demo' \
  -d '{
    "devices": [
      {
      "device_id": "device001",
      "entity_name": "urn:ngsi-ld:Device:001",
      "entity_type": "Device",
      "protocol": "MQTT_JSON",
      "transport": "MQTT",
      "attributes": [
        {"object_id":"c","name":"count","type":"number"},
        {"object_id":"s","name":"isRunning","type":"boolean"}
      ],
      "commands": [
        {"name":"start","type":"command"},
        {"name":"stop","type":"command"}       ]
      }
    ]
  }'
```

Figure 5: IoT Agent-Provisioning IoT Device

Once the provisioning is done, the Context Broker contains the entity that is the NGSI representation of the IoT device. For instance, Figure 6 shows the entity that will be created as a consequence of the request in Figure 5.

To send a command to the IoT device through a Context Broker, a REST request can be made that will be forwarded directly to the IoT Agent. To do it, /v2/entities/entityID/attrs or /v1/updateContext endpoints can be used. As the result of this request, the command attribute will be updated and it triggers the IoT Agent to send the request to the IoT device. For instance, Figure 7 shows how to send, using a curl request, a stop command to an IoT device through the Context Broker.

```

{
    "id": "urn:ngsi-ld:Device:001",
    "type": "Device",
    "TimeInstant":{"
        type":"ISO8601", "value":" ",
        "metadata":{}},
    "count":{
        "type":"number", "value":" ",
        "metadata":{}},
    "isRunning":{
        "type":"boolean", "value":" ",
        "metadata":{}},
    "start_info":{
        "type":"commandResult", "value":" ",
        "metadata":{}},
    "start_status":{
        "type":"commandStatus", "value":"UNKNOWN",
        "metadata":{}},
    "stop_info":{"
        type":"commandResult", "value":" ",
        "metadata":{}},
    "stop_status":{
        "type":"commandStatus", "value":"UNKNOWN",
        "metadata":{}},
    "start":{
        "type":"command", "value":"",
        "metadata":{}},
    "stop":{
        "type":"command", "value":"",
        "metadata":{}}
}
```

Figure 6: NGSI representation of IoT device

```

curl -iX PATCH \
  'http://<ContextBroker-Host>:1026/v2/entities/urn:ngsi-ld:
     Device:001/attrs' \
  -H 'Content-Type: application/json' \
  -H 'fiware-service: demo' \
  -H 'fiware-servicepath: /demo' \
  -d '{
  "stop": {
      "type" : "command",
      "value" : ""
  }
}'
```

Figure 7: NGSI request - sending a command to IoT device

2.2.3 NGSI-LD

NGSI-LD [5], already briefly introduced in the context of FIWARE in Section 2.2.2, is an information model and an API that is being standardized by the ETSI Industry Specification Group on cross-cutting Context Information Management (ETSI ISG CIM) and to which Fed4IoT is actively contributing. This subsection is structured as follows. First the evolution from NGSI to NGSI-LD is described with an illustration of the JSON serialization of an entity. Then the NGSI-LD Information Model is introduced. This is followed by an overview of the NGSI-LD API, including the logical architecture.

2.2.3.1 NGSI-LD evolves NGSI

Figure 8 presents an example of a *"vehicle"* context entity represented using NGSI, encoded in plain JSON. As we can see such entity comprises an identifier (id), a type and its attributes (brandName, isParked, location, speed).

```
{
    "id":"Vehicle:A100",
    "type":"Vehicle",
    "brandName":{
        "type":"string",
        "value":"Mercedes",
        "metadata":{}
    },
    "isParked":{
        "type":"boolean",
        "value":true,
        "metadata":{}
    },
    "location":{
        "type":"coords",
        "value":"41.2,-8.5",
        "metadata":{}
    },
    "speed":{
        "type":"number",
        "value":"80",
        "metadata":{}
    }
}
```

Figure 8: FIWARE-NGSI representation of a vehicle parked at a location

We also present an extended NGSI-LD description of the same entity in Figure 9, which is encoded in JSON-LD. This way we can compare the two representation formats.

It includes the explicit GeoJSON encoding of the vehicle's location, and the explicit representation of relationships. The explicit representation of relationships allows representing a property graph structure, where the entities are the nodes and the relationships the edges, which can be followed between entities. This enables the discovery of related entities and thus provides a richer context for a given entity.

```json
{
  "id":"urn:ngsi-ld:Vehicle:A100",
  "type":"Vehicle",
  "brandName":{
    "type":"Property",
    "value":"Mercedes"
  },
  "isParkedAt":{
    "type":"Relationship",
    "object":"urn:ngsi-ld:OffStreetParking:Downtown1",
    "observedAt":"2017-07-29T12:00:04Z",
    "providedBy":{
      "type":"Relationship",
      "object":"urn:ngsi-ld:Person:Bob"
    }
  },
  "speed":{
    "type":"Property",
    "value":80
  },
  "createdAt":"2017-07-29T12:00:04Z",
  "location":{
    "type":"GeoProperty",
    "value":{
        "type":"Point",
        "coordinates":[-8.5,41.2]}
  }
}
```

Figure 9: FIWARE-NGSI-LD representation of a vehicle parked at a location

2.2.3.2 NGSI-LD Information Model

The NGSI-LD information model is a meta model whose main concepts are *entities*, *properties* and *relationships*. The assumption is that the world consists of entities, which can be physical entities like a car or a building, but also more abstract entities like a company or the coverage area of a WLAN's access points. Entity instances have a URI as an identifier and a type, e.g. a car with identifier `urn:ngsi-ld:Vehicle:A4567` and of type `Vehicle`. Entities have properties, e.g. a `location` or a `speed`, as well as relationships to other entities, e.g. `isOwnedBy` or `isParkedAt`. Furthermore, properties and relationships can be annotated by properties and relationships themselves; e.g. a timestamp, the provenance of the information or the quality of the information, can be provided by nested properties/relationships. The underlying model thus represents a *Property Graph* and the respective concepts and relations are visualized in the UML diagram in Figure 10.

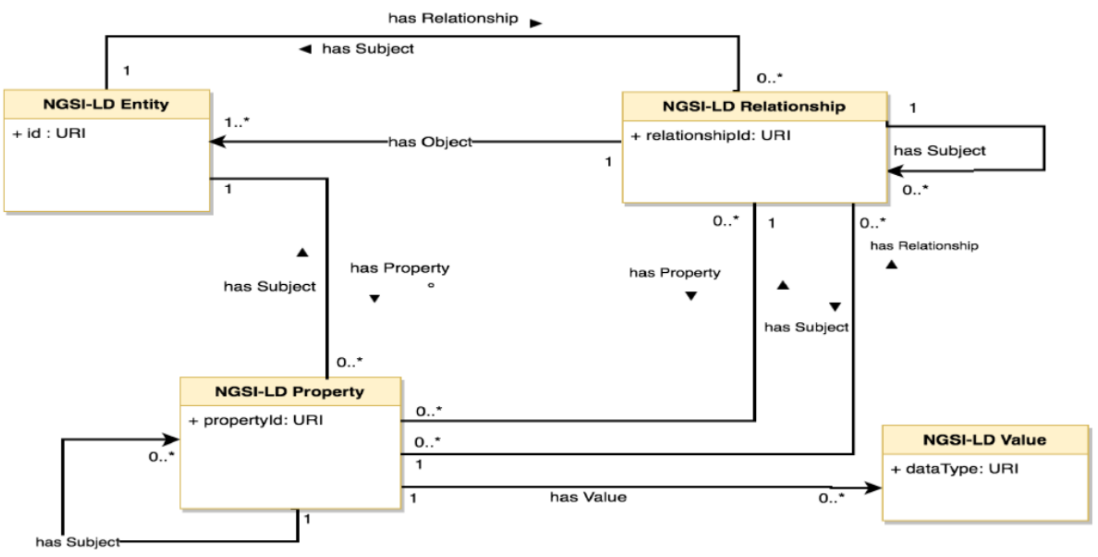


Figure 10: NGSI-LD concepts and relations

Figure 11 shows an example of an NGSI-LD property graph. The scenario is that, in an accident, a car hit a lamppost to which a camera had been attached. Thus there are three entities, which are of types `Vehicle`, `StreetFurniture` and `Sensor` respectively. Each of these entities has at least one property, e.g. the `Vehicle` with identifier `urn:ngsi-ld:Vehicle:A4567` has the property `brandname`, which has the value `"Mercedes"`. It also has the relationship `inAccident` whose target is the `StreetFurniture` with identifier `urn:ngsi-ld:SmartLamppost:Downtown1`. The relationship has one property `observedAt`, which is a timestamp with the time of the accident, and a relationship `providedBy` which relates to the police officer who entered the information.

The idea is that different elements of the graph can come from different data sources, but they can be easily combined as they relate to the same entity, i.e. in this case the police database could refer to the lamppost and the city database could contain the information about the attached camera, which might have been damaged as the result of the accident.

NGSI-LD is represented in JSON-LD and thus can have a grounding in RDF as JSON-LD is one serialization format for RDF. The NGSI-LD core model defining this grounding is shown in Figure 12.

We can see that NGSI-LD `Properties` and `Relationships` are subclasses of `rdfs:Resource`.

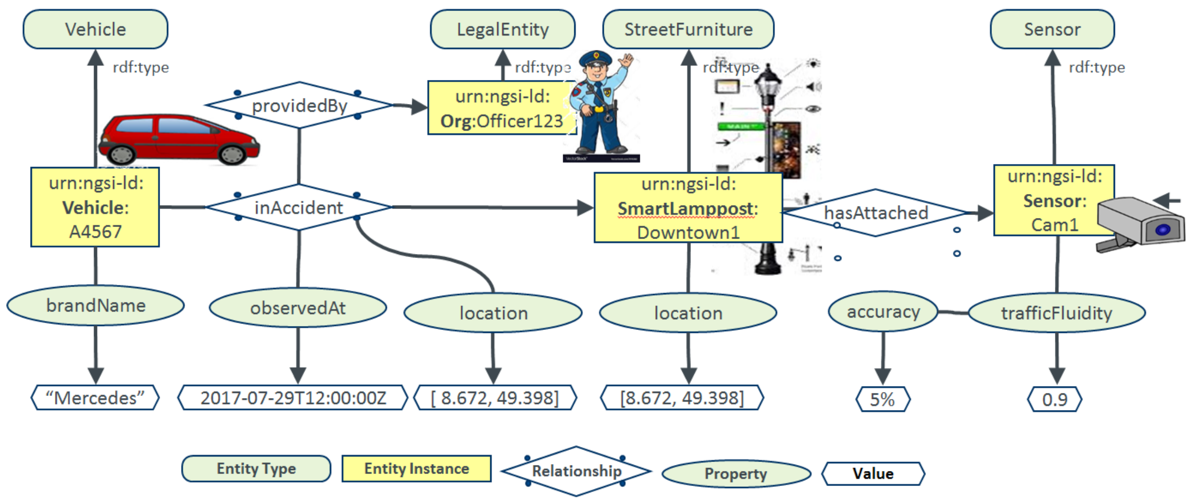


Figure 11: Example of an NGSI-LD property graph, *source: ETSI ISG CIM*

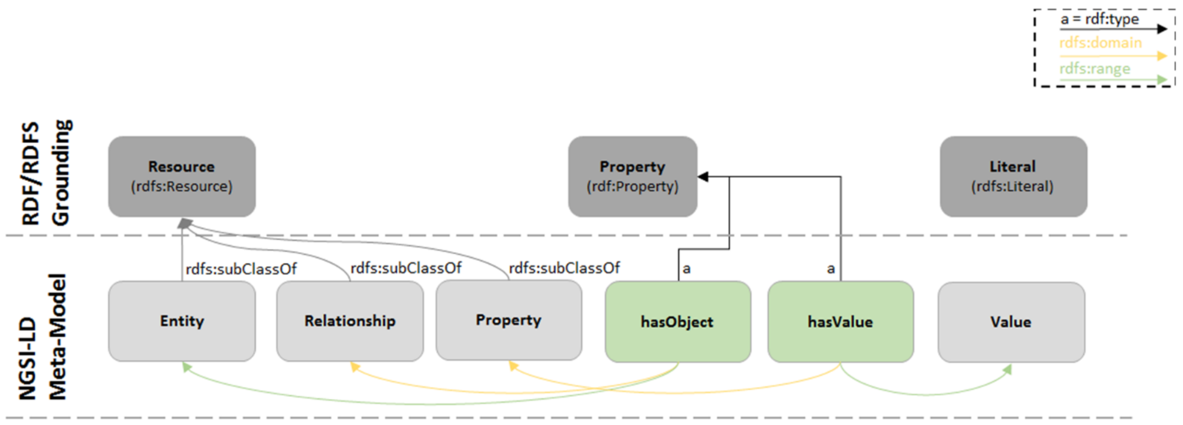


Figure 12: NGSI-LD Core Meta Model

The reason is that referring them to `rdf:Property`, instead, would be limiting to our purposes, because `rdf:Property` cannot be annotated as an instance with additional information, which is what we need for creating properties of properties, as we can see in the graph of figure 11. Conceptually this means that a statement about another statement is made, and this is not directly possible in RDF.

The way to deal with it is to use *rei-fication*, from Latin "making something a thing", and there are different ways to do it in RDF. We have chosen the so-called blank node reification. An example is shown in Figure 13.

The RDF property does not directly have the object as its target, but it points to a blank node instead, which has the Relationship as its type. The blank node then has a special rdf property `hasObject`, but can also have additional properties and relationships, such as the `observedAt` property in the example. All NGSI-LD Relationships have the `hasObject` property pointing to the entity that is the target of the NGSI-LD Relationship,

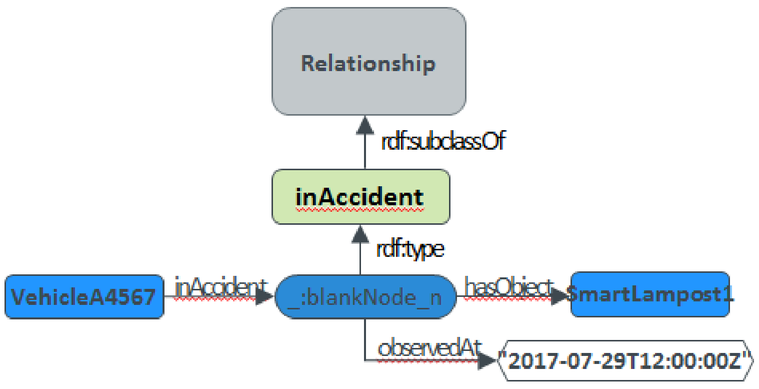


Figure 13: Example of blank node reification in NGSI-LD

and all NGSI-LD Properties have the `hasValue` property that point to the value of the NGSI-LD Property.

The advantage of the blank node reification is that in a JSON-LD serialization, the blank node is not explicitly visible, but is hidden in the JSON structure. An example of an NGSI-LD serialization is shown in Figure 14.

An important element in JSON-LD is the `@context`. It defines a mapping between the simple string terms used in the serialization for identifying entity types, properties and relationships, to specific concepts uniquely identified by a URI. While the NGSI-LD core terms are all defined in `http://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld`, the domain specific terms of the example like Vehicle and brandName have been directly defined in the custom provided `@context`. These URIs can be defined in an ontology. Thus a semantic grounding of the information is achieved and the information can be processed with semantic tools and be combined with other semantic information.

2.2.3.3 NGSI-LD Cross-Domain Ontology

In addition to the meta model, NGSI-LD defines some mandatory concepts that are used as part of the API. These are depicted in Figure 15.

In particular, some geographic properties represented in GeoJSON are defined, which are used in the API to specify geographic scopes, and temporal properties, which are used in the temporal API and unlike regular properties are not reified. Note how, in the example in Figure 14, the temporal property `observedAt` is provided, which is also defined as an NGSI-LD core term, and, as such, it is not separately listed in the custom `@context`.

2.2.3.4 NGSI-LD Application Programming Interface

The NGSI-LD API is an information-centric API based on the NGSI-LD Information Model. It supports the management and retrieval of entity-related information. When

