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 D2.2
# System Architecture - First Release

| | |
|---|---|
| **Deliverable Type:** | Report |
| **Deliverable Number:** | D2.2 |
| **Contractual Date of Delivery to the EU:** | 31/03/2019 |
| **Actual Date of Delivery to the EU:** | 31/03/2019 |
| **Title of Deliverable:** | System Architecture - First Release |
| **Work package contributing to the Deliverable:** | WP2 |
| **Dissemination Level:** | Public |
| **Editor:** | Andrea Detti (CNIT), Hidenori Nakazato (WAS) |
| **Author(s):** | Andrea Detti, Giuseppe Tropea (CNIT); Juan A. Martinez, Antonio F. Skarmeta (OdinS); Martin Bauer, Bin Cheng (NEC); Frank Le Gall (EGM); Hidenori Nakazato (WAS); Kenichi Nakamura (PAN); Tetsuya Yokotani (KIT) |
| **Internal Reviewer(s):** | Nicola Blefari Melazzi (CNIT) |
| **Abstract:** | The deliverable reports the first release of Fed4IoT system architecture |
| **Keyword List:** | IoT Virtualization, IoT Brokers, NGSI-LD, ICN |

# Disclaimer

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

# Table of Contents

# 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 |
| 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 |
| 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 the terms that have been considered relevant in 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 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. a "Mobius flavour" is related to a virtual silo with Mobius broker, a "MQTT flavour" refers to a virtual silo with 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 access the Fed4IoT VirIoT platform to develop IoT applications |

Table 2: Fed4IoT Dictionary

# 1 Introduction

## 1.1 Deliverable Rationale

This deliverable reports the first version of the Fed4IoT system architecture. It is centred on our concept of a virtualization platform for IoT. We also start to draw plans for possible implementations of the components of the architecture, as well as investigating alternatives.

## 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 | Initial Version | Andrea Detti, Giuseppe Tropea (CNIT), Juan A. Martinez, Antonio F. Skarmeta (OdinS), Hidenori Nakazato (WAS), Kenichi Nakamura (PAN), Tetsuya Yokotani (KIT) | 01/03/2019 |
| 0.2 | Section 2.3.1 and Section 6, Update Section 2.2.1 and 2.2.2, checked Section 5.1 | Martin Bauer, Bin Cheng (NEC) | 14/03/2019 |
| 0.3 | Section 2.2 and 2.3 | Juan A. Sanchez, Juan A. Martinez and Antonio Skarmeta (OdinS) | 14/03/2019 |
| 0.4 | Section 5 | Juan A. Sanchez, Juan A. Martinez and Antonio Skarmeta (OdinS) | 15/03/2019 |
| 0.5 | Update Section 2.2.1 and 6.4 | Frank Le Gall (EGM) | 15/03/2019 |
| 0.6 | Section 6.5 and Chapter 6 revision | Giuseppe Tropea (CNIT) | 21/03/2019 |
| 0.7 | Section 2.1 | Giuseppe Tropea (CNIT) | 22/03/2019 |
| 0.8 | Introduction e overall revision | Giuseppe Tropea (CNIT) | 23/03/2019 |
| 0.9 | Editing work | Andrea Detti (CNIT), Hidenori Nakazato (WAS) | 25/03/2019 |
| 1.0 | Quality review | Nicola Blefari Melazzi (CNIT) | 28/03/2019 |
| 1.1 | Final review | Andrea Detti (CNIT) | 31/03/2019 |

Table 3: Version Control Table

## 1.3 Executive summary

### 1.3.1 Deliverable description

This deliverable reports the first version of the Fed4IoT system architecture, and it also includes the description of possible development frameworks. 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 and FiWARE solutions), and finally we report on related projects. In section 3, we introduce some fundamental concepts of our architecture, named *VirIoT*, while in section 4 we describe the components of the architecture, their roles and relationships, in detail. 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. In section 6, we describe the NGSI-LD data model, its APIs and the ongoing standardization process carried out within the ETSI ISG named CIM, which our project (among other stakeholders) is contributing to. Indeed, the NGSI-LD data format has been chosen as the neutral data format to be used inside our architecture; in section 6.5 we describe in detail how our platform is going to exploit NGSI-LD as the neutral format and how we are going to implement mappers to translate from existing formats to the neutral format, and back; moreover, the NGSI-LD API and Broker will have a central role in the second release of the architecture, enabling support of semantic capabilities, as briefly described in section 7, which outlooks our plan for next release/second iteration at the architecture.

### 1.3.2 Summary of results

This deliverable presents the first version of the Fed4IoT system architecture, named VirIoT , which enables virtualization of IoT systems, formed by virtual things and brokers.

Our goal is to decouple developers of IoT applications from providers of things. VirIoT allows owners of IoT heterogeneous infrastructures to share them with many IoT application developers, which can simply rent the virtual things and the brokers their applications need. As described in the use-case deliverable D2.1, VirIoT can be useful for small stakeholders whose applications require large-scale IoT infrastructures, who are otherwise unable to handle the infrastructure deployment. 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.

# 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 1) 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 extends 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.

Though all of the IoT cloud providers have similar levels of functionality and en-

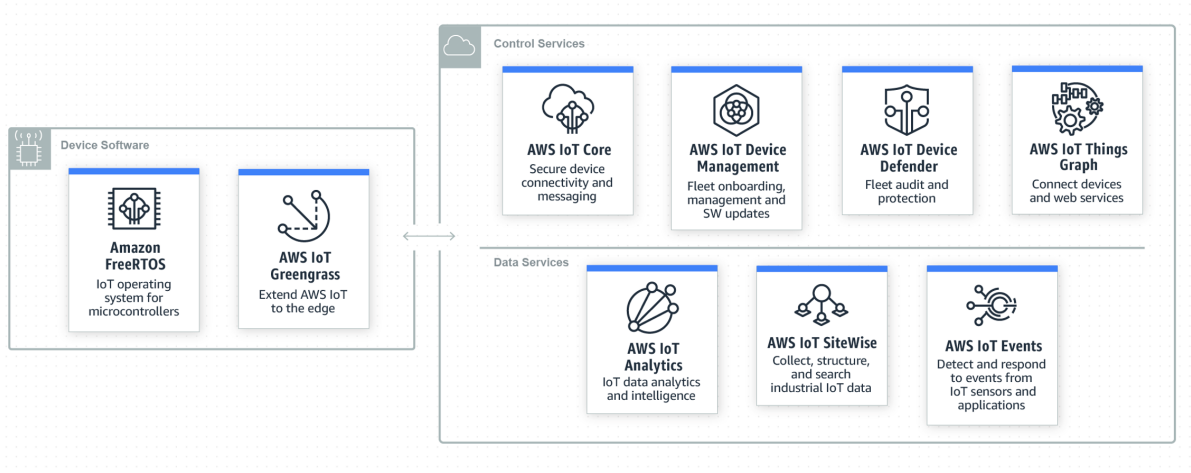


Figure 1: Amazon's AWS IoT ecosystem

terprise 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

While the platforms above mentioned 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. 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. In this sense, the VirIoT services are complementary to most of the existing solutions and can interoperate with them in an extended IoT chain (see bottom-right of figure 10).

From another perspective, Fed4IoT VirIoT system wants to push forward the role of IoT cloud providers. They should make the move towards acquiring (control of) the ever growing number of devices out there in the field, and by virtualizing them they should supply a scalable layer of horizontally share-able IoT resources to their customers, making them 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.

## 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 [6]. NGSI-LD is actually an evolution of NGSI specifications [5] 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 6 we present NGSI-LD, whose specification processes are supported by 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 is formed *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. 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. 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 usually run in ADNs and interact with the platform through a Common Service Entity (CSE), running in either in a 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 2. 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 .

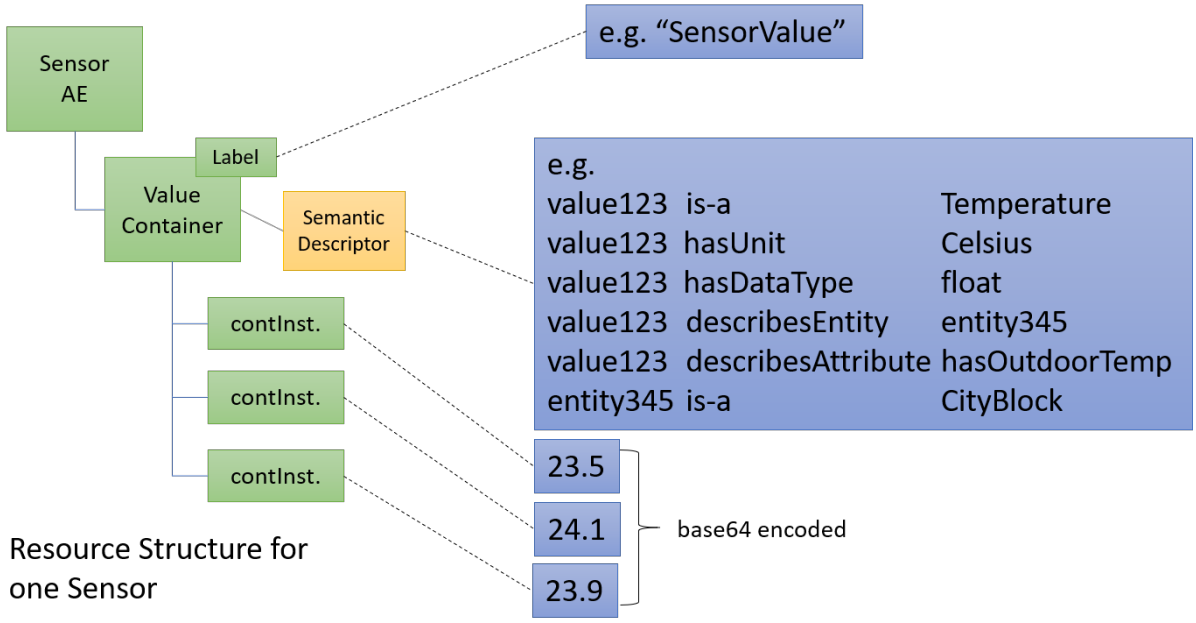


Figure 2: oneM2M resrouce tree

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 means in 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 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) [5, 9]. Additionally, thanks to the work of ETSI ISG CIM workgroup [6], 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 6, to thoroughly describe it. NEC is currently implementing its NGSI-LD Broker (Section **??**) with the intention of replacing the Aeron IoT Broker. In parallel, there exists an extension to ORION called ORION-LD[3] 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 3 shows the interactions between these components.

