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 3.1
Cloud Oriented Services - First Release

Deliverable Type:	Report
Deliverable Number:	D3.1
Contractual Date of Delivery to the EU:	30.11.2019
Actual Date of Delivery to the EU:	30.11.2019
Title of Deliverable:	Cloud Oriented Services - First Release
Work package contributing to the Deliverable:	WP3
Dissemination Level:	Public
Editor:	Michio Honda, Hajime Tazaki
Author(s):	Michio Honda and Bin Cheng (NEC); Hajime Tazaki (IIJ); Andrea Detti, Ludovico Funari and Giuseppe Tropea (CNIT); Juan Andrs Sanchez, Juan Antonio Martinez and Antonio Skarmeta (OdinS); Hidenori Nakazato, Kenji Kanai and Hidehiro Kanemitsu (Waseda)
Internal Reviewer(s):	Andrea Detti (CNIT)
Abstract:	This deliverable reports the first version of the Fed4IoT cloud-oriented services
Keyword List:	IoT virtualization, Light Compute Virtualization, ICN, FogFlow

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

Fed4IoT Glossary

Table 1 lists and describes terms relevant to this deliverable.

Term	Definition
FogFlow	An IoT edge computing framework that automatically orchestrates dynamic data processing flows over cloud- and edge- based infrastructures. Used for ThingVisor development.
Flexible Compute Virtualization	The Fed4IoT compute virtualizatin platform able to manage heterogeneous virtualization technologies (Docker, unikernel, etc.) over cloud- and edge- based infrastructures. Used for deployment of VirIoT components.
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, Orion ans Scorpio, can be considered as Brokers of oneM2M, NGSI and NGSI-LD 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 (as opposite to virtual) 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 as a service.
Virtual Silo	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	Image of a Virtual Silo, 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 accesses the Fed4IoT VirIoT platform to develop IoT applications.

Table 1: Fed4IoT Dictionary

1 Introduction

1.1 Purpose of the Document

This deliverable describes the Fed4IoT cloud-oriented services that are positioned, within the whole Fed4IoT architectural framework, as shown in Figure 1.

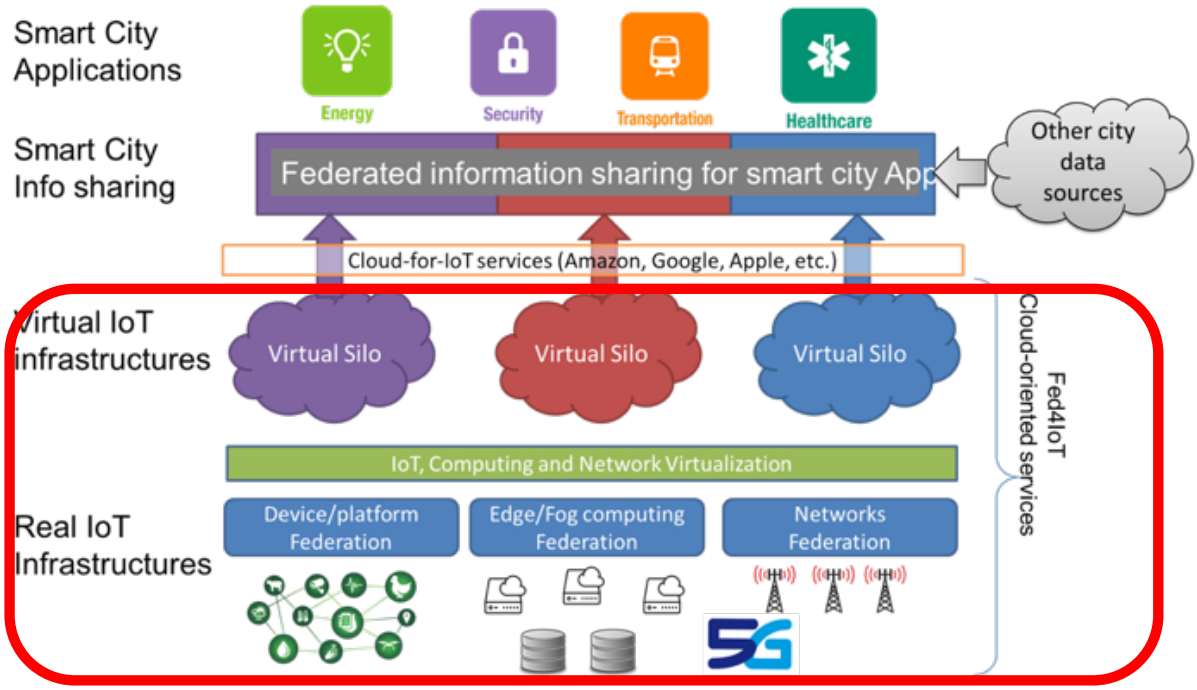


Figure 1: Fed4IoT Architectural Framework

1.2 Executive Summary

This deliverable presents the first release of Fed4IoT cloud-oriented services. Currently, only a part of them has been implemented, and the related release is internally numbered as v2.2. We provide details about the Fed4IoT IoT Virtualization Platform named VirIoT, already introduced in D2.2. We present optimization and design activities concerning specific core components, such as the ThingVisor. Finally, we describe alternative Compute Virtualization solutions to Docker to address either running low-power devices or specific application needs. The whole picture of these activities forms the so-called *Fed4IoT virtualization stack* shown in Figure 2.

1.3 Quality Review

The internal Reviewer responsible for this deliverable is Andrea Detti (CNIT).

Version Control Table			
V.	**Purpose/Changes**	**Authors**	**Date**
0.1	ToC	Michio Honda (NEC)	03/11/2019
0.2	First contribution round	ALL	20/11/2019
0.3	Last contribution round	ALL	27/11/2019
0.4	Final review	Andrea Detti (CNIT)	30/11/2019

Table 2: Version Control Table

2 The Fed4IoT Virtualization Stack

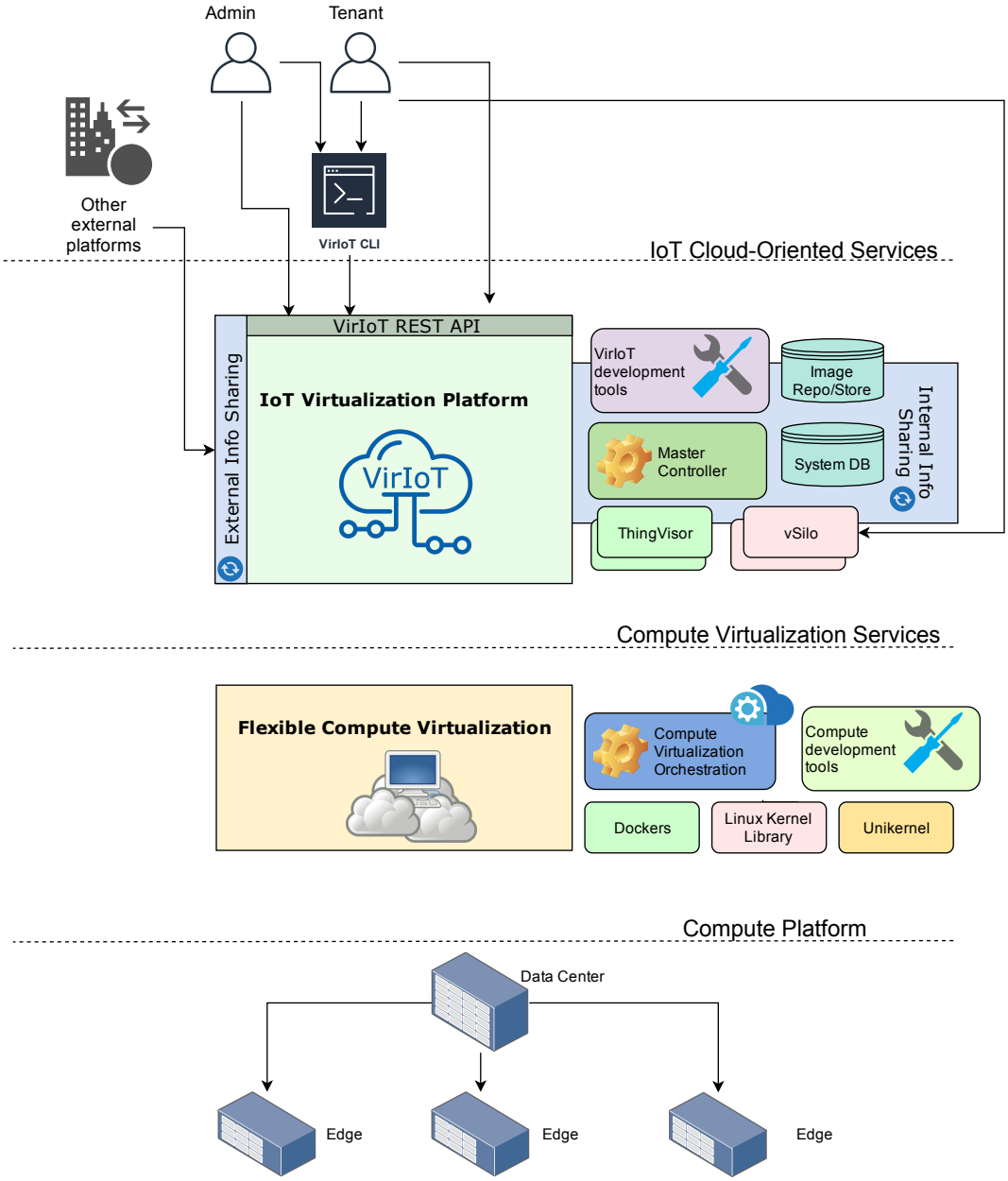


Figure 2: Fed4IoT Virtualization Stack

Figure 2 shows the stack of Fed4IoT virtualization services. Starting from the top of the figure we have the Fed4IoT **IoT Cloud-Oriented Services**, which offer IoT System-as-a-Service, where an IoT system, namely a Virtual Silo, is an isolated environment receiving data items coming from a configurable set of Virtual Things, and this information is exposed through a configurable IoT Broker inside the Virtual Silo. The IoT Cloud-Oriented Services are implemented by the VirIoT platform, previously introduced in D2.2 and now detailed in Section 3. VirIoT exposes a REST API and Command

Line Interface (CLI) to Administrators and Tenants, and also an External Info Sharing interface to other IoT platforms that wish to federate their information with information from VirIoT. Besides designing components proper of the VirIoT workflow, such as Master Controller (the VirIoT orchestrator), ThingVisors, Silo Controllers, etc..., we additionally offer VirIoT-tailored development tools (FogFlow and ICN Service Function Chaining) that simplify the development and deployment of complex and distributed ThingVisors. These VirIoT development tools are described in Section 4.

The VirIoT platform exploits compute virtualization services to deploy and run its components. In turn, compute virtualization services are based on a distributed cloud/edge computing platform. The compute virtualization architecture is flexible, i.e. its orchestrator can use different virtualization technologies to best fit the needs and constraints of the service and of the host device. For instance, for plain devices such as servers, Docker is the reference virtualization technology. For low-power devices (e.g. Raspberry PI) deployed at the network edge, the orchestrator can use the other innovative technologies the project is exploring such as Linux Kernel Libraries and Unikernels, which can be built by using Unikraft tools. Compute Virtualization Technologies are described in Section 5.

3 IoT Cloud-Oriented Services

Fed4IoT cloud-oriented services are implemented by the IoT virtualization platform named VirIoT, whose concepts have been presented in D2.2. VirIoT is made by several components and exploits the information coming from real IoT infrastructures to provide tenants with Virtual Silos (vSilos), which are virtual isolated IoT Systems receiving data generated by Virtual Things (vThings), exposed through an IoT broker technology of choice (e.g. the oneM2M Mobius server, the NGSIv2 Orion broker developed by FIWARE, or the new Scorpio NGSI-LD Context Broker by NEC, etc.) by the Tenant's applications.

Virtual Things are IoT data producers, which emulate the behaviour of real things. For instance we can have a real camera infrastructure connected to the VirIoT system, and a virtual "person counter sensor" (virtual thing), counting real-time number of persons in a room, thus producing "virtual" measurements that are generated by processing data coming from the real camera infrastructure. Within the VirIoT architecture, Virtual Things are implemented by specific components named ThingVisors, and one ThingVisor can implement more than one Virtual Thing.

A tenant can create a Virtual Silo and add to it (i.e. connect to it) its preferred set of Virtual Things, among the ones offered by the VirIoT. Virtual Silos differ each other both by the set of connected Virtual Things and by the IoT broker technology they use to expose them. When a tenant creates a Virtual Silo, the VirIoT platform runs a new instance of a so called Virtual Silo Flavour, which is an empty image of the Virtual Silo, only containing the specific IoT broker server and related Virtual Silo Controller. Such controller is a local agent interacting with the rest of the VirIoT architecture components (Master Controller, ThingVisors, etc.). As soon as a Virtual Silo runs, the tenant can connect to it the preferred Virtual Things.

As reported in D2.2, the VirIoT architecture is formed by the following components:

1. Master Controller: the main orchestration component, exposing a REST API to the tenants and to the administrator, and in turn interacting with other components, to manage ThingVisors and Virtual Silos. A python CLI, too, can be used to interact with the Master controller.

2. ThingVisors: a pluggable set of components, each of them implementing one or more Virtual Things.

3. Internal information sharing system: a data distribution system used to transfer, within the platform boundaries, the control messages and the data generated by the Virtual Things.

4. External information sharing system: a data distribution system used to transfer the data of Virtual Things that the VirIoT platform wishes to share with external IoT platforms.

5. Image repositories: image stores (e.g. DockerHub) hosting base images of Virtual Silos (Flavours) and ThingVisors.

6. Virtual Silos: instances of a Virtual Silo Flavour that include a Virtual Silo Controller and an IoT broker, and that receive and exposes data of a set of Virtual Things, as selected by the tenant.

7. System DB: database containing system information, mainly used to make the system components as much as possible stateless.

8. Compute Virtualization Layer: a compute virtualization architecture, exposing services for deploying different types of compute images (Docker, Unikernel, etc.) on a distributed and heterogeneous cloud platform formed by central and edge resources.

The VirIoT components are devised to run on cloud/edge/fog computing platforms, including 5G NFV platforms.

In what follows we describe the status of VirIoT components at the latest stable release (v2.2). Current implementation of the architecture is centralized and based on Docker container. The next release will be distributed on edge/fog nodes. Accordingly, we also report in this deliverable the advancements in light Compute Virtualization Technology (Unikraft, Linux Kernel Library) the project is proposing to enable deployment of the VirIoT components on edge/fog low-power devices, and finally we presents some tools for developing and orchestrating the internal components of complex and distributed ThingVisors.

3.1 VirIoT REST API and Command Line Interface

The following several sections give full details of all API functionality of the VirIoT platform, available both through a RESTful and through a command line option.

3.1.1 Registration

It registers a new user to the system. The administrator sends a request with a new triplet that needs to be registered, formed as the new user ID, the password and its role. A new user can either be registered as an admin or as a regular user.

Needed privileges: Administrator.

3.1.1.1 CLI

```
f4i.py register [-h] [-c CONTROLLERURL] [-u USERID] [-p PASSWORD] [-r
    ROLE]
```

CLI Example

```
python3 f4i.py register -c http://127.0.0.1:8090 -u tenant1 -p password -
    r user
```

3.1.1.2 REST API

POST register

```
127.0.0.1:8090/register
```

HEADERS:

Authorization: "Bearer" + ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
"message":registration_status
}
```

3.1.2 Unregistration

It unregister the user, deleting the credentials from the SystemDB.
Needed privileges: Administrator.

3.1.2.1 CLI

```
f4i.py unregister [-h] [-c CONTROLLERURL] [-u USERID]
```

CLI Example

```
python3 f4i.py unregister -c http://127.0.0.1:8090 -u tenant1
```

3.1.2.2 REST API

POST unregister

```
127.0.0.1:8090/unregister
```

HEADERS:

Authorization: "Bearer" + ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
"message":unregistration_status
}
```

3.1.3 Login

It lets a user login into the system. The Master Controller checks if the information given by the users are correct. In order to do so, it validates them with the information fetched from the SystemDB. If the operation is successful, the user receives an `access token` that allows the user to perform subsequent CLI commands.

Needed privileges: Regular user.

3.1.3.1 CLI

```
f4i.py login [-h] [-c CONTROLLERURL] [-u USERID] [-p PASSWORD]
```

CLI Example

```
python3 f4i.py login -c http://127.0.0.1:8090 -u tenant1 -p password
```

3.1.3.2 REST API

POST login

```
127.0.0.1:8090/login
```

HEADERS:

Authorization: "Bearer" + ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
"message":login_status
}
```

3.1.4 Logout

It logs out a user from the system. The `access_token` of the specified user is added to a local black-list, that will be used to deny a further login using the aforementioned token.

Needed privileges: Regular user.

3.1.4.1 CLI

```
f4i.py logout [-h] [-c CONTROLLERURL]
```

CLI Example

```
python3 f4i.py logout -c http://127.0.0.1:8090
```

3.1.4.2 REST API

POST logout

```
127.0.0.1:8090/logout
```

HEADERS:

Authorization: "Bearer" + $\langle JSONWebToken \rangle$
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
"message":logout_status
}
```

3.1.5 Create Virtual Silo

It creates a new Virtual Silo, unique for each tenant, with the characteristics and resources specified by the chosen flavour (default: *Mobius-base-f*). If debug mode is enabled, it will store information only in the SystemDB. This command will return the private IP address of the vSilo broker, the port to be used for accessing it and the port mapping by which it is possible to access the broker using the public IP address of the platform.

Needed privileges: Administrator.

3.1.5.1 CLI

```
f4i.py create-vsilo [-h] [-c CONTROLLERURL] [-s VSILONAME] [-t TENANTID]
    [-f FLAVOURNAME] [-d DEBUG_MODE]
```

CLI Example

```
python3 f4i.py create-vsilo -c http://127.0.0.1:8090 -t tenant1 -f Mobius
    -base-f -s Silo1
```

3.1.5.2 REST API

POST siloCreate:

```
127.0.0.1:8090/siloCreate
```

HEADERS:

Authorization:	"Bearer" + $\langle JSONWebToken \rangle$
Accept:	application/json
Content-Type:	application/json

RESPONSE:

```
{
  "creationTime":creation_time,
  "tvDescription":tv_description,
  "containerID":container_id,
  "thingVisorID":tv_id,
  "imageName":tv_img_name,
  "ipAddress":ip_address,
  "debug_mode":debug_mode,
  "vThings":v_things,
  "params":tv_params,
  "MQTTDataBroker":mqtt_data_broker,
  "MQTTControlBroker":mqtt_control_broker,
  "port":exposed_ports,
  "IP":floating_public_IP,
  "status":status
}
```

3.1.6 Destroy Virtual Silo

It deletes the unique, previously created Virtual Silo associated to the user. The tenant container will be deleted, as well as any data related to the user.

Needed privileges: Administrator.

3.1.6.1 CLI

```
f4i.py destroy-vsilo [-h] [-c CONTROLLERURL] [-t TENANTID] [-s VSILONAME]
     [-f]
```

CLI Example

```
python3 f4i.py destroy-vsilo -c http://127.0.0.1:8090 -t tenant1 -s Silo1
```

3.1.6.2 REST API

POST siloDestroy

```
127.0.0.1:8090/siloDestroy
```

HEADERS:

Authorization: "Bearer" + ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
    "force":force_condition,
    "tenantID":tenant_id,
    "vSiloName":v_silo_id"
}
```

3.1.7 Add Virtual Thing

It adds a new Virtual Thing to the indicated Virtual Silo. The vThing is created with the specified ID and it is associated to the tenant. It will fail if the silo does not exist.

Needed privileges: Administrator.

3.1.7.1 CLI

```
python3 f4i.py add-vthing -c <ControllerURL> -t <TenantID> -s <vSiloName>
    -v <vThingID>
```

CLI Example

```
python3 f4i.py add-vthing -c http://127.0.0.1:8090 -t tenant1 -s Silo1 -v
    weather/Tokyo_temp
```

3.1.7.2 REST API

POST addVThing

```
127.0.0.1:8090/addVThing
```

HEADERS:

Authorization: "Bearer" + ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
  "tenantID":tenant_id,
  "vThingID":v_thing_id,
  "creationTime":creation_time,
  "vSiloID":v_silo_id
}
```

3.1.8 Delete Virtual Thing

It deletes the Virtual Thing from the associated vThing of the tenant specified by its ID.
Needed privileges: Administrator.

3.1.8.1 CLI

```
f4i.py del-vthing [-h] [-c CONTROLLERURL] [-t TENANTID] [-s VSILONAME] [-
    v VTHINGID]
```

CLI Example

```
python3 f4i.py del-vthing -c http://127.0.0.1:8090 -t tenant1 -s Silo1 -v
     weather/Tokyo_temp
```

3.1.8.2 REST API

POST deleteVThing

```
127.0.0.1:8090/deleteVThing
```

HEADERS:

Authorization: "Bearer" + $\langle JSONWebToken \rangle$
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
  "tenantID":tenant_id,
  "vSiloName":v_silo_id,
  "vThingID":v_thing_id
}
```

3.1.9 Add ThingVisor

It adds a ThingVisor. The administrator can add a new ThingVisor from an image and a list of parameters written in JSON. It returns the ID of the newly added ThingVisor.

Needed privileges: Administrator.

3.1.9.1 CLI

```
f4i.py add-thingvisor [-h] [-c CONTROLLERURL] [-i IMAGENAME] [-n NAME] [-
    p PARAMS] [-d DESCRIPTION] [--debug DEBUG_MODE]
```

CLI Example

```
python3 f4i.py add-thingvisor -c http://127.0.0.1:8090 -i fed4iot/v-
    weather-tv:2.2 -n weather -p "{'cities':['Rome', 'Tokyo','Murcia','
    Grasse','Heidelberg'], 'rate':60}" -d "Weather ThingVisor"
```

3.1.9.2 REST API

POST addThingVisor

```
127.0.0.1:8090/addThingVisor
```

HEADERS:

Authorization: "Bearer" + $\langle JSONWebToken \rangle$
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
  'thingVisorID':thingvisor_id
}
```

3.1.10 Delete ThingVisor

It deletes a ThingVisor. In this way it will get rid of any information about the specified ThingVisor from the SystemDB.

Needed privileges: Administrator.

3.1.10.1 CLI

```
f4i.py del-thingvisor [-h] [-c CONTROLLERURL] [-n NAME] [-f]
```

CLI Example

```
python3 f4i.py del-thingvisor -c http://127.0.0.1:8090 -n weather
```

3.1.10.2 REST API

POST deleteThingVisor

```
127.0.0.1:8090/deleteThingVisor
```

HEADERS:

Authorization: "Bearer" + $\langle JSONWebToken \rangle$
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
  'thingVisorID':thingvisor_id
}
```

3.1.11 Add Flavour

It adds a new flavour to the database. The administrator specifies the new Flavour ID as well as its parameters. In particular, the operation is carried on if the image for the new flavour exists in the registered image repositories, saving the flavour status in the SystemDB, otherwise, the operation is cancelled by the Master Controller.

Needed privileges: Administrator.

3.1.11.1 CLI

```
f4i.py add-flavour [-h] [-c CONTROLLERURL] [-f FLAVOURID] [-s
    FLAVOURPARAMS] [-i IMAGENAME] [-d DESCRIPTION]
```

CLI Example

```
python3 f4i.py add-flavour -c http://127.0.0.1:8090 -f Mobius-base-f -s
    Mobius -i fed4iot/mobius-base-f:2.2 -d "silo with a oneM2M Mobius
    broker"
```

3.1.11.2 REST API

POST addFlavour

```
127.0.0.1:8090/addFlavour
```

HEADERS:

Authorization: "Bearer" + $\langle JSONWebToken \rangle$
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
  "flavourID":flavour_id
}
```

3.1.12 Delete Flavour

It deletes a flavour. It will reverse the operation seen in the previous Section 3.1.11, updating the information about the flavour specified by the command, deleting the stored data in the SystemDB.

Needed privileges: Administrator.