```
{
  "id":"urn:ngsi-ld:Vehicle:A4567",
  "type":"Vehicle",
  "brandName": {
    "type":"Property",
    "value":"Mercedes"
  },
  "inAccident": {
    "type":"Relationship",
    "object":"urn:ngsi-ld:SmartLamppost:Downtown1",
    "observedAt":"2017-07-29T12:00:00Z",
    "providedBy":{
      "type":"Relationship",
      "object":"urn:ngsi-ld:Org:Officer123"
    }
  },
  "@context":[
   "http://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld",
   {
   "Vehicle":"https://example.org/exampleOntology/Vehicle",
   "brandName":"https://example.org/exampleOntology/brandName",
   "inAccident":"https://example.org/exampleOntology/inAccident",
   "providedBy":"https://example.org/exampleOntology/providedBy"
   }
  ]
}
```

Figure 14: NGSI-LD serialization in JSON-LD

requesting information, applications specify what entity information they want to retrieve. This can be a specific entity, which is retrieved by its identifier, or entities can be discovered by providing the required type(s). Especially in the latter case, it is important for scalability reasons to scope and filter the results. For this reason a geographic scope based on a geographic property can be defined in GeoJSON and filtering based on property values or relationship objects is possible. There is also a paging mechanism. Two interaction styles are supported, a synchronous query/response and an asynchronous subscribe/notify style. For the subscribe/notify style, the notification condition can be specified, which may be based on changes or time intervals.

Figure 16 shows the main logical comments assumed by the NGSI-LD API. NGSI-LD does not define a specific architecture and instead has been defined with the intention of being usable in different concrete architectures. At the core is a component called *Broker*. *Context Consumer*s request information from the Broker using either the query or the subscribe/notify interaction style as described above. In the simplest setup, there is a central Broker that stores all information. In this case *Context Producer*s create and

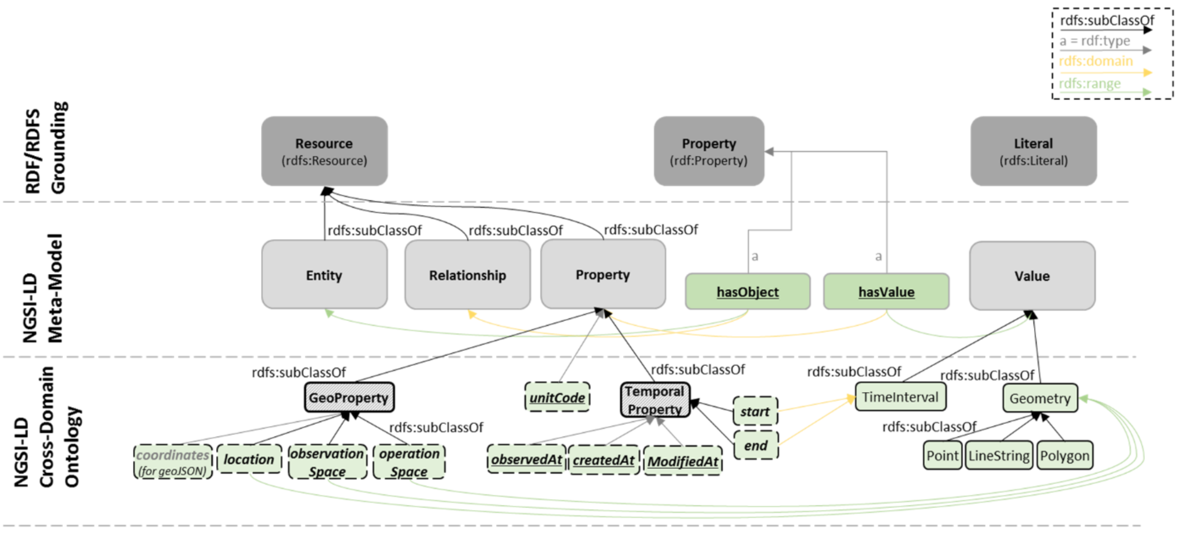


Figure 15: NGSI-LD Core and Cross Domain Model

update information in the Broker.

In a more advanced setup, not all information is stored by the Broker, but there are *Context Sources* which themselves implement the query and subscribe/notify functionality of NGSI-LD. Such a setup may be chosen for scalability reasons, or because different organizational units want to stay in control of the information. Context Sources can also be complete Brokers again, enabling the creation of a hierarchical federation of existing NGSI-LD deployments.

To enable a distributed scenario, Context Sources register what information they can provide with the *Registry Server*. Such a Registry Server can be implemented as a standalone component or be tightly integrated with one or more Brokers. Context Sources can register their information at different granularities. They can register a specific entity with properties and relationships, e.g. for `urn:ngsi-ld:Vehicle:A4567`, the source has `brandname`, `location` and `speed`. They can register that they have (any) information about a specific entity, e.g. `urn:ngsi-ld:Vehicle:A4567`. They can register that they have certain information about entities of a certain type in a certain area, e.g. entities of type `ParkingSpace` with properties `location` and `occupancy` in the area covering the city of Heidelberg. Finally, they can register that they have (any) information about certain types in a certain area, e.g. of type `Vehicle` in the city of Heidelberg. This provides flexibility and enables having a trade-off between the cost of keeping the index in the Registry Server up-to-date and the cost of requests, i.e. how many Context Sources have to be asked on a certain request.

In the context of Fed4IoT project, the Scorpio NGSI-LD Broker has been developed that supports multiple different architectures, i.e. you can use it in a small centralized deployment, as well as a hierarchically distributed and federated setting. The implementation is modular and micro-service based using Kafka as the message bus and PostgreSQL as a database with PostGIS to support geographic queries.

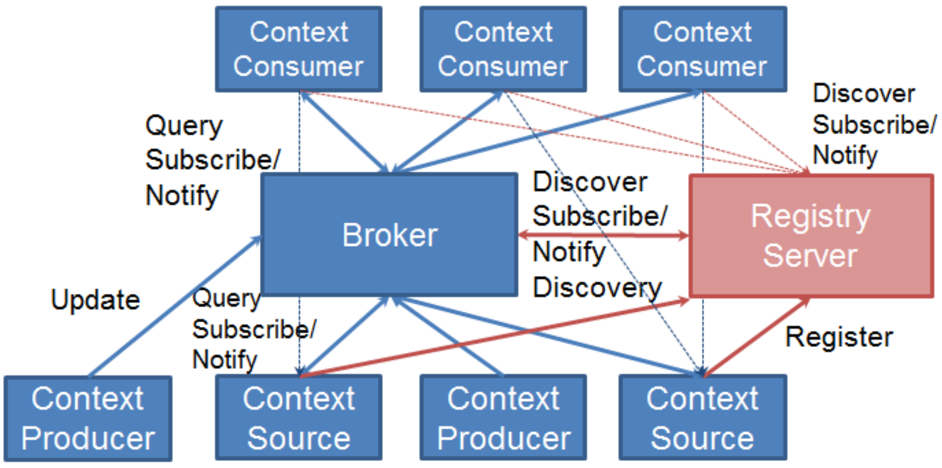


Figure 16: NGSI-LD logical architecture

There is currently no explicit support for actuation in the NGSI-LD API. Nevertheless, using some conventions, NGSI-LD Brokers can be used for the interaction between actuator clients and actuators/actuator proxies implementing the NGSI-LD API.

2.3 Related projects

In this section, we briefly describe past or ongoing project whose findings can be connected or inspirational to Fed4IoT.

2.3.1 Wise-IoT

The Wise-IoT project was a joint R&D project between Europe and South Korea from 2016 to 2108. The name *Wise-IoT* stood for *Worldwide Interoperability for SEmantics IoT*. The focus was on using semantics to create interoperability between different standard-based platforms and technologies, in particular *oneM2M* and *FIWARE*.

Key application domains were smart city and smart skiing resorts, and applications were developed that run both in Europe and South Korea, with a concept about how domains could be federated, so that users would seamlessly be able to use an application across countries, e.g. when requesting free parking spaces, depending on their location being in Santander or Busan, they would get the desired parking spaces in their vicinity. This federation concept was developed based on FIWARE NGSI, and it can also be realized using the Scorpio NGSI-LD Broker which is available to the Fed4IoT project.

One of the main aspects of Wise-IoT was the use of semantics to achieve interoperability. Ontology-based data models for the different application areas were defined, enabling translation of information between the different platforms and technologies. In particular, the concept of *Morphing Mediation Gateway (MMG)* was developed that allowed the dynamic instantiation of translation components making information from one platform available on the other, on the basis of the previously agreed ontology concepts.

The *Adaptive Semantic Module (ASM)* of the Morphing Mediation Gateway as shown in Figure 17 showcases the automatic, semantics-based translation from the oneM2M platform to the NGSI-based FIWARE platform. Raw information in the oneM2M platform was annotated with semantic information to provide the necessary basis for the translation. The ASM would discover semantically annotated resources and check whether a translation module is available. If that was not the case, it was checked whether such a module was available in a code repository and could be dynamically instantiated. Whenever such a module was available, the ASM would subscribe for new content instances becoming available in the oneM2M platform. This content information together with the semantic annotation was then used to translate the information into NGSI and push the information to the NGSI-based FIWARE platform, making the information originally only available to oneM2M applications also available to FIWARE applications. This translation functionality may also be useful to take raw information from a oneM2M platform, semantically annotate it, and make it available in the NGSI-LD format.

Results and translation approaches of Wise-IoT will be explored in Fed4IoT to translate information gathered from real IoT systems into the neutral format used inside the platform, namely NGSI-LD (see sec. 4). Besides, translators will be also used inside the so-called Virtual Silo Controller to translate from the neutral format to the format used by the Virtual Silo Broker.

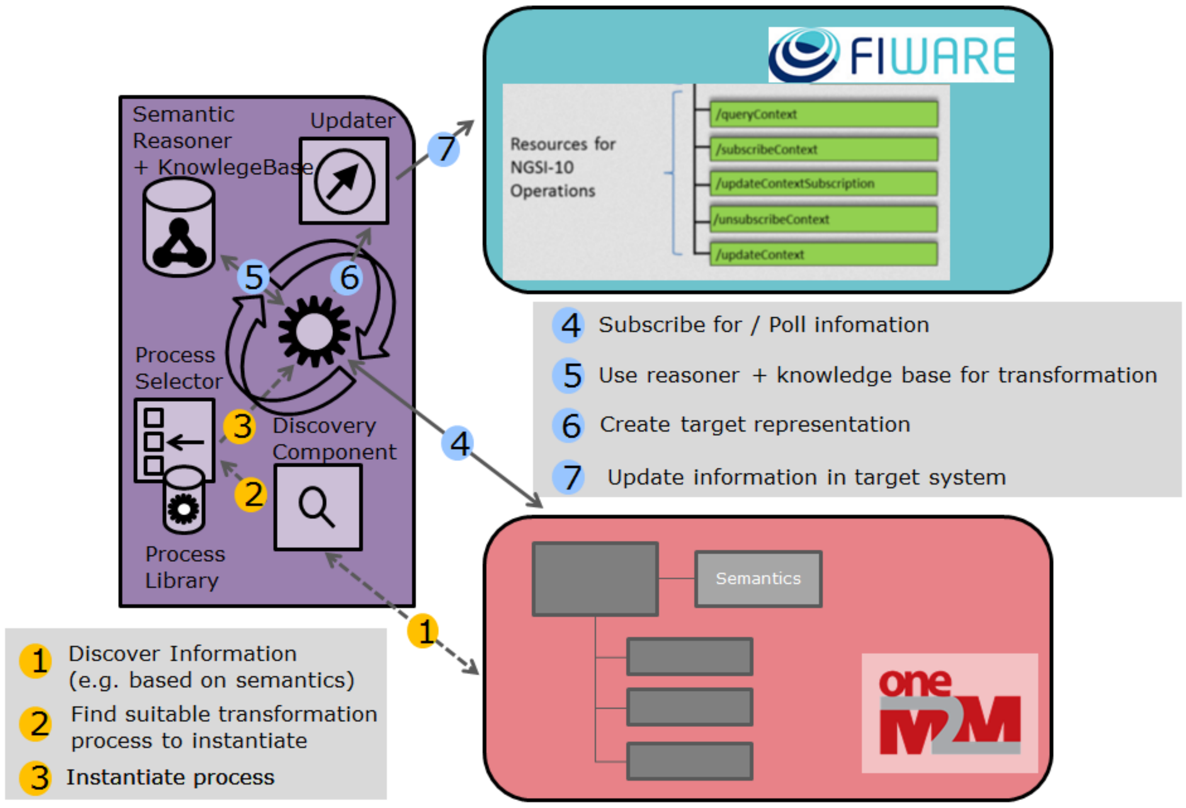

Semantic Reasoner + KnowlegeBase
Updater
Process Selector
Discovery Component
Process Library
FIWARE
Resources for NGSI-10 Operations
/queryContext
/subscribeContext
/updateContextSubscription
/unsubscribeContext
/updateContext
4 Subscribe for / Poll infomation
5 Use reasoner + knowledge base for transformation
6 Create target representation
7 Update information in target system
Semantics
oneM2M
1 Discover Information (e.g. based on semantics)
2 Find suitable transformation process to instantiate
3 Instantiate process

Figure 17: Adaptive Semantic Module of Morphing Mediation Gateway

2.3.2 CPaaS.IO

CPaaS.IO, *"City Platform as a Service. Interoperable and Open"* (see `http://www.cpaas.io`), started in 2016 and recently finished, is a joint R&D project between Europe and Japan. It aims at innovating in the scope of Smart Cities. This means creating value for the society and all actors in the city environment people, private enterprises, public administrations. To achieve this, the CPaaS.io platform combines the capabilities of the Internet of Things (IoT), big data analytics and cloud service provisioning with Open Government Data and Linked Data approaches.

The main focus in this project is to provide a federation of EU and JP platforms, that allows the secure exchange of information. For the EU side FIWARE relevant components such as Orion and Aeron brokers, IDAS for integrating the information coming from IoT devices, and COMET for generating historical information among others. Additionally, by using the NGSI interface and data model the integration of heterogeneous information was a success too.

Among other outcomes of this project, we highlight the development of a FIWARE GE (Generic Enabler) called FogFlow [10]. A framework for dynamically orchestrating IoT deployments in both edge and cloud planes. It also provides a GUI which allows users to easily define the tasks that must be executed by IoT Gateways or servers in the different planes. Its specification and documentation can be found in (`https://fogflow.readthedocs.io/en/latest/`).

Figure 18 presents a high-level view of this framework where we can understand its mode, of operation, as well as the interactions that this framework has with the producers and consumers of the information.

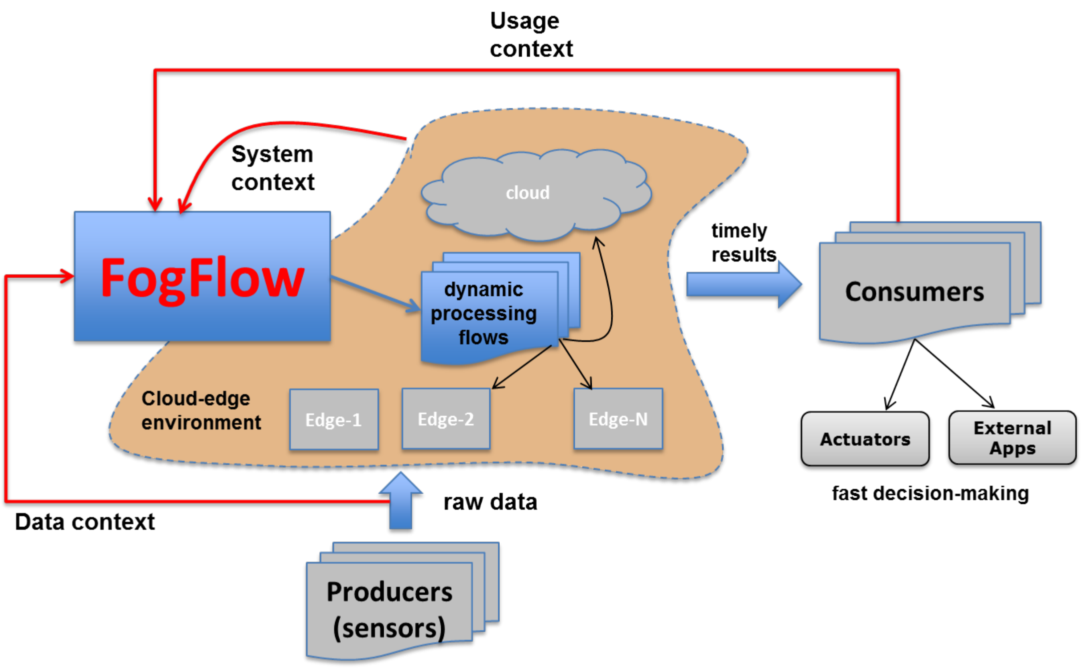


Figure 18: High-level view of FogFlow framework

According to this figure, Fogflow creates dynamic processing flows which can be deployed in both edge and cloud environments. The flows deployed at the edges process and aggregate the information coming from the producers and make it available to consumers for decision making thanks to the use of a Broker.

Fed4IoT is reusing and evolving FogFlow as a development framework for ThingVisors, as detailed in section 5.

2.3.3 IoT-Crawler

IoTCrawler [11], (`http://www.iotcrawler.eu`), is an on-going R&D project, which started in 2018, with the aim of creating a search engine for Internet of Things devices, making real-world data accessible and actionable in a secure and privacy-concerned manner.

In light of the overall architecture presented in Figure 19, there are different layers responsible for the integration of IoT frameworks; Security, Privacy and Trust; Crawling; Discovery and Indexing; Semantic search. It provides a distributed IoT framework based on brokers that use NGSI-LD for homogeneously representing the resources integrated into the IoTCrawler platform.

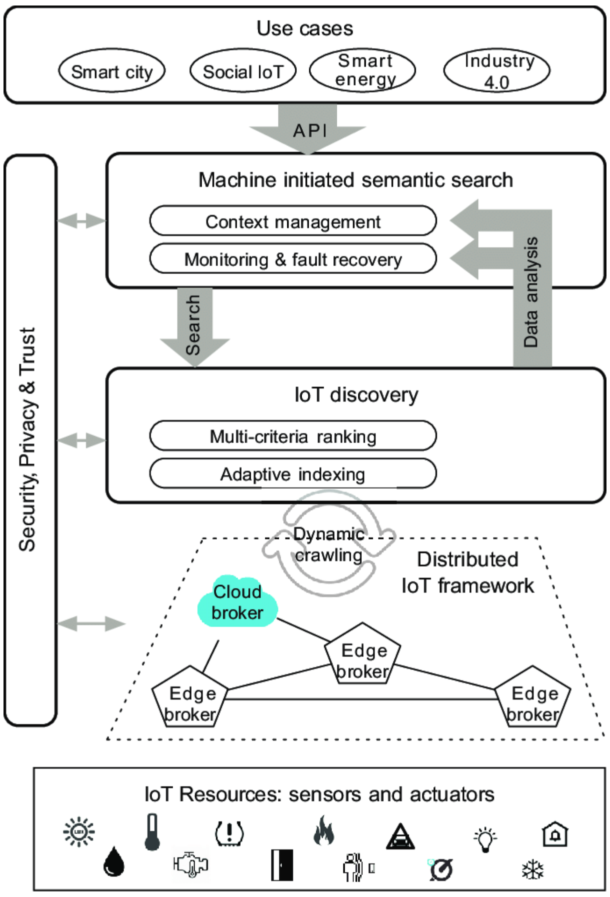


Figure 19: Overall architecture of the IoTCrawler framework

Security, Privacy and Trust is a transverse component responsible for providing secure exchange of information between the integrated IoT platforms and the users. This component takes into account not only the integration of different enablers for authentication, and privacy, but also the representation of security properties attached to the integrated information, so that it can be later used in the semantic search carried out by the final users.

Results of IoT-Crawler can be used in the second release of Fed4IoT architecture in order to support semantic search of virtual things, as briefly reported in sec. 5.

3 IoT Virtualization

Cloud Computing decouples infrastructure providers from application providers. The huge proliferation of web, mobile and machine-to-machine applications is undeniably an effect of this decoupling. Small/medium sized application providers can give vent to the invention of new applications by being free from the burden of maintaining and implementing complex ICT infrastructures; thanks to Cloud Computing they can rent them.

The Fed4IoT project wants to bring the same infrastructure/application providers decoupling to the world of the Internet of Things, where it is not yet present. In fact, nowadays, most of real-world IoT solutions operate within isolated *Silos* containing both the infrastructure and the full-stack software. For small stakeholders, the infrastructure provisioning might be a barrier that prevents their entering in the IoT arena, even though they might have innovative ideas. For instance, let us consider use cases for smart lighting or crime prevention systems in a big city. Tens of thousands of presence sensors, cameras and intelligent light bulbs are necessary, with a very high initial capital expenditure. Such high costs would be affordable to a small number of large corporations only, thus preventing fair competition and, even worse, slowing down the innovation pace, which instead is fast when thousands of small stakeholders take the field.

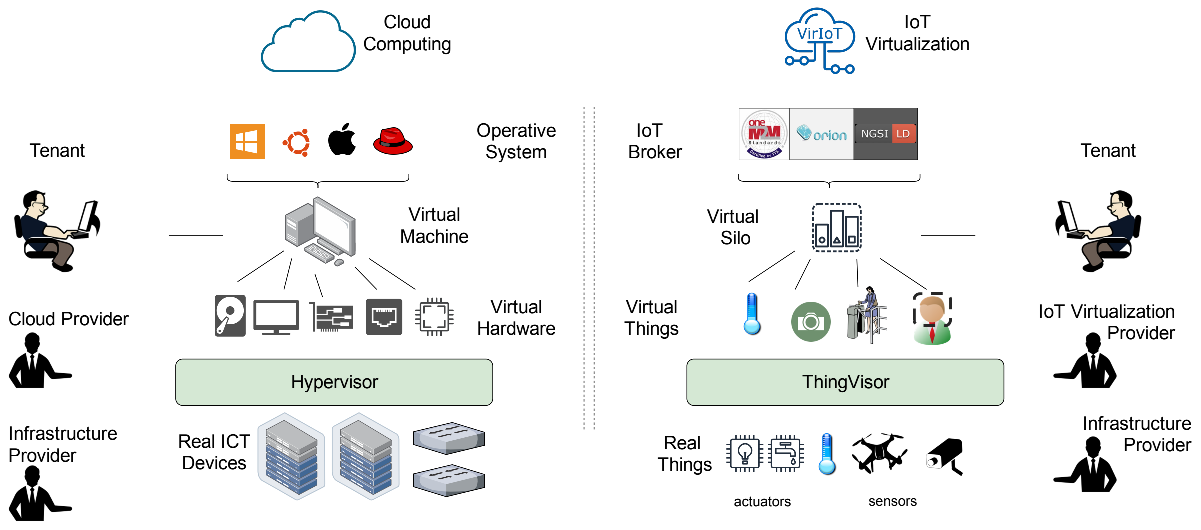


Figure 20: IoT virtualization paradigm

Figure 20 shows the **IoT virtualization paradigm** that we are promoting. To simplify the explanation we have shown a parallelism with cloud computing in the figure. Cloud computing platforms use a pool of real ICT devices (servers, network switches, storage, etc.) to offer virtual machines with configurable virtual hardware and operating system (OS). A key component of virtualization is the Hypervisor, a software that interacts with real hardware to create a virtual one.

Similarly, but for IoT applications, the Fed4IoT IoT virtualization platform, named

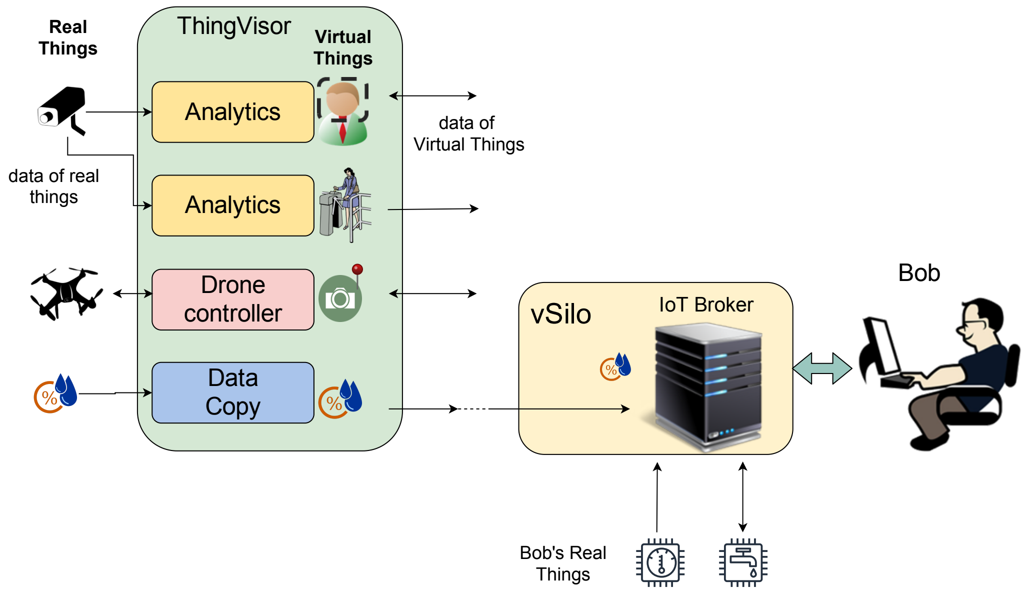


Figure 21: Virtual Things (vThings) and Virtual Silos (vSilos)

VirIoT , uses a federated pool of Real Things, including sensors and actuators, to offer **Virtual Silos** (akin to the virtual machines) consisting of a configurable set of **Virtual Things** (akin to the virtual hardware) and an IoT Broker (akin to the operating system) that exposes their data to the user. A Virtual Thing emulates the behavior of a Real Thing through the processing of data exchanged with real IoT devices, made available by different infrastructure providers. This elaboration/virtualization is performed by a software called **ThingVisor**.

In Figure 21, we have a ThingVisor that creates four Virtual Things: a face detector, a person counter, a moving camera that can be positioned in different places, and a hygrometer. The implementations of these Virtual Things provide the exchange of data with three Real Things, namely a camera, a hygrometer and a drone with an embedded camera. The figure shows that the things' virtualisation concept that we are considering may go beyond traditional data processing, since it can also involve "control" of the Real Things, such as drones or actuators. The virtual face detector and virtual person counter produce their data by performing analytics on the video stream coming from the real camera. The virtual hygrometer generates its data by merely copying data coming from the real hygrometer. Finally, the virtual moving camera is a camera that takes pictures at a very slow rate (e.g. one frame per hour) which a user/tenant can relocate to one or more given positions, such as interesting hot spots of a harbour in need of statistics or surveillance. In this last example, virtualisation is achieved by controlling the path of a drone to periodically drive it over the locations the tenants have chosen, and thereby taking a picture.

Figure 21 also shows the concept of Virtual Silo. Virtual Silos are isolated environments dedicated to a specific tenant for running his applications. A tenant can add data coming from the platform's Virtual Things to his Virtual Silo. Besides, he can also connect his Real Things to his Virtual Silo. Collectively, this data is exposed to the external world through a broker technology of choice. For instance, let us assume that Bob is a tenant who wants to develop a watering system for his house, and he is familiar with the FIWARE Orion Broker. Bob can create a Virtual Silo that embeds such a broker, connect his own thermometers and watering devices (actuators) to the broker, and he can then "rent" a virtual hygrometer for measuring air humidity outside his house, just because he does not own a real one. Data from the rented hygrometer reaches Bob's broker in the Virtual Silo, together with data from his sensors. So, Bob only sees his dedicated data set and broker, by accessing his Virtual silo, and the platform thereby provides for data and service isolation.

IoT application providers, like Bob, can simply rent a Virtual Silo with the necessary Virtual Things, avoiding the burden of deploying a real IoT system. Therefore, such an IoT virtualization paradigm decouples IoT infrastructure providers from application providers, making it possible for IoT infrastructure providers to make better use of their IoT devices by sharing their data with different application providers (tenants) and, for application providers, to dynamically configure the IoT infrastructure they need. Similar to the case of private clouds, the different providers can also be a single entity, using an IoT virtualization platform to run experimental services within the private infrastructure in use every day, by raising the security bar to run applications and things within isolated environments.

4 VirIoT System Architecture

Figure 22 shows the architecture of the Fed4IoT VirIoT platform.

- VirIoT integrates the emerging and promising European and Japanese IoT information standards into a single **IoT Virtualization platform** and makes users free to choose the standard they prefer for their Virtual Silos, which are IoT systems offered as a service.
- **OneM2M**, **NGSI** and **NGSI-LD** standards are supported for Virtual Silos, as well as simple MQTT connectors that allow Virtual Silos to be connected to IoT services offered by upstream cloud providers such as Azure, AWS, etc.
- VirIoT can be part of a larger **IoT federation** based on the NGSI-LD standard.
- By exploiting existing IoT and cloud platforms (such as oneM2M, FIWARE, Kubernetes, etc.), VirIoT establishes **a reference interoperability framework** that a pool of federated IoT and fog/edge/cloud resources can be built upon.
- VirIoT is **open-source** (https://github.com/fed4iot/VirIoT) and follows a **cloud-native micro-services design**, hence each component is an autonomous subsystem exposing network interfaces. This simplifies continuous integration and delivery of new components without interrupting the platform operations, and multiple developers can work on independent components, making faster the platform growth and innovation. Linux containers (e.g. Docker) are the preferred component packaging tool, supported by **Kubernetes**-based (k8s) orchestration to simplify continuous deployment.

In what follows we present the main components of the architecture, its procedures and its cloud/edge deployment using a Kubernetes installation that spans EU and JP.

4.1 Real Things

On the left of Figure 22 we have many IoT Systems, where an IoT System is made of a network of Real Things (sensors, actuators, etc.) exposing information through an IoT platform, such as FIWARE Orion or Mobius [7] oneM2M. Hence, an IoT System is formed by a collection of Real Things and by the platform that manages them. Information coming from different IoT Systems, and possibly from other data sources (e.g. open data), forms the Root Data Domain, with which VirIoT exchanges information.

4.2 ThingVisors

ThingVisors (TVs) are the bridges between the real and virtual worlds. Each ThingVisor is uniquely identified by a name (`TViD`). A ThingVisor manages the data of one or more

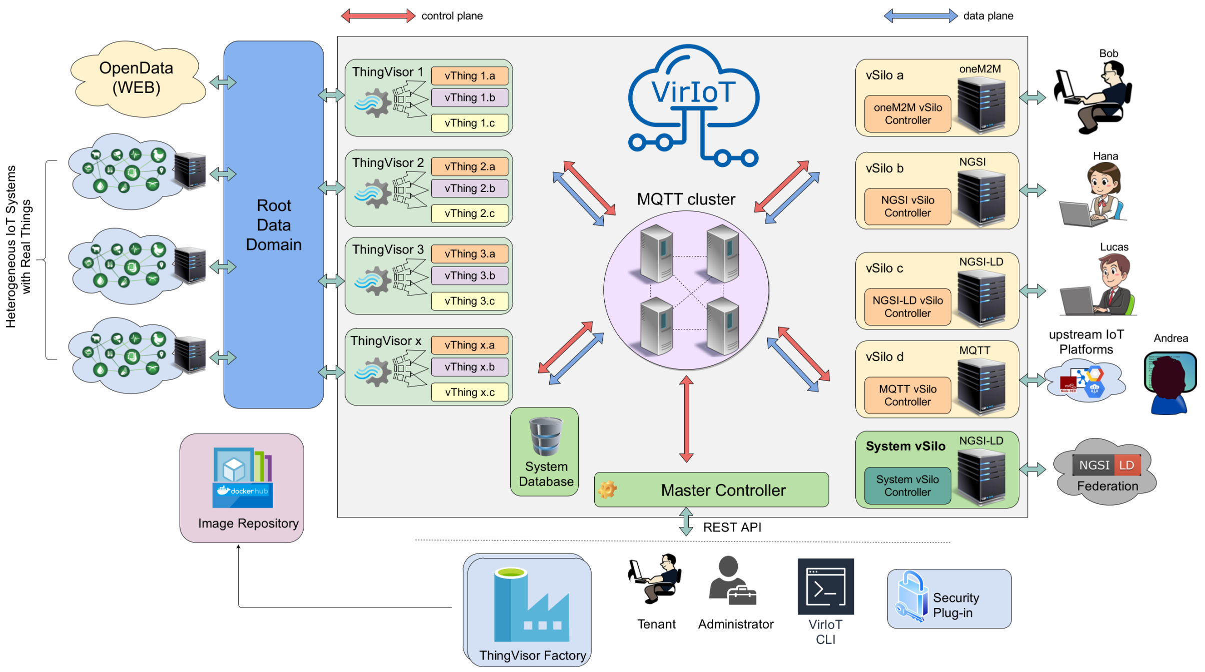


Figure 22: VirIoT System Architecture

Virtual Things, which could be virtual sensors or virtual actuators. Each Virtual Thing is uniquely identified by a name (`vThingID`[5]).

As shown in Figure 23, a ThingVisor interacts with Real Things by using their native data format. Native data items are processed to produce new data items that now belong to the Virtual Thing. These new items are produced in a *"neutral" data format* that can be translated to the data formats in use by different brokers of the Virtual Silos.

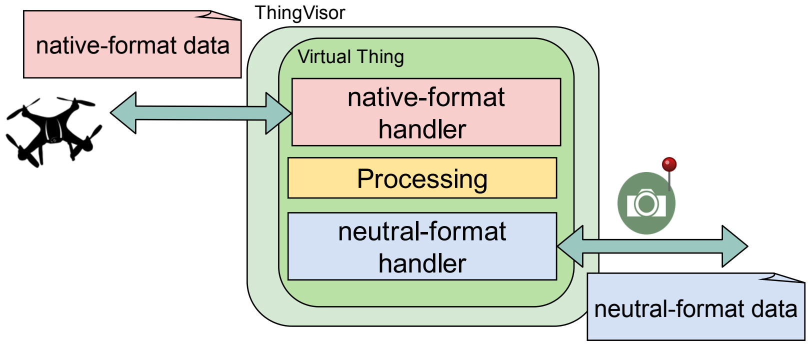


Figure 23: ThingVisor

[5]The `<vThingID>` must be equal to `<TViD>/<vThingLID>` where `<vThingLID>` is a local identifier, e.g. a random number

4.3 Virtual Silos

On the right of Figure 22 there are Virtual Silos (vSilos), which are used by tenants (Bob, Hana, Lucas and Andrea in the picture). Each vSilo is identified by a unique name (`vSiloID`). There could be different types of Virtual Silos, which differ in terms of broker type, ability to scale-ups, storage model, etc. We call *flavor* a specific configuration of a Virtual Silo, therefore a Virtual Silo is an instance of a given vSilo flavor. In Figure 22, different tenants use different vSilo flavours. Bob's vSilo includes a oneM2M broker (e.g. a Mobius server) to which his applications connects to. Hana uses a vSilo with a NGSI broker, e.g. FIWARE Orion. Lucas manages a vSilo with a NGSI-LD broker. Andrea uses a vSilo that exports data of his Virtual/Real Things via simple MQTT topics. This is kind of a *raw* vSilo, which can in turn be connected to an upstream IoT platform such as Node-Red or Google/Azure/Amazon IoT cloud services, according to the application design and deployment strategies.

Each vSilo includes an internal controller that is used to configure it (e.g. for adding or removing data of Virtual Things) and also to relay and translate the data items of selected Virtual Things from the ThingVisor to the vSilo's broker.

4.4 Master-Controller

The Master-Controller is the main element of the VirIoT control plane. It manages the deployment and delivery of new components in the systems as well as their configuration, following requests coming from administrator and tenants and by using an underlying cloud/edge platform offering containers-as-a-service, such as Kubernetes. To support its scale out, the Master-Controller is a stateless component. In fact, system state information, about Virtual Silos, Virtual Things, ThingVisors, etc. is stored in a System Database. Container images of vSilo flavors and ThingVisors are maintained by a specific Image Repository (e.g. Docker Hub).

4.5 Communications

VirIoT communication mechanisms can be classified as *internal* and *external*, as we describe in the following.

4.5.1 Internal Communications

Internal communications take place among VirIoT components, e.g. ThingVisors to vSilo, Master-Controller to vSilo, etc. Communications concern both the control and data planes. Control plane communications involves the exchange of messages used to configure VirIoT components; data plane communications involve the exchange of messages carrying IoT data items of Virtual Things.

Several aspects of VirIoT are very dynamic:

- VirIoT components can be instantiated and destroyed during run-time. For instance, vSilos can be created and destroyed, and different Virtual Things can be added and removed during the platform's lifetime;
- VirIoT intends to be a cloud-native platform where components can be re-positioned (e.g. moved from a central cloud to an edge one) or automatically re-instantiated after a failure (e.g. by Kubernetes), so a component can change its IP address during the platform's lifetime;
- the communication endpoints change frequently and many communications follow a one-to-many pattern. For example, a ThingVisor sends the data items or a control command concerning a Virtual Thing to all of the vSilos connected to it, and this list can change over time.

This dynamism makes it difficult to use a simple request-response scheme (e.g. RESTful HTTP) for internal communications, but it is perfectly supported by a publish/subscribe model. Therefore, VirIoT internal communications are based on MQTT protocol, also because there are implementations of MQTT (e.g. VerneMQ) that support clustering, a feature that well suits the distributed cloud/edge infrastructure of VirIoT . In fact, every VirIoT site includes a MQTT broker, and all together they form a single cluster (see D5.2 for an actual VirIoT deployment).

4.5.1.1 MQTT distribution system

As reported in Table 4, there are control-plane and data-plane topics, that are used system-wide.

Control topics are used by all components to receive (`c_in`) or send (`c_out`) *control messages*, such as `addVThing`, `createVSilo`, `createVThing`, etc. (see D3.1, D3.2). The `data_out` topics are mainly used to distribute the data items of Virtual Things to interested vSilos. When a new data item of a vThing is ready, the ThingVisor publishes it on the related vThing `data_out` topic. Virtual Silos that "have the vThing" are, in fact, subscribers of this topic, and therefore they receive the data items of the vThing. The `data_in` topics are used for point-to-point data-plane communication, e.g. to send commands from a vSilo to a ThingVisor, or vice-versa. Data plane communication uses neutral-format messages, described in Section 4.5.1.2.

Figure 24 shows an example of data-plane communication, which also highlights how the choice to use an MQTT cluster makes communications **bandwidth efficient** and, depending on the scenario, **with low latency**. We considered a VirIoT platform consisting of two cloud sites, an EU site and a JP site. A Thing Visor (TV1) is installed on the JP site and manages a Virtual Thing (vThing1.a). There are 4 vSilos and each one has that vThing inside. From the MQTT's point of view, the ThingVisor is the publisher to the data-plane topic `TV1/vThing1.a/data_out` and vSilos (specifically vSilo Controllers) are subscribed to the topic.

Concerning bandwidth efficiency, we notice that, when a new data item is created by the ThingVisor, it is published on `TV1/vThing1.a/data_out` topic. Only one copy of the data item is published and the MQTT cluster will build the multicast distribution tree to all subscribers accordingly. For example, in the EU-JP cross-border link only one version of the data element

Table 4: System Topics

Topic	Naming	Description
vThing (Data)	`vThing/<vThingID>/data_out`	Used by a ThingVisor to publish data items produced by a Virtual Thing.
vThing (Data)	`vThing/<vThingID>/data_in`	Used by a ThingVisor to receive data items / commands of a Virtual Thing. Used by virtual actuators.
vThing (Control)	`vThing/<vThingID>/{c_in,c_out}`	Used by a ThingVisor to send (`c_out`) and receive (`c_in`) control information related to an handled Virtual Thing (e.g. change data source, add face to match, motion threshold update, change virtual camera position, etc.)
ThingVisor (Control)	`TV/<TViD>/{c_in,c_out}`	Used by a ThingVisor to send (`c_out`) or receive (`c_in`) control messages related to the whole ThingVisor (e.g. pause, remove, activate vThing, etc.)
vSilo (Control)	`vSilo/<vSiloID>/{c_in,c_out}`	Used by the vSilo controller to send (`c_out`) or receive (`c_in`) control messages related to the specific Virtual Silo (e.g. add vThing, remove vThing, etc.)
Master (Control)	`master/{c_in,c_out}`	Used by the Master-Controller to send (`c_out`) or receive (`c_in`) control messages related to the system configuration

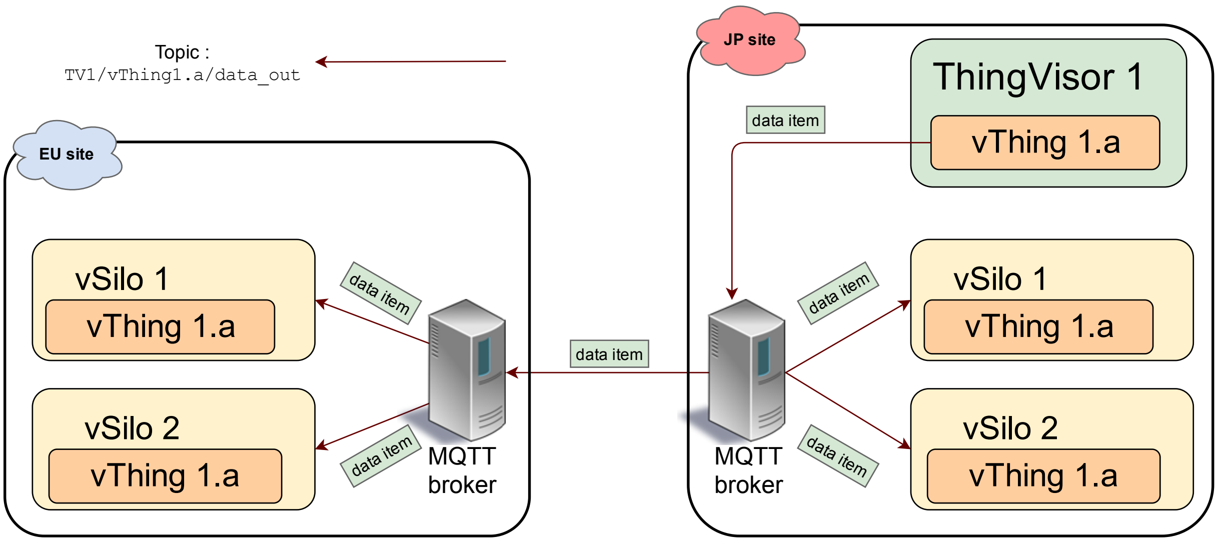


Figure 24: MQTT cluster communications

is transferred, and when it reaches the EU MQTT broker, the broker sends a copy to all vSilos subscribed to the topic.

Regarding the low latency feature, we notice that there may be some Virtual Things that are of local interest. For example, sensors or actuators in a meeting room in Japan may be of interest for room reservation applications running on Japanese phones. In this case, the related ThingVisor and vSilo can be deployed in the same JP site, so ThingVisor / vSilo communications will expewrience low latency.

4.5.1.2 Internal neutral-format for data-plane communications

VirIoT components use a specific format to communicate, in order to move data streams from producers to consumers. This internal communication of IoT data between ThingVisors and vSilos is carried out in a format that we call the VirIoT *neutral-format*. The neutral data format that we use inside our architecture is an IoT data representation format that can be easily translated to/from the different formats used by IoT brokers and IoT sensors, respectively. We argue that the NGSI-LD data format has this virtue and therefore the VirIoT neutral-format is based on NGSI-LD.

VirIoT components have to translate information gathered from real IoT systems into the neutral-format used inside the platform; specifically:

- for the generation of data items of vThings, ThingVisors translate from existing formats to the neutral-format. A ThingVisor receives data coming from physical sensors (or other sources, e.g. the web) and processes their native format in order to produce new data items in neutral-format, belonging to a vThing.
- for the control of actuators, ThingVisors also do reverse translation from neutral-format to existing formats. A ThingVisor receives command sequences from vSilos encoded using the neutral-format, which are addressed to an actuator vThing wrapping a controllable object, and translates them into whatever native format such actuations need be formatted, in order for the controlled physical object to understand them.
- for managing the received data items of vThings, vSilo controllers translate them, that are in neutral-format, into data formats in use by different internal IoT brokers. Each vSilo has a controller capable of translating neutral-format data items to the specific format used by its IoT Broker.
- for actuation purposes, vSilo controllers also do reverse translation of commands, that are in formats specific to their data IoT Broker, into neutral-format. Each vSilo controller may receive command sequences in Broker-specific format, and it converts them into neutral-format prior to addressing them to the destination actuator vThing.

Structure of the neutral-format - One basic concept in our virtualization architecture is that each vThing can produce and consume a stream of multiple different data items (possibly of different types) at a time. Thus, a vThing acts as a virtual data source producing data in the form of **a set** of NGSI-LD Entities. This is illustrated in Figure 25. We see that the neutral-format is composed of two main elements: ”meta” and ”data”.