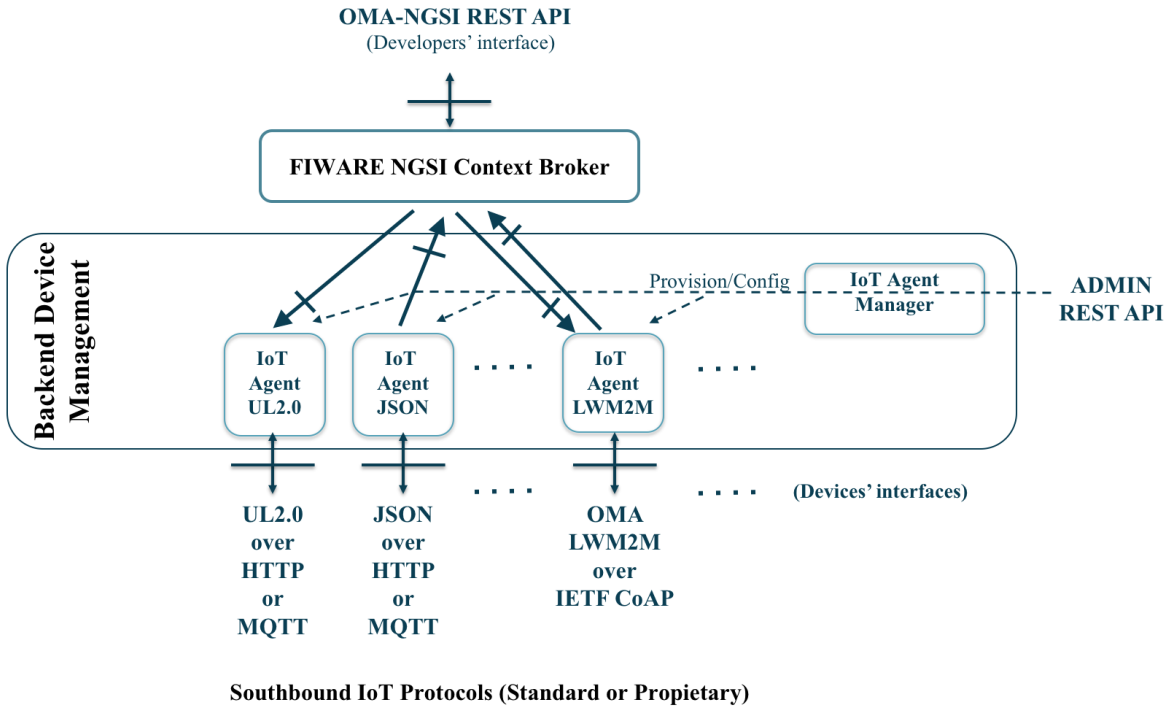


Figure 3: FIWARE-Diagram

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

## 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 NGSI-LD Broker which is becoming 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 4 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.

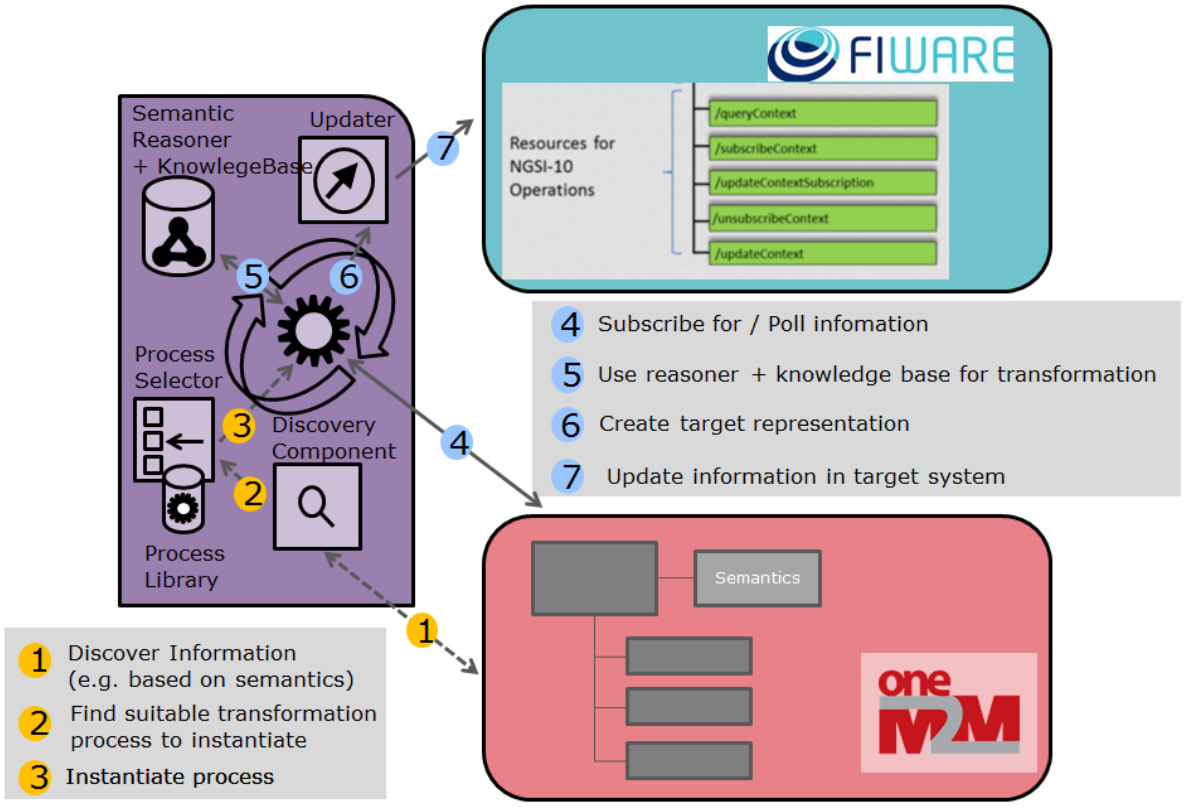


Figure 4: Adaptive Semantic Module of Morphing Mediation Gateway

### 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 new FIWARE GE (Generic Enabler) called FogFlow [13]. A framework for dynamically orchestrate 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 5 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.

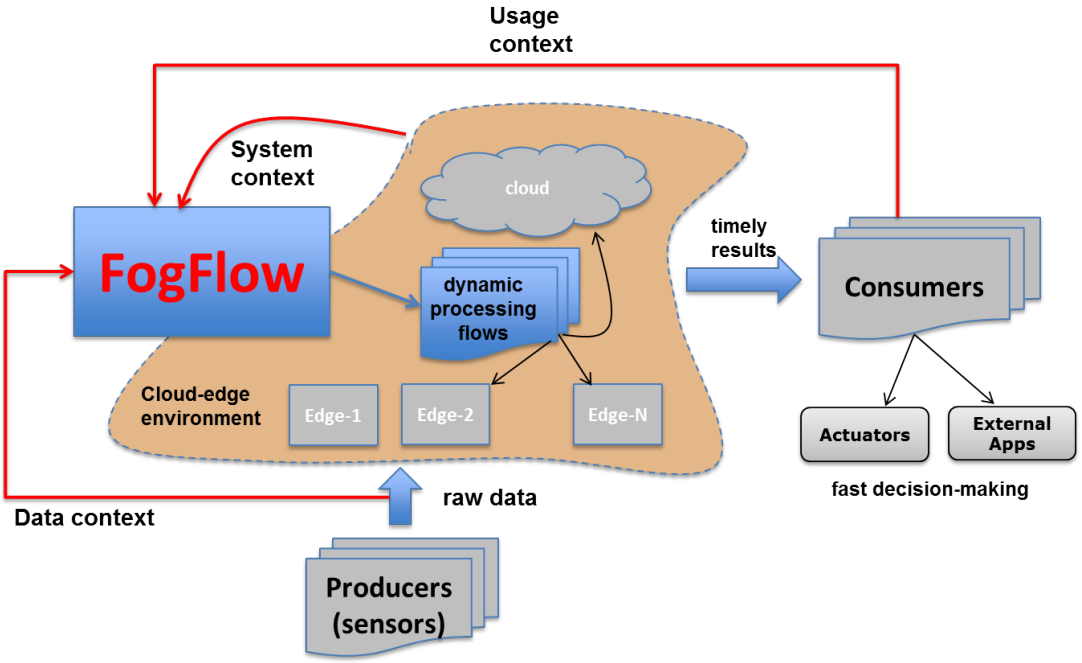


Figure 5: High-level view of FogFlow framework

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 plans to reuse and in case evolves FogFlow as a development framework for ThingVisors, as detailed in section 5.

### 2.3.3 IoT-Crawler

IoTCrawler [14], (`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 6, 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.

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

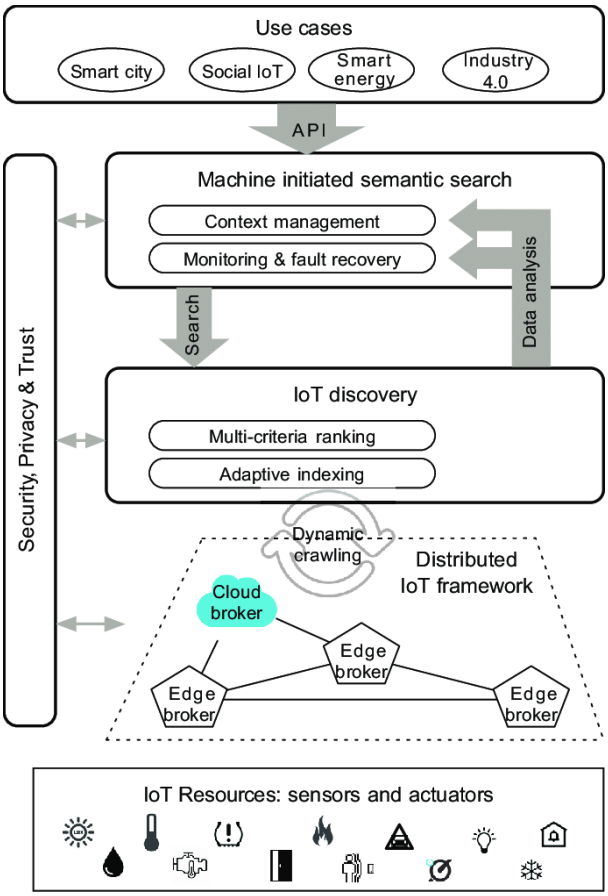


Figure 6: Overall architecture of the IoTCrawler framework

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 Concepts

## 3.1 Virtual IoT Systems

Nowadays, most of the 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 an insurmountable 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.

For almost all of today's applications running in production environments, Infrastructure-as-a-Service solutions have provided a convenient and widely adopted approach for renting the needed computing resources. Tenants can just focus on their applications, because the infrastructure, formed by computing, storage and network resources, is offered as a service by a cloud provider.

In this section, we introduce the key concepts behind the Fed4IoT system architecture, named the VirIoT platform, which re-uses cloud concepts but adapts them to the IoT world. VirIoT provides IoT developers with virtual IoT systems, named *Virtual Silos*[4], which are isolated environments that include *IoT Brokers* and a data domain dedicated to the single tenant. *Virtual Things* appear to a tenant as dedicated sensors that expose their data through a configurable broker technology (e.g. oneM2M [2], FIWARE [4], and the likes). Just like a traditional cloud offers virtual servers with configurable virtual hardware and operating system (OS), VirIoT offers Virtual IoT Systems with a configurable set of Virtual Things (akin to the hardware) and a Broker (akin to the OS).

VirIoT decouples IoT infrastructure providers from application developers, thus making possible: for the providers, to better use their IoT devices by sharing their data with different tenants, and, for the tenants, to configure the IoT infrastructure they need, quickly. Provider and tenant may also coincide, exploiting VirIoT for running experimental services within the private infrastructure in use every day, raising higher the security bar by running applications and their things inside isolated environments.

## 3.2 Virtualization Platform

Figure 7 visualizes the main concepts behind the VirIoT platform. On the left 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.

[4]In D2.1 and in the DoA, Virtual Silos was called IoT slice. We renamed them to avoid confusion with "5G" IoT slices. Indeed, VirIoT is at a higher layer and could use underlying 5G slicing and, in case, edge computing services for connecting and deploying its internal components, such as ThingVisors to Virtual Silo, etc. However, VirIoT can be deployment over plain Internet too.