3.1.12.1 CLI

```
f4i.py del-flavour [-h] [-c CONTROLLERURL] [-n FLAVOURID]
```

CLI Example

```
python3 f4i.py del-flavour -c http://127.0.0.1:8090 -n Mobius-base-f
```

3.1.12.2 REST API

POST deleteFlavour

```
127.0.0.1:8090/deleteFlavour
```

HEADERS:

Authorization: "Bearer" + $\langle JSONWebToken \rangle$
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
  "flavourID":flavour_id
}
```

3.1.13 Inspect Tenant

It dumps information about a tenant. In particular, it will print all the information about the vSilos and vThings associated to the specified tenant ID.

Needed privileges: Administrator.

3.1.13.1 CLI

```
f4i.py inspect-tenant [-h] [-c CONTROLLERURL] [-t TENANTID]
```

CLI Example

```
python3 f4i.py inspect-tenant -t tenant1
```

3.1.13.2 REST API

POST inspectTenant

```
127.0.0.1:8090/inspectTenant
```

HEADERS:

Authorization: "Bearer" + $\langle JSONWebToken \rangle$
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
  'vSilos':v_silo_description,
  'vThings':v_thing_id_description
}
```

3.1.14 Inspect Virtual Silo

It prints information about the Virtual Silos. In particular, it will print the description of the vSilo as well as its Virtual Things.

Needed privileges: Administrator.

3.1.14.1 CLI

```
f4i.py inspect-vsilo [-h] [-c CONTROLLERURL] [-v VSILOID]
```

CLI Example

```
python3 f4i.py inspect-vsilo -c http://127.0.0.1:8090 -v tenant1_Silo1
```

3.1.14.2 REST API

POST inspectVirtualSilo

```
127.0.0.1:8090/inspectVirtualSilo
```

HEADERS:

Authorization: "Bearer" + $\langle JSONWebToken \rangle$
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
  'vSilos':v_silo_description,
  'vThings':v_thing_id_description
}
```

3.1.15 Inspect ThingVisor

It prints out a description of the requested ThingVisor ID.
Needed privileges: Administrator.

3.1.15.1 CLI

```
f4i.py inspect-thingvisor [-h] [-c CONTROLLERURL] [-v THINGVISORID]
```

CLI Example

```
python3 f4i.py inspect-thingvisor -c http://127.0.0.1:8090 -v weather
```

3.1.15.2 REST API

POST inspectThingVisor

```
127.0.0.1:8090/inspectThingVisor
```

HEADERS:

Authorization: "Bearer" + ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
  "thingVisorID":thingvisor_id_description
}
```

3.1.16 List ThingVisors

It prints the information about the ThingVisors. In particular, it will dump all the details, including the parameters passed by the add command, as well as the vThings associated to the ThingVisors.

Needed privileges: Administrator.

3.1.16.1 CLI

```
f4i.py list-thingvisors [-h] [-c CONTROLLERURL]
```

CLI Example

```
python3 f4i.py list-thingvisors -c http://127.0.0.1:8090
```

3.1.16.2 REST API

GET listThingVisors

```
127.0.0.1:8090/listThingVisors
```

HEADERS:

Authorization: "Bearer" + ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
  "thingVisorDescription":thing_visor_description
}
```

3.1.17 List Flavours

It lists all the flavours saved inside SystemDB, previously created using the command in Section 3.1.11.

Needed privileges: Regular user.

3.1.17.1 CLI

```
f4i.py list-flavours [-h] [-c CONTROLLERURL]
```

CLI Example

```
python3 f4i.py list-flavours -c http://127.0.0.1:8090
```

3.1.17.2 REST API

GET listFlavours

```
127.0.0.1:8090/listFlavours
```

HEADERS:

Authorization: "Bearer" + ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
  "flavourDescription":flavour_description
}
```

3.1.18 List Virtual Silos

It interrogates the SystemDB to dump all the details about the Virtual Silos saved.
Needed privileges: Administrator.

3.1.18.1 CLI

```
f4i.py list-vsilos [-h] [-c CONTROLLERURL]
```

CLI Example

```
python3 f4i.py list-vthings -c http://127.0.0.1:8090
```

3.1.18.2 REST API

GET listVirtualSilos

```
127.0.0.1:8090/listVirtualSilos
```

HEADERS:

Authorization: "Bearer" + $\langle JSONWebToken \rangle$
Accept: application/json
Content-Type: application/json

RESPONSE:

```
{
  "vSiloDescription":v_silo_description
}
```

3.2 VirIoT Internal Procedures and Data Flow

This section describes the procedures and the related internal signaling that support both the management operations described in section 3.1 and the internal data flows that deliver the data.

VirIoT control commands (JSON)

{command:[value], [arg1]:[value], [argN]:[value]}

Command	arguments:value	Senders	Receivers	Description of Actions
addVThing	vSiloID: <vSiloID> vThingID: <ThingID>	Master Controller	vSilo Controller	Add a vThing to the vSilo whose ID is vSiloID
deleteVThing	vSiloID: <vSiloID> vThingID: <ThingID>	Master Controller	vSilo Controller	Disconnect the vThing from the vSilo whose ID is vSiloID
createVThing	thingVisorID: <thingVisorID> vThing:{ label: <vThingLabel> id: <vThingID> description: <vThingDescription> }	ThingVisor	Master Controller	Notify the creation of a new VirtualThing Note: <vThingID> must be equal to <thingVisorID>/<vThingLID> where vThingLID is a local identifier, e.g. a random number
destroyTV	thingVisorID: <thingVisorID>	Master Controller	ThingVisor	Destroy thing Visor request
destroyTVAck	thingVisorID: <thingVisorID>	ThingVisor	Master Controller	Destroy thing Visor ack
destroyVSilo	vSiloID: <vSiloID>	Master Controller	vSilo Controller	Notify the silo of an impending shutdown of the container
destroyVSiloAck	vSiloID: <vSiloID>	Vsilo Controller	Master Controller	Destroy virtual silo ack
deleteVThing	vSiloID: ALL vThingID: <ThingID>	ThingVisor	vSilo Controller	Delete from all silos the vThing whose ID is <thingVisorID>/<vThingLID>
getContextRequest	vSiloID: <vSiloID> vThingID: <ThingID>	vSilo Controller	ThingVisor	Get the current status of the virtual Thing whose id is vThingID
getContextResponse	data: [NGSI-LD Entity array] meta: { vThingID: <ThingID> }	ThingVisor	vSilo Controller	Response to getContextRequest message. The payload of the message is an array of NGSI-LD entities

Figure 3: Control commands exchanged among VirIoT entities

3.2.1 User management procedures

Figure 4 shows the register procedure used to insert a user in the VirIoT system. At the bottom of each figure, we will show the protocols used among the involved entities. We assume two possible roles for a user: Administrator or Tenant.

During the register procedure the Admin sends a register request (HTTPs POST) to the Master Controller using the `/register` REST resource, including JSON information such as `userID`, `password`, `role` that are related to the user the Admin is registering. Moreover, the Admin sends her `access_token` (a JSON Web Token - JWT) in the Auth header of the HTTP POST. In this and next figures, we use curly brackets to indicate JSON content and we will use the symbol # to indicate a generic value of the key. The Master Controller verifies the validity of the access token and consequently stores user

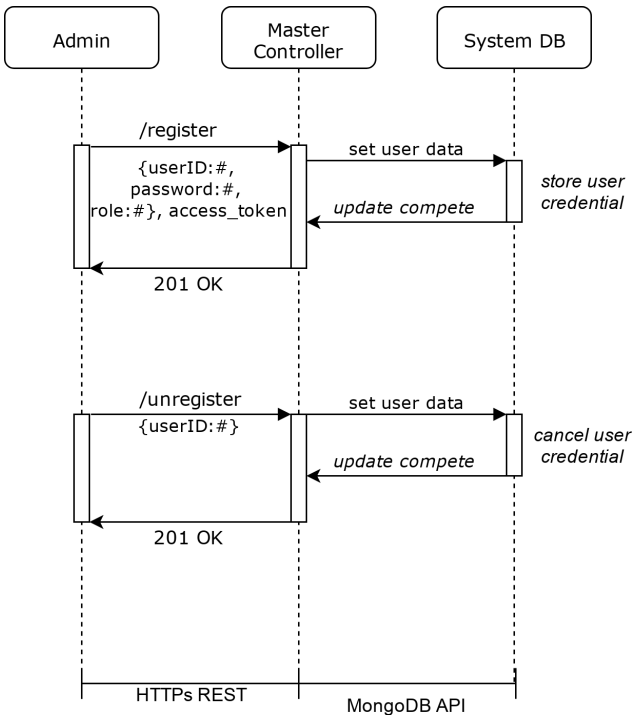


Figure 4: Register/Unregister procedure

information in the System DB, currently MongoDB. Only the HASH of the password is actually stored in the DB, so that only the user knows her real password. Figure 4 also shows the unregister procedure that is very similar, except that here the Master Controller removes user information from the System DB.

Figure 5 shows the login procedure that is used by a user (e.g. a tenant) to get her `access_token`. The user sends her credentials through HTTPs to the REST `/login` resource. The Master Controller accesses the System DB to fetch user info, then verifies the password HASH and, if the verification is successful, it sends back to the user her `access_token` that will be used within subsequent procedures for authentication purposes. Figure 5 also shows the logout procedure in which the `access_token` of the user is inserted in a local black-list, thus it can no longer be used.

3.2.2 Virtual Silo Procedures

This section reports procedures related to the creation and removal of a Virtual Silo (vSilo). We remind that a vSilo starts as an instance of a "void" Virtual Silo image that we named *Flavour*. A vSilo is deployed exploiting the service of a Compute Virtualization Layer, such as Docker/Kubernetes. In these specific cases a Flavour is a Docker image, and a vSilo is a Container/Pod. Besides Docker technology, other Light Compute Virtualization Technologies are under study as reported in section 5. However the current architecture version (V2.2) is centralized and based on Docker.

Figure 6 shows the procedure used by the Admin to add a Flavour to the VirIoT system. The Admin uses the `/addFlavour` REST resource to transfer information of the Flavour, such as description, image name (a URL), flavour identifier, etc. The Master

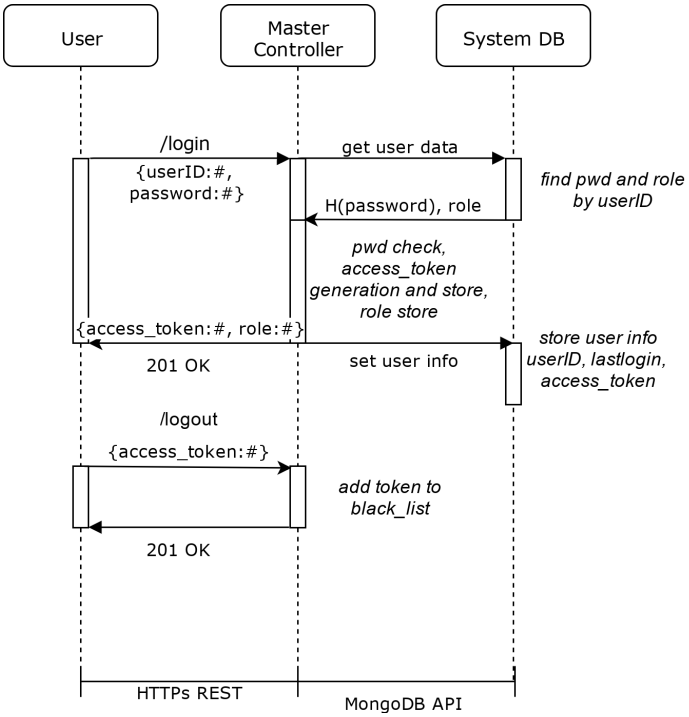


Figure 5: Login/Logout procedure

Controller register this information in the System DB, by setting the Flavour status as PENDING. Immediately, it sends back an HTTP 201 OK to the Tenant to quickly release the connection. We used such "transient" statuses in many other procedures to quickly disconnect the HTTP client, hence avoiding it to stay connected for many seconds, especially for those procedures that require long backend operations, such as the download of a Docker image. Thereafter, the Master Controller checks the availability of the image on the registered image repositories (e.g. DockerHub or local repo); if the image exists, the Master Controller changes the status to READY, otherwise, it cancels the pending entry in the System DB. Figure 6 also shows the delete flavour procedure in which the Master Controller removes flavour data from the System DB.

Figure 7 shows how a Tenant can create a Virtual Silo (vSilo). The Tenant sends to the Master Controller some information such as the identifier of the vSilo flavour to be used as the base image and the name she wishes to use for the Virtual Silo. The Master Controller registers the information of the new Virtual Silo in the System DB and sets the status as PENDING. Thereafter, it requests to the Compute Virtualization Layer to deploy the instance of the vSilo. The Compute Virtualization Layer downloads the related image (Flavour) from the Repository and runs a new instance of it. When the deployment is complete the Master Controller stores in the system DB the vSilo configuration information, such as the private IP address, the exposed port, the public IP address, the instance ID (e.g. Docker container name), etc. and set vSilo status as RUNNING.

Figure 8 shows how a Tenant can destroy a Virtual Silo (vSilo). The Tenant sends to the Master Controller information such as the identifier of the vSilo to be destroyed. The Master Controller gets the vSilo information from the System DB, sets the status as

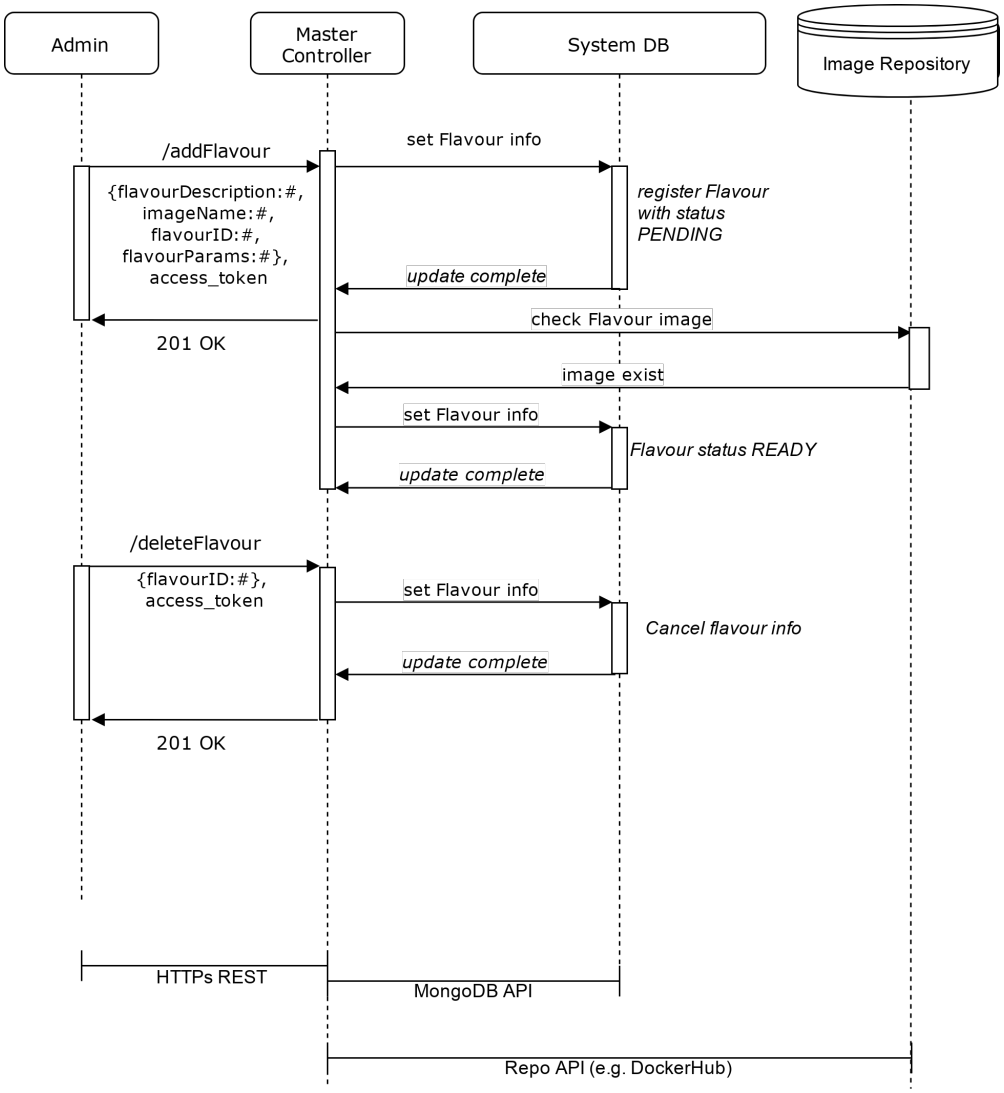


Figure 6: Add Flavour procedure

STOPPING and replies to the user with an HTTP 201 OK. Then, the Master Controller asks the vSilo Controller to start the destroy procedure. This message is a VirIoT *control command* delivered through the Internal Info Sharing VirIoT sub-system, which is developed by WP4 (see deliverables D4.x). In the current release (v2.2), the Internal Info Sharing is based on MQTT data and control topics, as reported in D4.1. However, many solutions are possible because the devised data and control messages are self-consistent and self-standing, i.e. the related actions solely depend on their own contents. Figure 3 shows the list of all VirIoT control commands, that will be used in the following.

When the vSilo Controller receives the `destroyVSilo` command it friendly closes all possible relations with external sources and then confirms to the Master Controller that it is ready to be destroyed with a `destroyVSiloAck` command. At the reception of this message, the Master Controller requests to the Compute Virtualization Layer to destroy the vSilo instance. When the destroy operation is completed, the Master Controller removes vSilo information from the System DB.

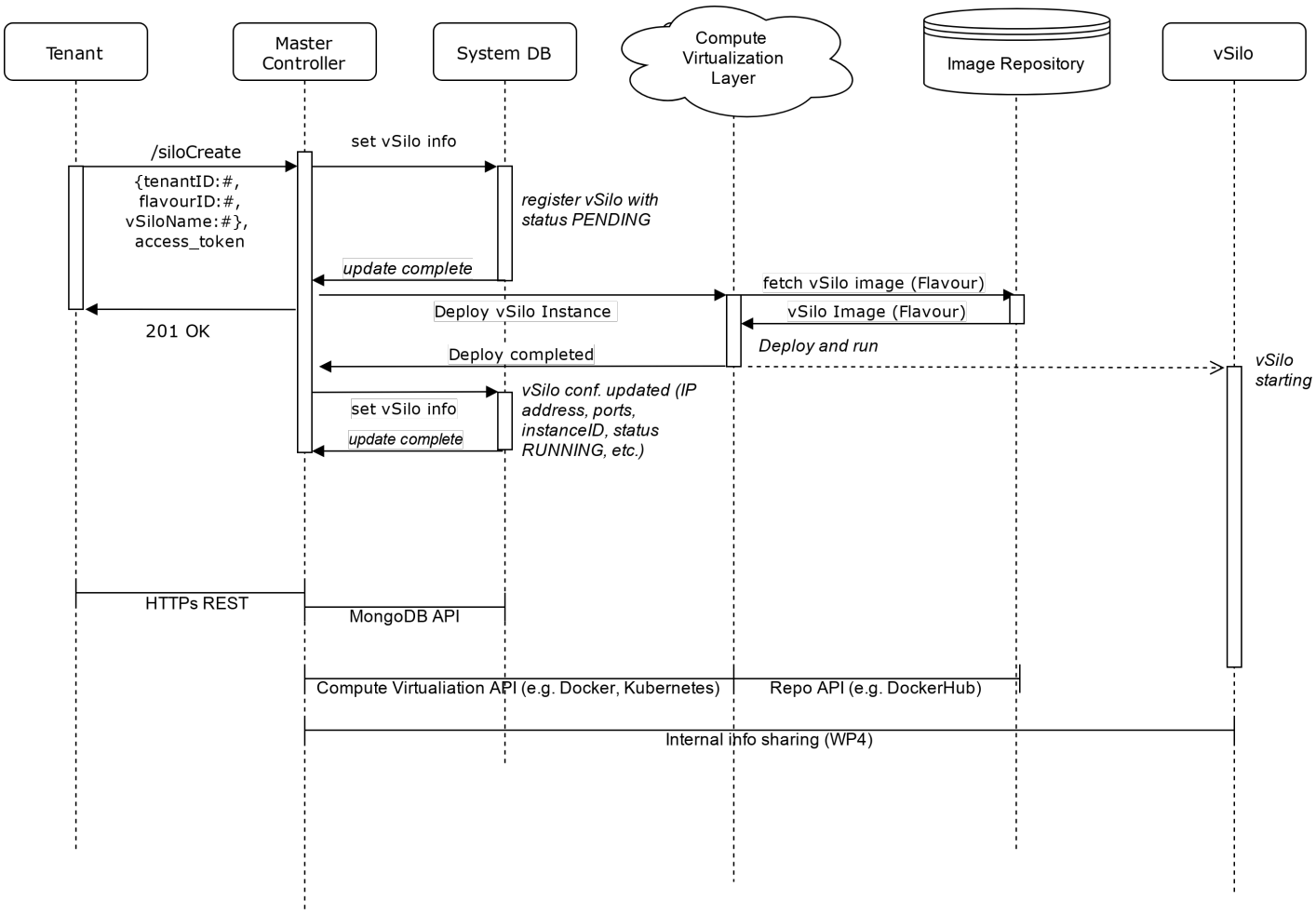


Figure 7: Create vSilo procedure

3.2.3 ThingVisor procedures

Figure 9 shows the procedure used by the Admin to add a ThingVisor to the VirIoT system. The Admin contacts the Master Controller through the `/addThingVisor` REST resource and sends ThingVisor information such as a description, its image name, the parameters that can be used to customize the added ThingVisor (e.g. info about the real sources that should be contacted by the ThingVisor), etc. The Master Controller registers ThingVisor information in the SystemDB, sets the status as PENDING, then sends back an HTTP 201 OK to the Admin. Thereafter, the Master Controller asks the Compute Virtualization Layer to deploy an instance of the ThingVisor. The Virtualization Layer fetches the image from the repository and runs it. In turn, the Master Controller updates the ThingVisor information in the System DB by setting its status to RUNNING.