```
{
  "meta": {
    information about the vThing, such as vThing identifier,
    wether is is an actuator or not, etc...
  }
  "data": {
    [
      NGSI-LD Entity 1,
      NGSI-LD Entity 2,
      ...
    ]
  }
}
```

Figure 25: Structure of the neutral-format messages

- The ”meta” element contains the identifier (in a mandatory field named `vThingID`), plus other optional information. For instance a metadata `"type" : "actuator"` field could also be there, if the vThing is capable of receiving actuating command sequences. All information that is common to all Entities of a vThing and pertaining to the vThing as a whole is carried within the ”meta” element of the neutral-format.
- The ”data” element contains the current set of the vThing's Entities, with their values.

For example the vThing acting as sensor for information about the Parking Sites in Murcia produces neutral-format containing a ”meta” element about the `murcia_parkingsite` vThing: the vThing identifier is there. Then there is a ”data” element containing the current set of its parkingsite Entities with all their attributes and their values. This is illustrated in Figure 26.

We can say that the vThing acts as wrapper, or producer, of NGSI-LD Entities. It is the source that produces/manages that data. This practically means there is a one-to-many relationship from vThing to Entities.

This relationship between the vThing and the Entities produced by the vThing, has to be carried all the way to the vSilo, and maintained locally in the vSilo, for the following motivations:

- Given a vThing identifier, the vSilo controller must know all Entities currently handled by the vThing (direct mapping). This is needed because, when the silo controller has to remove a vThing from the vSilo's broker, then it needs to delete all Entities under the specified vThing. There needs to be, somewhere in the vSilo, a map that keeps track of this relationship.
- Given the identifier of an Entity, the vSilo controller must know the corresponding vThing (inverse mapping). The reason is that, in case of actuators, when the user changes the Entity (in order to issue a command) in the vSilo's broker, the silo controller needs to know what is the target vThing to send the command to. There needs to be, somewhere in the vSilo, a (inverse) map that keeps track of this relationship.

Mapping of the neutral-format to external formats - For NGSI-LD to be able to effectively serve as a neutral format, two-way mappings must be designed. As said, on the one