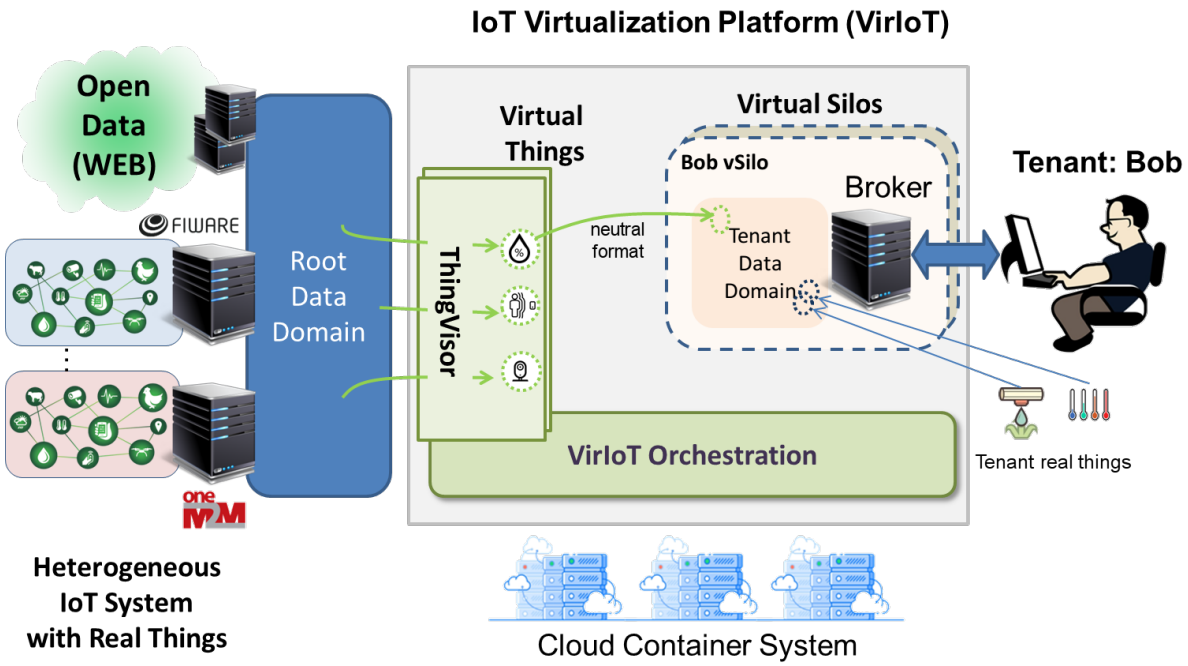


Figure 7: Fed4IoT VirIoT Platform

Information coming from different IoT Systems, and possibly from other data sources (e.g. open data), forms a *Root Data Domain*, from which VirIoT gathers information. Specifically, a group of VirIoT components named *ThingVisors* fetch the information and generate data items associated with *Virtual Things*. Consequently, a Virtual Thing is an emulation of a real thing that produces data obtained by processing/controlling data coming from the root data domain.

Figure 8 presents a diagram where we have emulated four virtual things (right side of the figure) out of three real things (left side of the figure). The three real things are a stationary camera, a camera-equipped drone and a thermometer, whereas the four virtual things are a face detector, a person counter, a moving camera and a thermometer.

The things' virtualisation concept that we are considering in this deliverable may go beyond traditional data processing, since it can also involve "control" of the real things. Let us explain this concept by detailing the virtualisation process made to obtain the virtual things presented on the right side of figure 8. The virtual face detector and virtual person counter obtain their data by performing analytics on the video stream coming from the real camera. The virtual thermometer obtains its data by merely copying data coming from the real thermometer. 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 9 depicts a general schema of how a ThingVisor receives data coming from one or more real things (or other sources, e.g. the web) and processes it in its native format,

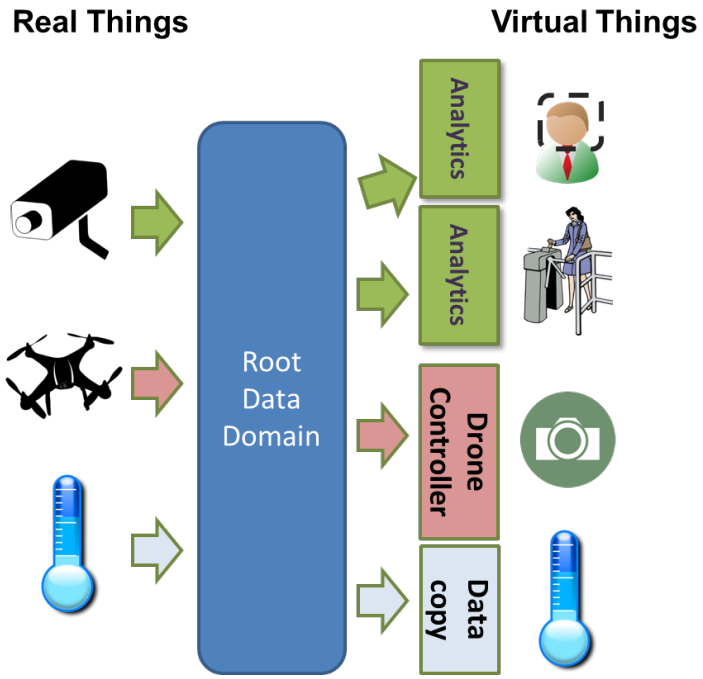


Figure 8: Virtual Things (vThings)

in order 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.

The VirIoT platform provides tenants with virtual IoT systems, dubbed *Virtual Silos*, which 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, such data comprises the tenant data domain which 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 hydrometer for measuring air humidity outside his house, just because he does not own a real one. Data from the rented hydrometer reaches Bob's broker in the silo, together with data from his sensors. So, Bob only sees his dedicated data set and broker, by accessing his silo, and the platform thereby provides for data and service isolation.

## 3.3 High-Level Usage Scenarios of the Platform

Similarly to cloud computing and as already described in D2.1, we envisage two possible high-level usage scenarios of the VirIoT platform, *public* and *private*.

In the public scenario, we foresee three distinct types of stakeholders: i) providers of real IoT systems offering their data in different formats, ii) providers of the VirIoT platform (or more than one VirIoT platform, competing for the market), which use

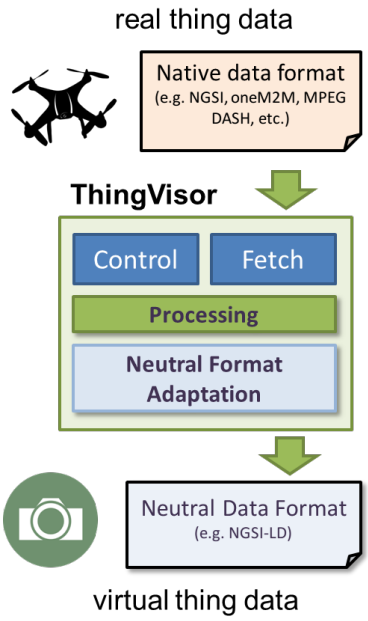


Figure 9: ThingVisor

this data to set up virtual things, iii) IoT application developers renting IoT virtual silos. This use case is going to be crucial in large-scale environments, such as smart cities, whereby the City owns several arrays of sensors and sells the raw data streams to a VirIoT provider, which acts as the intermediary between the vast amount of raw resources and the applications. Designers of smart city applications can instantiate silos as a service in the platform, gaining access to perhaps thousands of selected virtual things and a brokering environment of choice, without caring about infrastructural and data heterogeneity problems. Regarding data heterogeneity, we note that many IoT platforms cope with it by transforming external heterogeneous data items into an internal format (e.g. through proxies) and then by exposing that format to the final user, through a specific API. VirIoT makes a step forward: the data model and the API are a choice of the user rather than a platform one.

In the private scenario, the same actor owns both the infrastructure and the applications. She can use VirIoT both to enclose each IoT application in a small isolated environment (i.e. a virtual silo) and to support safe innovation, by decoupling the newly designed IoT applications from IoT services that are already in production. For instance, a company operating a smart harbour system may have a robust solution in place, where the existing application exploits various real sensors through a production broker. A novel version or an enhancement can be safely tested in a virtual silo, before final deployment in the production environment. Security-wise, a choice can be made as to what to expose to attacks from the outside. In short, a private approach to IoT virtualisation offers the same advantages a private cloud is nowadays offering to companies deploying their servers in virtual vs. bare-metal.

# 4 System Architecture - First Release

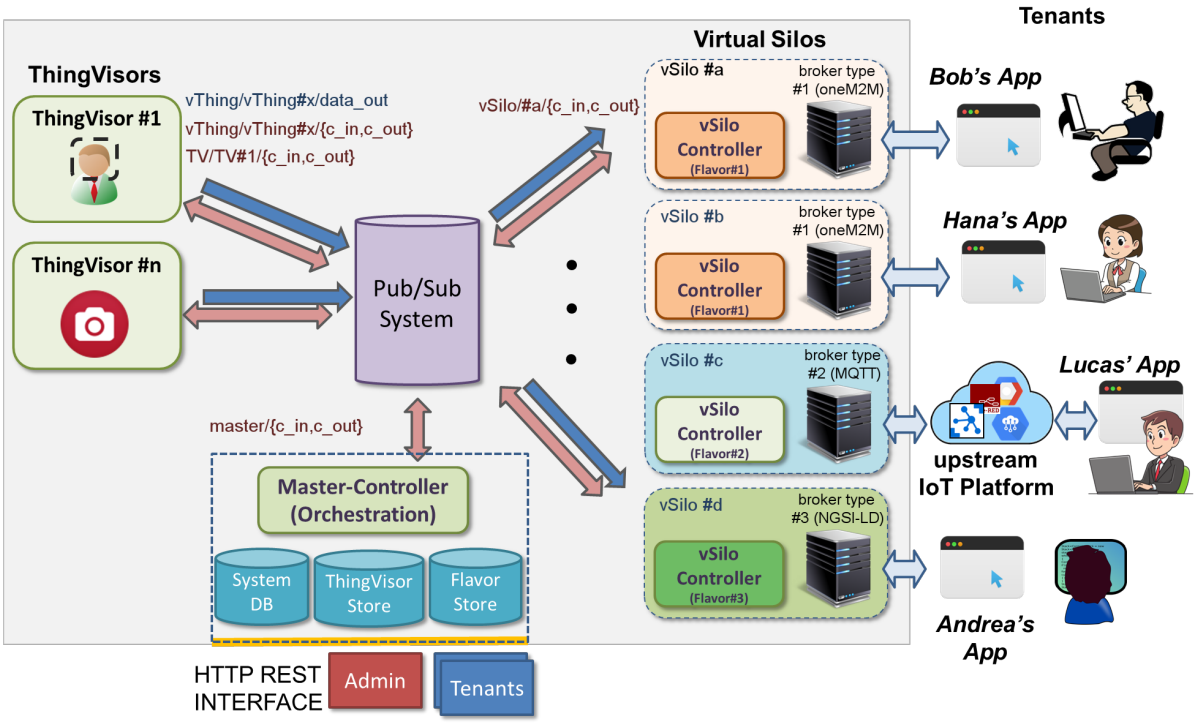


Figure 10: System Architecture - first release

Figure 10 shows the first release of the Fed4IoT VirIoT platform. This architecture follows a micro-services design, hence each component is an autonomous subsystem exposing network interfaces. This enables upgrades and updates to be injected 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) have been considered as the preferred component packaging tool, possibly supported by a container orchestration tool such as Kubernetes (k8s).

For external communications, the platform exposes an HTTP REST interface for the administrator and the tenants (users of the platform). Internal communications use a topic-based pub/sub system whose topics are reported in table 4 and detailed in the following.

There are control and data topics. Control topics are used by all components to receive (`c_in`) or send (`c_out`) control messages. Data topics are used to convey (`data_out`) the data items of virtual things.

On the left of figure 10 there are the ThingVisors (TVs), each uniquely identified by a name (`TViD`). A ThingVisor generates data items of one or more virtual things (vThings) and each virtual thing is uniquely identified by a name (`vThingID`[5]). The architecture is agnostic to the technology used to develop a ThingVisor, since ThingVisors run within an own container (or

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

Table 4: System Topics

| Topic | Naming | Description |
|---|---|---|
| vThing (Data) | `vThing/<vThingID>/data_out` | Used by a ThingVisor to publish data items of a virtual thing. |
| 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 |

k8s pod). However, it is necessary that it communicates with the other components through vThing and ThingVisor topics (again, see table 4).