A ThingVisor may manage more than one Virtual Things (vThings), which are not known apriori by the Master Controller. For this reason, when the ThingVisor starts, it sends to the Master Controller the `createVThing` command, which includes the list of vThings it is handling. At the reception of this message, the Master Controller updates the system DB with vThings' information. The running ThingVisor then starts to pro-

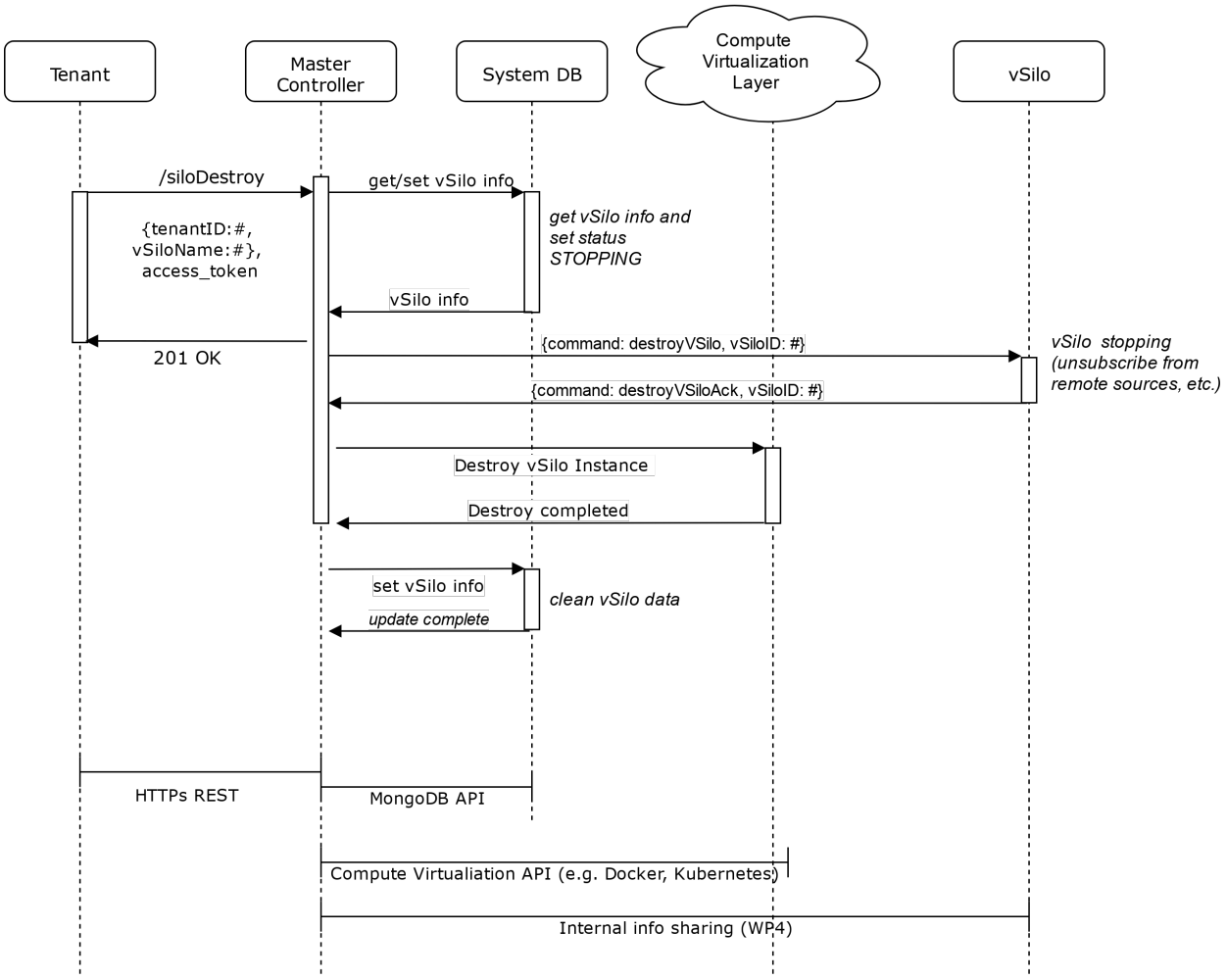


Figure 8: Destroy vSilo procedure

duce the data items (NGSI-LD format) of the managed vThings and send them to the interested vSilos by using the Internal Info Sharing System (see D4.x).

Figure 10 shows the procedure used by a Tenant to connect (add) a vThing to her vSilo. The Tenant sends vThing and vSilo information to the Master Controller by using the `/addVThing` REST resource. The Master Controller checks the existence of vThing and vSilo, and then send the `addVThing` control command to the Controller of the vSilo. The Controller configures the local broker of the vSilo, enables the reception of vThing data (e.g. by subscribing to the related data topic in case of MQTT-based Internal Info Sharing) and requests the latest data published by the vThing to the ThingVisor by using the `getContexRequest` control command. Meanwhile, the Master Controller updates the vSilo information on the System DB.

Figure 11 shows the procedure used by a Tenant to disconnect (delete) a vThing from her vSilo. The Tenant sends vThing and vSilo information to the Master Controller by using the `/deleteVThing` REST resource. The Master Controller checks the existence of vThing and vSilo, updates System DB hence removing the vThing from the vSilo, and then sends the `delVThing` control command to the Controller of the vSilo. The Controller

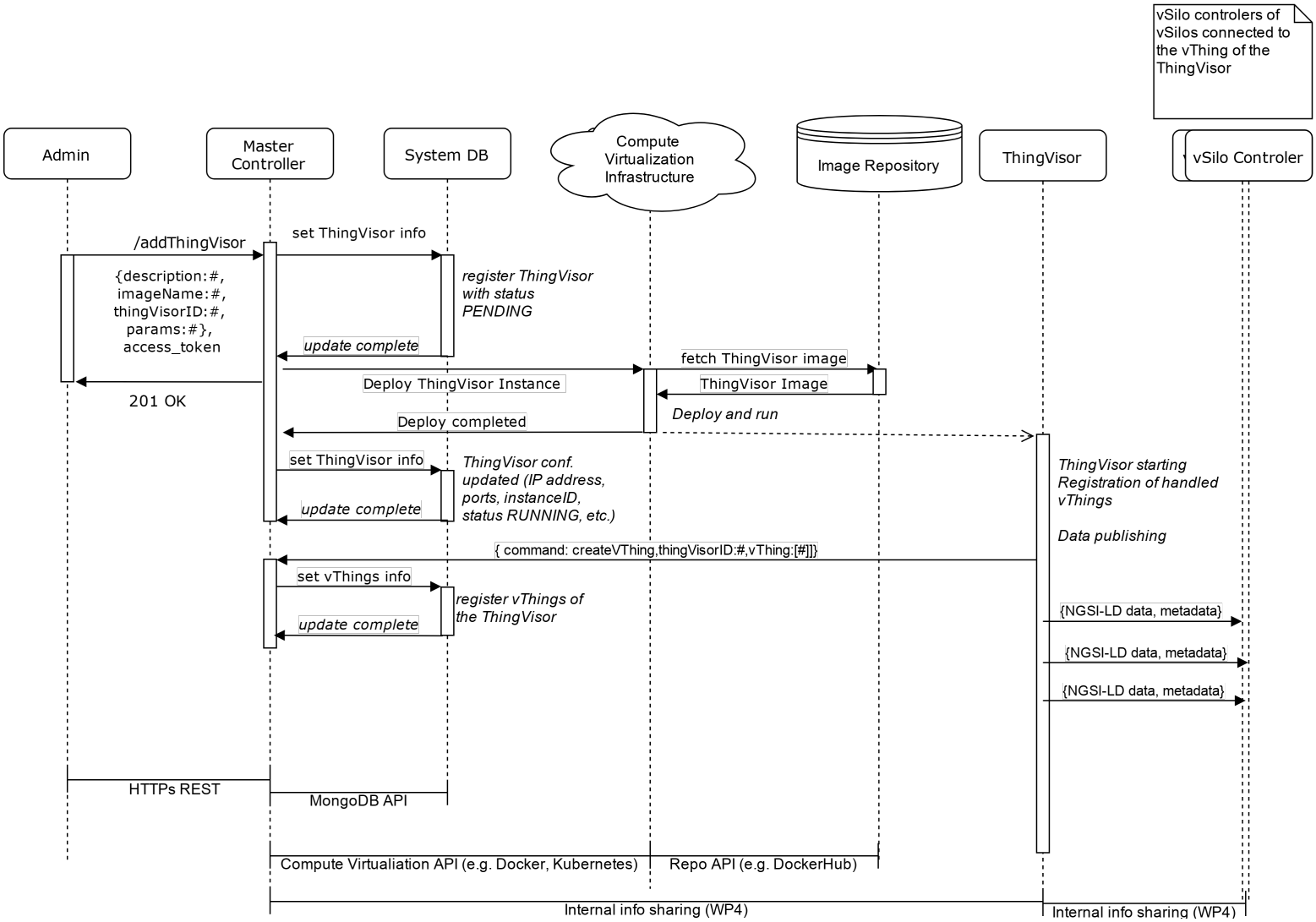


Figure 9: Add ThingVisor procedure

stops receiving vThing data and removes related entries from the local broker.

Figure 12 shows the procedure to remove a ThingVisor from the VirIoT system. The Admin contacts the Master Controller by using the `/deleteThingVisor` REST resource and passing the related ThingVisor ID. The Master Controller gets from the system DB information about the vSilos that are connected to the vThings handled by the ThingVisor, and sets the ThingVisor status as STOPPING. Then, the Master Controller sends `delVThing` control commands to the interested vSilos, so that they can remove the vThings of the ThingVisor from their brokers and stop receiving related data. Afterwards, the Master Controller asks the ThingVisor to stop itself by using the `destroyTV` control command. At the reception of this command, the ThingVisor revokes possible external states (e.g. subscriptions with remote sources, etc.) and then confirms that it ready to be destroyed by sending back the `destroyTVAck` control message to the Master Controller. When received, the Master Controller eventually asks the Compute Virtualization Layer to remove the ThingVisor instance.

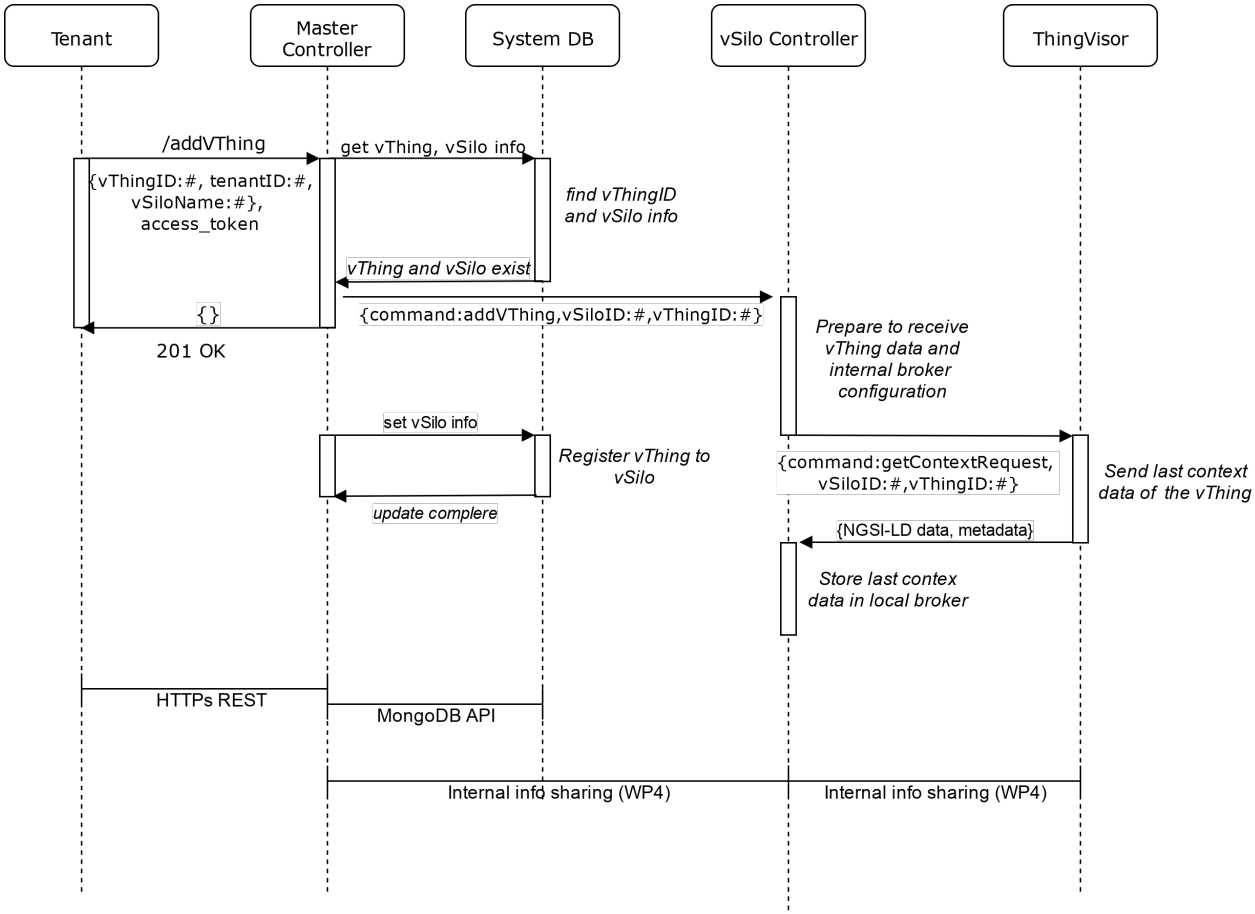


Figure 10: Add vThing procedure

3.2.4 SystemDB

The System DB stores the run-time configuration of VirIoT. Currently, the database in use is MongoDB. The data is organised in collections, which are (to some extent) analogous to tables in relational databases. A collection stores documents, which can be different in structure, and this is possible since MongoDB is a NoSQL and thus a schema-free database. For the time being, the information is classified into five collections, namely:

1. flavourCollection
2. thingVisorCollection
3. userCollection
4. vSiloCollection
5. vThingCollection

Table 3 shows the general description of the collections, as well as a practical example for each one of them. In general, any collection can be added, removed and retrieved using

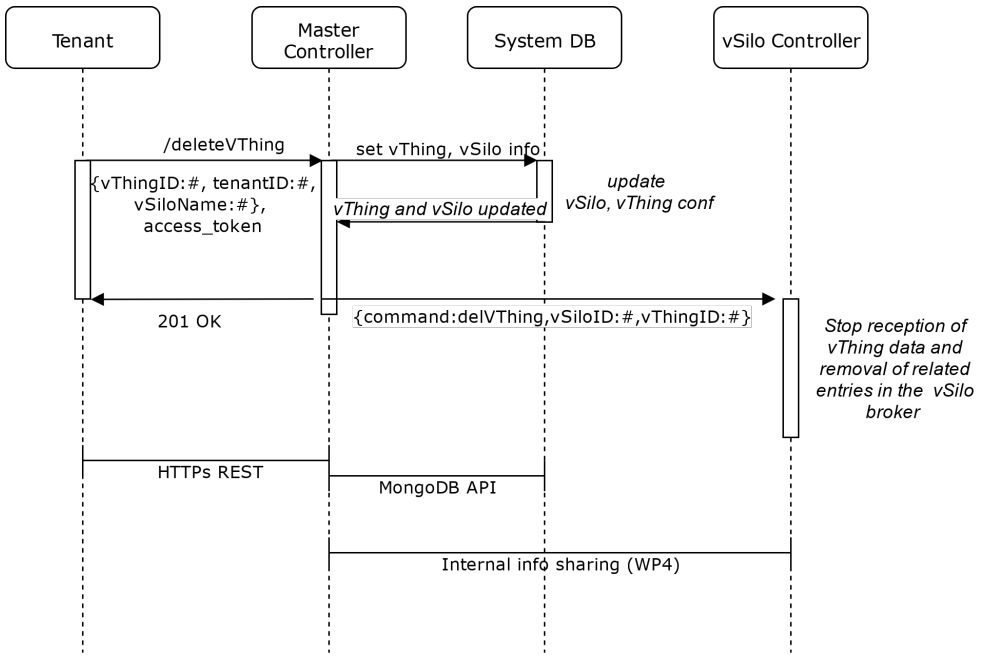


Figure 11: Delete vThing procedure

the Command Line Interface. When new information must be stored, a new collection is added inside the database.

The `flavourCollection` stores information about the flavours, like the used image name, that the Master Controller employs to check and download the image from the Image Repository. It is used, for example, when the administrator adds a new vSilo. Inside the `thingVisorCollection` information about the thingVisors is stored, such as its instance (e.g. container) ID, IP address, image name and all the vThings associated with it, as well as the IP address and port of the MQTT data and control broker, in case of internal information sharing based on MQTT (see D4.1). After the registration phase of a new user is completed, the `userCollection` is used to save her credentials, namely, her ID, password and role, along with her JSON Web Token that is essential to the user performing any action to the platform. The passwords are not in cleartext, but an hashed version of them is stored. Lastly, the lastLogin field is updated whenever the user logs into the system. The `vSiloCollection` holds the information about the vSilos. In particular, other than the IP address, ports, instance (container) ID and image name, it contains the user ID of the unique owner of the vSilo, the tenantID. Likewise, the collection that stores information about the vThings, the `vThingCollection`, contains the ID of the tenant and of the vSilo it is attached to, the vSiloID. It is a string formed by the tenant and vSilo ID, like it can be seen in the example in the Table 3, “tenant1_Silo1”.

3.3 Developed ThingVisors and Virtual Silo Flavours

This subsection reports the various ThingVisor and the different vSilo flavours we have developed so far.

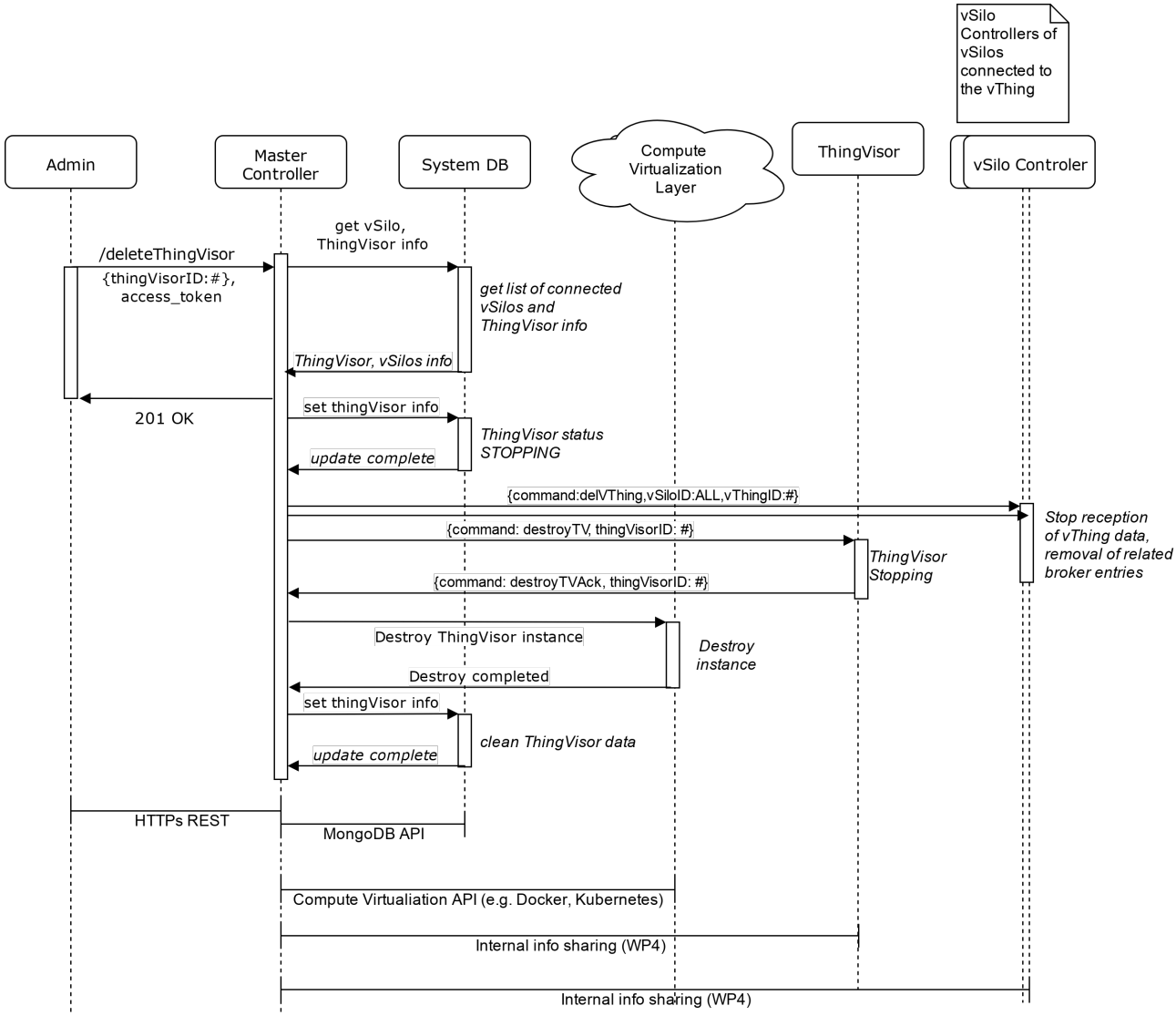


Figure 12: Delete ThingVisor procedure

We recall (please refer to D2.2 for the architecture details) that:

- ThingVisors fetch or receive data from remote systems (sensors, web services, Context Brokers, data providers, etc...), that comes in various formats, and create Virtual Things (vThings), which are able to present the same or new data, possibly after reshaping, manipulation or aggregation. Hence, ThingVisors extract data from the upstream systems that are part of the Root Data Domain and publish new data to downstream VirIoT components (e.g. vSilos) by means of the internal information sharing. ThingVisors are able to understand a variety of data formats and convert them to the internal NGSI-LD neutral format.

- vSilos declare their interest for specific vThings and dynamically incorporate them, hence getting related data from the internal information sharing, and they have the goal of exposing such data through systems and Brokers of choice. vSilos are able

Collection	JSON Description	Example
flavourC	'flavourID':flavour_id, 'status':status, 'flavourParams': flavour_parameter, 'imageName':image, 'flavourDescription': flavour_description, 'creationTime':creation_time	'flavourID':Mobius-base-f, 'status':ready, 'flavourParams':Mobius, 'imageName':fed4iot/mobius-base-f:2.2, 'flavourDescription':silo with a oneM2M Mobius broker, 'creationTime':2019-11-19T12:04:33.342169

thingVisorC	'thingVisorID':thing_visor_id,
	'status':status,
	'creationTime':creation_time,
	'tvDescription':tv_description,
	'containerID':container_id,
	'imageName':image,
	'ipAddress':ip_address,
	'debug_mode':debug,
	'vThings':
	[
	{
	'label':vt_label,
	'id':vt_id,
	'description':vt_description
	}
	],
	'params':parameters,
	'MQTTDataBroker':{
	'ip':data_broker_ip,
	'port':data_broker_port
	},
	'MQTTControlBroker':{
	'ip':control_broker_ip,
	'port':control_broker_port
	},
	'port':port,
	'IP':ip_address
		'thingVisorID':weather,
		'status':running,
		'creationTime':2019-11-19T16:16:15.010205,
		'tvDescription':Weather ThingVisor,
		'containerID':3fa61f1517339d19eb4f27f7eef15,
		'imageName':fed4iot/v-weather-tv:2.2,
		'ipAddress':172.17.0.3,
		'debug_mode':false,
		'vThings':
		[
		{
		'label':thermometer in Rome,
		'id':weather/Rome_temp,
		'description':current temperature,
		Kelvin
		},
		{
		'label':thermometer in Tokyo,
		'id':weather/Tokyo_temp,
		'description':current temperature,
		Kelvin
		},
		],
		'params':
		{
		'cities':
		[
		Rome,
		Tokyo
		],
		'rate':60
		},
		'MQTTDataBroker':{
		'ip':172.17.0.,
		'port':1883
		},
		'MQTTControlBroker':{
		'ip':172.17.0.1,
		'port':1883
		},
		'port':{},
		'IP':160.80.82.44

to understand the NGSI-LD neutral format and convert it to the data format of their Broker of choice.

userC	'userID':user_id, 'password':user_password, 'role':user_role, 'lastLogin':user_last_login, 'token':user_token	'userID':admin, 'password':pbkdf2:sha256:150000 $mWOSdeEB$09fa6, 'role':admin, 'lastLogin':2019-11-19T12:00:49.826222, 'token':eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
vSiloC	'vSiloID':vsilo_id, 'status':status, 'creationTime':creation_time, 'tenantID':tenant_id, 'flavourParams':flavour_params, 'containerName':container_name, 'containerID':container_id, 'ipAddress':vsilo_address, 'port':vsilo_port, 'vSiloName':vsilo_name, 'flavourID':flavour_id	'vSiloID':tenant1_Silo1, 'status':running, 'creationTime':2019-11-18T11:42:37.164069, 'tenantID':tenant1, 'flavourParams':'', 'containerName':tenant1_Silo1, 'containerID':5947 cf90fec920a732f71f46cd17c690a9a7c943c1 ec263d24026958f7d48b88, 'ipAddress':172.17.0.5, 'port': { '1883/tcp':32775, '9001/tcp':32774 }, 'vSiloName':Silo1, 'flavourID':mqtt-f
vThingC	'tenantID':tenant_id, 'vThingID':v_thing_id, 'creationTime':creation_time, 'vSiloID':v_silo_id	'tenantID':tenant1, 'vThingID':weather/Tokyo_temp, 'creationTime':2019-11-18T12:37:57.036506, 'vSiloID':tenant1_Silo1

Table 3: Collections stored inside MongoDB

3.3.1 ThingVisors

Table 4 summarizes the ThingVisors we have developed so far. Please contrast this table with Table 5, which is the complementary one for vSilos. The following sections give details about operation and implementation of each one of them.

3.3.1.1 Generic oneM2M ThingVisor

This ThingVisor is able to connect to data sources that follow the oneM2M standard, and it obtains data pieces coming from a set of specified oneM2M Containers that are hosted on one or more remote oneM2M platforms.