```
{
  "meta": {
    "vThingID":"thingVisor_ParkingSite/murcia_parkingsite",
  }
  "data": {
    [
      {
        "id":"urn:ngsi-ld:parkingsite:Aparcamiento:101",
        "type":"parkingmeter",
        "name":{
          "type":"Property",
          "value":"LIBERTAD"
        },
        "totSpacePCCapacity":{
          "type":"Property",
          "value":"330"
        },
        "isOpen":{
          "type":"Property",
          "value":true
        },
        "location":{
          "type":"GeoProperty",
          "value":{
            "type":"Point",
            "coordinates":[-1.128053634,37.99128625]
          }
        }
      },
      {
        NGSI-LD Entity of parking site number 102
      },
      ...
    ]
  }
}
```

Figure 26: Example of neutral-format message

hand, ThingVisors must be able to convert from the format of the data they use in the Root Data Domain, to the neutral format. On the other hand, for each vSilo flavour the related vSilo Controller must be able to convert from the neutral-format to the format in use by the internal broker.

In addition pass-through conversion to external NGSI-LD, our goal was to support converting to raw MQTT, plus two baseline formats: the former NGSIv2 in use by currently operational FIWARE Context Brokers, such as Orion, and the oneM2M format, currently operational in Mobius, openMTC, and others oneM2M brokers. We shall support bidirectional conversion, that is from NGSIv2 (or oneM2M) to NGSI-LD neutral format, and from NGSI-LD neutral format to NGSIv2 (or oneM2M).

We observe that a ThingVisor's developer knows the format and the semantics of the data (X) she is fetching from the Root Data Domain. Hence, she can implement her custom X→NGSI-LD mapping strategy inside the ThingVisor, and different ThingVisors can even use different mapping strategies while having the same output format, that is NGSI-LD. Consequently, there is no need for rigidly specifying a mapping strategy from the Root Data Domain to NGSI-LD, given that NGSI-LD should be able to represent whatever information coming from the Root Data Domain.

Conversely, developers of vSilo flavours have to devise controllers able to automatically convert data coming from *any* ThingVisor. Consequently, the conversion strategy from the NGSI-LD neutral-format to the format used in the specific vSilo flavour (e.g.: oneM2M, NGSIv2) must be specified. The issue of neutral-format mapping and its detailed specification is reported in deliverables D4.1 and D4.2.

4.5.2 External Communications

VirIoT has different types of communication paths to external actors. Those from ThingVisors to the Real Things, and those from Virtual Silos to the tenants are not regulated by the platform but implemented autonomously within these platform components. In this way VirIoT supports the continuous integration and delivery of new services and technologies. In fact, VirIoT is able to evolve to integrate new heterogeneous Real Things by simply adding new ThingVisors that, as proxies, manage the external/internal interoperability. At the same time, VirIoT is able to evolve to integrate new servers and data models to expose Virtual Things in Virtual Silos. Again, as a proxy, the Virtual Silo Controller manages internal/external interoperability.

The platform configuration is controlled by the Master-Controller that externally exposes an HTTPs REST API (see D3.1 and D3.2), supported by a command line interface (VirIoT CLI). The Master-Controller uses a "basic" JSON Web Token based access control, associated with the REST API. Other access control mechanisms can be added as *security plug-ins*, such as those described in Figure 40.

Finally, Virtual Things can also be included in a larger federation of IoT systems. For this purpose, external communications with the rest of the federation follow the NGSI-LD standard and are managed by the System vSilo, as described in Section 4.8.

4.5.3 End-to-End Data Flows

Figure 27 shows the data flows within the platform. The Real Things produce data flows that form what we have called **Root Data Domain**. Different colors are used to indicate

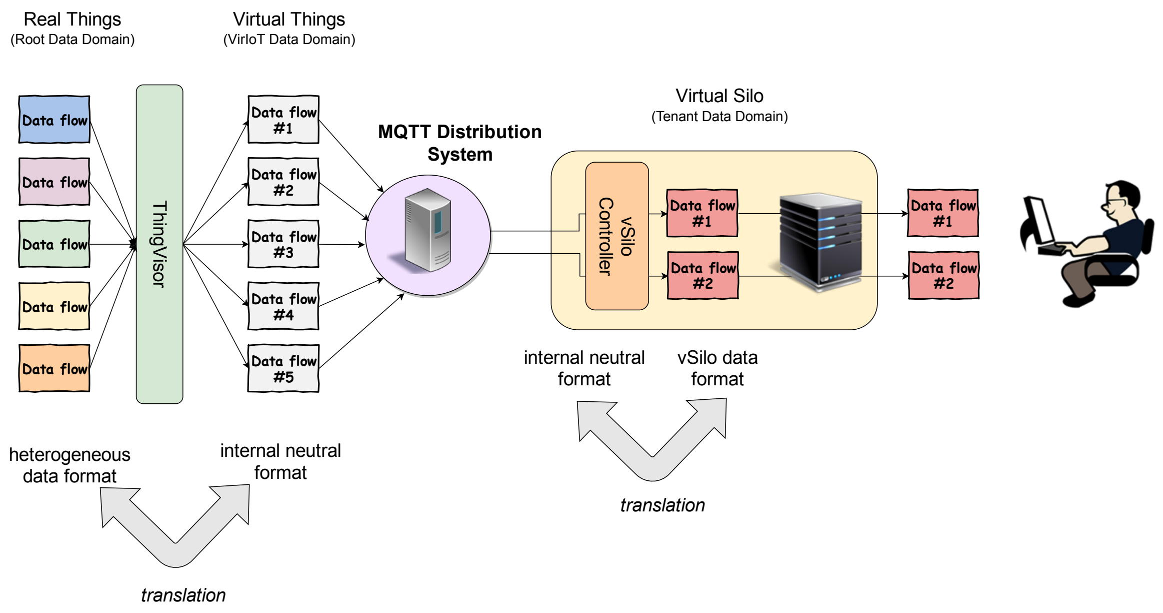


Figure 27: VirIoT data flows in case of Virtual Things emulating sensors, colors indicate data format

that these data streams can have a heterogeneous data format. The ThingVisors manage the interoperability and use these data flows to produce other data flows that use the common neutral-format (see Section 4.5.1.2). These flows form what we have called **VirIoT Data Domain** and each data flow is associated to a Virtual Thing. A tenant selects the Virtual Things that interest him and adds them to his Virtual Silo. The vSilo Controller brings the data streams of the selected Virtual Things into the Virtual Silo and performs the final translation, from the neutral-format to the data format used by the broker inside the Virtual Silo, e.g. NGSI, oneM2M, etc. This Virtual Silo data forms the **Tenant Data Domain**.

This workflow, where the data goes from left to right, refers to Real Things that are data producers, such as sensors. VirIoT also considers actuator virtualization, as shown in Figure 28. In our model, an actuator is driven by *commands*, similar to the FIWARE approach previously presented in Figure 5, and it is exposed as a Virtual Thing by a ThingsVisor. A user, who has access to an actuator, issues actuator commands in his vSilo using the vSilo data format. The vSilo Controller transforms the format into the neutral-format and routes the command flow to the ThingVisor that manages the Virtual Thing associated with the actuator. The ThingVisor finally transforms the command from the neutral-format to the specific format used by the real actuator (e.g. Philips Hue, Jetson Nano JetRobot, etc.). The next section 4.7 goes deeper on actuators.

4.6 Platform Deployment and Basic Procedures

VirIoT can be deployed both in a local test environment using Docker (Figure 29) and in a Kubernetes-based distributed production environment (Figure 30).

The Docker deployment provides that all containers of VirIoT components work in the same

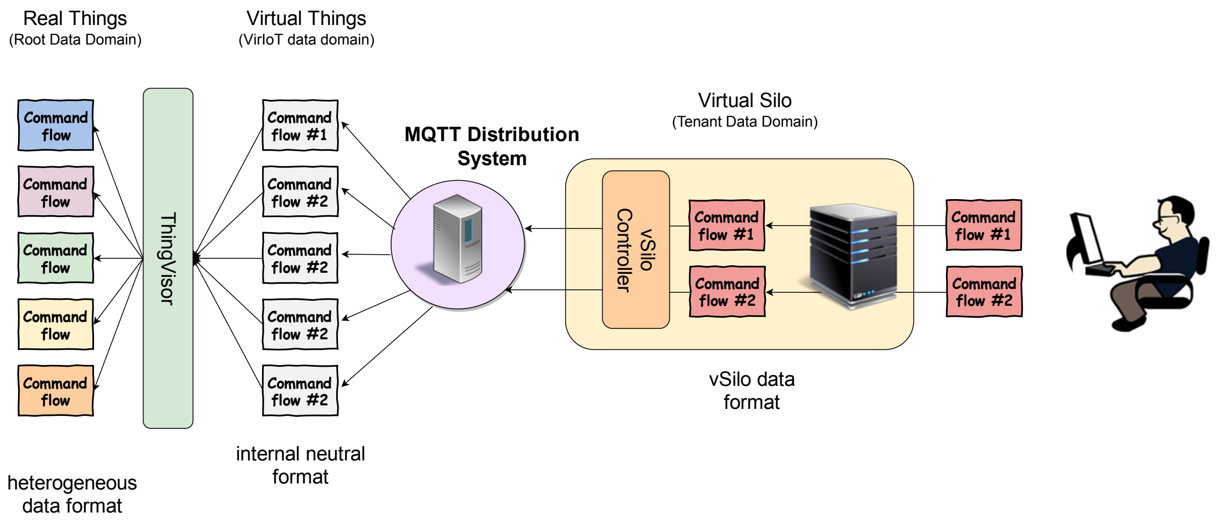


Figure 28: VirIoT data flows in case of Virtual Things emulating actuators, colors indicate data format

machine. In Figure 29 we have listed the components that form a VirIoT basic system, that is an empty system ready to host vSilos and ThingVisors. It is formed by the MQTT cluster, the system database and the Master-Controller.

The Kubernetes deployment instead exploits a distributed Kubernetes infrastructure made of (edge computing) working nodes, for example formed by some nodes in an EU zone and other nodes in a JP zone. The nodes in each VirIoT zone are labeled with the zone name and each zone has a gateway node that exposes a public IP address through which the cluster can be accessed. VirIoT components are Kubernetes PODS that can be distributed over the different zones through specific CLI commands or YAML files. For more details please refer to VirIoT GitHub website (https://github.com/fed4iot/VirIoT).

We now describe some basic procedures. We do not report here the control messages used in the procedures and the related message exchanges, which are described in D3.1 and D3.2. Similarly, data model translations made by ThingVisors and vSilo Controllers are reported in D4.1 and D4.2.

Add ThingVisor

The VirIoT administrator can request the addition of a new ThingVisor in order to extend the platform functionality. The ThingVisor's image has to be available on DockerHub. Consequently, the Master-Controller requires the underlying container platform (e.g. Kubernetes) to run the ThingVisor container in the VirIoT zone indicated by the administrator. As soon as it is operational, the ThingVisor subscribes to its input control topics to receive control messages from other components and starts publishing the data items of the Virtual Things it manages on the related topics `data_out` using neutral-format messages. These messages are distributed internally by the MQTT cluster.

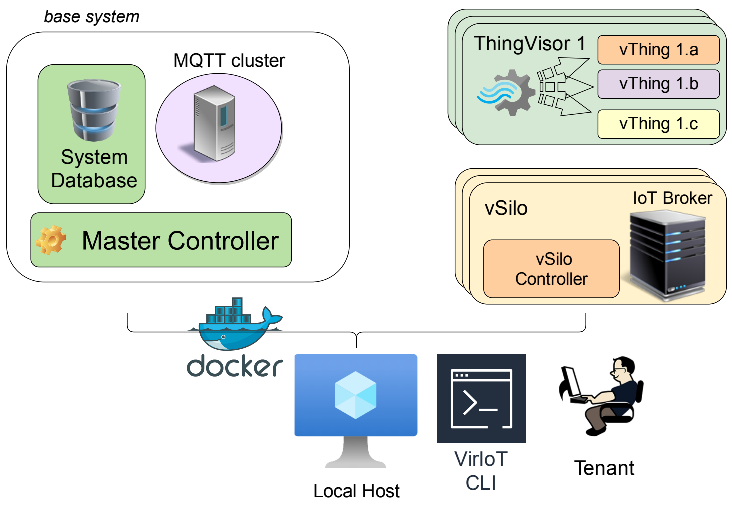


Figure 29: VirIoT Docker deployment

Create vSilo

To create a new Virtual Silo, its flavor should be available on DockerHub. When a tenant requests the creation of a new Virtual Silo in a requested VirIoT zone, the Master-Controller retrieves and executes an instance of the requested flavour image, providing the tenant with an IP address (gateway IP address) and a port where he can contact the IoT broker running within the Virtual Silo. As soon as it is operational, the Virtual Silo Controller subscribes to its input control topics in order to receive control commands from other components.

Add vThing

Subsequently, a tenant may request the addition of a Virtual Thing to his Virtual Silo. The Master-Controller, in turn, forwards this request to the Virtual Silo Controller by publishing a control message on the input control topic of the vSilo. As a result, the vSilo Controller creates the necessary structure on the vSilo broker to store incoming Virtual Thing data (e.g. create AEs and containers in the case of an M2M, etc.), and becomes a subscriber of the Virtual Thing `data_out` topic, thus starting to receive the relevant data items. When a data item is received, it is translated from the neutral-format to the format used by the vSilo broker and then inserted into the broker. This description refers to the case of Virtual Things that emulates IoT data producers, e.g. sensors. The Section 4.7 will deep dive on actuator workflow.

Access Control

Two different access control mechanisms have been designed in the frame of this project: the first one is based on a simple JSON Web Token (JWT) whereas the second is based on Distributed Capability-Based Access Control (DCapBAC). The former simply distinguishes, at API level, what can be executed by the administrator and by tenants. Therefore in terms of access control tenants are equal each other. On the other hand, the second mechanism can apply a finer and flexible access control, for example it can make differences between tenants. For example, a

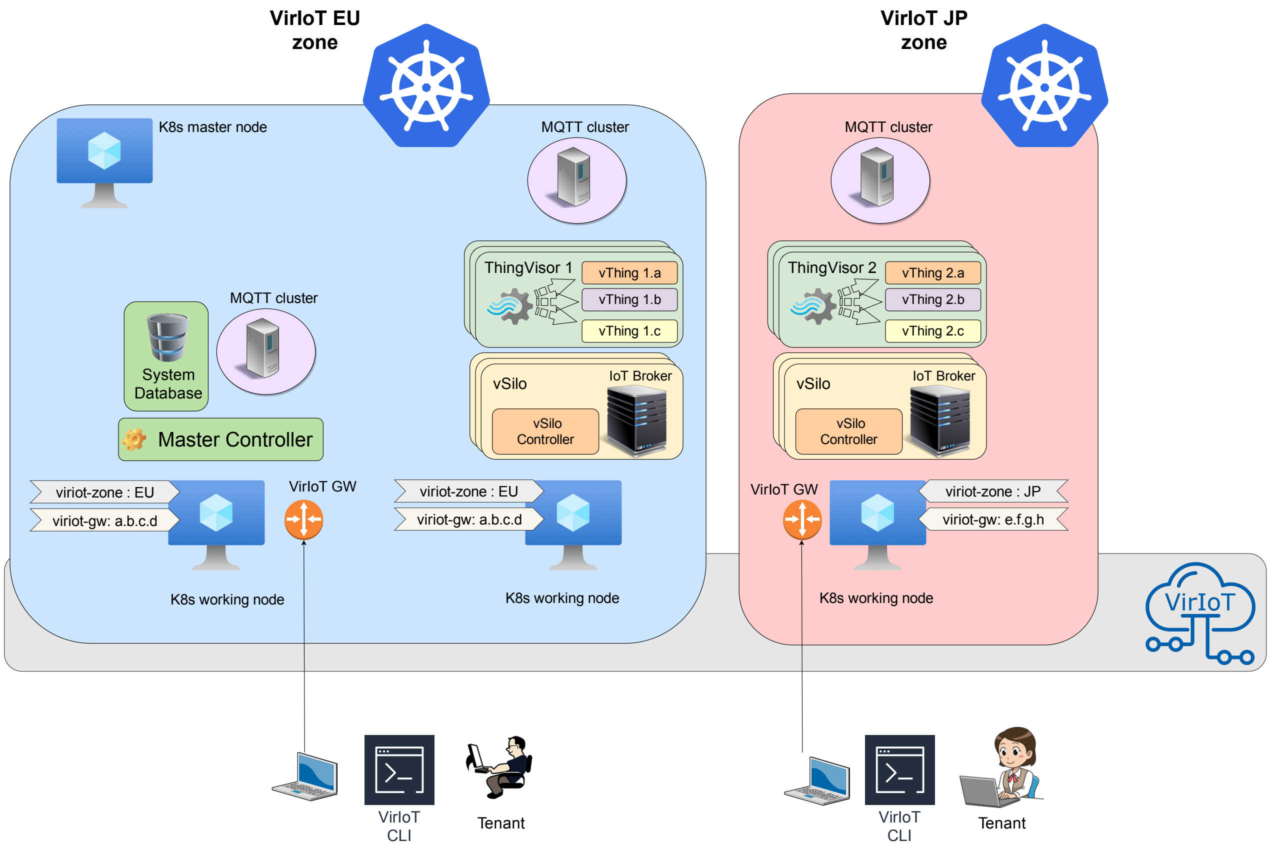


Figure 30: VirIoT Kubernetes deployment

tenant A can add vSilo Flavours, a tenant B cannot do so. More details on access control are reported in Section 4.9.

Failure management

VirIoT failure management is available in case of a Kubernetes deployment. VirIoT components are Kubernetes Pods, run as Kubernetes deployments and exposed through Kubernetes services (or similar). In case of failure of one of them, Kubernetes will run it again. Since many VirIoT components are statefull, to make this reboot effective, the Master-Controller stores all configuration information of Virtual Silos, ThingVisors, etc. in the System DB. When a component restarts, it reads its configuration in the System DB and then restores its status. The Master-Controller is stateless, so it can be easily replicated by Kubernetes for load balancing. The only "pet" component in the system is the System DB. Therefore, we used Mongo DB which easily supports replication on Kubernetes and uses persistent storage.

4.7 Virtual Actuators

VirIoT provides Virtual Things that may emulate the behavior of real sensors or actuators. The virtualization of a sensor is a rather plain process: it involves the elaboration of data flows coming from real sensors to generate the data flows of the virtual sensors, for example this is the case of the virtual person counter or the virtual hygrometer in Figure 21. The virtualization of actuators, on the other hand, requires a more complex process.

Actuators are things that can both change their state (light on/off) and take actions (move forward, detect face, etc.). There are different approaches to control an actuator via an IoT Broker. The simplest one provides that the actuator exposes its state (e.g. on/off) as variables in the broker, for example as an NGSI attribute or an oneM2M content instance, and the user can change the actuator's state by simply changing the associated value of the variable through the specific broker's interface. Albeit very simple, this approach has some limitations supporting actuator actions, because these are not always associated to a state of the actuator, or the action changes a state but this change takes a long time. For instance, in the case of a drone, action likes "take a picture" or "move to a way-point", can not be simply mapped to a state of the drone. To control actuator actions, it is better to interact with it by using *command requests* with associated properties (or parameters). To move the drone to a way-point, a user can generate a `point-toward-waypoint` command request that includes the way-point coordinates as property (see e.g. MAVLINK standard). By using command requests it is also possible to change the state of an actuator. As an example, a user can use the command request `set-on` with property `True` to switch on a light. Overall, we argue that a **command-oriented actuation model** can easily manage state changes and actions, and therefore VirIoT uses this model.

In addition to the use of commands, which, to some extent, has been already introduced by FIWARE (see Figure 6), we introduced the concept of **command QoS** to handle different actuators' use-cases. Specifically, we defined three levels of Quality of Service, which differ each other by the types of feedbacks sent back by the actuator after receiving a *command request*:

- QoS=0, the actuator sends no feedback after receiving a command request. Command feedback reduces the maximum rate of operations a user can achieve. Some applications, such as controlling the flight of a drone via joystick, need a high operation rate but no feedback. These applications can use QoS=0.

- QoS=1, the actuator sends back one or more *command result* messages after the actuation is executed. This QoS level is used by applications that need to receive one or more results/events as a consequence of the actuation action. For instance, a face detector sends back a command result every time the requested face is detected. A lamp that receives a `set-on` command request with parameter `True` can send back a command result `OK` when the lamp is switched on.

- QoS=2, extends QoS=1 with additional `command status` messages. After the reception of the command request, the actuator immediately starts to send back `command status` messages that inform the issuer about the status of the actuation action. This QoS level is meant for actuator actions that require a long execution time. For instance a `door-open` command request can be immediately followed by one or more `door-opening` command statuses, and can be eventually concluded by a `door-opened` command result.

As previously mentioned (Section 4.5.1.2), a VirIoT vThing acts as wrapper, or producer, of NGSI-LD Entities. In case of an actuator, each actuator is associated to a vThing and Figure 32 shows an example of a door actuator represented as an NGSI-LD entity. Apart from the `id` and the `type`, a VirIoT actuator entity has a set of properties that represent the status of the actuator. In the example they are the `position` and `lock` properties. Position property can

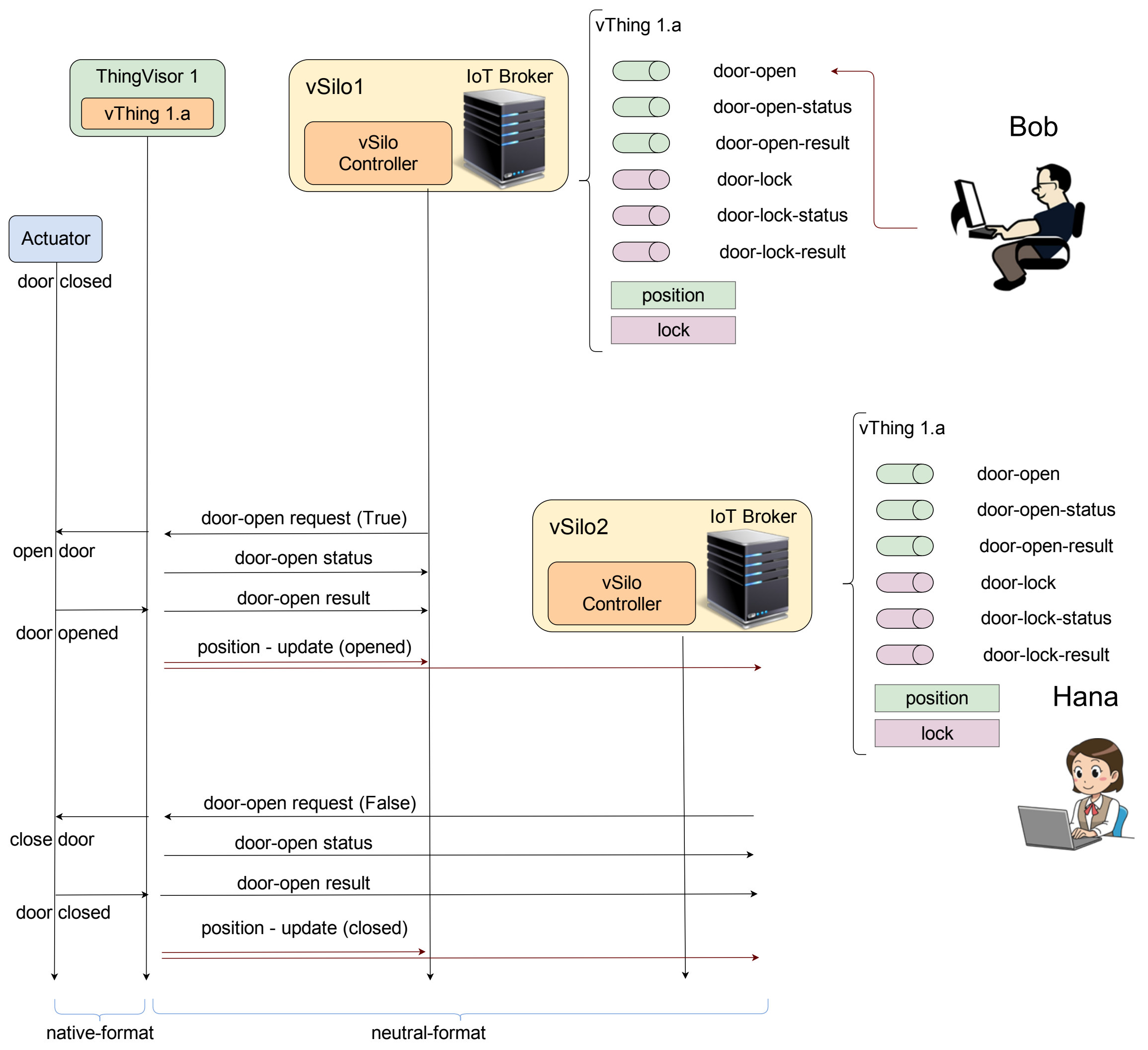


Figure 31: Actuation workflow

have value `opened` or `closed`, Lock property can have value `locked` or `unlocked`. Moreover, as a VirIoT convention, a `commands` property contains a list of commands supported by the actuator, namely `door-open, door-lock`.

Figure 31 shows the workflow related to a virtual actuator inside VirIoT (referring, without lack of generality, to the example of the door object presented in Figure 32). On the left, we have the door actuator that is exposed to the system as a Virtual Thing by its ThingVisor. Two users (Bob, Hana) have the right to control this actuator, and they have added that Virtual Thing to their Virtual Silos.

Within the IoT broker of a Virtual Silo, for each command, the VirIoT convention requires to setup three *command pipes*, mapped to three different properties:

- a property with the name of the command, e.g. `open-door`;
- the status property, which is the name of the command with `-status` as suffix;