On the right of figure 10 there are virtual silos (vSilos), which are used by tenants (Bob, Hana, Lucas and Andrea in the picture). Each silo is identified by a unique name (`vSiloID`). There could be different types of virtual silos, which differ in terms of broker type, scaling property, storage model, etc. We call *flavor* a specific configuration of a virtual silo, therefore a virtual silo is an instance of a given flavor. In figure 10, Bob and Hana have their own virtual silos (#a and #b, respectively) whose flavor is the same and includes a oneM2M broker to which their applications connect to. Lucas uses a virtual silo of a different flavor, instead, which exports the IoT data of his virtual/real things via simple MQTT topics. This is a kind of *raw* virtual silo, which can be in turn 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. Another flavor may include a NGSI-LD context broker that is able to talk to the upstream NGSI-LD-friendly App of yet another tenant, Andrea.

Each virtual silo includes an internal controller that is used to configure it (e.g. for adding or removing instances of virtual things) and also to relay and translate the data items of selected virtual things from the ThingVisor to the silo's broker. Again, the architecture is agnostic to the technology used to develop a virtual silo, as each silo runs in an own container (or k8s pod).

The Master-Controller manages deployment of new components in the systems as well as their configuration, following requests coming from administrator and tenants. System state information, about virtual silos, virtual things, ThingVisors, etc. is stored in a System DB. Container images of silo flavors and ThingVisors are maintained by specific object stores (e.g. Docker Hub).

## 4.1 Basic Procedures

We now describe some basic procedures.

The administrator can request to add a new silo flavor or ThingVisor, in order to extend the VirIoT platform's capabilities. Consequently, the Master-Controller inserts the new container image of the silo/ThingVisor into the proper stores and updates the System DB. In case of ThingVisors, it is then the underlying container platform that runs instances of them. As soon as it is up and running, a ThingVisor starts to publish data items of the virtual things it handles to the related data topics.

When a tenant requests creation of a virtual silo, the Master-Controller fetches and runs an instance of the image of the requested flavor, providing the tenant with an IP address and port where she can contact the broker running inside the virtual silo. Subsequently, a tenant can request virtual things to be added to her virtual silo. The Master-Controller, in turn, relays this request to the virtual silo controller through its input control topic. Consequently, the silo controller registers the necessary metadata in the silo's broker and becomes a subscriber of the virtual thing data topic, thereby starting to receive related data items. These data items are translated from the neutral format to the data model used by the silo's broker, and then they are eventually pushed to the broker. The Master-Controller stores all configuration information of the virtual silos in the System DB.

## 4.2 Virtual Actuators

Up to now, we have silently assumed that vThings are data producers, and we used pub/sub distribution model to easily support multi-tenancy: a virtual thing publishes data on the `data_out` topic and many tenants (subscribers) can exploit this data. But what if a vThing is an actuator? Does it still make sense to talk about multi-tenancy in this case, and what would it mean that many tenants share the same (virtual) actuator? At this stage, we are still exploring the multi-tenancy issue. Without multi-tenancy, VirIoT can provide a tenant with exclusive use of an actuator exposed as a vThing, just like, in the traditional cloud, a tenant can rent bare metal servers. For instance, an street lamps infrastructure provider can integrate into VirIoT a ThingVisor, which handles a set of vThings related to the lamps it wants to offer to the tenants' control. A tenant can insert in its vSilo the vThing that controls the street lamp close to her house to control on/off state, e.g. according to other IoT data available in her vSilo. To support the reception of triggering events (e.g. "light on") from the vSilo to the vThing, the platform uses a secure `data_in` topic that will be better detailed in the second release of the architecture.

Obviously, since the beginning VirIoT supports the use of actuators *own* by the tenant, i.e. neither virtual nor rented by VirIoT . This concept is shown in picture 8, bottom right, where the watering device (actuator) is a real thing directly connected to the tenant data domain.

## 4.3 Customizable Virtual Things

In many use cases, tenants need to customize the service offered by a ThingVisor. For instance, Bob may need a virtual thing providing face detection for his son, while Hana for her daughter. The function is the same, face detection, but the inputs and the outputs are different. For Bob, the input is an image of his son, the output may be the captured picture in case of face detection and this output must be transmitted only to the Bob virtual silo. The same reasoning can be repeated for Hana.

The support of ThingVisor customized services will be included in the second architecture release. Anyway, we present a very preliminary idea: the ThingVisor can create virtual things

*on-demand*, each of them offering the service needed by a specific tenant. For instance, in the previous example, Bob can request to add to his silo a face detector vThing, piggybacking an image of his son in the request. In turn, the related ThingVisor may create a dedicated vThing, whose `data_out` topic is only subscribed by Bob's silo controller. Hana can make the same for detecting her daughter. We point out that we are anyway exploiting the sharing of real resources because the ThingVisor uses the same video streams for the different face detection threads.

# 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 has the goal of publishing produced data on VirIoT system topics. In what follows we present two "instruments", FogFlow and ICN, that can be used as middleware to develop ThingVisors on top of the Root Data Domain.

## 5.1 FogFlow

Traditional Big-Data analysis approaches follow a scheme by which all the data sources are integrated in a single or distributed service where they can be exploited by using different techniques and technologies. This approach relies on a single point to make the aggregation and processing, requiring a high-performance machines for such an arduous task. Fog and Edge computing techniques promote a change in the paradigm, allowing edge nodes to make initial aggregating tasks that alleviate the final and more complex task carried out at cloud level. FogFlow [13] is an IoT edge computing framework that addresses this new change of paradigm.

Figure 11 presents the high-level architecture of this framework. As we can see there are

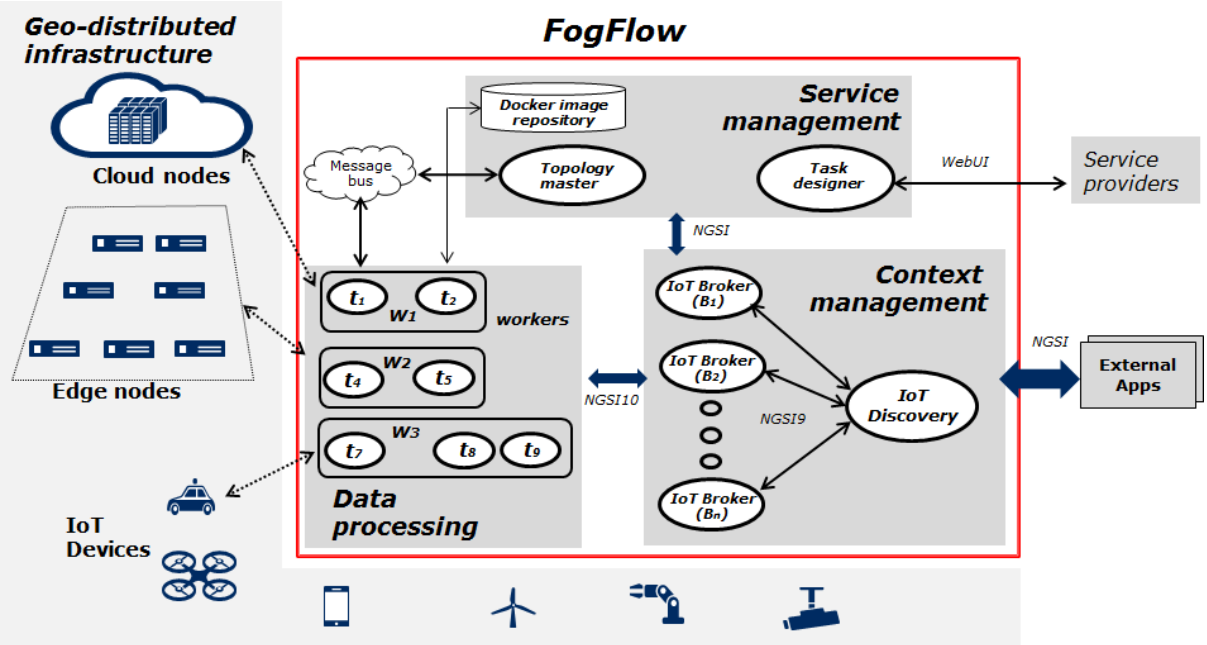


Figure 11: High-level view of FogFlow architecture

three divisions we highlighted in red a box:

- Service management: It comprises task designer, Topology Master (TM), and docker image repository, which are typically deployed in the cloud.
- Data processing: It consists of a set of workers ($w_1$, $w_2$, ..., $w_n$) to perform data processing tasks assigned by Topology Master. A worker is associated with a computation resource

in the cloud or on an edge node and can launch multiple tasks based on the underlying docker engine and the operator images fetched from the remote docker image repository.

- Context Management: Finally, this division comprises 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.

Therefore, each service to be deployed is made by a set of tasks that receive and send "flows" of data, either from IoT sources or from other tasks. These tasks, also called *fog-functions*, request IoT data to the internal Orion Broker. For its definition, FogFlow provides a graphical user interface which allows the user to easily define the consumed information per task, as well as the output information.

Focusing on the design of the ThingVisor and how FogFlow can help in its development and deployment, we present in Figure 12 the envisioned integrated view of both technologies. Information coming from both NGSI and non-NGSI compliant device in the Root Data Domain and/or services can be integrated, requiring for the latter the use of specific adapters. A FogFlow orchestrator takes care of deploying fog-functions to optimized locations that offer Cloud/Edge/Fog computing functionality (on the left of figure 12).

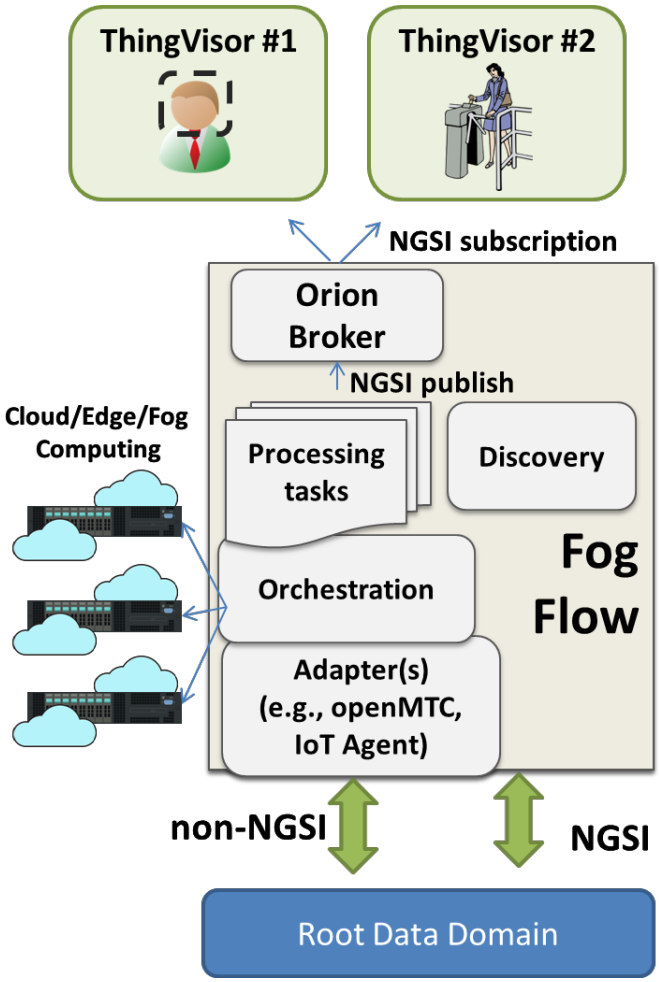


Figure 12: Integration of FogFlow with other components

Finally, data produced by a chain of fog-functions is published to the internal Broker and, this way, different ThingVisors can merely fetch and republish them to the VirIoT system topics.