The following box shows a typical set of parameters that can be used to configure the ThingVisor during the "Add ThingVisor" operation (please see Section 3.1.9 to understand how the "Add ThingVisor" operation is performed).

Table 4: Available ThingVisors

Data Type	Data Format	Interface with Remote System
Generic	oneM2M	pub/sub
Generic	NGSIv2	pub/sub
Parking-related	NGSIv2	pub/sub
Aggregated Parking Availability	NGSIv2	pub/sub
Weather-related	JSON	RESTful polling

Typical **oneM2M ThingVisor** Parameters

```
{'CSEurl':'https://fed4iot.eglobalmark.com','origin':'Superman', 'poaPort
    ':'8089','cntArns':['Abbas123456/humidity/value','Abbas123456/
    batteryLevel/value'],'poaIP':'52.166.X.X','vThingName':'EGM-
    Abbas123456-humidity','vThingDescription':'OneM2M humidity data from
    EGM Abbass sensor'}
```

The above configuration tells the ThingVisor to connect to a oneM2M system located at our EGM partner's infrastructure, and to subscribe to the Abbas123456 sensor's humidity readings. This humidity sensor is going to be virtualized inside our platform as a vThing whose ID is `EGM-Abbas123456-humidity`, and it comprises the actual value of humidity and the battery level of the sensor itself. The `poaIP` is the public IP address of our platform, so that the remote oneM2M system (a Mobius broker is this specific EGM deployment) knows how to notify the ThingVisor of new data.

Indeed, this ThingVisor is based on a oneM2M subscribe mode of operation, and at startup time it registers an own notification end-point into the remote oneM2M platforms, so as to be notified by the remote oneM2M systems whenever new data is produced for the specified Containers. During the initialization phase, the ThingVisor also requests the latest ContentInstances available inside all specified oneM2M Containers, so as to be ready to reply to possible getContextRequest commands coming from vSilos. Eventually, it announces the newly created vThing to the downstream components of the VirIoT system, specifically to the Master Controller (see Section 3.2.3 and Figure 9).

Upon arrival of a fresh piece of data, the ThingVisor carries out a translation from oneM2M to NGSI-LD and re-publishes the data internally, within VirIoT, making it available for downstream vSilos. The high-level operation is shown in Figure 13.

Thus, this ThingVisor is central to achieve seamless interoperability with oneM2M systems, in the forward direction (from oneM2M to other standards/platforms). Importantly, the mapping from oneM2M data to NGSI-LD data (i.e. to our neutral internal format) is **not automatic as the reverse one is** (i.e. the one from our neutral format to oneM2M, which is carried out along the mapping procedures we have specified in D2.2). The ThingVisor developer (or the ThingVisor logic) must know (or query) the underlying information model and resource structure of the oneM2M system, and decide what Containers are to be grouped together under the same vThing. The important thing to remark here is that this ThingVisor is capable of exploiting the (reverse of the)

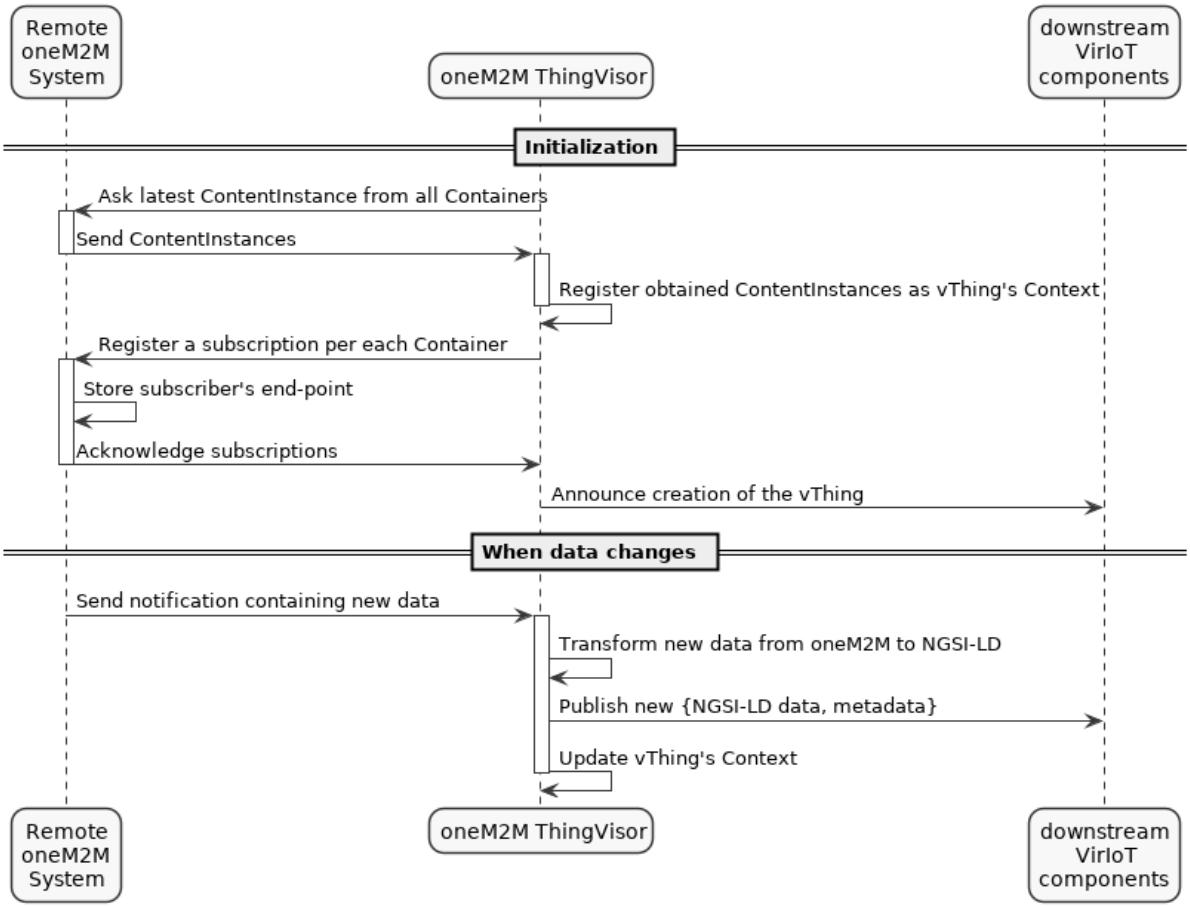


Figure 13: oneM2M ThingVisor

"one vThing" to "many Containers" mapping guideline that we have specified in D2.2. It is then possible to configure (as seen above) the ThingVisor with a list of oneM2M Containers that are to be grouped under the same vThing within VirIoT.

3.3.1.2 Generic NGSIv2 Greedy ThingVisor

This ThingVisor is able to connect to data sources that follow the NGSIv2 standard, and it obtains data entities coming from a remote Orion Context Broker (FIWARE platform).

The following box shows a typical set of parameters that can be used to configure the ThingVisor during the "Add ThingVisor" operation (please see Section 3.1.9 to understand how the "Add ThingVisor" operation is performed).

Typical **Greedy ThingVisor** Parameters

```
{'ocb_service':['trafico','aparcamiento','pluviometria','tranvia','
    autobuses','bicis','lecturas','gps','suministro'], 'ocb_ip':'fiware-
    dev.inf.um.es', 'ocb_port':'1026', 'notificacion_protocol':'http', '
    notify_ip':'X.X.X.X'}
```

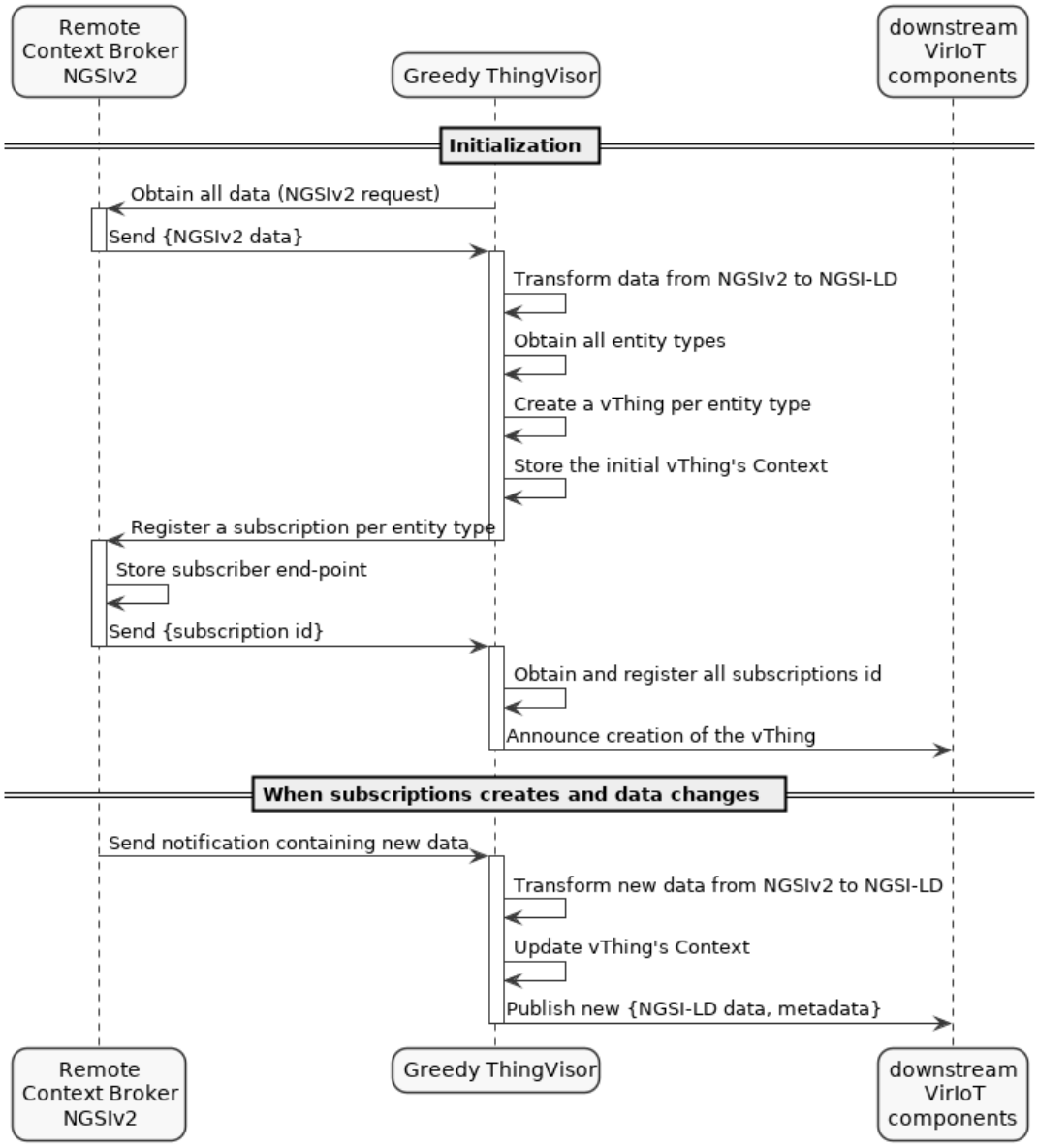


Figure 14: Generic NGSIv2 Greedy ThingVisor

The above configuration tells the ThingVisor to connect to a remote FIWARE-based platform exposing an NGSIv2 API, and more specifically to an Orion Context Broker GE (Generic Enablers in FIWARE terminology build an ecosystem of applications, services and data), to obtain all its data. This data is going to be virtualized inside our platform under different vThings, specifically one vThing per NGSIv2 entity type the remote FIWARE platform contains.

Further, this ThingVisor uses publication and subscription mechanisms to receive all the information coming from the selected platform. The `notify_ip` is the public IP address of our platform, so that the remote Context Broker knows how to notify the ThingVisor of new data. As said, in this case, this ThingVisor also sends a subscription request per entity type, to the FIWARE platform.

To close the initialization phase, ThingVisor sends `createVThing` messages about the

new vThings to the downstream components of the VirIoT system, specifically to the Master Controller (see Section 3.2.3 and Figure 9).

When ThingVisor receives the provider's Contex Broker notifications, it transforms the data from NGSIv2 to neutral NGSI-LD format and re-publishes the data internally, within VirIoT, making it available for downstream vSilos. The high-level operation is shown in Figure 14.

3.3.1.3 Smart Parking ThingVisor

This component is able to connect to data sources that follow the NGSIv2 standard, and it obtains data entities coming from a remote Orion Context Broker (FIWARE platform).

The following box shows a typical set of parameters that can be used to configure the ThingVisor during the "Add ThingVisor" operation (please see Section 3.1.9 to understand how the "Add ThingVisor" operation is performed).

Typical **Smart Parking ThingVisor** Parameters

```
{'ocb_ip':'fiware-dev.inf.um.es', 'ocb_port':'1026', '
    notificacion_protocol':'http', 'notify_ip':'X.X.X.X'}
```

Figure 15 shows how this component obtains data that is relevant to our Smart Parking use case from a particular Orion Context Broker that supports NGSIv2 API, and it exposes the corresponding payloads to other VirIoT components, by transforming them to our neutral NGSI-LD format.

The internal functionality of this ThingVisor is the same as mentioned above for the Greedy one, but this specific ThingVisor focuses only towards those data pieces that are relevant information for the Smart Parking use case.

3.3.1.4 Aggregated Parking Value ThingVisor

This component is able to connect to data sources that follow the NGSIv2 standard, and it obtains data entities coming from a remote Orion Context Broker (FIWARE platform).

The following box shows a typical set of parameters that can be used to configure the ThingVisor during the "Add ThingVisor" operation (please see Section 3.1.9 to understand how the "Add ThingVisor" operation is performed).

Typical **Aggregated Value ThingVisor** Parameters

```
{'ocb_ip':'fiware-dev.inf.um.es', 'ocb_port':'1026', '
    notificacion_protocol':'http', 'notify_ip':'X.X.X.X'}
```

Figure 16 shows how this component obtains data information relevant for the Smart Parking use case from a particular Orion Context Broker, processes and exposes an aggregated value in neutral NGSI-LD format. At this stage, the aggregated value is simply

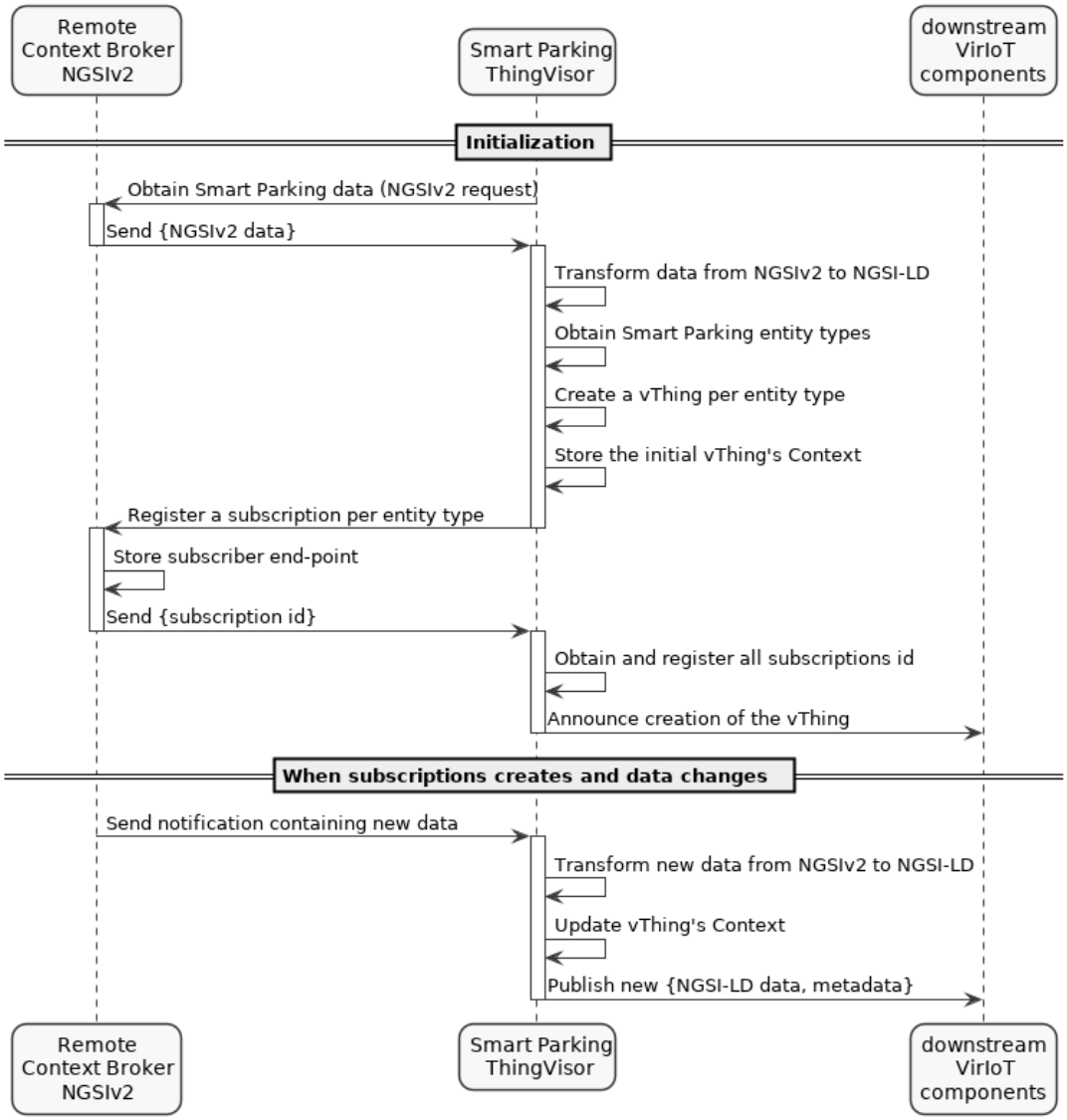


Figure 15: Smart Parking ThingVisor

the sum of free parking spaces. In order to accomplish the task, this ThingVisor receives data whenever there are changes, using publication and subscription mechanisms, and recalculates the aggregated value. Periodically, it verifies whether or not the aggregated value has changed since it was last published to other VirIoT components; only when this occurs, then ThingVisor re-publishes the data. The ThingVisor only needs one virtual thing to carry out its task.

3.3.1.5 OpenWeatherMap ThingVisor

This ThingVisor is able to provide virtual weather sensors (thermometer, barometer, etc.), for specified cities, by virtualizing information coming from the openweathermap.org service.

We have registered a Fed4IoT account on the openweathermap.org open API sys-

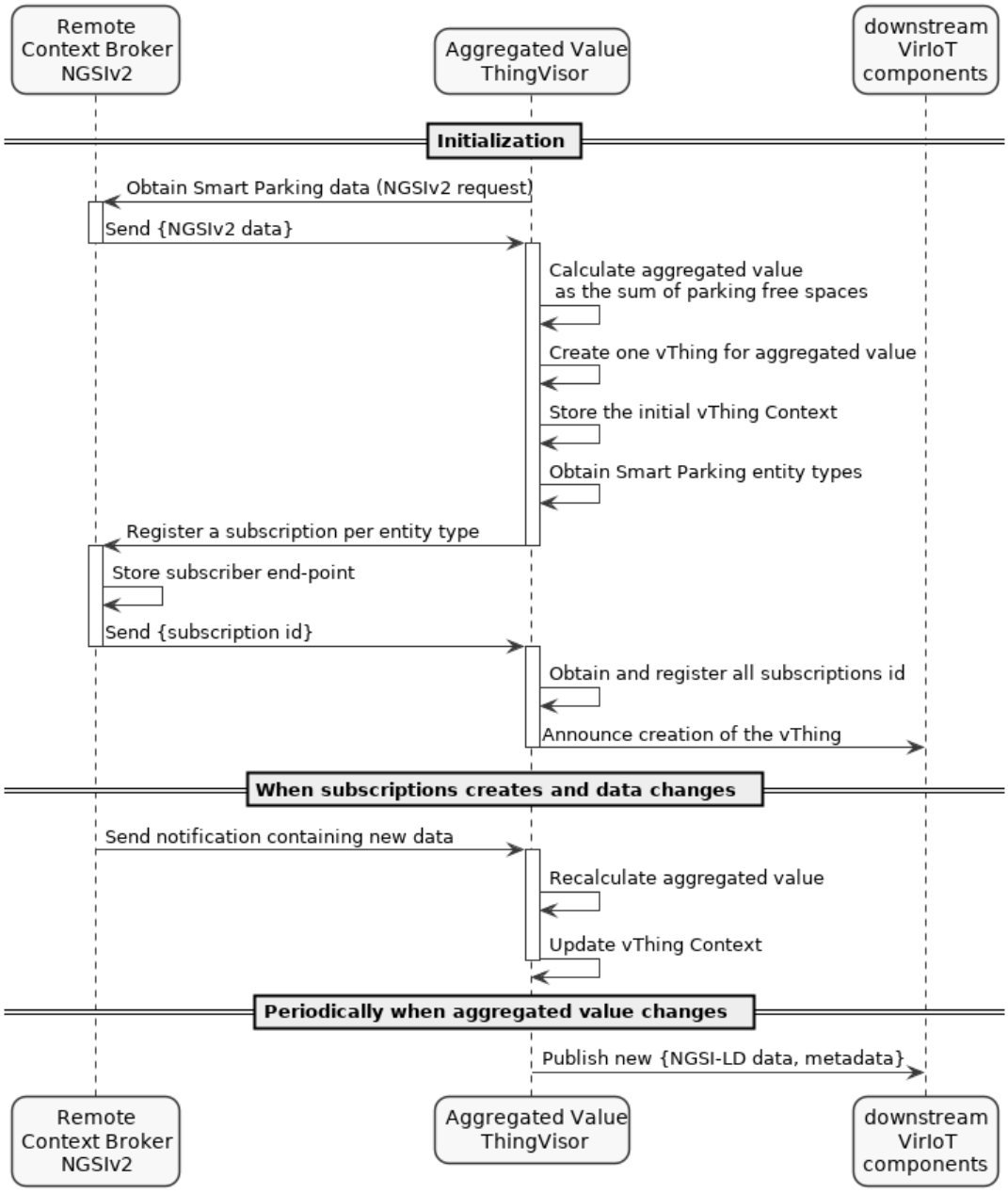


Figure 16: Aggregated Parking Value ThingVisor

tem, which allows our ThingVisors to periodically poll selected information from the openweather service, based on a custom configuration that is fed to the ThingVisor at startup time. Consequently, the OpenWeatherMap ThingVisor is able to create a set of vThings that act as virtual sensors for each measurable property (i.e. humidity, temperature, current atmospheric pressure) for each of the locations that have been specified at ThingVisor creation time.

The following box shows a typical set of parameters that can be used to configure the ThingVisor during the "Add ThingVisor" operation (please see Section 3.1.9 to understand how the "Add ThingVisor" operation is performed).

Typical **OpenWeatherMap ThingVisor** Parameters

```
{'cities':['Rome', 'Tokyo','Murcia','Grasse','Heidelberg'], 'rate':60}
```

The above configuration tells the ThingVisor to poll the openweather APIs every 60s, and it tells that the ThingVisor is to create a set of virtual sensors for all the specified cities and all the available measurable properties.

Figure 17 shows the flow of operations of the OpenWeatherMap ThingVisor.