```
{
 "id": "urn:ngsi-ld:room3:door1",
 "type": "Door",
 "position": {
  "type": "Property",
  "value": "closed"
 }
  "lock": {
  "type": "Property",
  "value": "locked"
 }
 "commands": {
  "type": "Property",
  "value": ["door-open","door-lock"]
 }
}
```

Figure 32: Example of VirIoT NGSI-LD entity associated with a door

- the result property, which is the name of the command with `-result` as suffix.

These pipes are used to exchange point-to-point command messages (request, status, result) between the ThingVisor and the Virtual Silos. For instance, the command property `door-open` is used to send command `door-open` requests, the other two properties are used to receive command status and result information, depending on the selected QoS. In Figure 31 we have considered QoS=3, therefore a command request is followed by a command status and a command result message.

We note that the proposed model with commands and pipes is **IoT broker independent**, in the sense that it can be adapted to the different broker technologies used within the Virtual Silo. For instance:

- In the case of a Virtual Silo with a oneM2M broker, each command property of the actuator can be associated to a oneM2M container whose name includes the name of the command. A command request is injected by the user in the oneM2M container as a content instance;

- In the case of a NGSI-LD Virtual Silo, the actuator is associated with an entity which has the same properties shown in Figure 32, plus the three properties per command mentioned above. To issue a command request, the user makes a NGSI-LD UPDATE of the value command property, e.g. `door-open`.

Details on command mapping among standards are reported in D4.2.

We now follow Figure 31 and we see that Bob (user of vSilo1) issues a `door-open` command request through its broker and receives status and result messages. Please notice that, in the example we are discussing, request, status, and result messages (which are shown in Figure 33) are JSON objects with a NGSI-LD Entity semantics, because (to simplify the example and mapping with the internal neutral-format) we are assuming that Bob's IoT Broker is of NGSI-LD flavour. Obviously, We note that, being the "value" element of the messages a plain JSON object, they can be handled by any IoT broker, not necessarily a NGSI-LD one. For instance, in case of a vSilo with oneM2M broker, the user simply injects or receives the "value" JSON object as the value of a content instance.

Even though the details of the information model used for commands will be described in D4.2, to simplify the explanation we briefly describe some fields by using Figure 33. The command request issued by Bob includes the command name (`door-open`), as a NGSI-LD property, whose value is a JSON object with several keys: the command parameter/value (`True`), the expected QoS (`2`), a unique command identifier, a command token. Command status and result messages, that follow the request, are a copy of the command request, but have a new value key, which indicates the value of the command status (e.g. `opening`) or result (e.g. `OK`). The semantics of `cmd-value, cmd-status` and `cmd-result` is application dependent, i.e. it is controlled by the ThingVisor designer, in order to best interact with the real actuator.

The command token is used to control access to the actuators by different Virtual Silos. Since different actuators may requires different control policies that can not b defined *a priori*, we preferred to leave the access control in the hands of the ThingVisor rather than implementing system-wide polices that could be not suited for unforeseen actuators, thus limiting the platform's evolution. For instance, if access control is needed, the ThingVisor can implement a specific `login` command, whose result value is the command token to be used in any command request that follows.

Once Bob has issued the command request (Figure 33a) to the command pipe of his vSilo IoT broker, the vSilo Controller fetches it (e.g. by polling the IoT broker or, better, by using a pub/sub scheme for low delay) and encapsulates it in a neutral-format message as shown by Figure 34 and Figure 31. This message is sent to the ThingVisor by using the related vThing `data_in` topic. The `data` part of this message contains the command request with an additional value key that is the `command notification uri`, representing a location where feedback (status, result) should be sent by the ThingVisor. In this case, the `cmd-nuri` contains the `data_in` topic of the issuing vSilo (vSilo1) so that command feedback (status and results) are sent to it, only. Neutral-format messages of command status and result follow the same structure.

After receiving the command request, the ThingVisor starts the execution of the actuation command, and it sends back a command status to vSilo1. At the end of execution, the ThingVisor sends to vSilo1 a command result. Bob monitors the command status and command result properties through his vSilo broker, to be aware of the evolution of the requested action. After the actuation is completed, the actuator changes its `position` state to `opened` and communicates this update to all connected vSilos.

The workflow of Figure 31 then shows how different actions can be similarly performed by the other user, at a later time, e.g. to close the door. Overall the exchange of command messages follows a one-to-one pattern: vSilo-ThingVisor and vice-versa; whereas the update of plain actuator properties (`position`, `lock`) follows a one-to-many pattern, from ThingVisor to all connected vSilos.

We finally observe that this actuation model can also be applied to a simple NGSI-LD system that does not include any virtualization platform in-between, by directly extending the NGSI-LD properties of the Entity shown in Figure 32 with the above mentioned command pipes properties. In this case a possible workflow involves an actuator proxy connected with the real

```
{
  "id": "urn:ngsi-ld:room3:door1",
  "type": "Door",
  "door-open" : {
   "type": "Property",
   "value": {
    "cmd-value":True,
    "cmd-qos":"2",
    "cmd-id":"123456",
    "cmd-token":"0x23456"
    }
  }
}
```

(a) Example of command request

```
{
  "id": "urn:ngsi-ld:room3:door1",
  "type": "Door",
  "door-open" : {
   "type": "Property",
   "value": {
    "cmd-value":True,
    "cmd-qos":"2",
    "cmd-id":"123456",
    "cmd-token":"0x23456",
    "cmd-status":"opening"
    }
   }
}
```

(b) Example of command status

```
{
  "id": "urn:ngsi-ld:room3:door1",
  "type": "Door",
  "door-open" : {
   "type": "Property",
   "value": {
    "cmd-value":True,
    "cmd-qos":"2",
    "cmd-id":"123456",
    "cmd-token":"0x23456",
    "cmd-result":"OK"
    }
   }
}
```

(c) Example of command result

Figure 33: Example of command messages

```
{"data":
 [{
  "id": "urn:ngsi-ld:room3:door1",
  "type": "Door",
  "door-open" : {
   "type": "Property",
   "value": {
    "cmd-value":True,
    "cmd-qos":"2",
    "cmd-id":"123456",
    "cmd-token":"0x23456",
    "cmd-nuri":"viriot:/vSilo/tenant1_vSilo1/data\_in"
        }
   }
 }],
 "meta": {"vThingID":"room3/door1", "vSiloID":"vSilo1"}
}
```

Figure 34: Example of neutral-format message embedding a command request

actuator and with a NGSI-LD broker containing the actuator entity. On the other side, there is a client that controls the actuator by issuing command requests on the command pipe properties of said NGSI-LD broker. The actuator proxy subscribes to all command properties. When the actuating client updates the value of a command property associated with a command request, the actuator proxy will receive the notification with the new value, that is the (JSON) command request. This will be translated into the proprietary format and forwarded to the actuator. The actuating client can, in turn, subscribe to the status and the result command properties. The actuator proxy updates the status of the actuation during the execution of the command, which is primarily relevant in the case of long-lasting actions, and finally writes the result, once the action has been completed. If the actuating client has subscribed to the status and result, it will receive the corresponding notifications. Independently from the command-related properties, other properties, which reflect the status of the actuator, will be updated.

4.8 System vSilo

The *System vSilo* is a special kind of Virtual Silo that is owned by the administrator of the VirIoT platform, and it contains all of the data produced by the platform, so as to make it accessible to external systems that may want to connect with VirIoT. Thus the System vSilo is used at system-level to *federate* VirIoT with external NGSI-LD platforms (Figure 16).

The System vSilo's broker exports data from all the Virtual Things, and its data is stored directly in NGSI-LD. By transparently using the internal neutral-format that is used throughout VirIoT, no conversion step is needed on the System vSilo's silo controller. System vSilo exposes information coming from vThings through a prototype NGSI-LD Context Broker that implements a sub-set of the full NGSI-LD API specification, plus some additional features that support the *discovery* of available vThings.

4.8.1 System vSilo Discovery Functionality

The System vSilo has several features to allow users query for available Entity Types and Attributes. Discovery means the ability to find out what kind of information is available in an NGSI-LD system. These features are additional to what is part of the current NGSI-LD specification.

Version 1.2.2 of the currently NGSI-LD API only supports the request for entities based on specific entities identifiers, entity types and attributes. This implies that the user has to have a detailed *a priori* knowledge of the data structures. When this is not the case, we implemented requests for available entity types in the System vSilo's discovery capabilities. Additionally, since attributes are on the same indexing level as entity types, requests for available attributes are also supported by the System vSilo discovery feature. The System vSilo exposes information about all data types currently available in the platform under the `/types` REST endpoint; similarly, it exposes information about all data types currently available in the platform under the `/attributes` REST endpoint. This information is enriched by joining it with information coming from the System DB, which has more detailed knowledge about vThings, that it has received from descriptors declared at vThing (and ThingVisor) creation time.

It must be noted that Fed4IoT is actively participating to the upgrade process of the NGSI-LD specification, and the next version of the NGSI-LD API (versions v1.3.1 and 1.4.1 are already planned) will include more advanced discovery functionality than what is currently available in v1.2.2, and some of this new functionality stems from what is part of System vSilo.

4.8.2 How System vSilo Captures Data from ThingVisors

Sytem vSilo "captures" all data coming from all vThings, automatically. At startup it asks to the System DB for the list of all currently existing vThings, then iterates through them to get their current data (see flows 1A and 1B in Figure 35).

From then on, whenever a ThingVisor produces a "create VThing" message because it wants to instantiate a new vThing, the System vSilo intercepts it, and then it adds the corresponding vThing to itself. Thus, the System vSilo:

- subscribes to the internal `TV/+/c_out` MQTT channel, in order to grab all commands from ThingVisors;
- for each "create vThing" message, it invokes an add vThing command on the Master-Controller, adding the corresponding vThing to itself, as depicted in workflow 1, 2, 3, 4 of Figure 35.

This way, each newly created vThing is automatically added to the System vSilo, making its data available through the System vSilo's NGSI-LD data broker. In addition, through the System vSilo's NGSI-LD data broker, other NGSI-LD brokers, external to our platform, can federate to VirIoT, sharing data by means of the NGSI-LD API. Given proper access control, they can access all data items coming from the Root Data Domain and, through the concept of virtual actuators we have developed (see section 4.7), they can send commands to actuators physically located within the boundaries of our Root Data Domain.

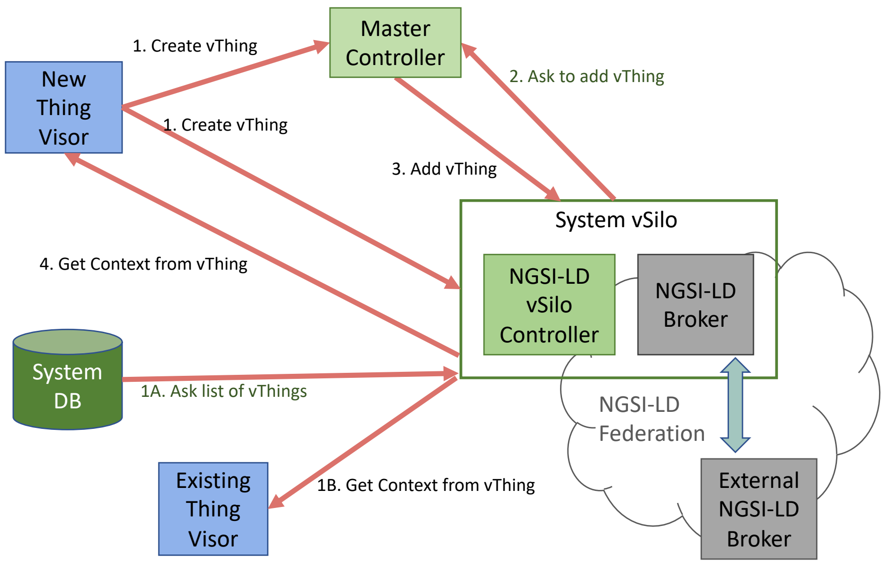


Figure 35: System vSilo's workflows

4.8.3 Querying Data in System vSilo

The System vSilo implements the standard NGSI-LD API, as specified in the NGSI-LD API HTTP binding. The main entry-point for querying data is the `/entities` REST resource. For instance, when a Weather ThingVisor is added to the system, the Sytem vSilo will immediately start incorporating data from all virtual things managed by the Weather ThingVisor, making them available as NGSI-LD Entities under the `/entities` endpoint.

Let's assume we want to know where there is a high level of humidity, then we can issue a query, as follows:

```
GET /ngsi-ld/v1/entities/?q=hygrometer.value≥60
```

And the System vSilo will respond with an array of all currently matching Entities within the system, as shown in Figure 36.

4.9 Security Aspects

Security is a maximum relevance aspect that has been considered in the Fed4IoT project. The project considers an architecture which has many external interaction points. Currently, these points are the following ones:

- `System Silo` which supports external communication

```
[
{
  "id": "urn:ngsi-ld:Tokyo:humidity",
  "type": "humidity",
  "hygrometer": {
    "type": "Property",
    "value": 94
  }
}
,
{
  "id": "urn:ngsi-ld:Grasse:humidity",
  "type": "humidity",
  "hygrometer": {
    "type": "Property",
    "value": 64
  }
}
]
```

Figure 36: Structure of the neutral-format messages

- `ThingVisors` which receive data from providers, transform and push them into the federated architecture. They can send commands to data root domain too.
- `Virtual Silo` which receives the updated information from ThingVisors and transforms it again to fit end-user. It can send commands from a tenant to data root domain through ThingVisors too.
- `Master-Controller` which offers an API to deploy and configure the system.

In this scenario, two different access control mechanisms have been designed:

- A JSON Web Token (JWT) access control mechanism
- An access control mechanism based on capabilities called Distributed Capability-Based Access Control which also considers the interaction with an Identity Manager for storing the different entities in the platform.

These two mechanisms have been designed in a compatible way so that both mechanisms can coexist.

4.9.1 JWT based Access Control

This mechanism was already proposed in deliverable D4.1 and has been implemented following the diagram presented in Figure 37.

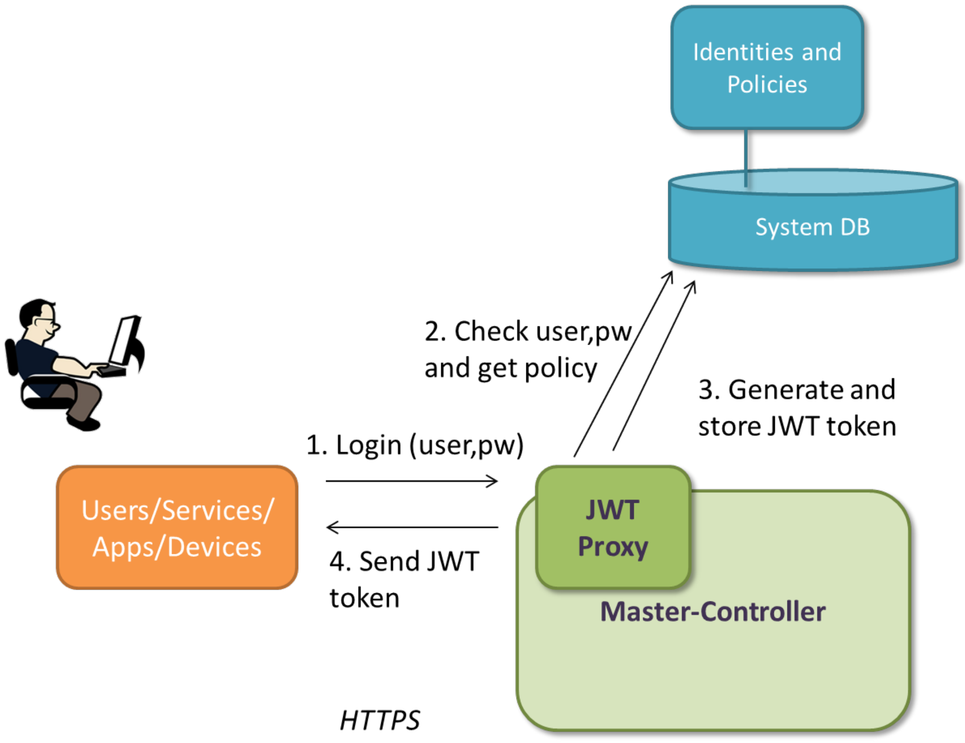


Figure 37: JWT-based login

So, basically, after the user introduces her credentials into the system, the system validates the policy stored in an information repository. After a positive validation, it generates a JSON Web Token, which follows the structure presented in Figure 38.

This token must be included in subsequent requests to the system, so that it can validate the fact that the user has the right permissions to perform the action included in the request.

4.9.2 Distributed Capability-Based Access Control (DCapBAC)

The second access control mechanism that we propose in our architecture, integrates both an Identity Manager, and a well-known authorisation framework, which is XACML. The former component is responsible for storing the identities that can interact with the system, by specifying also a set of attributes which actually define an identity, such as name, email, organisation, or even other extra features.

The XACML component is a framework which roughly defines access control policies based on triplets (subject, resource, action). The advantage of this second mechanism is that it decouples the authorisation process into two different phases: one for requesting access control to a specific resource, and another for accessing it. This way, after the first phase, the user or service receives an authorisation token called Capability Token, which is used in subsequent queries, during the second phase, meanwhile the Capability Token is valid.