## 5.2 Information Centric Networks

This section initially reports an introduction on Information Centric Networks (ICNs) and then an outlook about how this technology can be exploited for devising ThingVisors.

An ICN is a communication architecture providing users with data items rather than end-to-end communication pipes. ICN can support both request-response and publish-subscribe [17] communication patterns. Its services resemble that of content delivery networks, but with packet-level granularity.

The network addresses are hierarchical names (e.g. `rs1/lamp/1502`) that do not identify end-hosts but data items (e.g. status of the lamp n. 1502 in the IoT Real System n.1). Figure 13 depicts the model of an ICN node and packets.

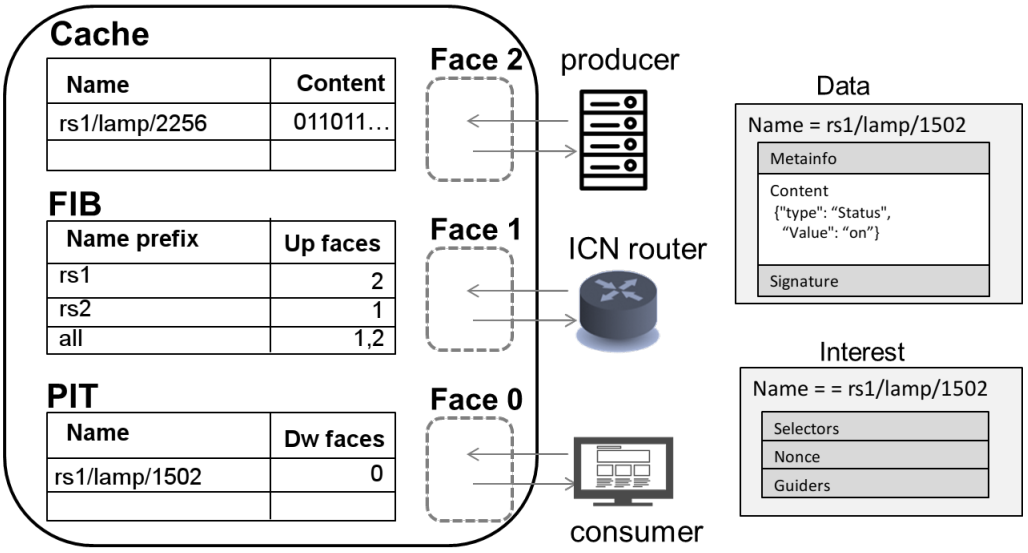


Figure 13: ICN forwarding engine model and packets

A data item and its unique name form the so-called *named object*. A named object is actually a small data unit (e.g., 4kB long) and may contain an entire content (e.g., a document, a video, etc.), or a chunk of it. The names used for addressing the chunks of the same content have a common prefix (e.g., `rs1/lamp/1502`) followed by a sequence number identifier (e.g. `s1, s2, s3, etc.`).

An ICN is formed by nodes logically classified as consumers, producers, and routers. Consumers pull named objects provided by producers, possibly going through intermediate routers. The consumer-to-producer path is labeled as upstream; the reverse path as downstream.

Any node uses the forwarding engine shown in figure 13 and is connected to other nodes through channels, called *faces*, which can be based on different transport technologies such as Ethernet, TCP/IP sockets, etc.

The data units exchanged in ICN are *Interest packets* and *Data packets*. To download a named object, a consumer issues an Interest packet, which contains the object name and is forwarded towards the producer. The forwarding process is referred to as routing-by-name, since the upstream face is selected through a name-based prefix matching based on a *Forwarding*

*Information Base* (FIB) containing name prefixes, e.g., `rs1` in figure 13. The FIB is usually configured by routing protocols, which announce name prefixes rather than IP subnetworks [10]. During the Interest forwarding process, the node temporarily keeps track of the forwarded Interest packets in a Pending Information Table (PIT), which stores the name of the requested object and the identifier of the face from which the Interest came from (downstream face).

When an Interest reaches a node (producer or an intermediate router) having the requested named object, the node sends back the object within a Data packet, whose header includes the object name. The Data packet is forwarded downstream to the consumer by *consuming* (i.e., deleting) the information previously left in the PITs, like bread crumbs.

Each forwarding engine can cache the forwarded Data packet to serve subsequent requests of the same object (*in-network caching*). Usually, the data freshness is loosely controlled by an expiry approach. Any Data packet includes a metainfo field reporting the freshness period specified by the producer, which indicates how long the Data can be stored in the network cache.

The forwarding engine also supports *multicast/anycast distribution* both for Interest and Data packets. Interest multicasting takes place when there are more upstream faces for a given prefix (e.g., `all` in figure 13) and the incoming Interest is forwarded to all of them. In case of Interest anycasting, a routing strategy properly select the best output face according to given routing metrics. Data multicasting is implemented as follows: when a node receives multiple Interests for the same object, the engine forwards only the first packet and discards the subsequent ones, appending the identifier of the arrival downstream faces in the PIT; then, when the requested Data packet arrives, the node forwards a copy of it towards each downstream face contained in the PIT.

ICN security is built on the notion of *data-centric* security: the content itself is made secure, rather than the connections over which it travels. The ICN security framework provides each entity with a private key and an ICN digital certificate, signed by a trust anchor, and uniquely identified by a name called *key-locator* [11]. Each Data packet is digitally signed by the content owner and includes the key-locator of the digital certificate to be used for signature verification. For access control purposes, Interest packets can be signed too.

An ICN uses *receiver-driven flow/congestion control*. To download a content formed by many chunks, the consumer sends a *flow* of Interest packets, one per chunk, and receives the related flow of Data packets. Flow and/or congestion control are implemented on the receiver side by limiting the number of in-flight Interest packets to a given amount (aka pipeline-size), which can be a constant value or a variable one, e.g., controlled by an AIMD scheme [12].

### 5.2.1 ICN Service Function Chaining

We can build ThingVisors by chaining functions and exploiting Information Centric Networking (ICN) technology [15, 16], as well. We can take advantage of the name-based delivery of ICN so as to build a ThingVisor based on a chain of in-network functions. We can better explain such composition referencing Figure 14, where we present an example of a ThingVisor processing the information coming from a camera. It comprises three chained tasks, which we reference by their task names (TN in the following): image capture, human detection, and face detection. Expressing these tasks by using unique and representative names, we can create a sequence of in-network functions by a sequence of names.

Thus, we can exploit name-based delivery (routing-by-name) to form ThingVisors out of in-network functions, where an in-network function can be either a sub-task of a ThingVisor or another ThingVisor. This approach can then enable chaining of ThingVisors to create a different ThingVisor.

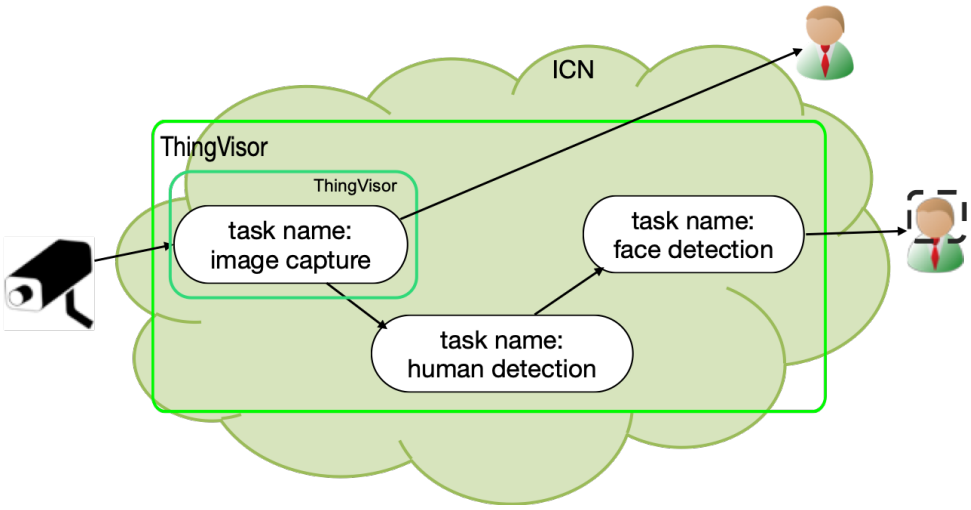


Figure 14: ThingVisor on ICN

Whenever the IoT systems forming the Root Data Domain are able to provide replicas of in-network functions under the same name, the name-based anycast delivery capability of ICN can be in turn exploited to properly select the best replica, thus increasing system reliability and network efficiency.

Root Data Domains with a variety of information models and APIs are connected to an ICN by adapters, which translate the models to our neutral format, namely NGSI-LD (see Figure 15). Accordlingly, we are proposing to use the neutral format also inside the ThingVisor ICN development framework.

One Root Data Domain adopting oneM2M standard publishes thermometer readings through a rendezvous node. The reading is supplied to "TN: Data copy" to be used within the VirIoT platform properly exposed as ThingVisor. A surveillance camera is connected through the adapter "TN: camera adapter." The "TN: image capture" requests an image from the camera using the request-response communication in ICN. The retrieved image is first stored at the task and forwarded to "TN: human detection" and then "TN: face detection" to find out a particular person. After "TN: image capture," the communication follows the publish/subscribe model. For our IoT platform, both the publish/subscribe model and the request-response interactions are needed, in order to have flexibility for reducing energy consumption.

A deployment management functionality decides where to place the tasks. Tasks can be placed at the cloud, at the edge node, or even at the intermediate routers in ICN. Routing functionality chooses task instances with the same task name (anycasting) to have good reliability, load balance, and efficiency.