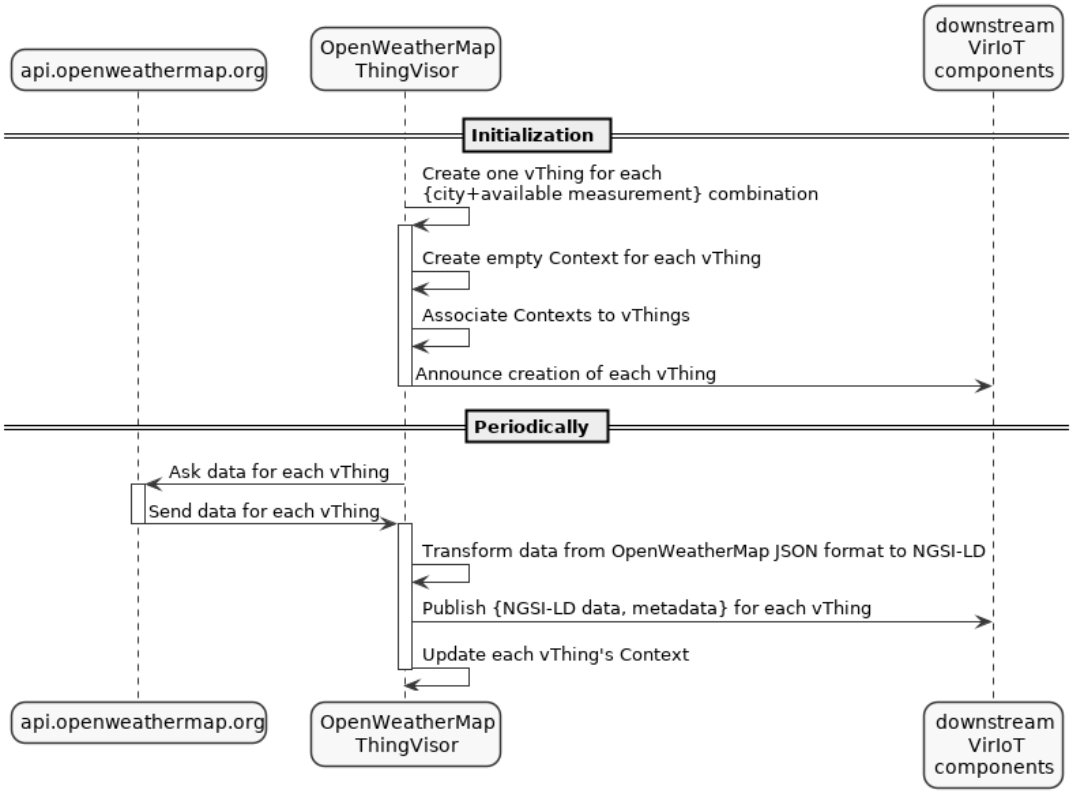


Figure 17: OpenWeatherMap ThingVisor

3.3.2 Virtual Silo Flavours

vSilos are the IoT backend systems that support the Applications being developed by every user/tenant of the VirIoT platform. Tenants can already choose among a variety of vSilos flavours (images of vSilos), but adding new flavours is possible.

Every vSilo is composed of two main building blocks:

- the vSilo Controller. It is a custom software piece that is able to interpret the incoming NGSI-LD data and metadata produced by the upstream VirIoT components (i.e. ThingVisors). Additionally, it is in charge of transforming such data (and possibliy metadata) into the destination native format of the vSilo Broker.

- the vSilo Data Broker. This is a standard IoT data broker (such as FIWARE's Orion, oneM2M's Mobius, NEC's Scorpio, a Mosquitto server, etc...) that is incorporated into the vSilo and serves data coming from the virtual things (that have been added to the vSilo) to the external IoT Application.

The vSilo Controller and the vSilo Data Broker collaborate in a simple manner, as exemplified in Figure 18.

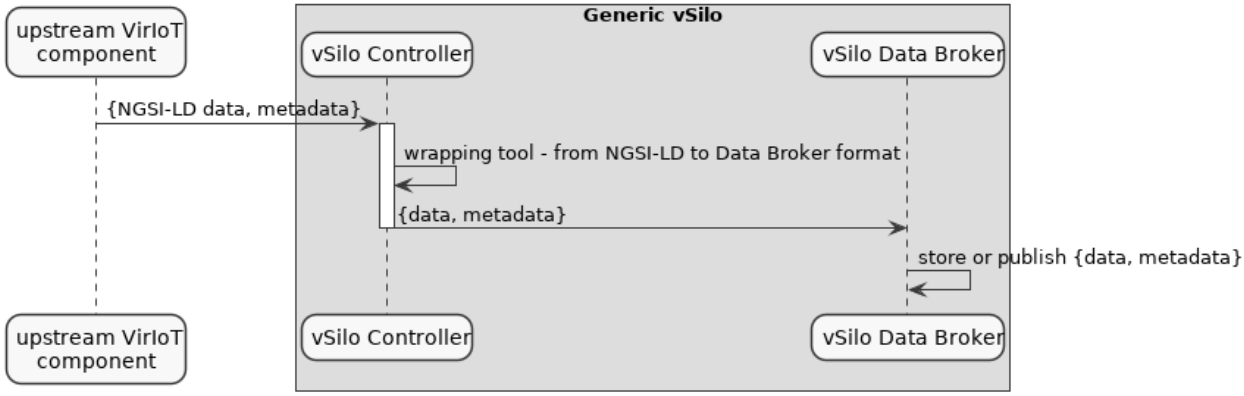


Figure 18: Generic vSilo operations

The internal communication channel between controller and broker is usually either RESTful or based on pub/sub. Table 5 summarizes the vSilos we have developed so far. The following sections give details about operation and implementation of each one of them.

Table 5: Available vSilo Flavours

IoT Broker	Data Format	Broker Developer	Broker Interface
Mobius	oneM2M	OCEAN open alliance	RESTful
Scorpio	NGSI-LD	NEC	RESTful
Orion	NGSIv2	FIWARE	RESTful
Mosquitto	JSON	Eclipse Foundation	pub/sub

3.3.2.1 Mobius oneM2M Flavour

This vSilo Flavour contains a vSilo Controller that is able to transform NGSI-LD data and metadata into oneM2M data and metadata. It performs a straightforward mapping between NGSI-LD and oneM2M along the guidelines for automatic translation that we have defined in D2.2. Further details about the interoperability of NGSI-LD with oneM2M, and about the mapping of metadata, are given in the D4.x deliverables. The relevant piece of information here is that this vSilo includes a Mobius server, which is the open source IoT server platform based on the oneM2M standard that has received certification by TTA (Telecommunications Technology Association). This oneM2M certification makes it one (of the two) servers designated as golden samples (please see `http://onem2mcert.com/sub/sub02_07.php`.

3.3.2.2 Scorpio NGSI-LD Flavour

Scorpio is the NGSI-LD-capable Context Broker under active development by NEC. The Scorpio vSilo is able to replicate upstream NGSI-LD data to the broker. Specifically, it has the ability to map, into NGSI-LD, both data and metadata by exploiting NGSI-LD Relationships between Entities and vThings. This approach is a consequence of our information model design, which is also meant to support discovery and indexing of vThings inside the VirIoT platform, in general. More details can be found in deliverables D4.x.

3.3.2.3 Orion NGSIv2 Flavour

This vSilo flavour contains the vSilo Controller that receives NGSI-LD representation of the vThings, previously sent by the ThingVisors, through the internal information sharing. The vSilo Controller processes the NGSI-LD payload and builds the corresponding payload in NGSIv2 format using a wrapping tool. After performing this transformation task, the vSilo Controller can forward and store the NGSIv2 to an Orion Context Broker using the NGSIv2 RESTful API. This broker is included in vSilo component too.

3.3.2.4 Mosquitto Raw JSON Flavour

This virtual silo flavour is designed so as to export the IoT data coming from its virtual 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. Specifically, this vSilo incorporates a Mosquitto server instance. Upon receiving data, the silo controller simply publishes any data payload to a Mosquitto topic that is named according to the tenant name and the vThing data is coming from: `tenant_id/v_thing_id`. Any Application can then subscribe to this tenant-specific topic, and can receive a raw copy of data and metadata.

4 ThingVisor Advanced Orchestration and Development Tools

4.1 FogFlow

This section introduces the high level system design of FogFlow and its latest programming model, which could be further used to support the implementation of more advanced ThingVisors. Later, it describes how FogFlow is integrated with the other Fed4IoT components to support the development and management of ThingVisors. More detailed information on how to use FogFlow is provided by the FogFlow online tutorial [1].

4.1.1 System Overview

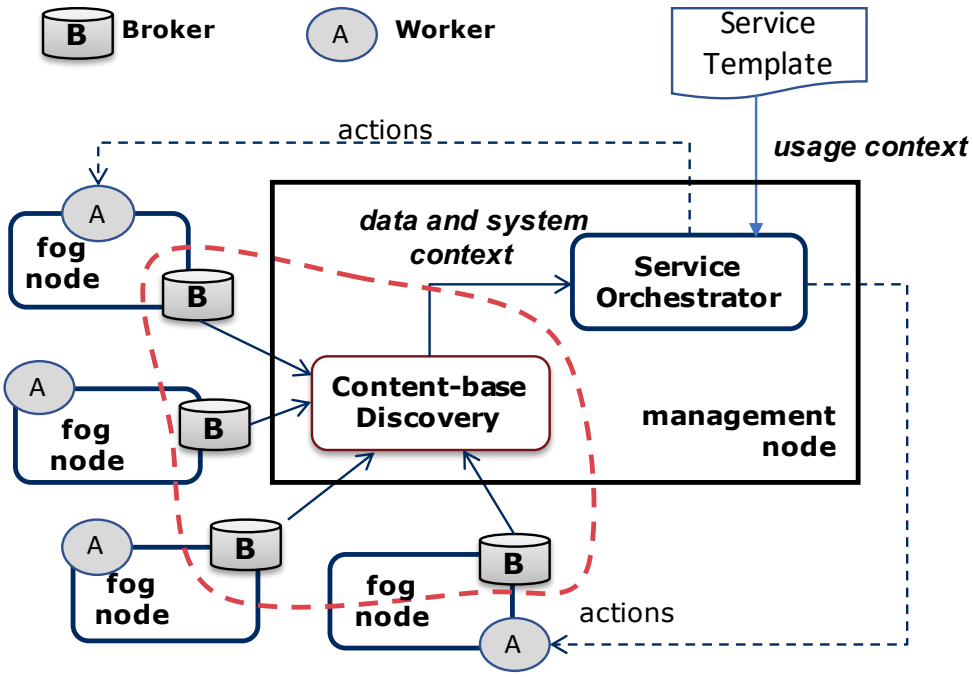


Figure 19: System Overview of FogFlow

FogFlow is an open source fog computing framework that can dynamically orchestrate IoT services over cloud and edges on-demand, in order to fulfill high-level "service intention" expressed by service consumers, which could be external applications or any IoT devices. Figure 19 shows a high level view of the FogFlow system. It consists of a number of *fog nodes*, each of which runs a *Broker* and a *Worker*. A management node runs two centralized components, namely *Discovery* and *Orchestrator*. Each node is a Virtual Machine (VM) or physical host deployed either in the cloud or at edges. All fog nodes form a hierarchical overlay based on their configured GeoHash IDs. All data in the system is represented as entities saved by a Broker and indexed by the centralized Discovery for discovery purposes. The data can be raw data published by IoT devices, intermediate results generated by some running data-processing tasks, or data available

at a resource, reported by fog nodes. When a fog function is registered, Orchestrator will subscribe to the input data of the fog function to Discovery. Once the subscribed data pieces appear or disappear in the system, Orchestrator will be informed, and it can then take orchestration actions accordingly, which will be carried out by an assigned worker.

4.1.2 Intent-based Programming Model

As illustrated by Figure 20, in FogFlow an IoT service is represented by a service topology and a set of user-defined intents.

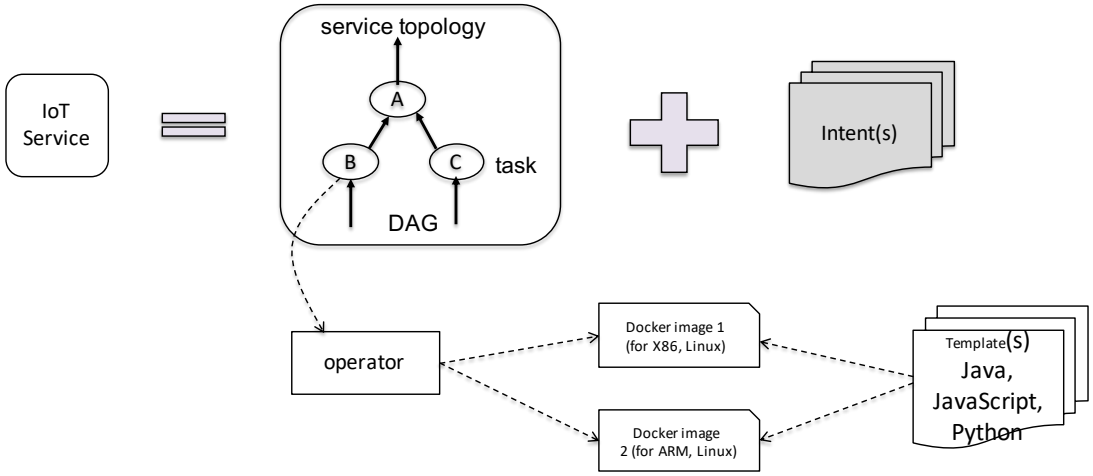


Figure 20: Service Model in FogFlow

The service topology is a graph of tasks, and each of them is supposed to perform some type of data processing. Tasks in the same topology are linked with each other, based on the dependency of their data inputs and outputs. Each task is annotated by service designers via a graphical editor, to define their input and output data and to also define a granularity feature that determines how input data should be divided into task instances, for parallelization of computation. Each task instance runs within a docker container. By design, a service topology only defines the data processing logic of an IoT service.

To trigger the service topology in FogFlow, service consumers need to define an intent to express their high-level goals of using such as IoT service. More specifically, as illustrated by Figure 21, an intent can be customized to cover the following goals:

1. *service topology* that defines which service logic to be triggered;
2. *geoscope* that defines the scope to select the input data for applying the selected service topology;
3. *service level objective (SLO)* that defines the service level objective to be achieved, in terms of latency requirements, bandwidth saving, or privacy/security needs;
4. *priority* that defines how the triggered service deployment could utilize the shared infrastructure resources with the other existing services.

With such an intent-based programming model, FogFlow is able to dynamically orchestrate concrete service deployment plans to meet any user-defined intents in a more flexible way, even for the same service topology.

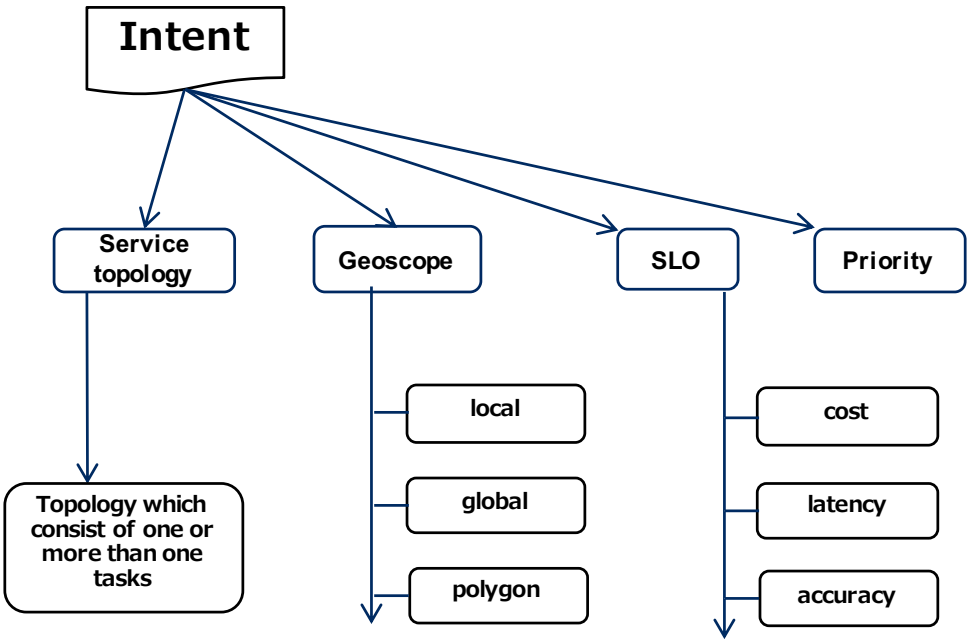


Figure 21: Intent Model

As shown in Figure 22, programming a FogFlow service includes three key elements: *operator*, *service topology*, and *intent*. An operator represents a type of data processing unit, for example, calculating the average temperature, performance the face matching of two face images. A service topology represents the computation logic of the IoT service and consists of several linked operators annotated with their data inputs and outputs. An intent is a JSON object that follows the proposed intent model to define a customized requirement of how the service topology should be triggered.

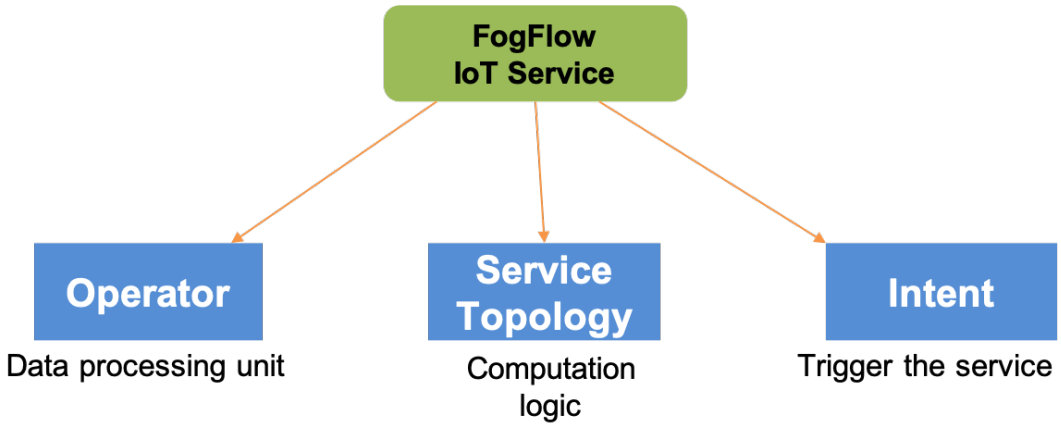


Figure 22: Three key elements to program an IoT service in FogFlow

Currently, FogFlow can support serverless fog computing by providing so-called Fog Function, which is a common easy case of the intent-based programming model. As

illustrated in Figure 23, Fog Function represents a common special case of the generic intent-based programming model in FogFlow, meaning that a fog function is associated with a simple service topology that includes only one task (a task is mapped to an operator in FogFlow) and an default intent that takes "global" as its geoscope. Therefore, when a fog function is submitted, its service topology will be triggered immediately once its required input data is available.

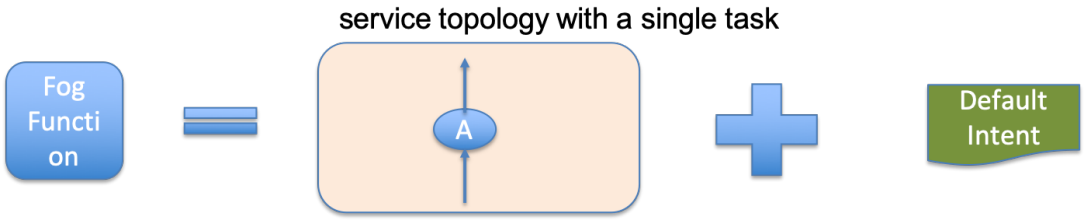


Figure 23: FogFunction as a simple case of service topology in FogFlow

4.1.3 Context Aware Service Orchestration

Another unique feature of FogFlow is context aware service orchestration, meaning that FogFlow is able to orchestrate dynamic data processing flows over cloud and edges based on the following three types of contexts, including:

Data context: the structure and registered metadata of available data, including both raw sensor data and intermediate data. Based on the standardized and unified data model and communication interface, namely NGSI, our system is able to see the content of all data generated by sensors and data processing tasks in the system, such as data type, attributes, registered metadata, relations, and geo-locations.

System context: available resources at each fog node. The resources in a cloud-edge environment are geo-distributed and they are dynamically changing over time. As compared to cloud computing, resources in such a cloud-edge environment are more heterogeneous and dynamic.

Usage context: high level usage intentions defined by service designers to indicate what their fog functions should be used in the system, such as which type of results is expected under which type of QoS within which geo-scope.

Figure 24 shows the major procedure for Orchestrator to orchestrate fog functions based on the update notification of context availability of their input data, provided by Content-based Discovery. More specifically, the following four basic orchestration actions are designed to dynamically orchestrate tasks for each registered fog function.

- *ADD_TASK*: To launch a new task with the given configuration that includes the initial setting of its input streams. When launching a new task, the Worker first fetches the Docker image for this task and then launches and configures this task within a dedicated Docker container. After that, the Worker subscribes the input entity to the context management system on behalf of the running task so that the input streams can be received by the running task; in the end, the newly created task is reported back to the orchestrator.

- *REMOVE_TASK*: To terminate an existing running task with the given task ID. When terminating an existing task, the Worker not only stops and removes its corresponding Docker container, but also unsubscribes its input streams so that the context management system does not end up with lots of unavailable subscribers.
- *ADD_INPUT*: To subscribe to a new input stream on behalf of a running task so that the new input stream can flow into the running task.
- *REMOVE_INPUT*: To unsubscribe from some existing input stream on behalf of a running task so that the task stops receiving entity updates from this input stream.

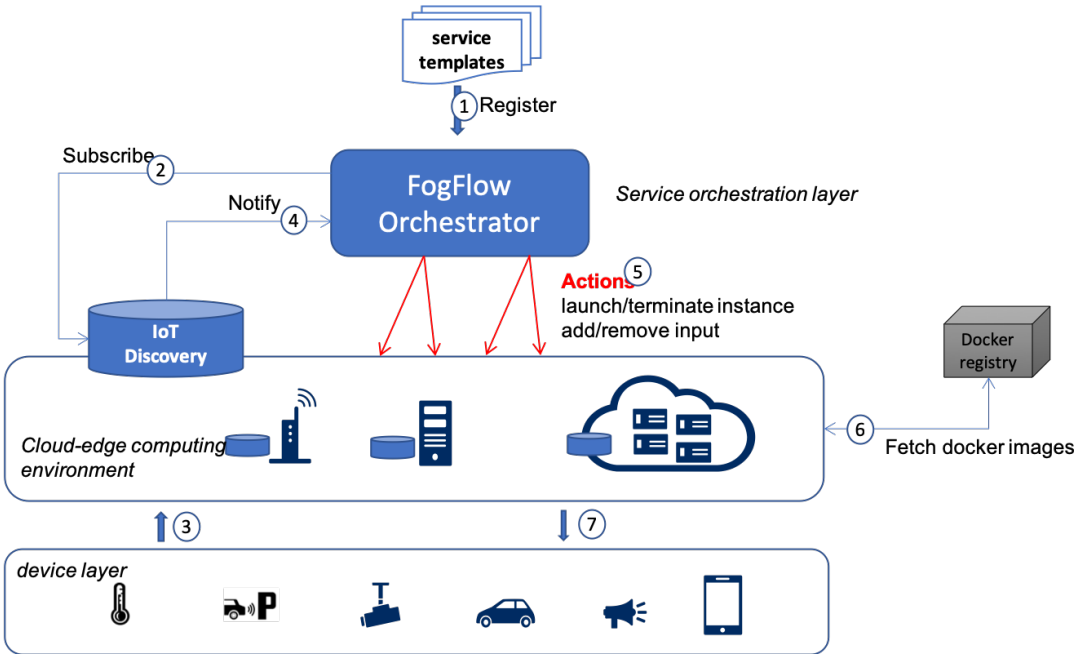


Figure 24: Data-driven orchestration

4.1.4 FogFlow-based ThingVisor

FogFlow provides an advanced programming model to implement various ThingVisors that can take advantage of edge computing. For example, in the person finder application that we have as one of our use cases, when creating a Virtual Thing for a camera, we need to trigger a face-matching task to check if the camera is capturing the person (a lost child, for instance). In order to reduce the bandwidth consumption, it is better to offload the face-matching task to an edge node that is close to the camera. Using FogFlow, we can outsource some data-intensive data processing tasks down to the edge and then produce the required Virtual Things or their attributes with low bandwidth consumption and delay.

Also, with the intent-based edge programming model in FogFlow, we can implement different types of ThingVisors in a more flexible way. For example, we can implement a very specific ThingVisor that is used to create only one specific Virtual Thing; however, we can also implement some generic ThingVisors that can be used to create a number of

Virtual Things with the same kind on-the-fly for a given geo-scope. In addition, with the fog function programming model, we can implement some ThingVisors that can create Virtual Things automatically. In this case we can create virtual things without explicitly calling the `/addVThing` API.