```
{
        'iat': 1568105212,
        'nbf': 1568105212,
        'jti': '060e7128-7cde-40da-9731-e1c3439bc926',
        'exp': 1572425212,
        'identity': 'admin',
        'fresh': False,
        'type': 'access',
        'user_claims': '{"role": "admin"}'
}
```

Figure 38: JWT token payload

Identity Manager

The Identity Manager (IdM) Enabler offers the FIWARE Keyrock RESTFull API for authentication, based on the OAuth 2.0 protocol, which is described in the RFC6749 standards. This protocol supports several grants ("methods") types for a client application to acquire an access token, including Authorization Codes, username and password, etc. An access token can be used to authenticate a request to an API endpoint, such as a specific API endpoint of the VirIoT Master-Controller. The IdM provides functionalities to gain an identity within the system and to manage the access privileges. Specifically, the Keyrock Identity Manager implementation defines the following actors and roles:

- `User`. Any signed-up user who is able to identify herself with an email and password. Users can be assigned rights individually, or as a group.
- `Application`. Any application consisting of a series of micro-services.
- `Organization`. A group of users who can be assigned a series of rights. Altering the rights of the organization affects the access of all users of that organization. Users within an organization can either be members or admins. Admins are able to add and remove users from their organization, while members merely inherit the roles and permissions of an organization. This allows each organization to be responsible for their members and removes the need for a super-admin to administer all rights.
- `Role`. A role is a descriptive bucket for a set of permissions. A role can be assigned to either a single user or an organization. A signed-in user gains all of the permissions from all of her own roles plus all of the roles associated with her organization
- `Permission`. An ability to do something on a resource (e.g. API endpoint) within the system.
- `X-Subject-Token`. After a user has supplied her username and password, the server returns an access-token in `X-Subject-Token` header.

Authorisation

In a typical DCapBAC scenario (Figure 39), an entity (subject) tries to access a resource (e.g. API endpoint) of another entity (target). Usually, a third party (issuer) generates a token for the subject specifying which privileges it has. Thus, when the subject attempts to access a resource hosted in the target, it attaches the token which was generated by the issuer. Then, the target evaluates the token granting access to the resource. Therefore, a subject which wishes to get access to certain information from a target, requires sending the token together with the request. Thus, the target device that receives such token get to know the privileges (contained in the token) of the subject, and it can act as a Policy Enforcement Point (PEP). This simplifies the access control mechanism, and it is a relevant feature in IoT scenarios, since complex access control policies are not required to be deployed on end devices.

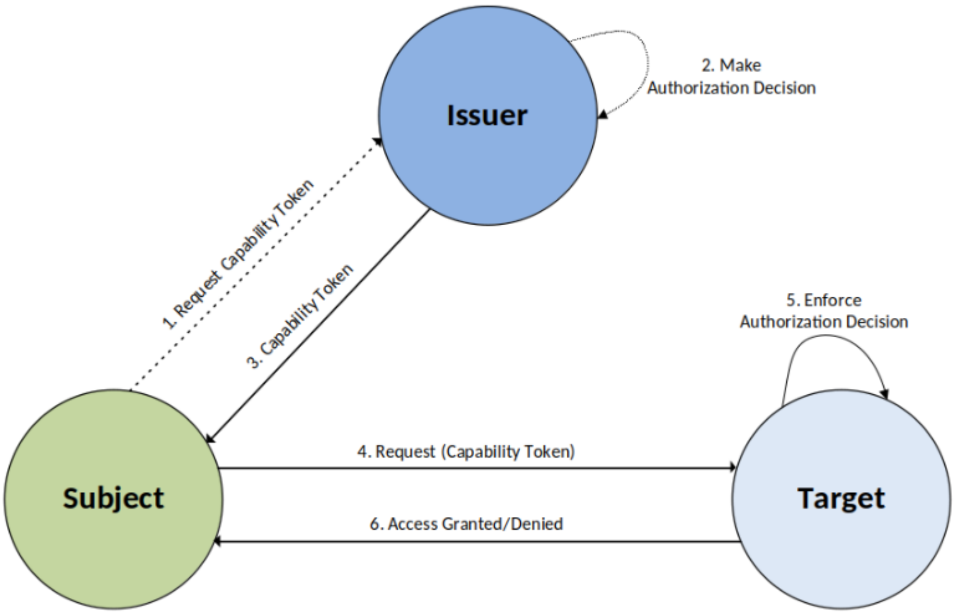

Issuer
2. Make Authorization Decision
1. Request Capability Token
3. Capability Token
5. Enforce Authorization Decision
Subject
4. Request (Capability Token)
6. Access Granted/Denied
Target

Figure 39: DCapBAC scenario

VirIoT integration

Figure 40 shows the integration of this access control technology to the VirIoT platform, as a kind of plug-in chained with the basic JWT access control already implemented within the Master-Controller.

In a schematic way DCapBAC's components are as follows:

- `Security enablers` based on XACML framework (PAP, PDP) to manage access control policies, and to decide who can access a resource and what actions can one perform over the resource. Policies are represented as triplets (subject, resource, action).
- `Capability Manager` (and `Capability Token`). It takes care of:
  - Access to authentication component to validate authentication token (returned by IdM).

- – Access to XACML framework for validating authorization requests.

  This component generates an authorization token, named Capability Token, which is then required to access the resource.

- PEP-Proxy, which validates the authorisation. Before performing any action the PEP-Proxy carries out validation, and, if validation is passed, it then forwards the command/message to the Master-Controller.

Under this scheme, tenants (or even administrators), in order to interact with VirIoT for configuration (e.g. create a vSilo, add a vThing, etc.) or maintenance purposes, go through the Identity Manager (IdM) and DCapBAC components, rather than directly accessing the Master-Controller API. The PEP_proxy has full system administration rights, and it interacts with the Master-Controller through the basic JWT access control technology previously described.

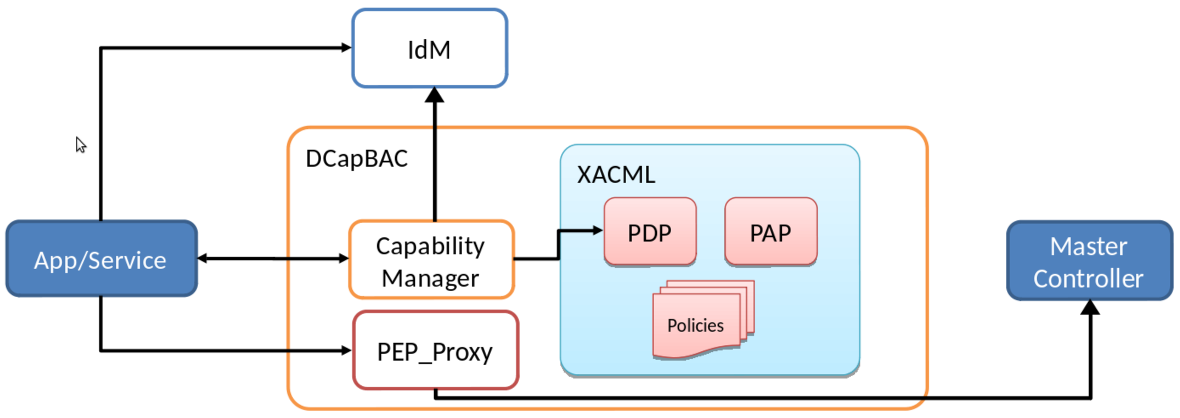


Figure 40: Security Relationships Components

The sequence diagram in Figure 41 shows the whole interaction that must be performed in order to grant to a user/service access to a specific REST resource of the Master-Controller API. As we can see, after an authentication process, the User receives an authentication token, this must be introduced in the authorisation query which is addressed to the Capability Manager. This component is responsible for translating this request to an XACML one, which is routed to the XACML-PDP component. It validates the XACML authorisation query by checking the XACML policy file. If there exists a matching policy, the XACML-PDP will issue a positive verdict, which is received by the Capability Manager. This, in turn, generates an authorisation token, called Capability Token which is sent to the User. This Capability Token must accompany the query issued to the system, so that the PEP_Proxy can verify that the query issued by the User is granted. This process is done without querying any other third party, so it is a straightforward task. After validating the query, the PEP_Proxy will forward the query to the Master-Controller by using plain JWT, as well as the corresponding response to the User.

As PEP_Proxy request example, Figure 42 shows how to access the resource, using a curl request, through the `Cap.Token` obtained in capability manager request. This request will return the response of the specific Master-Controller API.

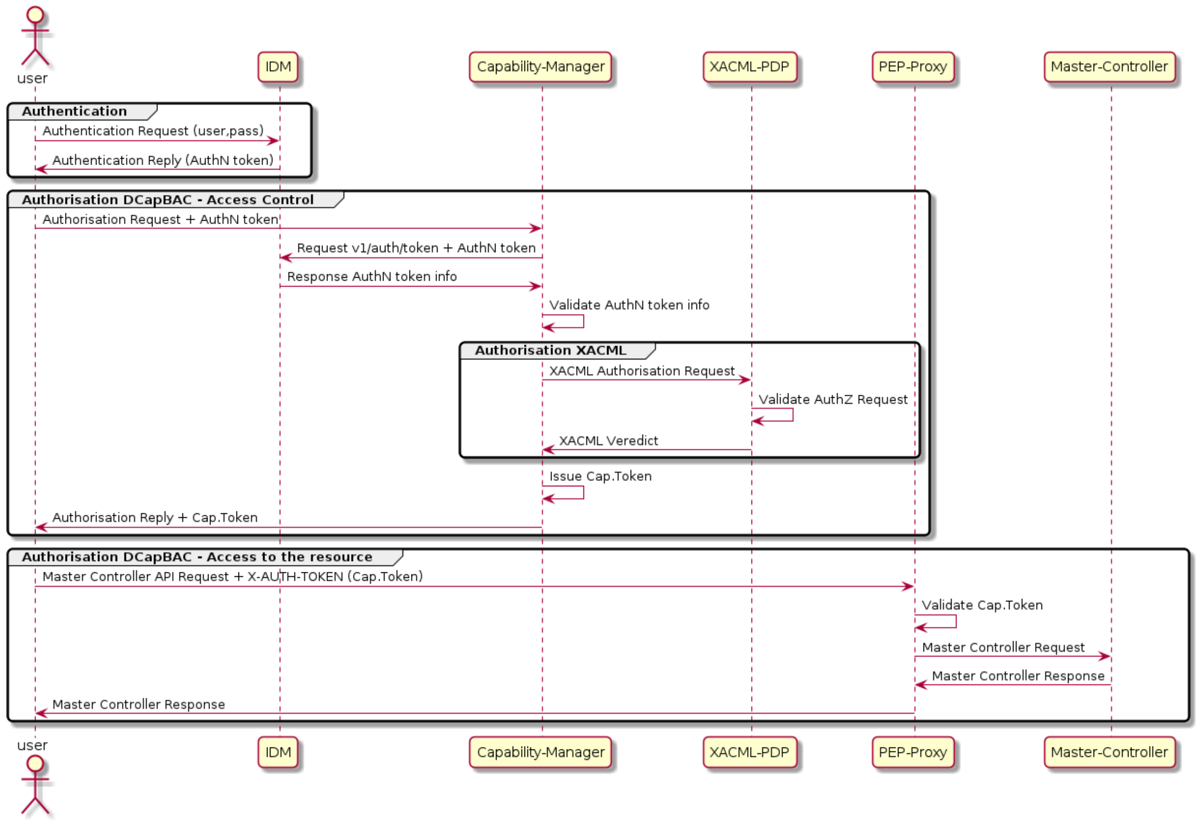

user
IDM
Capability-Manager
XACML-PDP
PEP-Proxy
Master-Controller
Authentication
Authentication Request (user,pass)
Authentication Reply (AuthN token)
Authorisation DCapBAC - Access Control
Authorisation Request + AuthN token
Request v1/auth/token + AuthN token
Response AuthN token info
Validate AuthN token info
Authorisation XACML
XACML Authorisation Request
Validate AuthZ Request
XACML Veredict
Issue Cap.Token
Authorisation Reply + Cap.Token
Authorisation DCapBAC - Access to the resource
Master Controller API Request + X-AUTH-TOKEN (Cap.Token)
Validate Cap.Token
Master Controller Request
Master Controller Response
Master Controller Response

Figure 41: Security diagram