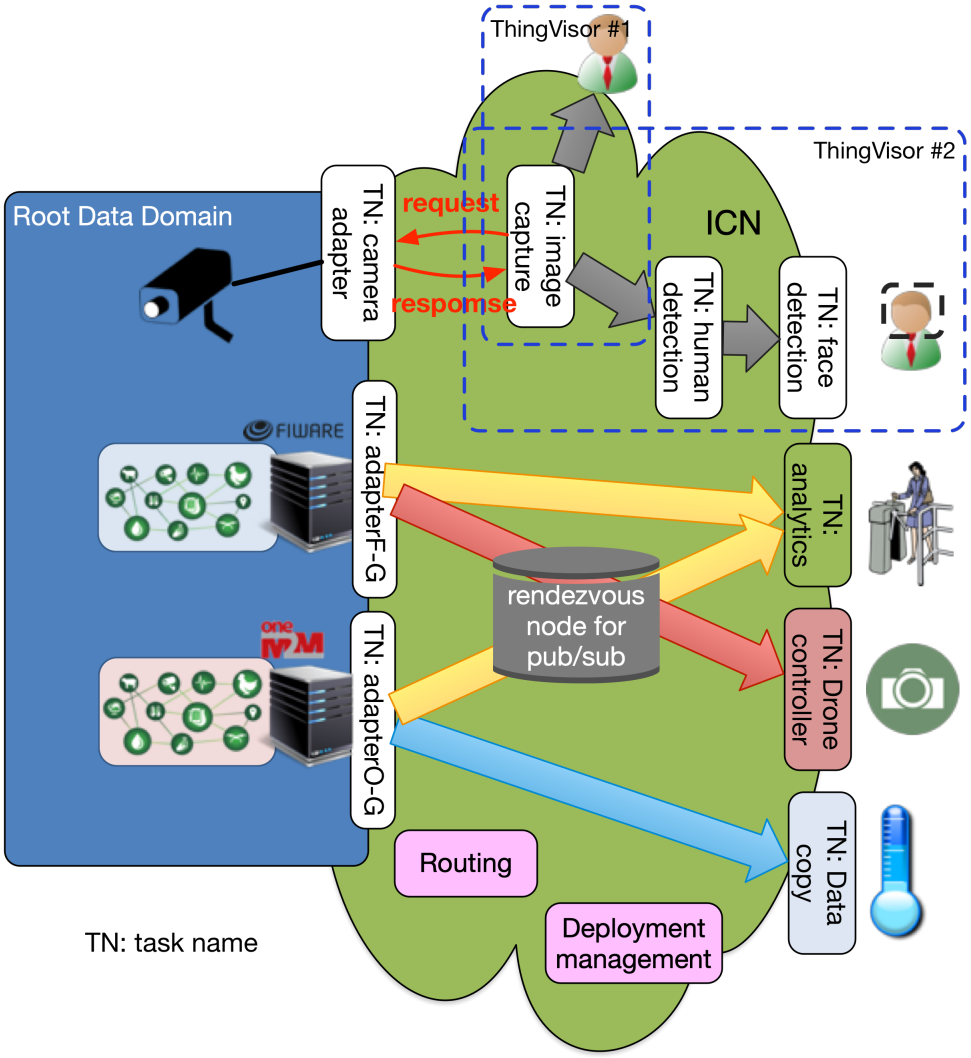


Figure 15: Bridging between Root Data Domains and VirIoT with ICN

# 6 NGSI-LD: neutral-format and Broker

NGSI-LD [6], as already 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 section is structured as follows. Section 6.1 illustrates the JSON serialization of an entity, comparing between NGSI and NGSI-LD. Section 6.2 introduces the NGSI-LD information model. Section 6.3 gives an overview of the NGSI-LD API. Section **??** introduces the NGSI-LD Broker, highlighting how it/certain components can be used in Fed4IoT. Finally, Section 6.5 describes how the NGSI-LD information model and representation will be used as the *neutral format* in Fed4IoT.

Please notice that, in the following we interchangeably refer to NGSI or NGSIv2. NGSIv2 is the latest released specification of NGSI.

## 6.1 NGSI-LD evolves NGSI

Figure 16 presents an example of a *"vehicle"* context entity represented using NGSI. 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 16: FIWARE-NGSI representation of a vehicle parked at a location

We also present a possible, richer, NGSI-LD description of the same entity, in Figure 17. This way we can compare the two representation formats as well as identify some advantages,

such as the relationships between entities and the explicit GeoJSON encoding of the vehicle's location, provided by NGSI-LD.

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

Figure 17: FIWARE-NGSI-LD representation of a vehicle parked at a location

## 6.2 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 18.

Figure 19 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,

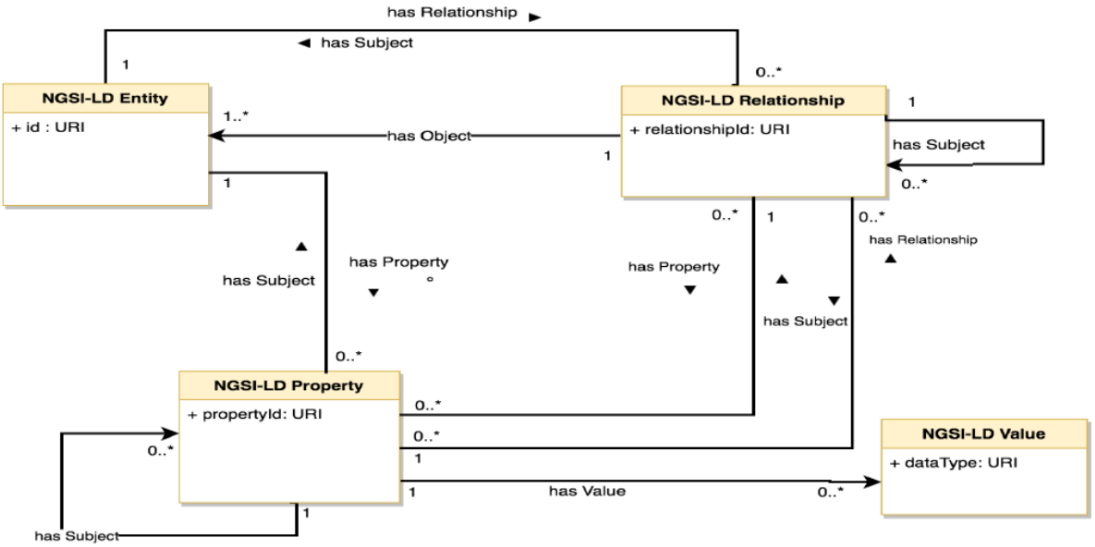


Figure 18: NGSI-LD concepts and relations

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.

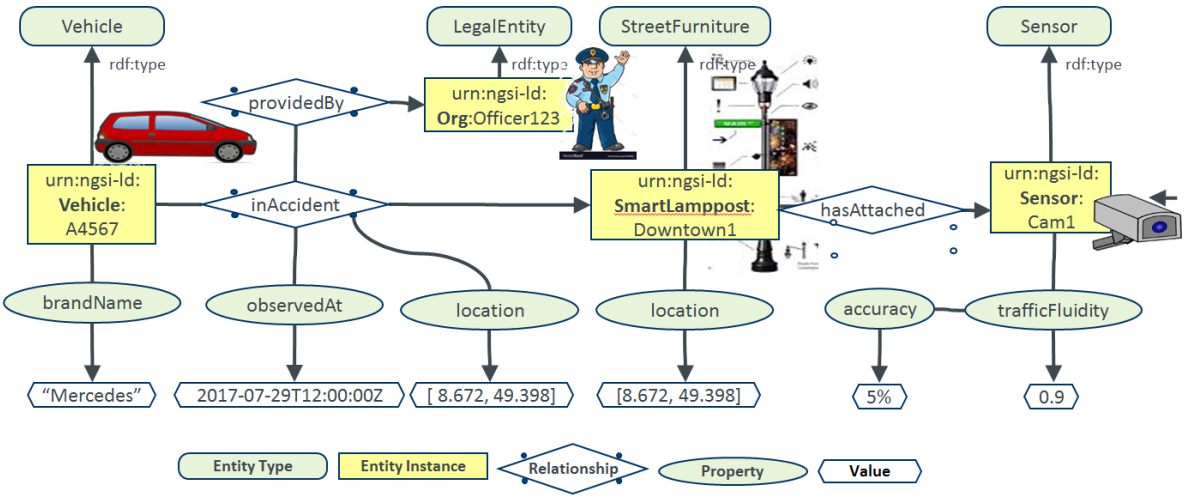


Figure 19: Example of an NGSI-LD property graph, *source: ETSI ISG CIM*

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 20.

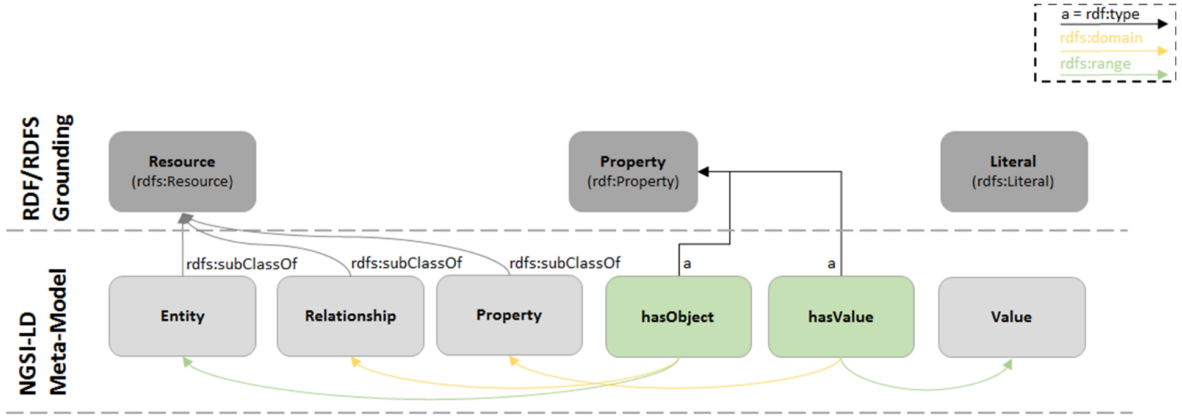


Figure 20: NGSI-LD Core Meta Model

We can see that NGSI-LD `Properties` and `Relationships` are subclasses of `rdfs:Resource`. 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 19. 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 21.