Figure 25 shows how FogFlow works with the other components in Fed4IoT. FogFlow can be used to implement different types of FogFlow-based ThingVisors, each of which is implemented as a FogFlow service based on the intent-based programming model. Assume that those FogFlow-based ThingVisors are developed and registered in FogFlow via the GUI of FogFlow Task Designer. Then those ThingVisors can be managed by the APIs (`/addThingVisor` and `/deleteThingVisor`) of Fed4IoT Master Controller via a generic FogFlow-ThingVisor, which is a dockerized application to communicate with the running FogFlow system for managing specific FogFlow-based ThingVisors. Since each FogFlow-based ThingVisor is implemented as a FogFlow service, adding or deleting ThingVisor is equal to enabling or disabling a FogFlow service via a customized intent.

For the management of Virtual Things, there are two cases:

- *pull-based data source*: the device data to create the Virtual Thing is not available in FogFlow, the `/addVThing` interface must be explicitly called to provide the device profile, which will trigger FogFlow to launch some data processing tasks for creating the Virtual Thing. For the use case of lost child finder, to create the Virtual Things of cameras, we need to call `/addVThing` to provide the camera profile, such as the accessible URL of the camera stream.

- *push-based data source*: the required data source to create the Virtual Thing is already available in FogFlow. For example, some temperature sensors joins FogFlow and reports its profile to the internal context management system in FogFlow; or the required data sources are produced by some other Virtual Things already and we need to aggregate those data sources in FogFlow to further create a new aggregated Virtual Thing. For example, we can create a Virtual Thing for a city by taking the temperature data from all virtualized temperature sensors from that city. In this type of case, once we enable the defined FogFlow service, the associated Virtual Things can be created automatically on the fly. This is no need to further call the `/addVThing` interface. Therefore, FogFlow can help to reduce the effort of managing Virtual Things.

Once a Virtual Thing is created in FogFlow, its information will be registered and forwarded to the internal information sharing system of VirIoT and it will be shared to the other applications via a Virtual Silo. To support this feature, we are currently extending the intent model in FogFlow to cover a new aspect, called destination, which can tell FogFlow where to publish the generated Virtual Things.

4.2 Service Function Chaining

Service function chaining (SFC) is a technology to compose network functions by chaining component sub-functions. With the advancement of computer hardware technology, even functions to implement networking services, such as packet forwarding, firewall, and load balancing, can be implemented by means of high-level software programming. Such

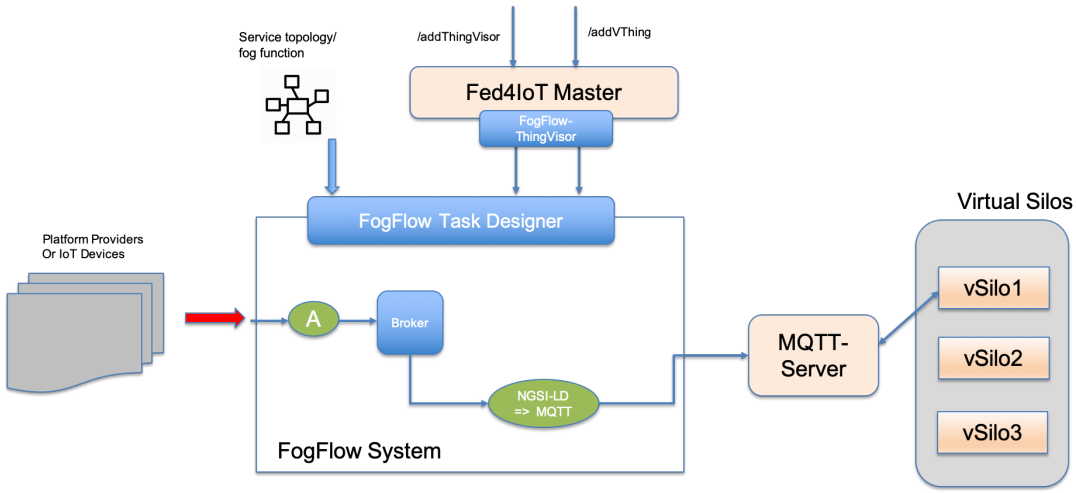


Figure 25: FogFlow in VirIoT

technology is called software-defined networking (SDN) or network-function virtualizaiton (NFV) depending on the focus of the topic. Several networking services are performed in sequence in data centers. For example, a packet requesting a web page is first processed by the firewall to check the sanity of the packet, and then passed to L7 load-balancer to forward the packet to an appropriate server.

The same technology can be applied to realize vThings by means of ThingVisors. A video frame captured by a surveillance camera may be processed by a ThingVisor to detect human or animal, extract a human face or an animal figure, and then the output picture is further processed by another ThingVisor to identify the age of the person or to classify the animal. The information may be further forwarded to yet another ThingVisor to generate statistical data.

As stated in the paragraph above, SFC is used to construct ThingVisors (Figure 26). Video images provided by the camera directly connected to VirIoT environment are captured by the image capture function. The captured images as well as the images provided by one of the Root Data Domain are processed by the human detection function to find humans in the captured images. Then the detected human images are further processed to extract faces. The captured images by the directly connected camera are provided to Virtual Silos as vThing data. The extracted face images are also provided as a vThing to Virtual Silos. Each output of the sequence of functions executed by the service function chaining can be provided as a vThing, too. Accordingly, the chain of functions is a realization of a ThingVisor.

There are a few technical challenges in implementing service function chaining. They are:

- the mechanism to execute service functions in VirIoT,
- the location to execute functions and the choice of available functions to be used in a particular chain, and
- the communication mechanism to execute functions.

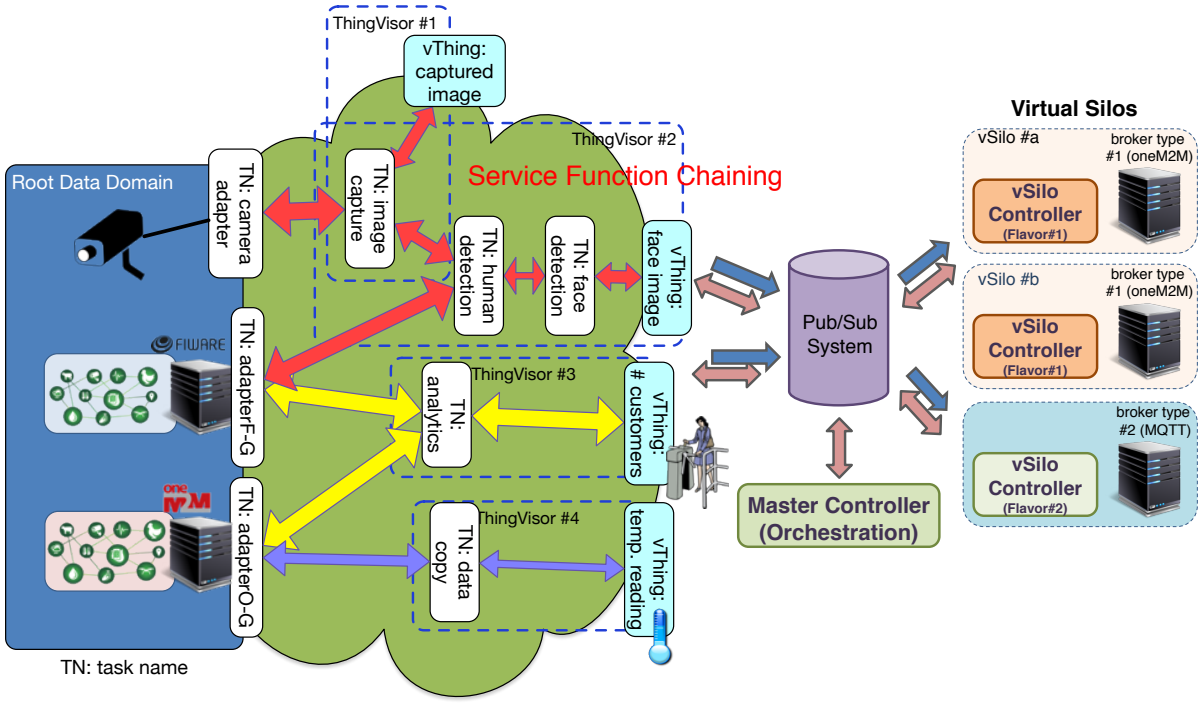


Figure 26: Service Function Chaining in VirIoT

4.2.1 Implementation of Service Functions

To execute each Service Function (SF) in any network nodes without considering hardware dependence, SF should be implemented on computer virtualization environments. In the first step, we select Docker as the virtualization platform. Because SF implement only a partial functionality of an IoT service, SF needs to clearly define input and output data types. In addition, to participate in SFC, each SF has a unique name. In case of IoT services using video cameras (e.g., video surveillance service), typical names of SFs are "capture image/video", "encoding", "object detection", and "streaming."

4.2.2 Deployment and Selection

An instance of each SF in a chain is required to be placed on a node such as a route or an edge computing node, or a VM before processing the chain for SFC. If no node have any instance of an SF, it must be downloaded from an *SF Repository* to the node. In this context, the downloading time of a large size SF can be a bottleneck of the whole processing time for the chain. In another context, if multiple chains must be processed simultaneously, several SFs with the same type may be used by different SFCs. If such SFs are processed on different nodes, all SFs must be downloaded from the repository; that is, multiple SF downloading can degrade the SFC throughput. Then, one of the objectives of SFC in VirIoT is to achieve effective processing for each chain, i.e., minimizing the response time with the small number of SF instances and computational resources. The points to be satisfied in VirIoT are described as follows:

- **SF deployment:**
  SF instances must be allocated to nodes that contribute to optimize the response

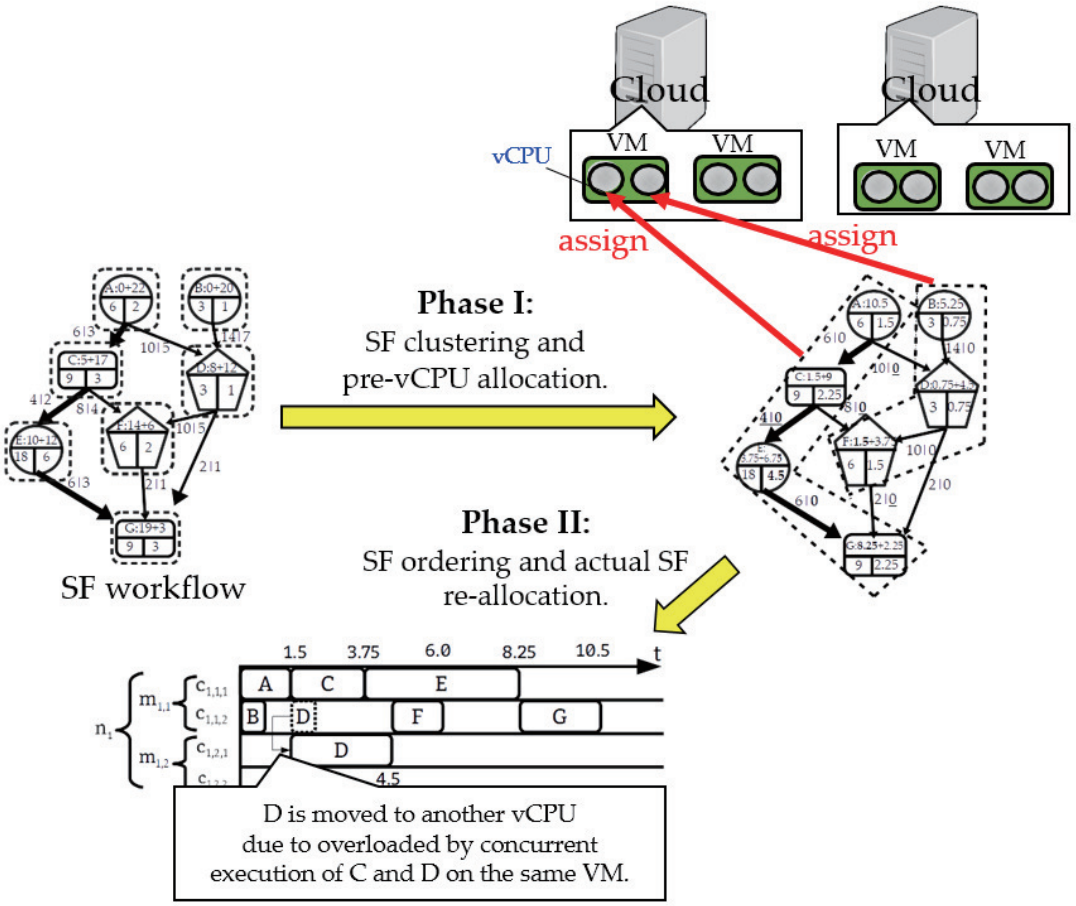


Figure 27: Procedures of SF-CUV algorithm.

time. This optimization becomes more crucial when no SF is deployed, i.e., the initial state in VirIoT. When SF instances are been deployed on a node that has already downloaded the SF, the node can initiate SF instances without downloading the SF. SF instances should be deployed on nodes where chains can be effectively processed.

- **SF selection:**
  Once an SF instance is deployed on a node, it should be shared among many chains using the same SFs in order to avoid redundant SF download and execution. Thus, a criterion for sharing the SF instances among chains is needed.

We designed the algorithm to perform SF deployment and SF selection simultaneously. The main idea behind the algorithm is to cluster SFs as an allocation unit to a node. In particular, the SF clustering algorithm for scheduling SFs, namely SF-clustering for utilizing vCPUs (SF-CUV), is directed to minimize the response time with a small number of vCPUs (virtual CPUs) and SF instances. SF-CUV consists of two phases, i.e.,

1. SF clustering and pre-vCPU allocation phase (phase I).
2. SF ordering and actual SF re-allocation phase (phase II).

In phase I, an accurate scheduling priority is derived. In phase II, the accurate scheduling priority is determined and the SFs are moved to another vCPU for sharing to avoid redundant SF downloading procedures. Thus, each SF can be scheduled to effectively minimize the response time with a small number of vCPUs and SFs. In particular, the second phase minimizes the number of allocated SFs by sharing them in the same vCPU, and it also minimizes the number of allocated vCPUs. Thus, many SFCs can be effectively and simultaneously executed in the system.

Figure 27 shows the whole procedures of SF-CUV. In this figure, the assumed function chain is a workflow-type chain, that is a general form of SFC. Phase I outputs two SF clusters by SF clustering steps. This phase also outputs the mapping between each SF cluster and each vCPU as an allocation target. This allocation can derive an accurate scheduling priority for phase II. Then the actual SF ordering is performed at phase II. In this phase, D is moved to another VM because the concurrent execution of C and D on a VM exceeds the predefined CPU load. From those procedures, in total three vCPUs on two VMs are used for processing the SF workflow.

4.2.3 Communications Mechanisms for Service Function Chaining

SF instances placed in network nodes, edge computing facilities, and clouds exchange messages to realize SFC. The initiator of the message exchange in SFC may not be SF instances but can be external sources such as user triggers or IoT devices generating their output. The communication is not end-to-end but hop-by-hop in this respect. In this project, we are investigating three alternatives as the communication mechanism: 1) IP-based point-to-point mechanism, 2) Topic-based publish/subscribe over IP, and 3) ICN.

4.2.3.1 IP-based Point-to-Point

IP-based point-to-point mechanism is the simplest approach and uses IP addresses for routing of SFC. To chain the SFs, each SF needs to know the IP address of the next SF. In the preliminary SF workflow, we assume that a controller of the workflow knows IP addresses of every computing node in the system and assigns the appropriate IP address and port of the next SF instance after the optimal SF deployment plan is determined.

4.2.3.2 Topic-based Publish/subscribe over IP

Topic-based Publish/subscribe (pub/sub) is another approach to realize SFC. Unlike the IP-based point-to-point, Pub/sub solves the routing problem is SFC by using topic name. As explained in 4.2.3.1, IP-based point-to-point approach assumes one controller knows IP addresses of all computing nodes, and this assumption potentially contains a scalablity issue. In contrast, in the pub/sub-based approach, the routing of SFC can be easily managed by only defining topics for pub/sub messages. In a preliminary implementation, the topic is specified by device ids and names of SFs such as "[pub/-sub://]DeviceID/FunctionA/FunctionB/Function C."

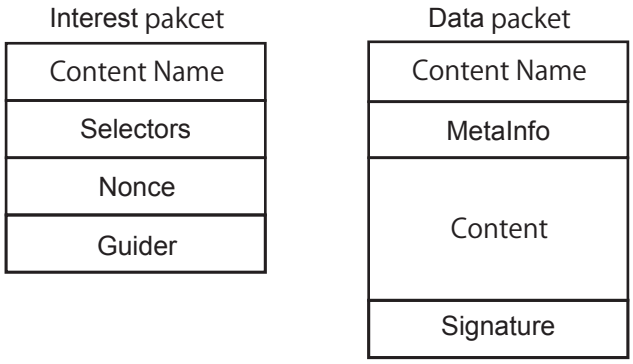


Figure 28: Original NDN Packet Format

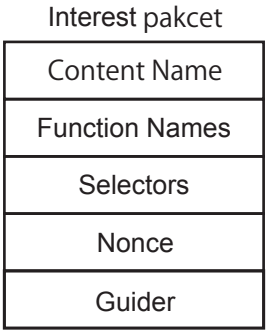


Figure 29: Extended Interest Packet Format

4.2.3.3 Information Centric Networking

Name or identifier assigned to a content is the address to be used to forward packets in information-centric networking (ICN). By making use of this feature of ICN, the functions with the same capability can be identified by the same name regardless of their implementation. IoT devices with the same funcionality can also be identified by the same name. In this respect, ICN is a promising communication mechanism among IoT devices and service functions.

We adopted one typical incarnation of ICN: Named-Data Networking (NDN) or Content-Centric Networking (CCN), which uses request-response communication model. A content is requested by *Interest* packet and the content requested by the Interest packet is delivered by *Data* packet. The content to be requested is identified using a hierarchical name such as "rs1/lamp/2256" and the name is specified in both the Interest packet and Data packet. The formats of Interest packet and Data packet in NDN are shown in Figure 28.

We extended the Interest packet format to include the names of functions as shown in Figure 29. The functions specified in the Function Names field are applied to the content requested in the Content Name field. A sequence of function names can be specified in the function names field. Suppose three functions F1, F2, and F3 are to be applied to the content specified in the content name field. Then, the three function names are specified in the function names field in the reverse order of the application, i.e., F3:F2:F1. The Interest packet forwarded through the three functions F3, F2, and F1 in that order, and

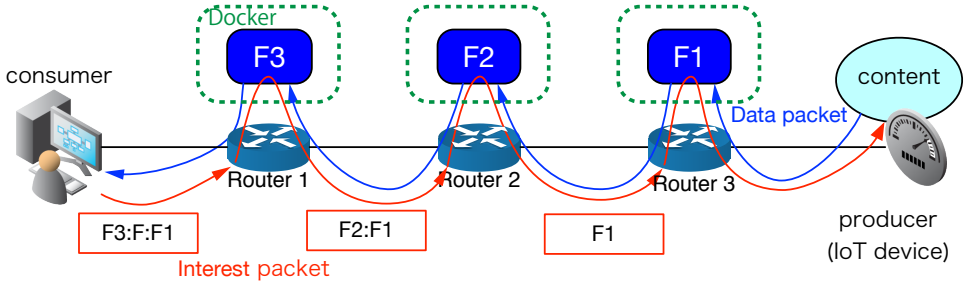


Figure 30: Example of Function Chaining

finally to the content specified by the content name field (Figure 30). The content is first delivered to F1 in the Data packet. F1 applies its processing on the content and pass to the next function F2. F2 applies its function on the received content in the Data packet, and so on. The consumer who originally dispatched the Interest packet receives the content after application of all the three functions.

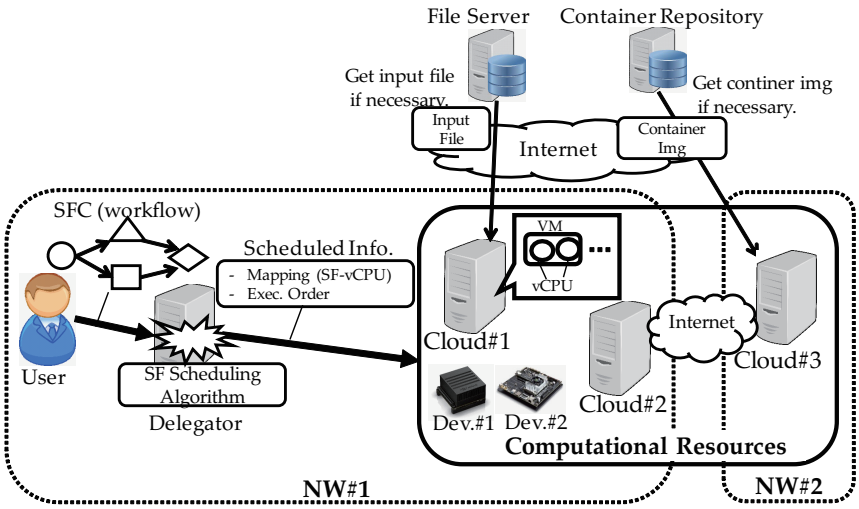
We implemented service functions as Docker containers. When a service function is installed with an NDN router, its routing table, *Forwarding Information Base* (FIB), is configured so that the Interest packet with the name of the installed service function is forwarded to the Docker container. By implementing service functions by Docker containers, service functions can be dynamically added to and removed from NDN routers regardless of the execution environment of NDN routers. Also, we confirmed activation time of service functions in Docker containers is about 1/20 of virtual machine implementation of service functions.

When the router receives an Interest packet with the name of a service function which is offered at the router, it removes the corresponding service function name from the function names field of the packet, and then forward the Interest packet to the Docker container offering the service function according to the information found in its FIB. The service function further forwards the Interest packet back to the NDN router. The NDN router then forwards the Interest packet towards the next service function by simply forwarding the packet referring to what is specified in the function names field because the next function is what is specified as the first function in the Function Names field now.

After consuming all the function names, the function names field becomes empty. Then NDN routers forward the Interest packet using its content name field. The Interest packet eventually received by the source of the content and the source creates a Data packet. The Data packet is sent back towards the consumer on the path of created by the Interest packet. Since the path includes service functions, the functions process the Data packet and further send the result back towards the consumer.

4.2.4 Performance Evaluation

We implemented a workflow engine to incorporate SF scheduling algorithms including SF-CUV for the performance verification in a real environment. We assume that each SF is performed in a heterogeneous distributed environment where nodes have different configurations, such as VMs in cloud, edge computing facilities, and edge devices. In this context, we conducted the performance comparisons in terms of the response time and



Hardware	Parameter	Value
NW#1 Router#1	Bottleneck BW to external hosts	100Mbps.
	Internal BW	1Gbps
Cloud#1	# of VMs	15VMs
	# of vCPUs	2vCPUs x 4VMs 4vCPUs x 11VMs
	vCPU Freq.	{2.0, 2.5, 3.0} GHz.
	BW to Router #1	1Gbps
Cloud#2	# of VMs	6VMs
	# of vCPUs	2vCPUs x 4VMs 4vCPUs x 2VMs
	vCPU Freq.	{1.0,2.0, 2.5} GHz.
	BW to Router #1	1Gbps
Dev.#1	Type	NVIDIA Jetson AGX Xavier (2.26 GHz x 8Cores, 16GB RAM)
Dev.#2	Type	NVIDIA Jetson TX2 (2.0 GHz x 6Cores, 8GB RAM)

Hardware	Parameter	Value
NW#2 Router#2	Bottleneck BW to external hosts	100Mbps.
	Internal BW	1Gbps
Cloud#3	# of VMs	8VMs
	# of vCPUs	4vCPUs x 4VMs 4vCPUs x 2VMs
	vCPU Freq.	{1.0, 3.0, 3.5} GHz.
	BW to Router #2	1Gbps

Figure 31: Experiment Environment

resource utilization to verify the practicality of SF-CUV. We compared SF-CUV with five other algorithms. They are CUV-FIX, HEFT, PEFT, CAP-based, and COM-based.