```
curl -iX <action> <'pepproxy endpoint><resource>' \
  -H 'x_auth_token: <capability_token>' \
  -d '...'
```

Figure 42: PEP_Proxy request - accessing to specific API

4.10 Comparison with Related Works

Many IoT systems are in operation, but stay confined within their own silos. The current issue in IoT systems development is inter-operation among IoT silos. There are many attempts, besides Fed4IoT, to "punch holes" into those silos and let them work together. We already know that (as stated in Section 3), to attain the above mentioned interoperability, Fed4IoT adopts mechanisms based on ThingVisors and Virtual Silos. What the mechanisms intrinsically provide is:

- uniform access to IoT devices in different and separate data domains, and
- free choice of development and execution environment by the creator of the IoT-application.

In this section, we compare our approach with other attempts that provide interoperability among IoT silos.

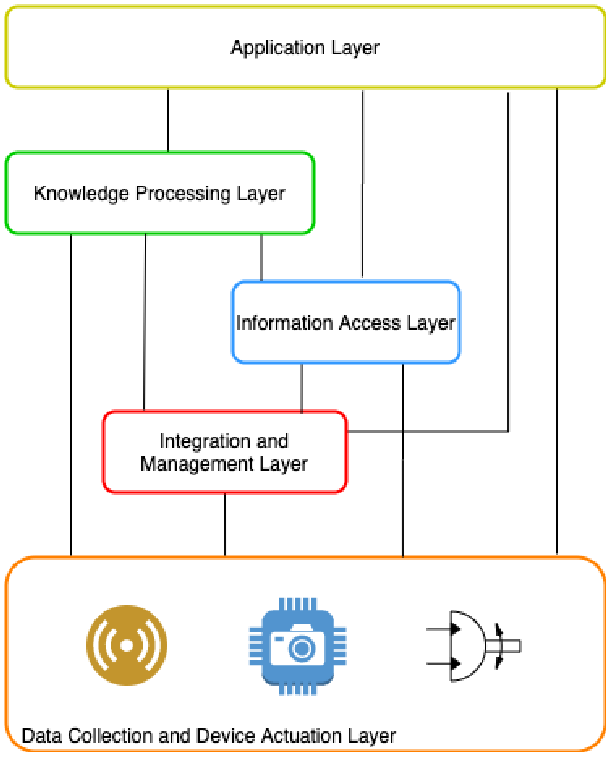
One of the projects aiming to make IoT systems work together is Wise-IoT. We briefly revised it in Section 2.3.1. The focus of Wise-IoT is unified access to different IoT devices and platforms. This unified access includes semantic unification. The semantic mapping among different IoT platforms is resolved by pre-defined ontologies agreed among the platforms.

The high-level architecture of Wise-IoT is as shown in Figure 43.

Wise-IoT adopts oneM2M at the Integration and Management Layer, and it assumes IoT devices can connect to the Integration and Management layer because oneM2M is a worldwide standard already supporting interworking among different technologies. In other words, Wise-IoT does not assume heterogeneity in the connection between its system and external data domains. Therefore, the project does not provide a mechanism like the ThingVisor, able to connect heterogeneous IoT devices in the Root Data Domain.

From the perspective of the IoT application, Wise-IoT assumes that IoT applications must conform to interfaces provided by the Wise-IoT system. We don't find any mechanism to adapt to the preference of the application, like Virtual Silo are in Fed4IoT.

CPaaS.io is a project we have introduced in Section 2.3.2. The main objective of CPaaS.io is to develop a smart city platform. In the project, the smart city platform of CPaaS.io is implemented in two flavors: *Ubiquitous ID 2.0* (u2) architecture and FIWARE. The two platforms are federated at application layer using linked data and common API as shown in Figure 44. Since the two platform are quite different, the federation is a loose one. Some APIs are provided by both the platforms, but what the applications can do using the common APIs is limited. That is, the APIs provided to smart city applications sitting on top of Figure 44 need to pay attention to the differences among the platforms where IoT devices are deployed. The smart city applications do not have the freedom to chose their preferred IoT platform, unlike what we provide by means of Virtual Silos. Full access to IoT devices is not as easily achievable as with the uniform view provided by ThingVisors.


Application Layer
Knowledge Processing Layer
Information Access Layer
Integration and Management Layer
Data Collection and Device Actuation Layer
Figure 43: Wise-IoT Architecture

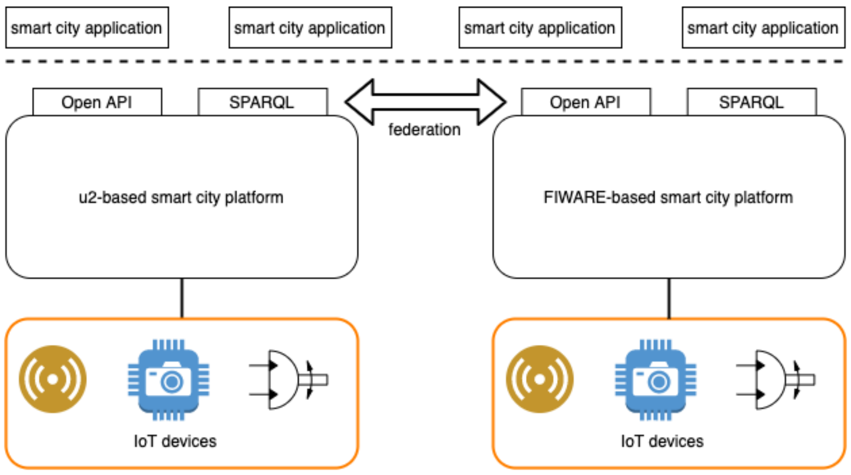


Figure 44: CPaaS.io Architecture

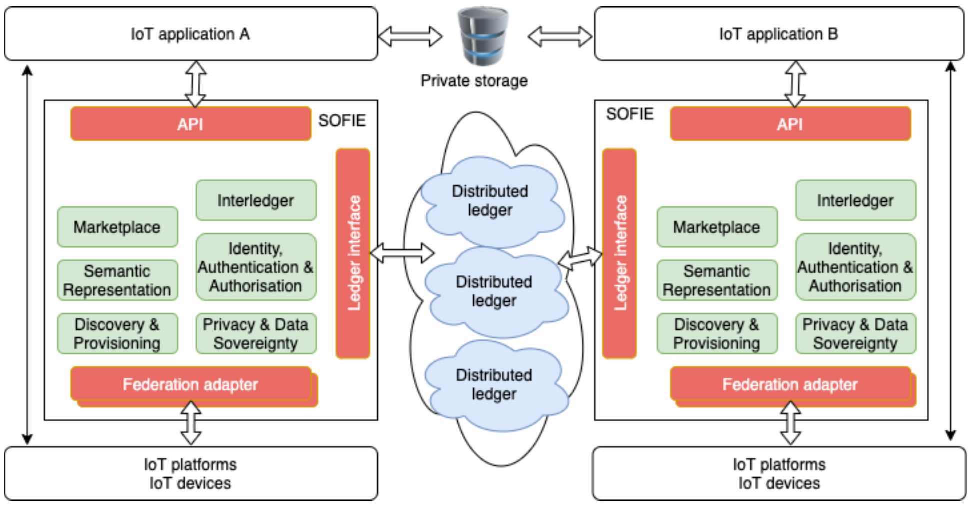


Figure 45: Architecture of SOFIE

SOFIE is a Horizon2020 project inaugurated in 2018, which is still under development at the time of this writing. SOFIE stands for “Secure Open Federation for Internet Everywhere”. The goal of SOFIE is linking heterogeneous IoT platforms without changing existing IoT platforms. The focus in SOFIE lies on security, and SOFIE exploits distributed ledger technologies (DLTs) to achieve secure federation of IoT platforms and applications. Federation in SOFIE means connecting IoT applications that are originally constructed in silos and make them work together.

The architecture of SOFIE is shown in Figure 45. IoT data is directly communicated between the IoT platforms and an IoT application. Control messages only, related to authorisation, events, payment, etc., go through SOFIE. The SOFIE architecture is defined as a framework that only designs types of functionalities, without a definite implementation. The reason is the variety in applications to which SOFIE is to be applied. Due to the variety, no single API definition can serve all the applications. In other words, different APIs are defined for different applications. This design concept is clearly different from VirIoT , where IoT applications assume an IoT platform they can chose among available standard IoT platforms, and all messages transparently reach sensors, going through VirIoT .

Similarly to Fed4IoT, in SOFIE a federation adapter is prepared for each IoT platform. The role is similar to ThingVisors. In this way, the existing IoT platform can be connected to SOFIE without any modification. Depending on the variety in IoT platforms and IoT devices to be connected to an IoT application, the corresponding federation adapters need to be designed, just like ThingVisors.

SOFIE focuses on secure data exchange among applications in different silos. Distributed ledger technology is used for that purpose, and the concept of a secure data path is constructed through the distributed ledgers.

SymbIoTe is a short hand of *Symbiosis of smart objects across IoT environment* and is another Horizon 2020 project conducted between 2016 and 2018. The aim of SymbIoTe is to federate vertical IoT platforms, supporting creation of cross-domain applications over heterogeneous IoT platforms. Figure 46 shows the architecture for IoT platform federation in SymbIoTe.

SymbIoTe provides two types of federation among different IoT platforms. One is provid-

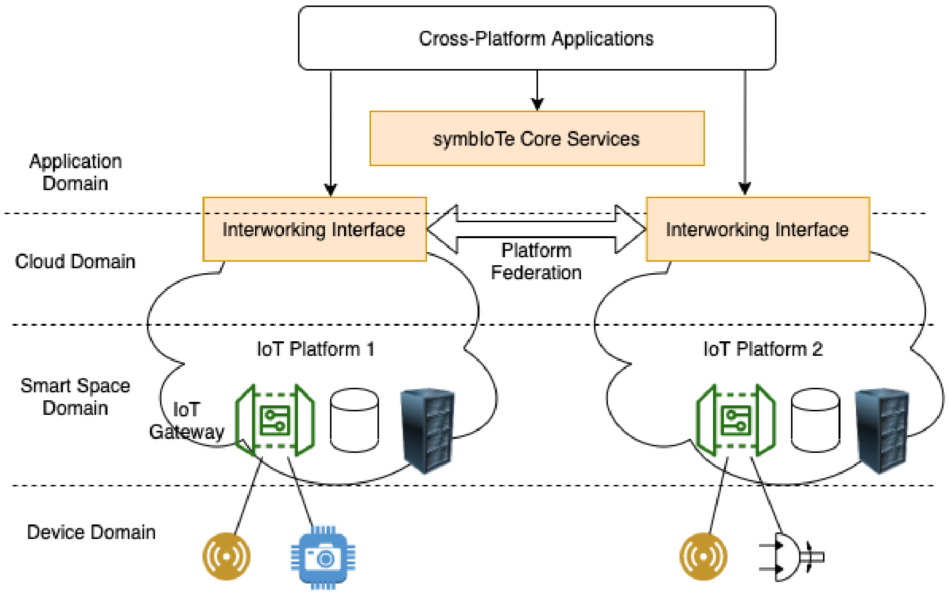


Figure 46: SymbIoTe Federation Model

ing a common interface to cross-platform applications. Then, cross-platform applications can access multiple IoT platforms with the same interface. The other is federation between IoT platforms which makes IoT devices accessible directly from other IoT platforms. The first type of federation is what is provided by VirIoT . We do not provide the second federation capability.

To facilitate the above mentioned two types of federation, SymbIoTe requires additional components in federating IoT platforms. For the first type of federation, each IoT platform needs to be equipped with a *Resource Access Proxy* (RAP) to unify the access interface. It is like ThingVisors, though there is only one RAP in a platform while ThighVisors intrinsically exist on a device-by-device basis. RAP exists in the IoT platform and ThingVisors exist in VirIoT environment though the difference is just conceptual. RAP can be placed within SymbIoTe system to keep the IoT platform intact.

Cross-platform applications in SymbIoTe need to adapt to the interface provided by SymbIoTe in order to access IoT device data attached to IoT platforms. The interface is independent from the IoT platforms where the IoT devices are attached, though. On the other hand in VirIoT , IoT applications have freedom to choose a select environment and a preferred IoT platform where the application executes, making the IoT platform executing applications independent from the IoT platform where devices are attached.

5 ThingVisor Design Technologies

How to exploit agile and innovative technologies in the field of services' development, be they spot-on centred on IoT or bordering it, is the topic of this chapter. The goal is for us to scout solutions able to shape, guide, and support the implementation and deployment of ThingVisors.

To this end, solutions providing service function chaining are valuable candidates. We can devise a ThingVisor as formed by an ordered set of tasks, composing in this way a service function chain where the last task, or even the intermediate-task output, has the goal of publishing produced data on VirIoT system topics. In what follows we present two "instruments", FogFlow and VirIoT ThingVisor Factory, that can be used as middleware to develop ThingVisors on top of the Root Data Domain.

5.1 FogFlow ThingVisor Factory

Traditional Big-Data analysis approaches follow a scheme by which all the data sources are uploaded to a single place (e.g., a cluster or a data center) where further data aggregation and processing can be carried out by different big data processing frameworks, like Apache Spark and Flink. This approach has two limitations. First, it relies on a single point to do the aggregation and processing, therefore introducing lots of network traffic, high cost, and longer delay when collecting data from various IoT devices. Second, data aggregation and processing tasks are usually programmed via a job with closed interface, to be shared only within a single application per application provider, and there is no standard data model or open interface to enable the sharing across applications and providers.

FogFlow is an open source fog computing framework, initiated by NEC, in order to address the limitations of existing big data processing frameworks, especially in the context of IoT. It supports not only the new fog computing paradigm to move data processing close to data sources, but also the sharing and interoperability of data aggregation and processing tasks across applications and providers, via the widely used standard data model and open interface NGSI/NGSI-LD. Furthermore, FogFlow can launch data processing tasks seamlessly over cloud and edges with an optimized deployment in terms of where to deploy each task and how much resources to be assigned to each task.

The high-level system architecture of FogFlow is illustrated in Figure 47. The figure includes the FogFlow framework, *geo-distributed infrastructure resources*, and FogFlow's connection with the users (system operator and service developers) and external applications through its API and interfaces. Infrastructure resources are vertically divided as *cloud*, *edge nodes*, and *devices*. Computationally intensive tasks such as big data analytics can be performed on the cloud servers, while some tasks such as stream processing can be effectively moved to the edge nodes (e.g., IoT gateways or endpoint devices with computation capabilities). Devices may include both computation and communication capabilities (e.g., tablet computer) or only one of them (e.g., beacon nodes advertising Bluetooth signals). The FogFlow framework operates on these geo-distributed, hierarchical, and heterogeneous resources, with three logically separated divisions: *service management*, *data processing*, and *context management*. We have highlighted these three divisions in the red box of Figure 47. As we can see there are:

- Service management: it includes task designer, topology master (TM), and docker image repository, which are typically deployed in the cloud. Task designer provides the

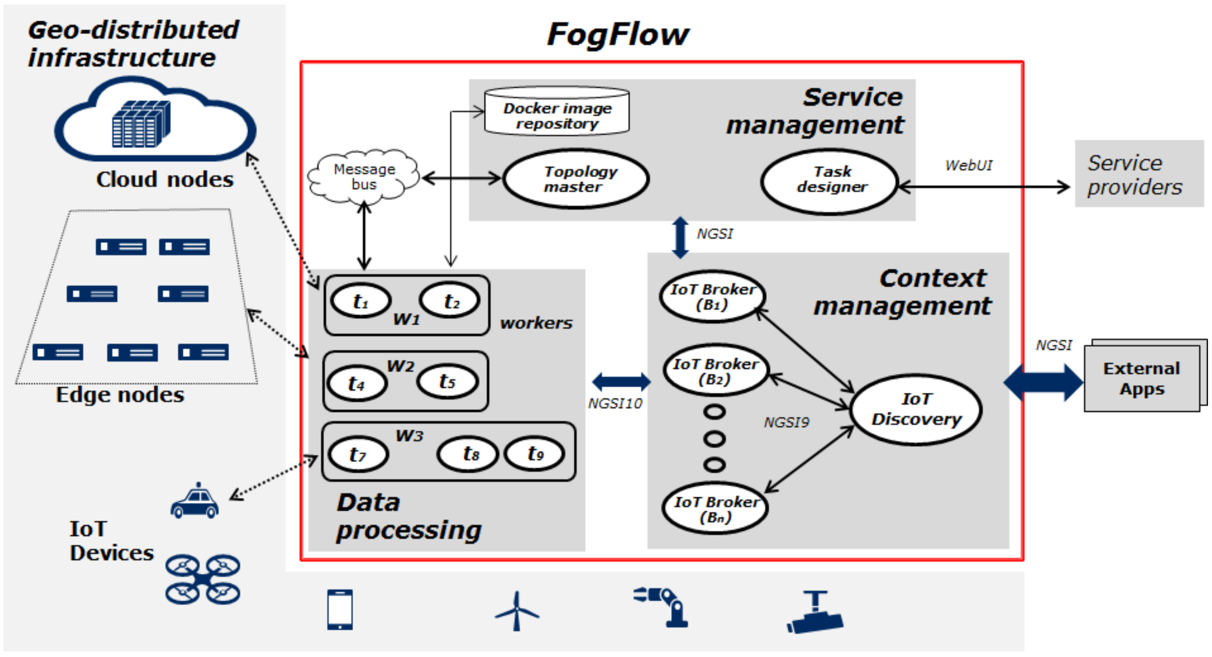


Figure 47: High-level view of FogFlow architecture

web-based interfaces for the system operators to monitor and manage all deployed IoT services and for the developers to design and submit their specific services. Docker image repository manages the docker images of all dockerized operators submitted by the developers. TM is responsible for service orchestration, meaning that it can translate a service requirement and the processing topology into a concrete task deployment plan that determines which task to place at which worker.

- Data processing: it consists of a set of workers $(w_1, w_2, \cdots, w_m)$ to perform data processing tasks assigned by TM. A worker is associated with a computation resource in the cloud or on an edge node. Each worker can launch multiple tasks based on the underlying docker engine and the operator images fetched from the remote docker image repository. The number of supported tasks is limited by the computation capability of the compute node. The internal communication between TM and the workers is handled via a Advanced Message Queuing Protocol (AMQP)-based message bus such as RabbitMQ to achieve high throughput and low latency.

- Context Management: it includes a set of *IoT Brokers*, a centralized *IoT Discovery*, and a *Federated Broker*. These components establish the data flow across the tasks via NGSI and also manage the system contextual data, such as the availability information of the workers, topologies, tasks, and generated data streams. IoT Discovery handles registration of context entities and discovery of them. This component is usually deployed in the cloud. IoT Brokers are responsible for caching the latest view of all entity objects and also serving context updates, queries, and subscriptions. In terms of deployment, IoT Brokers are distributed on the different nodes in the cloud and on the edges. They are also connected to the other two divisions (workers, task designer, TM, external applications) via NGSI.

In FogFlow, each service includes three key elements: *operator*, *service topology*, and *intent*. An operator represents a type of data processing unit, for example, calculating the average temperature, performance the face matching of two face images. A service topology represents the computation logic of the IoT service and consists of several linked operators annotated with their data inputs and outputs. An intent is a JSON object that follows the proposed intent model to define a customized requirement of how the service topology should be triggered.

To trigger the service topology in FogFlow, service consumers need to define an intent to express their high-level goal from the following perspectives: 1) *service topology* that defines which service logic to be triggered; 2) *geoscope* that defines the scope to select the input data for applying the selected service topology; 3) *service level objective (SLO)* that defines the service level objective to be achieved, in terms of latency requirements, bandwidth saving, or privacy/security needs; 4) *priority* that defines how the triggered service deployment could utilize the shared infrastructure resources with the other existing services.

FogFlow is used to realize advanced ThingVisors in a flexible and efficient manner. As illustrated in Figure 48, the data processing logic of each ThingVisor can be programmed as a FogFlow service using the intent-based programming model in FogFlow. The FogFlow service consists of one or more than one dockerized functions, which can be dynamically configured and deployed over cloud or edges in FogFlow. For each type of ThingVisor, its corresponding FogFlow service can fetch data from the Root Data Domain and then create the up-to-date representation of Virtual Things via some data processing tasks.

The FogFlow-ThingVisor in Fed4IoT is an advanced ThingVisor that performs two types of interactions on requests:

- On receiving the request of adding, updating, or removing ThingVisor, the FogFlow-ThingVisor will interact with the running FogFlow system to dynamically start or stop a pre-defined FogFlow service by sending a customized intent object accordingly and then use the received notification from FogFlow to create or update Virtual Things.

- Simmetrically, the FogFlow-ThingVisor will respond to the request of adding or removing Virtual Things, deciding which newly created Virtual Things are to be announced to the Fed4IoT pub/sub system, so that the created Virtual Things can be used by Fed4IoT applications within different Virtual Silos.

5.2 VirIoT ThingVisor Factory

In addition to the FogFlow ThingVisor Factory, which helps creating ThingVisors that run in a FogFlow environment, we are developing another ThingVisor Factory, VirIoT ThingVisor Factory, which is designed for, and works with, the kubernetes environment. The VirIoT ThingVisor Factory is a platform that can provide functionalities for designing, developing, and deploying ThingVisors on-demand for IoT service/application developers, or "tenants". Through VirIoT ThingVisor Factory, developers can interactively design and develop their own ThingVisors, including *private ThingVisors* that produce tenant-specific Virtual Things, such as a face detection ThingVisor for a specific person. In addition, developers need not pay attention to installation of IoT devices as long as they exist in the VirIoT environment, and to deployment of required data processing functionalities over the Internet.

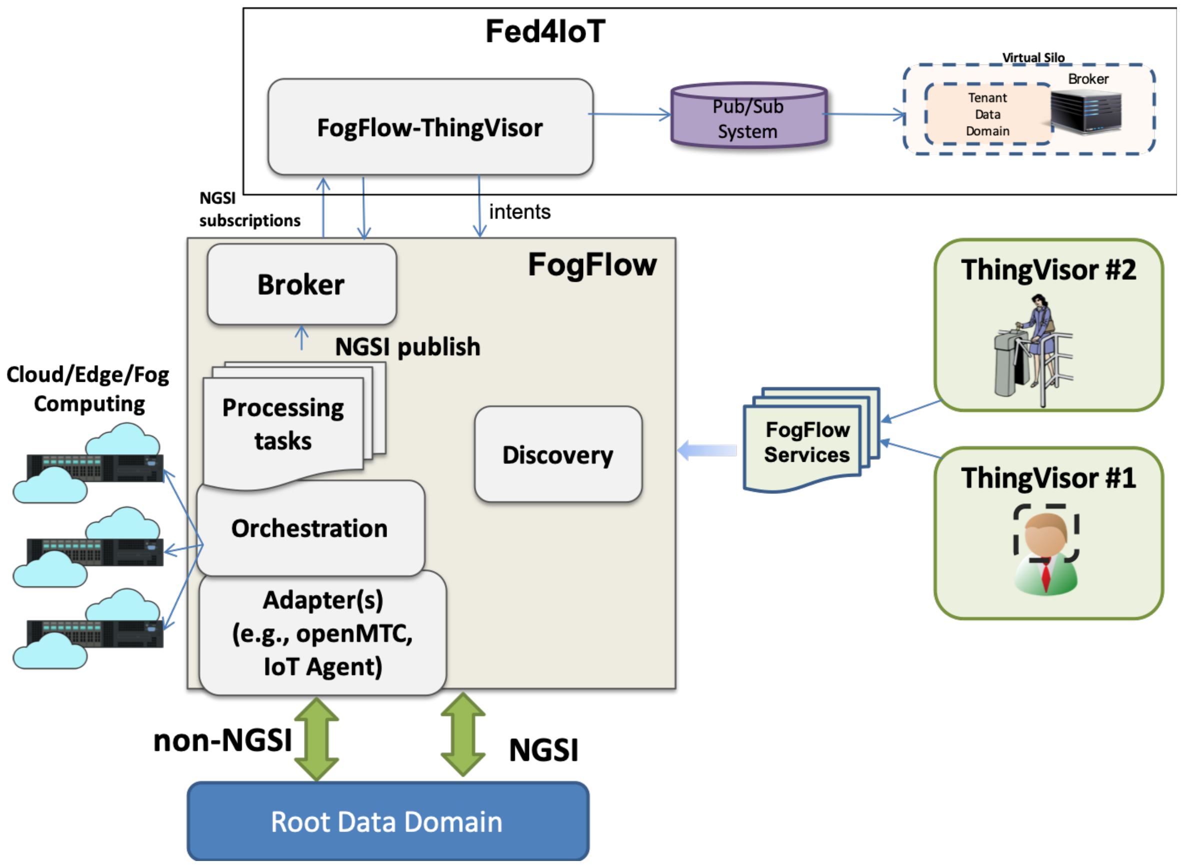


Figure 48: Integration of FogFlow with other components

The concept of our VirIoT ThingVisor Factory is illustrated in Figure 49, which shows a ThingVisor Factory providing a private ThingVisor that detects Alice's face in a picture taken by a fixed camera.

The ThingVisor detecting Alice's face is a private one because such a ThingVisor needs to provide access policy (and control), and it has to guarantee data protection.

In another use case, ThingVisor Factory provides another on-demand ThingVisor that produces new Virtual Things by chaining ThingVisors, i.e. by utilizing service function chaining (SFC). For instance, on-demand ThingVisor #3 in Figure 49 produces new Virtual Thing as "flying surveillance camera" by chaining ThingVisor #3, which is a copy of on-demand ThingVisor #1, and Drone Control (pre-deployed ThingVisor #2) in the system.

One of the challenging issues in ThingVisor Factory is how to deal with diverse requirements from IoT application providers, by satisfying their networking and computing demand in a user-friendly way, e.g., autonomous, instinctive, and interactive way. To address this issue, VirIoT ThingVisor Factory is equipped with two functionalities:

- Dataflow-programming-based graphical user interface (GUI).
- Service function chaining-based ThingVisor construction.

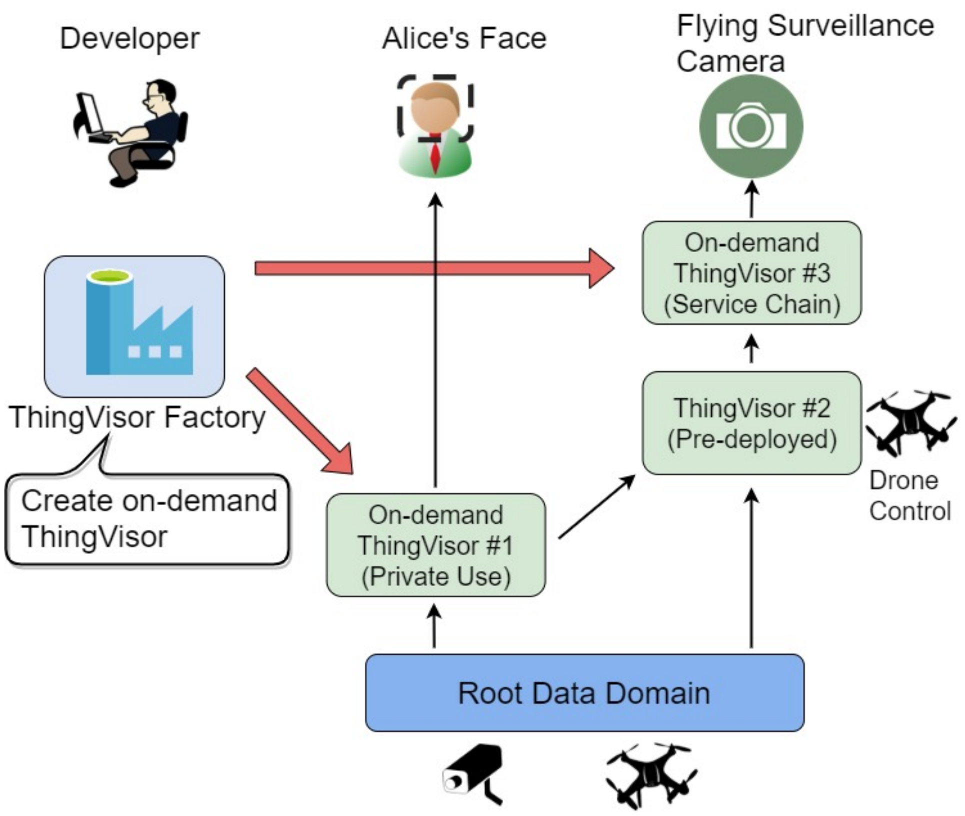

Developer
Alice's Face
Flying Surveillance Camera
On-demand ThingVisor #3 (Service Chain)
ThingVisor Factory
Create on-demand ThingVisor
ThingVisor #2 (Pre-deployed)
Drone Control
On-demand ThingVisor #1 (Private Use)
Root Data Domain

Figure 49: Concept of VirIoT ThingVisor Factory

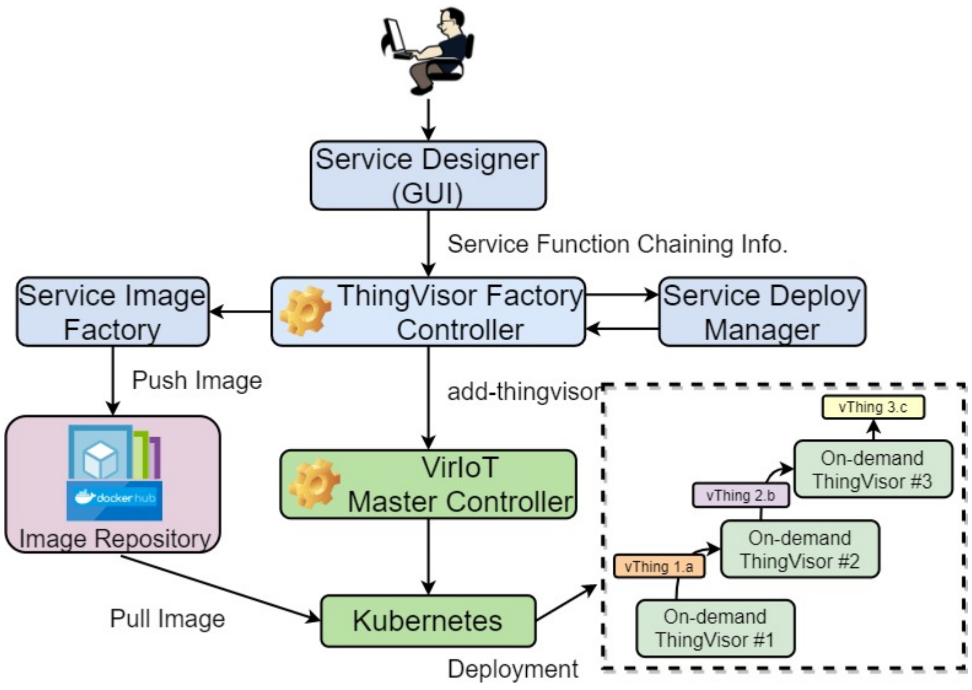


Figure 50: Architecture of VirIoT ThingVisor Factory

5.2.1 Architecture of VirIoT ThingVisor Factory

The architecture of VirIoT 's ThingVisor Factory is shown in Figure 50, where red function blocks represent the components of VirIoT ThingVisor Factory and blue function blocks are represent components of VirIoT system. As shown, VirIoT ThingVisor Factory provides modules that complement VirIoT functionality, and it supports the implementation and deployment of a chain of ThingVisors on demand. In order to provide a user-friendly platform and autonomous management, ThingVisor Factory mainly composes three key functionalities: Service Designer, Service Image Factory and Service Deploy Manager.

Through the VirIoT ThingVisor Factory, IoT service/application developers, or tenants, can develop their own ThingVisors. This operation is done from the GUI provided by Service Designer. In VirIoT ThingVisor Factory, Virtual Things are defined as outputs from a chain of service functions created by dataflow programming. After the developer completes designing ThingVisors and their chains, Service Designer outputs JSON serialized service function chaining information corresponding to the designed chains of ThingVisors. The service function chaining information indicates a specification of a ThingVisor chain and is composed of the information of required service functions, or ThingVisors, and connectivity among the ThingVisors. Based on the service function chaining information, Service Image Factory creates Docker Images for required ThingVisors and pushes those images to a ThingVisor repository, such as Docker Hub. Meanwhile, Service Deploy Manager determines the deployment plan of dockerized service functions, by considering networking and computing conditions of network nodes (e.g., Kubernetes nodes).

These functionalities are operated with APIs provided by ThingVisor Factory Controller. After all preparation is done (i.e, service functions are stored on Docker Hub, and deployment plan of them are defined), ThingVisor Factory Controller invokes an “add ThingVisor” command on the VirIoT Master-Controller in order to deploy the ThingVisor on the platform.

5.2.2 Communication Protocols for Service Function Chaining

Before introducing the key component of ThingVisor Factory, this subsection summarizes communication protocols for service function chaining. Service function chaining is one of technologies regarding Software Defined Networking (SDN) and Network Function Virtualization (NFV). Because the service function chaining can be realized by in-network processing, it is possible for network operators to manage networking and computing resources efficiently and flexibly. While the service function chaining is also one of promising solutions in order to realize Thing Virtualization in the Fed4IoT VirIoT system, operation and management of service function chaining heavily depends on communication protocols. As described in deliverable D3.1, in order to realize the service function chaining, there are three candidates of communication protocols: P2P over IP, Pub/Sub over IP and ICN.

- P2P over IP: In this model, the service function chaining is operated in SDN manner, like OpenFlow. The service functions (or virtualized network functions) are identified by using specific tags called Network Service Header (NSH) tags. The NSH tag is embed in IP packet header, and IP routers forward the IP packets according to the NSH tag. The architecture of this service function chaining is standardized as IETF RFC7665 [12] and RFC8300 [13].
- Pub/Sub: In this model, unlike the previous one, the routing for the service function chaining is handled on the application layer. The service functions are identified by topic names for Pub/Sub communications, such as MQTT and Apache kafka. The service function subscribes a specific topic name to receive a required data and publishes an alter topic name to produce a processed data. In this model, because the routing can be managed on the application layer, it is easy to implement, but, the load balancing of Pub/Sub Broker is an important issue.
- ICN: In this model, the communication model is completely different from the previous two models. Because ICN routing is ideally resolved by not IP address but content name, the communication model indicates request-response-type communication. ICN service function chaining is required to the management of routing table called forwarding information base (FIB) on ICN router layer like P2P base and the management of interest name on the application layer like Pub/Sub base. Although, ICN service function chaining is a challenging topic, the ICN capabilities, such as in-network caching and in-network processing, potentially provides more efficient and flexible operation compared to the other two models. Detailed description of route resolution of ICN service function chaining is represented in Service Deploy Manager section.

In the next three Sections, we present here the detailed description of three key components of VirIoT ThingVisor Factory.

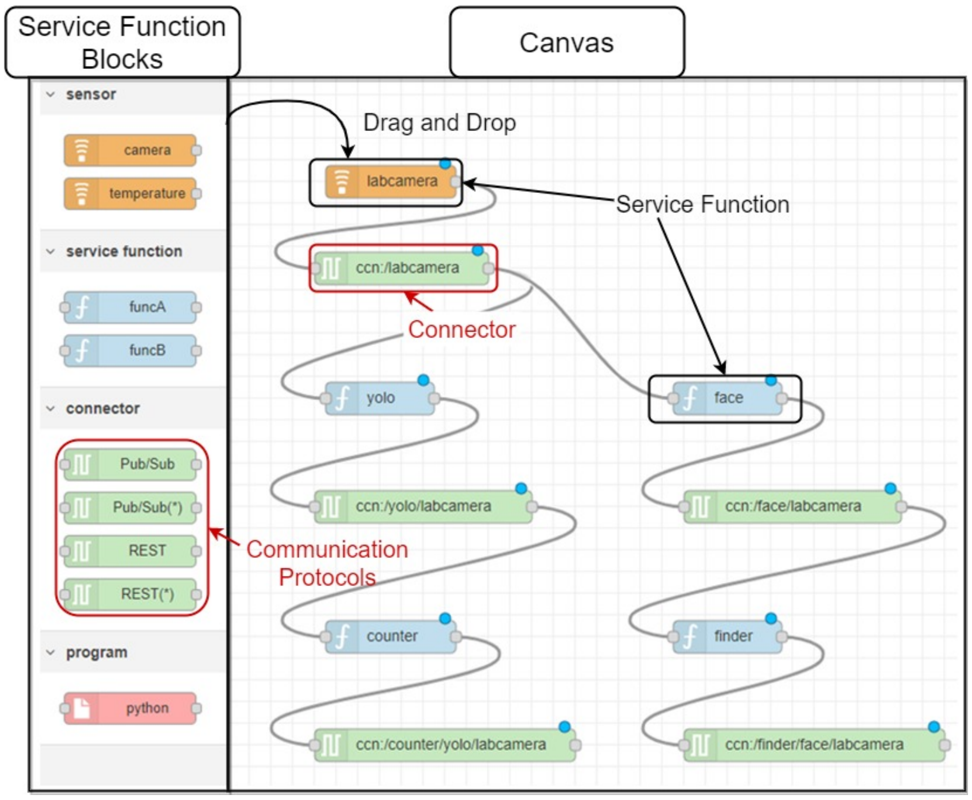
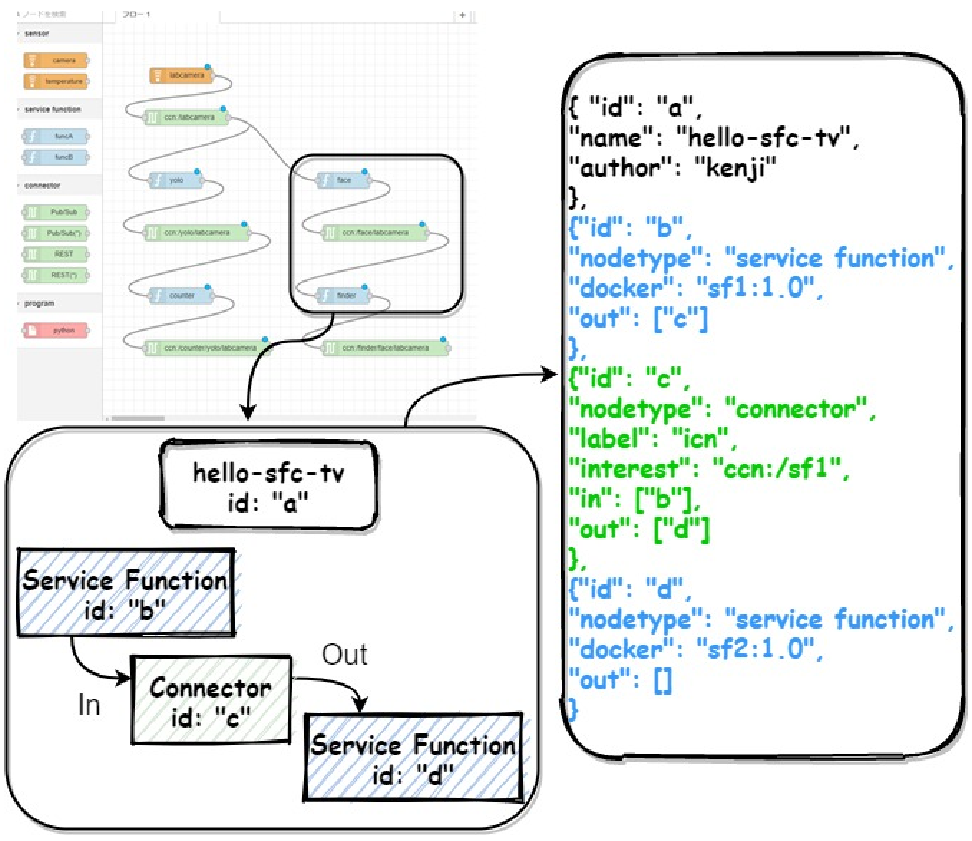


Figure 51: GUI of Service Designer

5.2.3 Service Designer

The first key component of our VirIoT ThingVisor Factory is the Service Designer. Because ThingVisor Factory is required to develop on-demand ThingVisor instinctively and interactively, Service Designer provides dataflow programming-based GUI as shown in Figure 51. The Service Designer abstracts the common functionalities as *service function blocks* and visualizes as colorful *blocks.* Service Designer is similar to (and based on) NodeRED, but, unlike NodeRED, Service Designer can specify the network connectivity between service function blocks. More specifically, the developer can specify the communication protocols for service function chaining, such as P2P, Pub/Sub and ICN as mentioned in the previous subsection. This is the biggest different aspect against NodeRED, and ThingVisor Factory aims at network-wide deployment (and operation) of service functions while NodeRED can only deploy (and operate) on the local NodeRED environment.

In order to design ThingVisor instinctively and interactively, Service Designer abstracts the typical service functions as blocks. Service function blocks compose “sensor,” “service function,” “connector,” and “program”. First, a sensor block provides a functionality of retrieving Real Things and Virtual Things such as temperature values, surveillance camera images or video. The sensor block contains the information of how to access Thing (e.g., API, broker information, and protocol), Thing ID and meta data of Thing (e.g., geolocation). Second, a service function block provides a functionality of IoT data processing, such as statistic analysis and image processing. This block contains the information regarding data processing engine, such as input