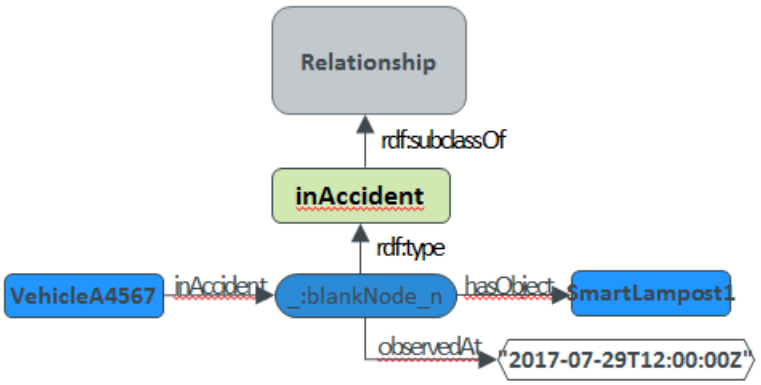


Figure 21: Example of blank node reification in NGSI-LD

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, 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 22.

```
{
  "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 22: NGSI-LD serialization in JSON-LD

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.

### 6.2.1 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 23.

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

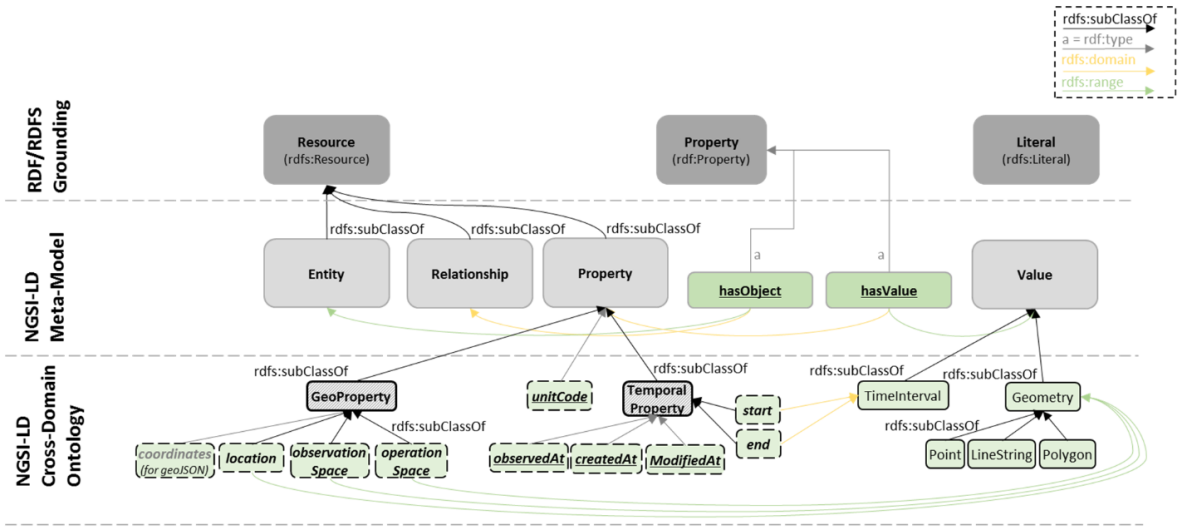


Figure 23: NGSI-LD Core and Cross Domain Model

Figure 22, 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`.

## 6.3 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 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 24 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 update information in the Broker.

In a more advanced setup, not all information is stored by the Broker, but there are *Context Source*s 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 stand-alone component or be tightly integrated with one or more Brokers. Context Sources can register

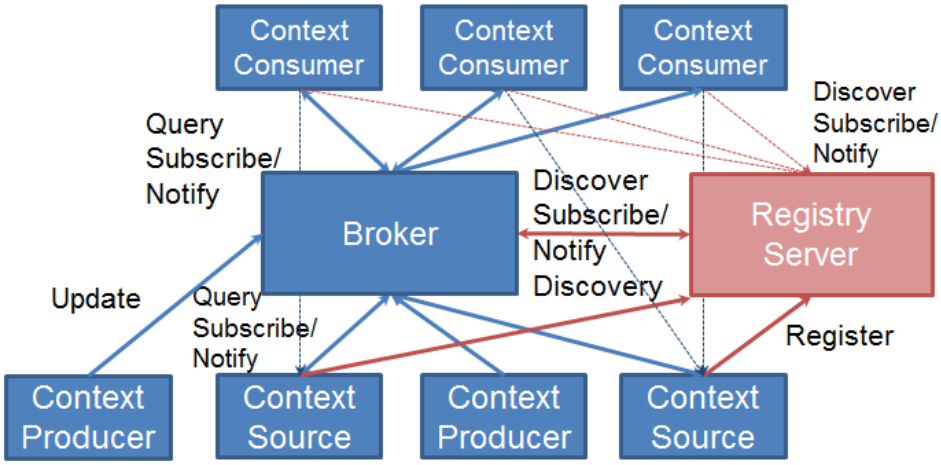


Figure 24: NGSI-LD logical architecture

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.

## 6.4 NGSI-LD Broker and Architecture

In the context of Fed4IoT project, we are developing an NGSI-LD Broker 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 microservice based using Kafka as the message bus and PostgreSQL as a database with PostGIS to support geographic queries.

The NGSI-LD Broker can be used as the Broker for an NGSI-LD Virtual Silo, making information available to user applications based on NGSI-LD. When using NGSI-LD as neutral format, the related Silo Controller is completely straightforward, as all data notification can directly be used for creating or updating the contained entity information in the NGSI-LD Broker, making it directly available without needing an additional translation step.

Another possible use of NGSI-LD architecture for VirIoT is to support the *semantic discovery* of available vThings, for which the use of the NGSI-LD Registry Server should be further investigated. For each vThing, the related ThingVisor would register the NGSI-LD information it can provide into an internal NGSI-LD Registry Server and the discovery interface of the Registry could then be used by the user to find suitable vThings for her virtual silos.

## 6.5 NGSI-LD as neutral format

For NGSI-LD to be able to effectively serve as a neutral format, two-way mappings must be designed. On the one 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 Silo Controller must be able to convert from the neutral format to the format in use by the internal broker. As show in figure 7, in addition to the raw MQTT pass-through and to the native NGSI-LD (for NGSL-LD silo flavour), our minimum goal is to support two baseline formats: the former NGSIv2 in use by currently operational FIWARE Context Brokers, such as Orion, and the oneM2M format, currently operaitonal 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 semantic 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 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. Accordingly, in the following sections we present our first round of design decisions concerning translation between formats and we give priority to the mappings from NGSI-LD to FIWARE/oneM2M, i.e. those used by vSilo controllers.

### 6.5.1 Mapping between NGSI and NGSI-LD

We have already mentioned in section 2.2.2 that NGSI has evolved into NGSI-LD, allowing for a richer representation of information. It may thus seem somewhat obvious that mapping from NGSI to NGSI-LD is straightforward, while the reverse can be more challenging. This is not always the case. For instance, in NGSIv2 there is no constructs to indicate explicit relationships between Entities. But, in time, to compensate for this lack of cross-referencing between objects, FIWARE introduced the convention that the name of attributes representing a relationship should start with "ref". NGSI-LD fixes this, by introducing explicit Relationships. Hence a well-thought mapping algorithm must implement a non-trivial amount of intelligence, and it should be able to correctly map "ref" NGSIv2 attributes to NGSI-LD Relationship blocks.

Suppose we have the following NGSI Entity, which is information about a bike parking place in Murcia, taken from their publicly available pool of data, served by the currently online Orion Context Broker.

```
{
  "descripcion": {
    "metadata":{},
    "type": "string",
    "value": "Santo%20Domingo"
  },
  "geoposicion":{
    "metadata":{},
    "type": "coords",
```

```
    "value": "37.987769,-1.129766"
  },
  "id": "AparcamientoBicis:180",
  "libres":{
    "metadata":{},
    "type": "number",
    "value": "16"
  },
  "type": "Sensor"
}
```

The link `http://telefonicaid.github.io/fiware-orion/api/v2/stable`, section named "Specification", neatly describes the FIWARE NGSI data model for context information, a simple information model based on the notion of context entities. The main elements of this NGSIv2 data model are: context entities, attributes and metadata. For the sake of readability, we summarize in the following the central notions from the above specification.

Each entity has an entity id. Furthermore, NGSI enables entities to have an entity type. Entity types are semantic types that are intended to describe the type of thing represented by the entity. For example, a context entity with id *sensor1* could have the type *temperature-sensing-device*. Attributes are properties of entities. For example, the current speed of a car could be modeled as attribute *current-speed* of entity *car1*. Attributes have an attribute name, an attribute type, an attribute value and metadata. The attribute name describes what kind of property (of the entity) the attribute value represents. The attribute value contains the actual data. Optional metadata describe properties of the attribute value like e.g. accuracy, provider, or a timestamp. Normally, attribute types are informative and processed in an opaque way. Nonetheless, the special attribute types DateTime (identifies dates, in ISO8601 format), geo:point, geo:line, geo:box, geo:polygon and geo:json (have semantics related with entity location) are used to convey a special meaning.

From the above description, it can be noticed that in NGSI, both the metadata and the special attribute types are used to represent common information typically found in every attribute. One of the goals for the NGSI-LD evolution has been to clean up these two different methods for representing meta-information (timestamp, location, who is the provider of the values, etc.) that is common to most of the attributes regardless of the specific vertical application. Thus, the concept of Property was introduced in NGSI-LD to subsume the attribute's type and attribute's metadata, and a number of cross-domain properties has been defined (please see section 6.2, specifically the Cross-Domain Ontology definition).

This gives birth to the mapping strategy described in table 5.

An example NGSI-LD entity equivalent to the previous NGSI entity, built by applying the guidelines in table 5, is then the following:

```
{
  "id":"urn:ngsi-ld:Sensor:AparcamientoBicis:180",
  "type":"Sensor",
  "descripcion":{
    "type":"Property",
    "value":"Santo%20Domingo"
  },
  "location":{
    "type":"GeoProperty",
```

Table 5: Guidelines for mapping between NGSI and NGSI-LD

| NGSIv2 | NGSI-LD |
|---|---|
| entity id | entity id |
| entity type | entity type |
| attribute blocks | heuristics to decide whether the attribute is a property or a relationship:<br>• if attribute name is prefixed by "ref" then map to an attribute block of the entity<br>• else map to a property block of the entity |
| attribute value | • property value if mapping to a property<br>• relationship object if mapping to a relationship |
| attribute type | information conveyed by the NGSI attribute type is:<br>• inferred automatically if type was a JSON standard type<br>• encoded in the name of the property/relationship if type was mapped to a cross-domain concept such as "createtAt", "observedAt", "location" |
| attribute metadata | create a sub-property only if not exploited for the main property/relationship mapping |

```
10      "value":"{\"type\":\"Point\", \"coordinates\":[37.987769,-1.1
           29766]}",
11      "former-ngsi-attribute":{
12        "type":"Property",
13        "value":"\"geoposition\": {\"metadata\":{},\"type\":\"
             coords\",\"value\":\"37.987769,-1.129766\"}"
14      }
15    },
16    "libres":{
17      "type":"Property",
18      "value":"16"
19    },
20    "@context":[
21     "http://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld",
22     "http://example.org/murcia-context.jsonld"
23    ]
24  }
```

In the above transformation we see how the mapping algorithm should be able to infer that the "geoposition" attribute refers to a geographic position of the entity, by looking not only at the name itself, but also at the type ("coords") and most importantly at the value itself, and by carrying out a sanity check on the numbers. Consequently, we are able to map the attribute to a NGSI-LD-specific GeoProperty, by forming the correct GeoJSON construct at the value.

Similarly, we need to develop appropriate heuristics targeting NGSI metadata blocks that carry information about timestamps or about owners or creators of the entities. The goal is to be able to exploit as much as possible the Cross-Domain Ontology.

Additionally, we have adopted a strategy to retain the original/former NGSI attributes, embedded as sub-properties within the newly instantiated NGSI-LD properties (namely, the "former-ngsi-attribute" property inside the "location" property, which retains the original block as a JSON-encoded string at the "value" key). This strategy may be particularly useful for our virtualization purposes, whenever the neutral format is then going to be re-mapped to the output brokers, for instance to a NGSIv2 output broker. We would then be able to not loose any information in the process. This is maximally important when we apply our heuristics in mapping attributes to properties or relationships, as the process is potentially error-prone.

### 6.5.2 Mapping between NGSI-LD and oneM2M

In the following we investigate the mapping with oneM2M. Specifically, we are trying to design a mapping strategy between oneM2M resources such as Application Entities, Containers and ContentInstances), and NGSI-LD Entities.

Suppose we have the following NGSI-LD Entity, that comprises three Properties, where `@context` is omitted for brevity:

```
{
  "id": "urn:ngsi-ld:Sensor:AparcamientoBicis:180",
  "type": "Sensor",
  "descripcion": {
    "type": "Property",
    "value": "Santo%20Domingo"
  },
  "location":{
    "type":"GeoProperty",
    "value":"{\"type\":\"Point\", \"coordinates\":[37.987769,-1.1
       29766]}"
  },
  "libres": {
    "type": "Property",
    "value": "16"
  }
}
```

We started by evaluating the following two strategies, as a base for an automated algorithm, that are both able to map an NGSI-LD Entity to a native oneM2M resource (please consult section 2.2.1 for a brief recap on the oneM2M resource tree structure).

- Strategy 1: every NGSI-LD Entity is mapped to a different oneM2M Application Entity
- Strategy 2: every NGSI-LD Entity is mapped to a different Container within the same oneM2M Application Entity

The first mapping principle is to consider each NGSI-LD Entity as an application on its own, that has a fine-grained structure comprising several top-level Containers. Then, each Property or Relationship of the original NGSI-LD Entity is mapped to a different top-level Container resource of such oneM2M Application Entity. Consequently, property values (or relationship objects) are mapped to ContentInstances of the respective top-level Container that represents the property.

Table 6: Guidelines for mapping between NGSI-LD and oneM2M

| NGSI-LD | oneM2M |
|---|---|
| entity | top-level container |
| entity id | top-level container resourceID |
| entity type | top-level container labels |
| property/relationship | sub-container |
| property/relationship name | sub-container resourceName |
| property/relationship type | sub-container labels |
| property value/relationship object | content instance of the sub-container |
| sub-property/sub-relationship | sub-sub-container (container nested within sub-container) |

The second mapping method is more coarse-grained, and it groups all NGSI-LD Entities under one common NGSI-LD Application Entity. This means that top-level Containers now represent NGSI-LD Entities, not Properties/Relationships. This strategy then exploits Container nesting, hence using sub-container resources to represent Properties or Relationships, so that each NGSI-LD Entity is considered as a top-level Container and be created, together with all other entities, inside the networked oneM2M AE where they belong.

We can see that oneM2M is extremely flexible in the structuring of data sources. Nonetheless, different resource types come with different levels of overhead when creating them. For instance, at creation time, Application Entities need to explicitly register themselves to the higher-level resource that coordinates them, i.e. the Common Services Entity (CSE). This is because Application Entities in oneM2M capture the concept of data producers (and consumers), rather than the concept of the produced data items. Furthermore, each AE can be the originator of a request to be granted a set of access operations (such as CREATE, RETRIEVE, UPDATE, DELETE, DISCOVERY and NOTIFY), giving birth to a complex system of access control policies for those Application Entities that are going to operate on certain Containers.

This is why we think Strategy 1 may be overkill in many practical use cases, and Straregy 2 is preferred for our mapping between NGSI-LD and oneM2M. In NGSI-LD, entities are passive data items, that much better align to the Container concept of oneM2M, rather than the concept of Application Entity.

Table 6 summarizes our preferred strategy.

This may, however, not always be the perfect choice because both standards allow for a great amount of flexibility and there may be applications already in place that make certain rigid assumptions about how the data producers and the data items are organized. Moreover, applications will want to discover data items, based on attributes' names and labels. Nonetheless, our first implementation of the mapper is going to be based on Strategy 2. We will collect feedback from the use cases and keep other possibilities open.

The oneM2M standard allows discovery of resources based on a number of attributes, labels and semantic descriptors, although it must be noted that not all of the existing oneM2M implementation, if any, as of today (March 2019), implement the discovery via semantic descriptors. In our mapping strategy we are going to exploit labels as much as possible.

Concerning values of the NGSI-LD properties, they are always going to be mapped to oneM2M resources of type ContentInstance. ContentInstances make no assumptions regarding their content, and they behave as blackboxes, so that not much is done on their content, except retrieving it. Most of the time the content is base64-encoded. This is the reason why our

idea is to put the whole property into the content instance, including types, so that it can be unboxed in a straightforward way, without loosing any information, for example whenever we need to reverse map from oneM2M back to NGSI-LD. Furthermore, the type is mapped onto the labels of the Container, too, in order to enable discovery by means of queries/filters on the labels' values. Similarly, oneM2M ContentInstances do not express metainformation about the instance's units or geographic position, essentially treating the data points as opaque. If the system (or data producer) wants to specify complex metadata about oneM2M resources, then the semanticDescriptor resource type is available. It enables attaching RDF triples to the resources, and the oneM2M specification describes how to build queries and semantic filters to discover resources based on the semantic descriptors. The ontologyRef attribute of the semanticDescritpor is used for specifying what ontology is used as the basis for the semantic content of a semantic descriptor.

In the example that follows, we see how it would be possible to exploit the (sub-)Container's semantic descriptor to capture the fact that the property of the original NGSI-LD entity concerns its geographic location. Let's refer to the example NGSI-LD entity we presented above, which contains information coming from the Murcia bike parking place (a single vThing), and let's apply our mapping guidelines to it:

- an Application Entity is created and registered to the CSE of the Silo broker. The AE resource name may be equal to the vThingID, e.g. "BikeParking". The AE is the contact point for retrieving information about all bike parking places of Murcia. This strategy avoids creating a huge number of AE resources
- "id":"AparcamientoBicis:180" (which is a specific parking place) → becomes a top-level Container within the AE (i.e. every NGSI-LD Entity is mapped to a different top-level Container)
- "type":"Sensor" → becomes a "labels" of the top-level Container
- "libres" property → spawns a sub-Container. It becomes the sub-Container's name
- "type":"Property" of "libres" → becomes labels of the "libres" sub-Container
- "value":"16" → spawns a ContentInstance of the "libres" sub-Container
- "location" → another sub-Container
- it is inferred that "location" refers to a geographic position, hence a semantic descriptor of the "location" sub-Container is created. RDF triples are inserted stating the fact that the Container's data concerns geographic positions or geometries
- "descripcion" → another sub-Container

Also the other values become ContentInstances of their respective sub-Containers. It must be noted that the whole JSON block of the "location" Property:

```
1 {
2   "type":"GeoProperty",
3   "value":"{\"type\":\"Point\", \"coordinates\":[37.987769,-1.129
      766]}"
4 }
```

becomes the ContentInstance, so that no info is lost. This holds true for all properties/relationships in the original NGSI-LD entity.

Basically this approach can be summarized as: every Property/Relationship of the NGSI-LD Entity becomes a sub-Container; the values become ContentInstances, and the Entity itself becomes the top-level Conbtainer.

It is possible to do query search&discovery, by exploiting the fact that we copy the names to the labels. Whenever the mapping algorithm is able to infer complex data types, semantic descriptors can additionally be created.

Figure 25 represents that mapping as resource tree, having one AE for all bikeparkings and assuming that the CSE of the vSilo has name "Murcia", i.e. the tenant identifier. A containers group could allow to query all the bikeparkings at once. This perspective is interesting from the Murcia city point of view. However, it is not possible to *a priori* exclude that bikeparking managers may prefer to have an alternative with one AE per bikeparking. We will plan a first implementation of the mapping module following the preferred guidelines, and evaluate its effectiveness agains the project's available data and use cases.

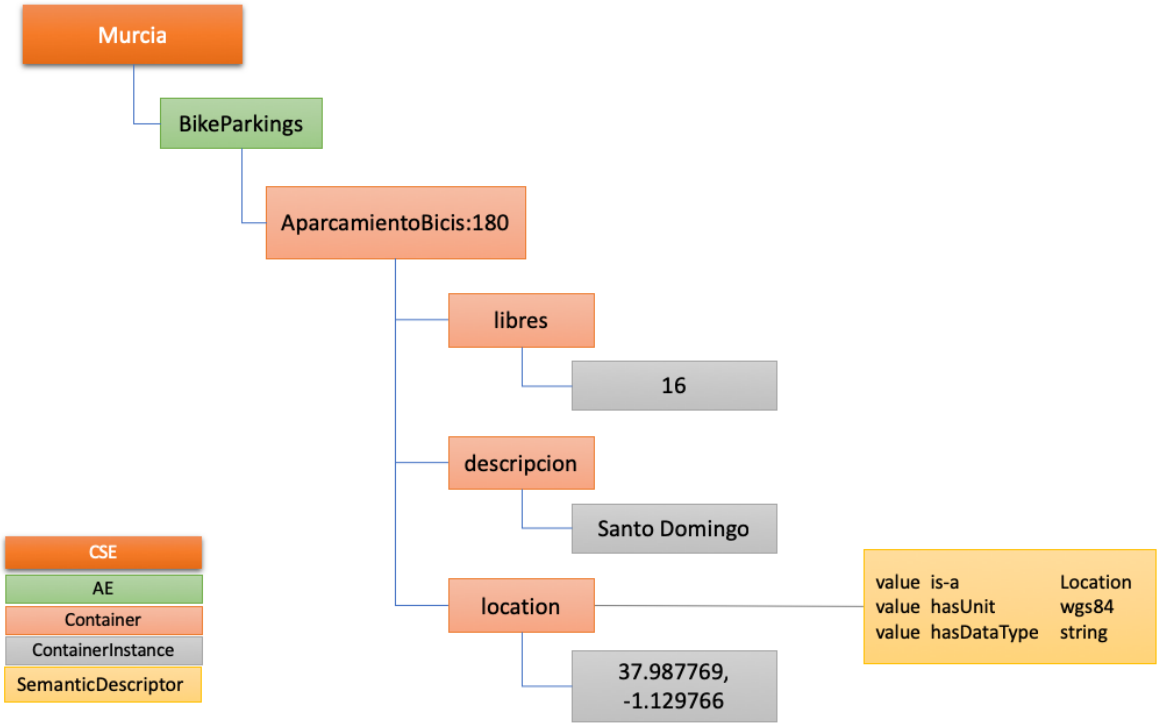


Figure 25: oneM2M instantiation of the NGSI-LD bike parking entity

# 7 System Architecture - Outlook on second release

The architecture presented in this deliverable includes all the key, main concepts such as ThingVisor, Silo, Silo controller, pub/sub internal communication system and so forth. We also considered FogFlow and ICN as development frameworks for ThingVisors. In this section, we envisage some possible extensions for the second release of the architecture.

Currently, the architecture is meant to be deployed in a single datacenter. However, it may be useful to move functions close to the data rather than vice-versa. For instance, let us consider a ThingVisor that carries out face recognition. The ThingVisor receives the full HD video stream from a camera located at the edge of the network. Accordingly, it is better to deploy the ThingVisor at the edge of the network rather than in a central data center. The second architecture release will consider the support of a distributed cloud system, up to device level (fog computing). Regarding orchestration functions, evolved cloud platforms will be considered including Kubernetes and/or ETSI MANO to foster exploitation of the VirIoT platform for 5G providers. Indeed, 5G providers can be providers of VirIoT as well. As another extension of the architecture, application of ICN in addition to the Pub/Sub system connecting ThingVisors and Virtual Silos will be considered, in order to have more flexibility in the communication among them.

Another evolution of the architecture will concern semantic aspects. Currently, ThingVisor developers must know where to fetch information, and also tenants must know which Virtual Thing to add to their virtual Silos, in terms of Virtual Thing identifier (vThingID). For the second iteration on the design of architecture, we plan to introduce semantic discovery services, which allow to ThingVisor developers to search sources of information through semantic queries. Besides, tenants can add Virtual Things to their virtual Silos just providing a semantic description of "what" they want. For instance, a tenant can ask to add a "sensor of temperature in Rome". Consequently, VirIoT will find the most suitable group of ThingVisors that provide such kind of information. One of them is then actually attached to the tenant virtual Silo, while the others are used in case of failure of the active one, i.e. the platform provides for the migration from one ThingVisor to another for reliability purposes.

Moreover, the algorithms for data conversion to and from the neutral format will be more formally defined, based on feedback from preliminary implementation.

Last but not least, the current release of the system architecture does not consider security aspects. In the second release, we will handle security issues, including authorization and authentication of the users, of system components and of data. As an example, we would have to deal with how to verify the authenticity of control and data messages distributed in the pub/sub system, and on how to control access to system elements (virtual Silos) from the user and/or from another system element.

# 8 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 different virtualization goal: to offer virtual things to the users lacking them, alleviating developers of the burden of buying and deploying IoT devices and services needed by their applications. Such virtual things are obviously based on real things, just like virtual machines are based on real hardware. We think that such a concept of sharing the IoT hardware infrastructure is not so 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] OMA, Open Mobile AllianceTM. [Online]. Available: http://www.openmobilealliance.org

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

[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] A. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang, "Nlsr: named-data link state routing protocol," in *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*. ACM, 2013.

[11] Y. Yu, A. Afanasyev, D. Clark, V. Jacobson, L. Zhang *et al.*, "Schematizing trust in named data networking," in *Proceedings of the 2nd International Conference on Information-Centric Networking*. ACM, 2015, pp. 177–186.

[12] S. Salsano, A. Detti, M. Cancellieri, M. Pomposini, and N. Blefari-Melazzi, "Transport-layer issues in information centric networks," in *Proceedings of the second edition of the ICN workshop on Information-centric networking*. ACM, 2012, pp. 19–24.

[13] 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.

[14] 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.

[15] V. Jacobson, D. K. Smetters, J. D. Thornton, M. Plass, N. Briggs, and R. Braynard, "Networking named content," *Commun. ACM*, 2012.

[16] L. Bracciale, P. Loreti, A. Detti, R. Paolillo, and N. B. Melazzi, "Lightweight Named Object: an ICN-based Abstraction for IoT Device Programming and Management," *IEEE Internet of Things Journal*, 2019.

[17] A. Detti, D. Tassetto, N. B. Melazzi, and F. Fedi, "Exploiting content centric networking to develop topic-based, publish–subscribe MANET systems," *Ad hoc networks*, 2015.