CUV-FIX is a variant of SF-CUV, where each allocated vCPU in the clustering phase is fixed in the SF ordering phase. Heterogeneous earliest finish time (HEFT)[2] is a well-known task scheduling algorithm that is widely employed in real systems. Predict-EFT (PEFT)[3] is a variation of HEFT that outputs a better schedule length than HEFT. Thus, we adopted PEFT as a state-of-the art task scheduling algorithm. In capacity-based approach (CAP-based), the order for SF selection is based on the increasing order of the finish time with the average processing speed and the average communication bandwidth. Then, the selected SF is allocated to the idle time slot of the vCPU with the largest residual capacity. This approach is adopted by [4, 5]. Communication locality-based approach (COM-based) tries to minimize the communication time among SFs, i.e., the output data from one SF is sent to the nearest node having sufficient capacity to accommodate the target SF. Such data locality-based SF allocation has been reported in the literature[6, 7].

Figure 31 shows the environment in which we conducted the performance comparison.

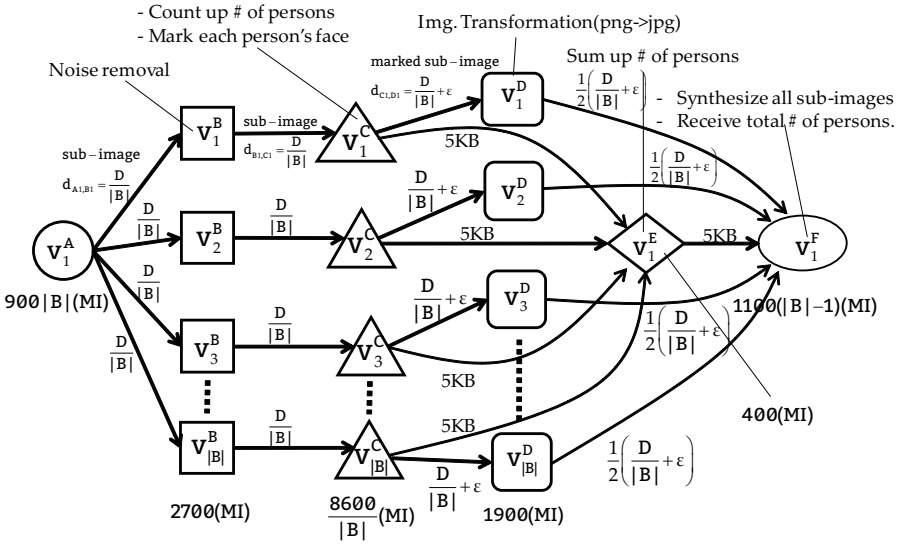


Figure 32: Applied workflow structure.

In this environment, we set up a heterogeneous computing environment in two different networks, i.e., NW♯1 and NW♯2. In these networks, we deployed computer hosts on which VMs run through Apache CloudStack. Each VM works on Ubuntu Desktop 18.04.3 LTS through KVM hypervisor. Although the mapping between each CPU core and each vCPU is typically controlled by a hypervisor, in this experiment, we manually mapped each CPU core and each vCPU by CPU pinning in advance. The reason is that we assume that each SF is allocated to each vCPU by an SF scheduling algorithm. In this environment, we assume that one SF corresponds to one Docker container that is stored in the "Container Repository" in Figure 31. Thus, if a node has no SF for execution, it downloads the SF from the Container Repository by SCP and then starts execution. If an execution of an SF requires one input file and the node to execute the SF has no input file, it downloads the input file from the "File Server" by SCP. Thus, we assume that the file transfer involving the input file and Docker image may occur.

Figure 32 shows the applied workflow of the SFC. The workflow has six types of SF, i.e., A, B, C, D, E, and F, and every SF is executed on a Docker in which OpenCV is pre-installed. Then, we generated the OpenCV-enabled Docker image and compressed it as 1.22 GB tar file that we name "BaseSF-tar", and it was stored in the "Container Repository" in Figure 31. As the workflow has six types of SF, we generated six different Docker images based on "BaseSF-tar", i.e., every SF Docker image has the same file size (1.22 GB).

Figure 33 shows the comparison results of the degree of SF sharing in the real environment. In this figure, the horizontal axis represents the number of partitions $|B|$, i.e., the number of instances of SF B. At $|B| = 2$, the degree of SF sharing is nearly the same among the algorithms. The reason is that the number of SFs ($|V| = 3(|B| + 1) = 9$ is not sufficient to provide the difference in the degree of SF sharing, i.e., no SF has not been moved in the second phase, "SF ordering and actual vCPU allocation" phase, in SF-CUV.

If $|B|$ is larger, the difference increases and SF-CUV outperforms the others in terms

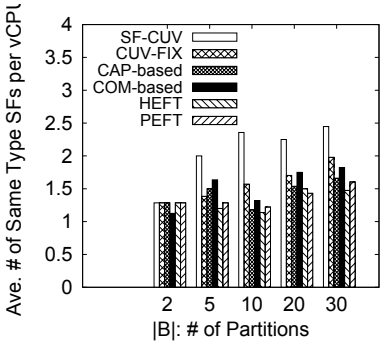
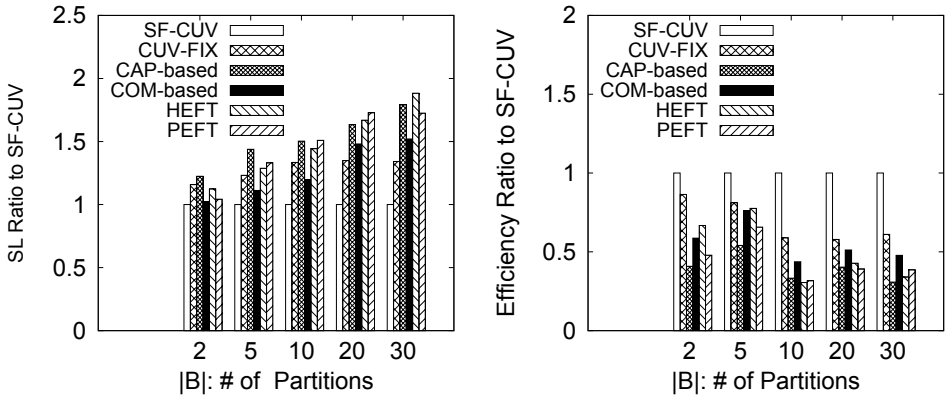


Figure 33: Degree of SF sharing.



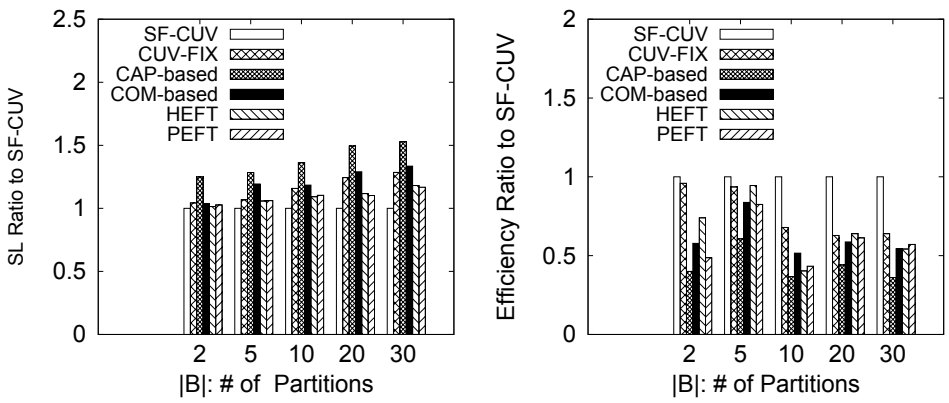
(a) response time Ratio (b) Efficiency Ratio

Figure 34: Comparisons of no SF pre-deployment.

of the degree of SF sharing. In particular, the difference between SF-CUV and CUV-FIX increases, i.e., more SFs have been moved to be shared in the second phase compared to the case of $|B| = 2$. From this result, SF-CUV can effectively share SFs when $|B|$ is larger than 5.

Figure 34 shows the comparison results in the case of no SF pre-deployment, where Figure 34 (a) shows the comparison results for the normalized response time by setting SF-CUV response time to 1.0, and Figure 34(b) shows the results for the normalized efficiency by setting SF-CUV efficiency to 1.0. In (a) and (b), in all cases of $|B|$, SF-CUV outperforms the others in terms of the response time and efficiency. In particular, in Figure 33, the degree of SF sharing in HEFT and PEFT is worse than that in the COM-based approach, and in Figure 34(a), their response times are also worse.

Figure 35 shows the comparison results in terms of the response time and efficiency when SFs are deployed in advance. In this case, every type of SF is deployed before scheduling SFs, i.e., no SF downloading is required. Thus, this case corresponds to the ideal situation that we can obtain the information about where and which SF should be executed in advance. SF-CUV outperforms the others in Figure 35(a) and (b). From those obtained results, we conclude that SF-CUV satisfies the requirement of response time minimization with a small number of computational resources with and without SF pre-deployment.



(a) response time Ratio (b) Efficiency Ratio

Figure 35: Comparisons of SF pre-deployment.

5 Flexible Compute Virtualization Architecture

In order to enable IoT virtualization and its applications and orchestration, it is essential that the Fed4IoT IoT Virtualization Platform (VirIoT) can use heterogeneous compute virtualization infrastructures at scale, beyond boundaries of service domains, organizations, policies and stakeholders. We named this component as *Flexible Compute Virtualization* architecture (see Figure 2), whose services should be able to i) handle heterogeneous virtualization technologies at the server level, ii) provide uniform means for the building of different virtualization images (VM, containers, etc.), and iii) manage the orchestration of a distributed computing resources platform formed of edge and central data centers. Towards these ends, at the Flexible Compute Virtualization Architecture, we need to address the following four problems:

- **Resource efficiency.**
  Local, individually owned IoT infrastructures, such as those in public areas of the smart cities, offices and ordinary homes, normally have relatively small amounts of resources in comparison to centralized data centres. Typically they deploy IoT gateway devices based on embedded boards, which manage sensors, cameras and/or actuation devices. Similarly, at the *edge* of big clouds, relatively a small numbers of racks will usually cover the cloud owner's locally offered services, and connect to the centralized data centres as much as possible. The individual servers in the racks will have high compute capacity, but at a limited scale due to physical space constraints, and yet they are required to serve millions of clients at high request rates.

- **Deployment flexibility.**
  Ideally, app developers would like to be able deploy their new services in any platform, but in reality individual IoT infrastructures will provide different platforms: one would provide container instances for efficiency and manageability, but another would offer VM instances for isolation. It is clearly painful if the users or developers need to adapt or re-implement their services to the different platforms. In other words, we need to decouple the platform adaptation from service implementation and provisioning. But today, there exists no such framework.

- **Resource isolation.**
  IoT gateway devices and edge servers will need to accommodate compute instances, such as VMs or containers, which have different requirements in terms of performance, policies and security, within the same bare-bones hardware or cluster. Naïve design would always instantiate compute resources as VMs, which have strong isolation property between each other, or against the underlying hypervisor. However, as their footprint is very large, this strategy could result in low efficiency. On the other hand, instantiating all of the compute resources as containers, which share the OS kernel components, results in poor resource isolation in terms of performance and privacy, or violates security requirements.

- **Functional flexibility.**
  Today's applications, even those inside IoT gateways, are highly complex. For example, the use of NGSI/oneM2M/NGSI-LD for Virtual Silos, in reality, requires the

IP/TCP/TLS/HTTP protocol stack. Individual protocol implementations would also be very complex. For example, we need sophisticated congestion control and loss recovery algorithms of TCP for timely data delivery over the Internet, which is much more than the legacy TCP specification or RFC793. Further, if the devices are exposed to the Internet, they also require general security mechanisms like a firewall. At the other end of the spectrum, if the device requires extremely simple a service, for instance sending static IPv6 packets at a constant interval without delivery guarantee, the fully-fledged protocol stack is overkill. Provisioning functionality, kernel modules or libraries bloat instance footprints. This exacerbates service density and resiliency (e.g., to deploy or migrate services) issues.

In the next subsections we begin to describe the first steps we have taken in designing the Flexible Compute Virtualization architecture. We started with the design of the *server level* architecture, i.e. the set of functionality needed on the servers where the VirIoT components are to be virtualized. The main novelties, i.e. the use of Unikernels and LibOS, were introduced with the aim of distributing VirIoT components on low power servers, located on the edge of the network, i.e. where the use of simple VMs or Docker containers might not be possible.

Orchestration and integration of geo-distributed servers/data-centers will be the focus for upcoming work and forthcoming deliverables, instead.

5.1 Server level compute architecture

In order to have both efficiency and flexibility, we need to be able to decouple each other the three fundamental dimensions: functionality, isolation and service implementation, and maximize for efficiency at the same time. Accordingly, at the server level, we want to design a Flexible Compute Virtualization (FCV) architecture in which the same VirIoT component can be easily packaged by means of different virtualization technologies to adapt to different IoT compute infrastructures, while using a unified building tool: Unikraft. In addition to targeting the platform, Unikraft may build application code in the optimal way, in order to minimize resource utilization and security and privacy requirements. To this end, we use two essential technologies, which can be used alone or in combination:

- **Unikernel**
  In our flexible virtualization architecture, we achieve resource efficiency by enabling *Unikernel* instances wherever possible. A Unikernel is a monolithic, single-address space kernel that additionally embraces application(s) in it, without a user-space context. It thus avoids context switches that are expensive for low-latency, high-throughput services. Further, since the kernel can be application-specific and thus minimalistic, disk or memory footprints of Unikernel can be very small. Last but not least, for the same reason, many specialization opportunities in the kernel-level services are available. Unikernels can be built by using different tools, including our Unikraft, which embeds in the Unikernel image the so called Unikraft libraries.

- **Library Operating Systems**
  Typically, system-level services, such as device I/O, file systems and network pro-

tocols, are implemented as a part of the operating system kernel, usually in monolithic form. When using the Library Operating Systems (LibOSes) approach, on the other hand, we run such OS kernel components as user-space libraries, named Linux Kernel Libraries (LKL). This provides high flexibility of deployment, because portability of the user-space code is much higher than that of the kernel code. Since LibOSes usually port a f ull-fledged, production-quality OS kernel code into the user-space, the applications that link LibOSes can benefit from the rich functionality.

Unikernels and LibOSes concepts provide us a significant starting point to enable the flexible virtualization platform that we need especially when a low footprint is necessary, but we must address many technical challenges. Problems in Unikernels include difficulty to build optimized instances for different applications. The developer needs to understand OS features or libraries required by the individual application.

Further, since Unikernels often require custom components, it is sometimes impossible to meet functional requirements. For example, advanced TCP features, such as TCP BBR or Rack, are impossible to use in the Unikernels available today. LibOSes, although being able to address both of the above problems with Unikernels, can only run as user-space applications, meaning that LibOS instances cannot be Unikernels.

In summary, we are able to progressively implement our FCV architecture by facing the following three technical problems:

- Possibility to assemble VirIoT components as Unikernels that are as feature-rich as regular OS kernels.
- Ability to implement VirIoT components exploiting a LibOS both as a Unikernel, and as a real OS kernel.
- Automatic instantiation and deployment in a distributed cloud of heterogeneous optimized images (Docker, plain VMs, Unikernels, etc.).

As shown in Figure 36, the result is that VirIoT components, such as ThingVisors or Virtual Silos, requiring different system features depending on their functionality and services, can be instantiated anywhere, and they can be packaged as:

- **Linux containers**, which include the VirIoT components' software, related libraries and binaries, and possibly additional libraries (Unikraft Libraries or Linux Kernel Libraries) enabling the easy repackaging of the image in different formats, e.g. Unikernel image. The supported containers will be the plain Docker ones[1], but also those based on Unikraft Libraries or Linux Kernel Libraries that may require a different container runtime (e.g. runu instead of runc).
- **Light Virtual Machines based on Unikernels**, which include the software of the VirIoT component and either Unikraft or Linux Kernel Libraries. Light VMs will run on top of an off-the-shelf Hypervisor (possibly bare-metal), such as Linux KVM.

[1]This is the current implentation of VirIoT

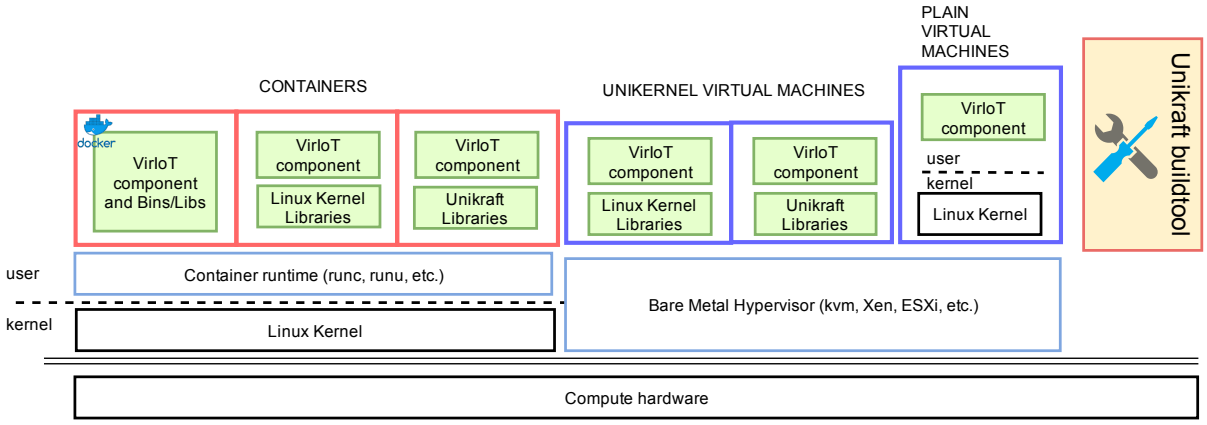


Figure 36: Flexible Compute Virtualization Architecture, server level

- **Plain Virtual Machines**, including a fully-fledged kernel and the VirIoT component software.

The related images (containers or VMs) can be built by using the Unikraft unified tool depicted as a vertical rectangle in the rightmost position of the figure .

In this section we describe this, server level, compute virtualization architecture, referring to Figure 36, and we give the links to the paragraphs of the next sections, where the technical solutions are detailed.

As a design recommendation, when complex ThingVisors or vSilos are to be implemented, then the choice for having a light VM will be to use a LibOS images Section 5.3, possibly packaged as Unikernel. Otherwise, Unikraft libraries (refer to Section 5.2) and Unikernels will be used if ThingVisor or VirtualSilo do not require much system-level functionality, for the highest efficiency and smallest footprint. Such a design recommendation can be applied only when the the platform permits deployment of VMs, i.e. has a suitable Hypervisor. Otherwise the target will be a container of user-space application. In any case, Unikraft build tool will hide the target platform differences from the developers.

5.2 Unikraft

Unikraft was motivated by a number of past Unikernel or related projects, such as Mirage [8], OSv [9], ClickOS [10] and IncludeOS [11], which demonstrate the small footprint and scalability of Unikernels, and performance enabled by related technologies such as user-space packet I/O frameworks [12, 13] and scalable software switches [14, 15]. Unikernels are small because they do not need the generality required by general-purpose, monolithic kernels that support a wide range of applications; instead, individual Unikernels support only their target applications, possibly exploiting domain-specific optimization opportunities, such as known workloads. Small-footprint Unikernels are particularly useful when they are deployed in resource-constrained environments, such as IoT gateways, and when massive numbers of instances (e.g., ThingVisor and VirtualSilo) must be deployed (e.g., at the IoT edge).

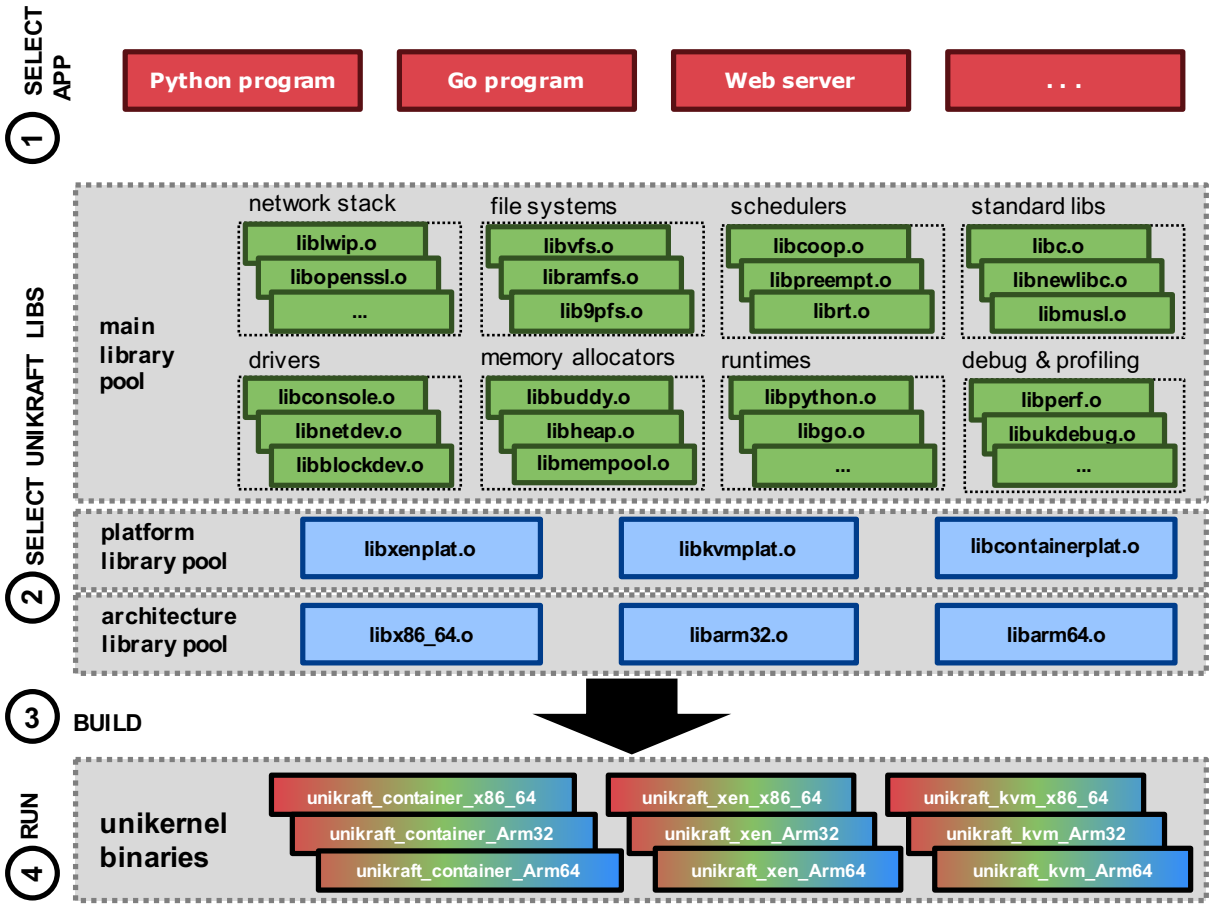


Figure 37: Unikraft Concepts.

Unfortunately, these large advantages do not come for free. Unikernels suffer from the rather tedious process or high engineering costs of composing a different Unikernel for every application. For example, since ClickOS always runs Click as its application, different packet processing applications can be implemented taking the advantages of the flexibility offered by Click modular router [16]. However, it cannot go beyond Click or packet processing applications; building a new application requires the whole Unikernel redesigned. The similar goes true for Mirage; the applications must be written in OCaml, significantly limiting design and optimization opportunities. Unikraft is designed exactly to solve these problems.

Unikraft is an end-to-end framework that builds Unikernels based on modular, fine-grained libraries using its unified built tool (Figure 37). These libraries include memory allocators, networking interface abstraction, various scheduling algorithms, and standard-compliant glibc. Because of the library nature, Unikraft can be used not only to produce Unikernels, but also to build user-space applications based on the same libraries. This is a great advantage, because the application can be built in the form of any of Unikernel or user-process (i.e., container application), while guaranteeing the identical functionality, which largely contributes to enabling our flexible compute virtualization architecture (Figure 36). Unikraft is an active open source project under Linux foundation. Figure 37

illustartes the Unikraft framework.