```
{ "id": "a",
"name": "hello-sfc-tv",
"author": "kenji"
},
{"id": "b",
"nodetype": "service function",
"docker": "sf1:1.0",
"out": ["c"]
},
{"id": "c",
"nodetype": "connector",
"label": "icn",
"interest": "ccn:/sf1",
"in": ["b"],
"out": ["d"]
},
{"id": "d",
"nodetype": "service function",
"docker": "sf2:1.0",
"out": []
}
```

Figure 52: Example of JSON-Serialized Service Function Chaining Information

and output data formats (or types) and docker image name. It also includes meta data of service function (e.g., description of service function and authors name). Next, a connector block, which is one of important blocks, provides a definition of network connectivity between blocks. This block contains the information of communication protocols, such as Pub/Sub or ICN, and it also contains the information of input and output functions in order to describe the chaining operation of service functions. At this moment, the developer can specify topic name or interest name to publish Virtual Thing when the developer select Pub/Sub or ICN as a communication protocol. Finally, a program block provides a functionality where the developer can write his/her own functionality in specific programming languages, such as Python, similar to NodeRED (e.g., NodeRED allows programmers to write programs in JavaScript).

By drag and drop operations, the developers can select the service functions, and by connecting the blocks with lines, the developers can define the service functions as service function chainings. Thus, Service Designer can provide simple, instinctive and interactive platform to design on-demand ThingVisors. After the developer designs ThingVisor, Service Designer generates and outputs JSON serialized service function chaining information based on the required blocks as shown in Figure 52. By using the service function chaining information, ThingVisor Factory prepares docker images regarding the required service functions and determines the deployment plan, including routing resolution, in case of ICN protocol.

5.2.4 Service Image Factory

The second key component of the ThingVisor Factory is the Service Image Factory. Service Image Factory mainly provides a functionality of automatic dockerization of service functions if necessary. First, Service Image Factory parses the service function chaining information produced by Service Designer and figures out whether the service functions need to be dockertized or not. The simplest case is that the developer only selects pre-defined (or pre-developed) service functions. In such case, service functions, including communication protocol, have already dockerized, and Service Image Factory just modifies operations of service functions by only changing a configuration level, such as changing the names of topics, transmission interval or other parameters regarding data processing. When the developers request to compose their own service functions which specified by program block in Service Designer, Service Image Factory parses their requests and generates docker images, including network functionality (i.e., communication protocol), according to their source codes. In order to simplify docker image creation, Service Image Factory provides a template of service function. The template guides the developers to program service functions which are certainly dockerized and executed. The template also contains the functions of communication protocols.

After Service Image Factory prepares docker images regarding service functions, Service Image Factory pushes the docker images to ThingVisor repository (e.g., Docker Hub) in order to be able to be pulled from VirIoT master controller.

5.2.5 Service Deploy Manager

The third key component of the ThingVisor Factory is the Service Deploy Manager. In order to deploy dockerized ThingVisor autonomously and construct service function chaining, Service Deploy Manager mainly provides two functionalities: determination of optimal deployment plan and route resolution for ICN. In the determination of optimal deployment plan, as presented the detailed description in Deliverable 3.1, Service Deploy Manager derives optimized network nodes (e.g., Kubernetes nodes) where the dockerized ThingVisors are deployed. In the operation, Service Deploy Manager considers not only networking and computing resource usages of the network nodes, but also physical locations of the network nodes (i.e., Japan or European and edge or cloud). This optimization is modeled as a workflow scheduling problem and implemented as a workflow engine.

In the route resolution for ICN, this functionality is required only for ICN usage case. ICN is one of good candidates of communication protocols for realizing service function chaining. Unlike TCP/IP, including Pub/Sub model over IP protocol, ICN requires to solve networking routes by using not IP table but forwarding information base (FIB). in ICN-based service function chaining, (autonomous) FIB management is one of challenging issue. To simplify the FIB management, ThingVisor Factory adopts a centralized management approach by referring to OpenFlow controller.

Before introducing FIB management, a determination method of FIB is presented. FIB mainly composes (content) name prefix and upstream faces. To determine (or build) FIB in advance, (ideally) ICN routers need to know correct upstream faces corresponding to name prefixes. In general, it is quite difficult to know such information in advance, however, ThingVisor Factory can determine the correct upstream faces and name prefixes regarding service function chaining as shown in Figure 53. As shown in the figure, this is because, at first, through Service

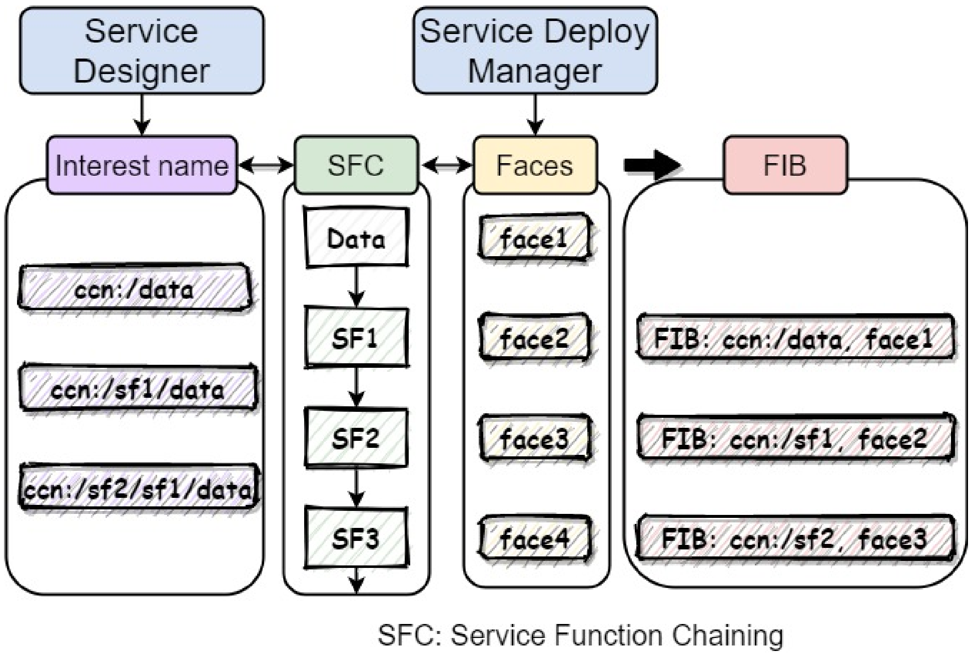


Figure 53: FIB Construction in VirIoT ThingVisor Factory

Designer, the developer designs the service function chaining and specifies the content names corresponding to the service functions. In other words, ThingVisor Factory can know the exact interest names transmitted by the service functions in order to retrieve contents. In addition, after the service function chaining is designed, Service Deploy Manager determines the deployment nodes, and this means that ThingVisor Factory can know the exact faces corresponding to the content names produced by the service functions. Thus, ThingVisor Factory can determine FIB from the specification of service function chaining in advance.

After ThingVisor Factory figures out the determination of FIB, ThingVisor Factory needs to populate such FIB information to ICN routers. As mentioned before, by referring architecture of OpenFlow, ThingVisor Factory adopts centralized management approach. In other words, ThingVisor Factory controller distributes the FIB information to ICN routers. A simple architecture of FIB distribution is shown in Figure 54. As shown in the figure, there are two strategies: ICN/IP Hybrid case and pure ICN case. In ICN/Hybrid case, data plane is operated over ICN protocol and control plane is operated over IP protocol. More specifically, FIB distribution can be handled on the control plane, and FIB information multicasts with a topic specified by the Pub/Sub communication. This approach is easy to implement, but control traffic may become large. On the other hand, in pure ICN case, both data and control planes are operated over ICN protocol, and FIB information is distributed by the ICN manner (e.g., exchange Interest and Data packets). This approach requires more complex implementation such as FIB management for FIB distribution. In addition, realization of push delivery over ICN may be required.

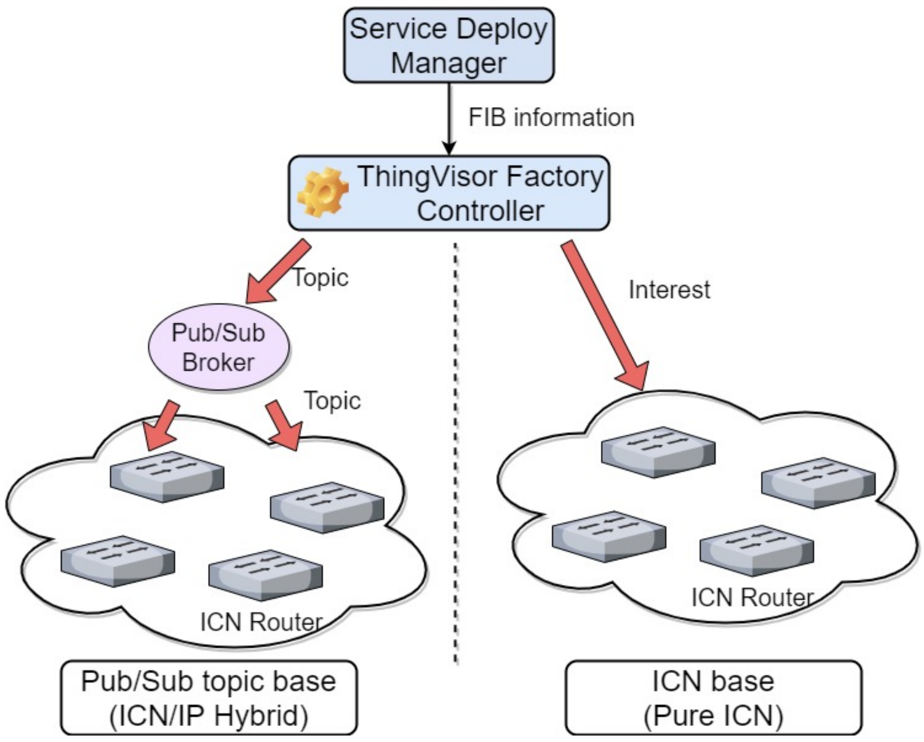


Figure 54: Architecture for FIB Information Distribution

6 Conclusions

Current IoT cloud solutions, usually, offer virtual data hubs for collecting, analyzing and distributing information coming from things owned by the user. In this deliverable we presented VirIoT , the Fed4IoT platform that has a set of different and distinctive virtualization goals:

- to offer **Virtual Things** to users who lack them, alleviating developers of the burden of buying and deploying IoT devices and services needed by their applications. Virtual Things are a smart proxy to IoT sensors or actuators, capable of complex interaction with real things or other producers/consumers of information;
- to provide access to the data items managed by the Virtual Things through existing, standard IoT broker technologies, among which the developers can freely choose. Therefore, our goal is to solve **IoT interoperability and virtualization** issues in a single platform, without requiring external actors to adapt their streams, environment or interfaces to a given solution. We do this without introducing a new IoT standard or layer;
- to provide users with IoT systems-as-a-service, or **Virtual Silos**: isolated environments made of Virtual Things and an IoT broker.

We think that such a concept of IoT virtualization is not much explored in the IoT (cloud) arena, thus we suggest this project is a step forward in a stimulating and promising direction.

References

[1] W. Lumpkins, "The internet of things meets cloud computing [standards corner]," *IEEE Consumer Electronics Magazine*, 2013.

[2] S. K. Datta, A. Gyrard, C. Bonnet, and K. Boudaoud, "oneM2M architecture based user centric IoT application development," in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on.* IEEE, 2015.

[3] H. Park, H. Kim, H. Joo, and J. Song, "Recent advancements in the Internet-of-Things related standards: A oneM2M perspective," *ICT Express*, 2016.

[4] FIWARE home page. [Online]. Available: https://www.fiware.org/

[5] ETSI GS CIM 009. Context Information Management (CIM): NGSI-LD API. [Online]. Available: https://docbox.etsi.org/ISG/CIM/Open

[6] OMA, Open Mobile AllianceTM. [Online]. Available: http://www.openmobilealliance.org

[7] Mobius IoT Server Platform. [Online]. Available: http://developers.iotocean.org/archives/module/mobius

[8] A. Glikson, "Fi-ware: Core platform for future internet applications," in *Proceedings of the 4th annual international conference on systems and storage*, 2011.

[9] M. Bauer, E. Kovacs, A. Schülke, N. Ito, C. Criminisi, L.-W. Goix, and M. Valla, "The context API in the OMA next generation service interface," in *Intelligence in Next Generation Networks (ICIN), 2010 14th International Conference on.* IEEE, 2010.

[10] B. Cheng, G. Solmaz, F. Cirillo, E. Kovacs, K. Terasawa, and A. Kitazawa, "Fogflow: Easy programming of iot services over cloud and edges for smart cities," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 696–707, 2018.

[11] A. F. Skarmeta, J. Santa, J. A. Martínez, J. X. Parreira, P. Barnaghi, S. Enshaeifar, M. J. Beliatis, M. A. Presser, T. Iggena, M. Fischer *et al.*, "Iotcrawler: Browsing the internet of things," in *2018 Global Internet of Things Summit (GIoTS).* IEEE, 2018, pp. 1–6.

[12] "Ietf rfc 7665 service function chaining (sfc) architecture," 2015.

[13] "Ietf rfc 8300 network service header (nsh)," 2018.