5.2.1 Support for Containers

Although Unikraft has a large potential, some important features required to achieve Fed4IoT's flexible compute virtualization architecture are unavailable. The first one is the support for containers managed by Docker framework. Containers are lightweight compute instances that have individual file system trees and network interfaces. Container instances share the OS kernel, but compute resources are isolated by the kernel mechanism called *cgroup*. Containers are typically managed by some frameworks, such as Docker [17] and Cloud Foundry [18], to package and deploy container images.

To integrate with the Fed4IoT virtualization stack, adding support for containers in Unikraft necessitates a top-down approach. As described in Section 3, Fed4IoT can use FogFlow for the design of ThingVisors, and FogFlow relies on Docker. Therefore, our goal here is that Unikraft supports Docker-based containers.

OCI Image Format.

We have decided to add support for OCI Image Format [19] (OCIIF), which is a popular container image format defined by Open Container Initiative [20], as Unikraft images. A single image consists of metadata about the contents, dependencies of the image, and filesystem changeset that describes the serialized filesystem and changes made on it.

Unikraft provides support for running a container application. Currently to run Unikraft application within a container we need to fetch the OCI runtime, generate a runtime specific configuration file config.json and a root file system containing the application to execute. This provides us a bare minimal support for running a Unikraft application within a container. The container platform is integrated within Unikrafts build system as an external platform. The user of the platform configures Unikraft build system to create an container image with the structure shown in Figure 38.

The build system gives user the following configuration options: With this configuration you can run a Unikraft application within a container. The Unikraft build system produces the rootfs and a config.json which the container runtime uses to run Unikraft application within a container. The rootfs of the container contains the Unikraft application and the device file needed by the application. There are no external libraries needed the application is a self sufficient image containing all the libraries it requires. The oci runtime does not setup the network interfaces for the container application. We need to explicitly configure the network interface.

Boottime networking.

Since Docker communicates with container instances for a number of reasons, we need to enable this. In particular, since the original Unikraft images are instantiated by Xen control plane without the use of any networking features, we need to enable Unikraft images to support network interfaces and configuration.

To support network interface within a container environment we need to establish a pair of host and guest virtual Ethernet interface. The guest interface is added to the network namespace belonging the container. The guest interface acts as the network interface through which the application within the container interact with the outside world. On the host end, a bridge is created and the host end of the virtual Ethernet is

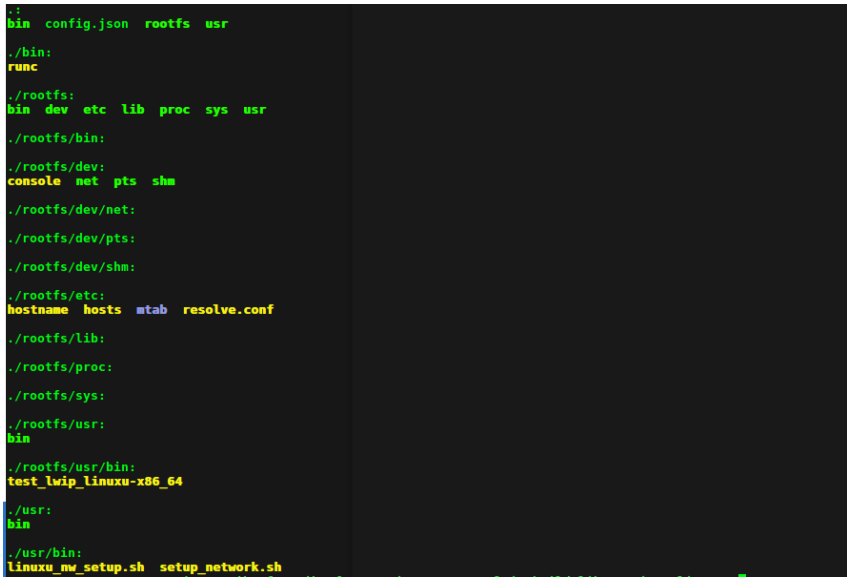


Figure 38: Unikraft Container Image Configuration.

configured to be a slave of the bridge device. The network is setup as a prestart hook. The hook script is called before switching the mount namespace within the container.

For Unikraft to establish a network connection in the container environment, we create a tap interface in the container environment. The tap device read onto the traffic received on the guest end of the virtual Ethernet. The tap interface is created and associate with a bridge on Unikraft boot up. Since Unikraft has user level network stack lwip running within it the l2 packets received on the tap device are forward into the lwip stack and processed by the application. The diagram below gives an overview of the design.

A post start hook setup the network interface. Figure 41 illustrates how it is established.

5.3 Linux Kernel Library

5.3.1 Background

The growth of container architecture and its ecosystem, and development, integration and deployment of programs is nowadays not a headache anymore, thanks to a powerful and flexible environment of underlying virtualization technologies. At the same time, when facing, for instance, the development of a large smart-city platform involving distributed computation across cloud and edge devices, the conventional container system meets stricter requirements of even smaller footprint of the execution environment while avoiding functional degradation to the container runtime. Although the conventional container architecture, or operating system virtualization based on software partitioning, with the namespace technique, has a lightweight advantage when compared to the full machine virtualization with hardware-assisted partitioning, however, as several studies reveal [21, 22, 23, 24], containers still have room for even lighter weight when considering the execution of a set of small programs (a.k.a. micro-services).

Figure 39: Unikraft Network Configuration.

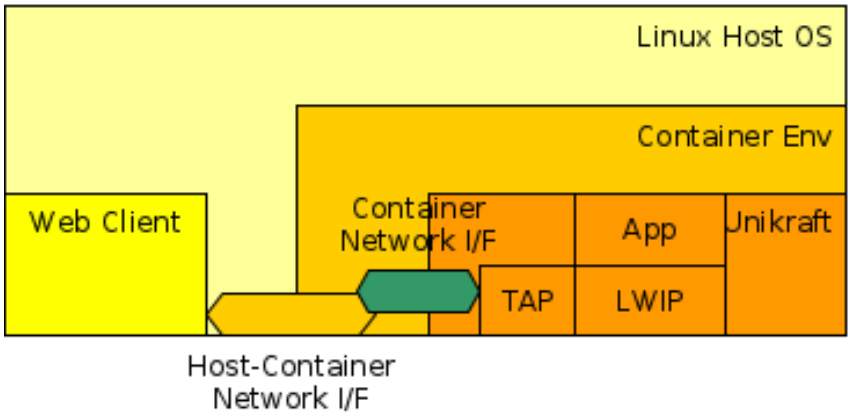


Figure 40: Unikraft Application and Networking in Container Environment.

5.3.2 Existing Solutions

Prior works have also tried to address the issue. Their approaches are varied: [21] tried to reduce the size of container image by splitting into slim and fat images, where the fat image contains tools and the slim image only contains main application so that runtime footprint could be reduced. This approach could offer full feature-set of underlying host systems but supported platforms are limited to host systems if there are no emulation of the deployed images.

Unikernels ([8][9][25]) offer specialized, small guest kernel over hypervisor which has smaller resource usage by combining user- and kernel-space in their runtime memory layout. This integration also contributes smaller runtime overhead by eliminating context switches which happen during I/O operations. Moreover, due to its nature of hypervisor use, the restriction of underlying system is also relaxed, so that we can cover more various devices. However, even several unikernels implementations ([26][9]) offer the binary compatibility to Linux, but the compatibility layer is always incomplete as underlying kernel (in unikernel) is not Linux.

According to the observation from past studies, the lightweight property of virtual-

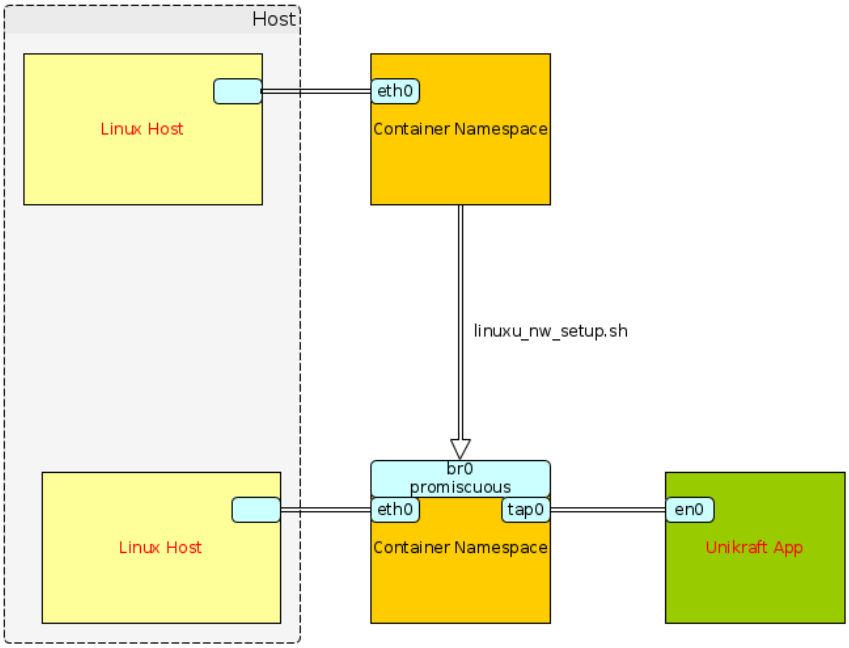


Figure 41: Host Networking Setup with Unikraft and Container.

ization with the various platform support is achieved by paying additional tax for the feature richness. We are trying to fill this gap in our proposed software.

5.3.3 Linux Kernel Library: Rich Feature-set with Specialized Kernel

Our motivation here to develop Linux kernel fitting into our requirements to provide specialize Linux kernel feature without losing the original, mature Linux kernel. Thus the binary compatibility property is simply one of the feature that the original Linux kernel can.

Our design decision follows *stand on the shoulders of giants*, where we will try to avoid writing code and rather we will try to re-shape the mature code base to address more specialized requirement. Thus, Linux Kernel Library (LKL) was designed. LKL is originally aiming to allow reusing the Linux kernel code as extensively as possible with minimal effort and reduced maintenance overhead. It was proposed around 2007 but we have recently developed the LKL so that we can address further use cases, including lightweight property of application runtime.

Unlike other userspace ports of the Linux kernel such as User-mode Linux (UML) [27], LKL aims to be a reusable library in a variety of environments so that programs can link to the components of OS features implemented in the Linux kernel as a library OS. As illustrated in Figure 42, LKL introduces a hardware-independent architecture in the Linux kernel tree by decoupling the LKL host environment (machine/environment-dependent code) from the Linux kernel, largely contributing to the platform independence of this library.

As LKL takes the minimum-additional-code approach, it has other benefits. An application running with the LKL has its richness of Linux features, such as the latest protocols or algorithms inherited from the Linux kernel. This is not possible for other

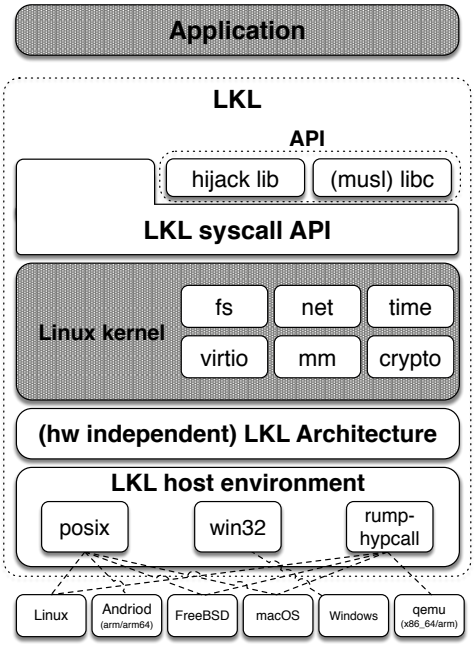


Figure 42: Structure of LKL as a portable and reusable library of the Linux kernel.

supersized kernel approach as those implement Linux feature from scratch. Additionally, thanks to the decoupled design of machine/environmental dependent code into the host environment, the porting effort to non-Linux platform is relatively easy. The latest LKL can run on top of not only Linux host, but also Windows, FreeBSD, macOS, inside UEFI bootloader, Android phone (which is hard to upgrade kernel).

5.3.4 Docker Integration

Interfacing to container infrastructure with LKL is important since VirIoT platform (V2.2) is already developed on Docker/Kubernetes infrastructure, thus LKL with container interface would not request changes to the VirIoT orchestrator (master controller). Our docker integration is not just to provide a wrapper script for LKL-ed programs in order to use via docker client command, but more tightly coupled with the infrastructure by offering plug-in module container runtime. With this runtime, named `runu`, users benefits of LKL with fully-featured, and customized Linux kernel for container instance.

The implementation of the `runu` runtime has two aims: 1) to invoke a LKL-ed process that includes libOS and 2) to bridge the interface of the standardized runtime specification by Open Container Initiative [20]. The implementation allows us to replace the default runtime environment (`runc`) with our custom one via a command-line option for the container invocation to preserve the portability of the container usage model.

Because of the cross-platform support of the Go language, the implementation of `runu` is quite straightforward on both Linux and macOS, which are the currently tested platforms. We also have integrated 9pfs server functionality[2] to share a filesystem layout from a container image to a container instance.

[2] `https://github.com/docker/go-p9p`

5.3.5 Preliminary Evaluations

Our preliminary evaluations consist of two parts to confirm 1) the lightweight property of container instantiation, and 2) the measurement study of feature richness of network stack.

In the mind of serverless platform use cases, where the container instances are frequently started and terminated upon the requests from users, we measured the duration of a simple Python program execution that is ready to listen to a socket. Although the result involves other preparatory elements, such as filesystem and network interface creation and several message interactions between the container engine and runtime to manage container life-cycle, we used the `docker run` command to invoke a container instance because it reflects the typical use cases of serverless deployments.

The experiments were conducted on two machines, Dell PowerEdge R330 with a 4-core 3.8 GHz Xeon E3-1200 and 64 GB memory, interconnected via an Intel X540 10 Gbps link. The machines ran the Linux 4.18.5 kernel in the Fedora release 28 and used Docker 18.06.1-ce for the container framework. To compare `runu` (our implementation for container runtime to instantiate LKL-ed applications) with other approaches, we used the Kata container [28] version 1.8.0 with the 4.19.28-48.1 Linux kernel, gVisor [29] from the git e9ea7230 revision, Nabla containers [30] from git 2cecc88 revision, a native Linux application on the host kernel without containers, and `runu` runtime environment. The base Linux kernel in the LKL was version 4.19.0.

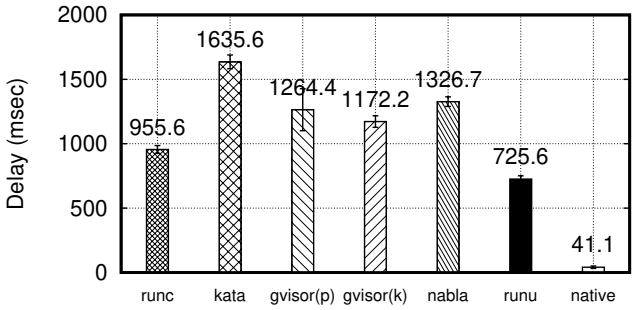


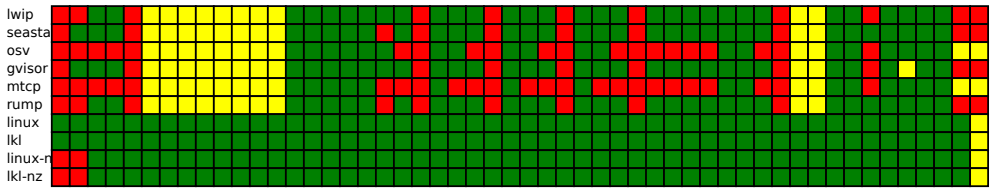
Figure 43: The duration of Python script execution from 30 measurement iterations (with the mean values).

Figure 43 plots the result of this measurement with the standard deviation from 30 repetitions. The duration of the (Linux) native Python program is about 41 ms, and this can be used as a baseline for typical process instantiation in the host system. The standard Docker runtime environment (`runc`) takes 956 ms, requiring configurations such as signal handler installations, system call filtering to the host kernel, followed by multiple process invocations. gVisor (`gvisor`(p), gVisor with ptrace system call trap, and `gvisor`(k), with a trap via a KVM) do not utilize the namespace facility, but require even more time (1264 ms and 1172 ms) because of the overhead of its context switches across multiple system calls. Nabla container (`nabla`) shows longer duration to others except Kata, 1327 ms, and this is also slower than that presented in their paper [25] because our measurement involves multiple containers and OCI runtime interactions while their paper may not include these considerations. The Kata container (`kata`) takes 1635 ms, and this is shorter than a typical virtual-machine instantiation but slightly longer than

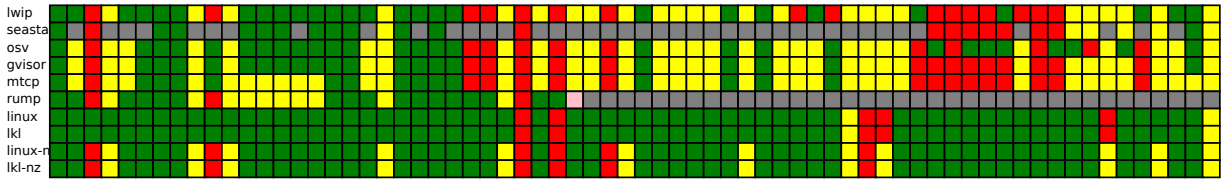
that required by the other approaches and represents the cost of transparent hardware virtualization. `runu` runtime environment takes 726 ms, and this is the fastest among all OCI runtimes including the result of `runc`, although it also involves an additional filesystem mount to load Python library files.

Next, we tried to measure the level of maturity of network stack implementation by testing the conformance of the implementation. We used Ixia IxANVL (automated network validation library) [31], a software utility, to validate network protocol compliance and interoperability. By running a set of test suites to verify the behavior of network stack, based on standard specifications of IETF RFCs, the tool reports the number of successful tests for each network stack implementation. We use the reported number as the level of maturity of network stacks.

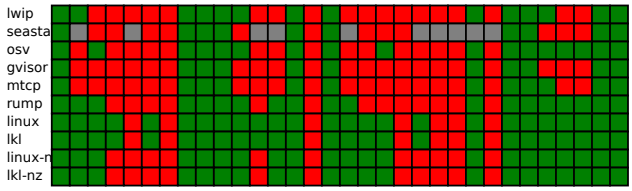
We used lwIP (git 7b7bc349 revision), Seastar (git c19219ed revision), OSv version v0.24, gVisor (git faa34a0 revision), mTCP (git 611cc05d revision), rump kernel (git f10683c revision of buildrump.sh), LKL (git 5221c547af3d revision- based on Linux 4.16.0 version), and native Linux kernel (4.15.0-34 version). We used IxANVL version 9.19.9.32 (Linux) and ran the following test suites for the conformance tests: ARP, IPv4, and ICMPv4.



(a) ARP



(b) IPv4



(c) ICMPv4

Figure 44: Conformance test results (IxANVL) for network protocol based on RFC specifications (Pass=green, Failed=red/yellow).

Figure 44 visualizes the result of this measurement. The matrix in the figure is interpreted as follows: the x-axis is a sequence of test cases, and green box indicates that a test is passed (succeed) while red and yellow box are failed tests. The gray box indicates that the test are not conducted due to the restriction of a particular implementation (e.g., following tests are not able to conduct due to previous test's failure). Of all the tests conducted, the conformance of LKL is the best to the others while achieving the identical

results with the test of typical Linux kernel.

5.3.6 Further Steps

We have accomplished, by the several results described in this deliverable, the design and implementation of a working container and virtualization system using LKL. Our next step is to further eliminate unnecessary part of image contents, in order to reduce the storage and memory footprint. Small storage footprint will contribute the shorter download time of container images, while small memory footprint contributes to relax runtime limitations under scarce-resource environments.

6 Conclusion

This deliverable has described the many details of the the Fed4IoT Virtualization Stack, including both IoT (previously reported in D2.2) and compute virtualization aspects. The whole system is centered around a plethora of facilities such as tools to ease development and deployment tasks (i.e. FogwFlow, Service Function Chaining) as well as a Flexible Compute Virtualization based on Unikernels and library OSs, which focuses on lightweight and rich environments. Those components are working together towards the goal of a widely-distributed, inter-operable platform for IoT virtualization, which can span state-of-the-art cloud- and edge- based infrastructures.

Further studies are still ongoing, and the next D3.2 release of the deliverable will cover, at least, the compute orchestration technology for edge/cloud deployment of Fed4IoT components (considering also 5G/NFV solutions), more advanced implementations (e.g. ThingVisors and vSilo Flavours using FogFlow, Unikernels, etc.) that consider distributed deployment on edge nodes together with light compute virtualization, and system level performance evaluations, which are still missing at this stage.

References

[1] NEC Labs Europe, "FogFlow tutorial," https://fogflow.readthedocs.io/, (Accessed November 26th 2019).

[2] H. Topcuoglu, S. Hariri, and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, March 2002.

[3] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, March 2014.

[4] M. C. Luizelli, W. L. da Costa Cordeiro, L. S. Buriol, and L. P. Gaspary, "A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining," *Computer Communications*, vol. 102, pp. 67 – 77, 2017.

[5] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, "Multi-objective scheduling of micro-services for optimal service function chains," in *2017 IEEE International Conference on Communications (ICC)*, May 2017, pp. 1–6.

[6] L. Wang, Z. Lu, X. Wen, R. Knopp, and R. Gupta, "Joint optimization of service function chaining and resource allocation in network function virtualization," *IEEE Access*, vol. 4, pp. 8084–8094, 2016.

[7] M. T. Beck and J. F. Botero, "Scalable and coordinated allocation of service function chains," *Computer Communications*, vol. 102, pp. 78 – 88, 2017.

[8] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *Acm Sigplan Notices*, vol. 48, no. 4, pp. 461–472, 2013.

[9] A. Kivity, D. Laor, G. Costa, P. Enberg, N. HarEl, D. Marti, and V. Zolotarov, "Osv-optimizing the operating system for virtual machines," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 61–72.

[10] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 459–473.

[11] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," in *2015 ieee 7th international conference on cloud computing technology and science (cloudcom)*. IEEE, 2015, pp. 250–257.

[12] L. Rizzo, "Netmap: a novel framework for fast packet i/o," in *USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 101–112.

[13] Intel, "Intel DPDK: Data Plane Development Kit," http://dpdk.org/, Sep. 2013.

[14] M. Honda, F. Huici, G. Lettieri, and L. Rizzo, "mswitch: a highly-scalable, modular software switch," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM, 2015, p. 1.

[15] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 117–130.

[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[17] Docker, "Docker: Enterprise Container Platform ," https://www.docker.com.

[18] Cloud Foundary, "Open Source Cloud Application Platform ," https://www.cloudfoundry.org.

[19] OCI Image Format Specification, https://github.com/opencontainers/image-spec.

[20] OCI Initiative, "OCI Image Format Specification ," https://www.opencontainers.org.

[21] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight OS containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 199–212. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/thalheim

[22] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC18)*, 2018.

[23] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018, pp. 133–146.

[24] R. Koller and D. Williams, "Will Serverless End the Dominance of Linux in the Cloud?" in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: ACM, 2017, pp. 169–173. [Online]. Available: http://doi.acm.org/10.1145/3102980.3103008

[25] D. Williams, R. Koller, M. Lucina, and N. Prakash, "Unikernels as processes," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: ACM, 2018, pp. 199–211. [Online]. Available: http://doi.acm.org/10.1145/3267809.3267845

[26] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A binary-compatible unikernel," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019.

New York, NY, USA: ACM, 2019, pp. 59–73. [Online]. Available: http://doi.acm.org/10.1145/3313808.3313817

[27] J. Dike, "User Mode Linux," in *Proceedings of the 5th Anual Linux Showcase and Conference*, ser. ALS'01. USENIX Association, 2001, pp. 3–14.

[28] The OpenStack Foundation, "Kata Containers," https://katacontainers.io/, (Accessed Aug 15th 2018).

[29] Google Inc., "gVisor: Container Runtime Sandbox," https://github.com/google/gvisor, (Accessed May 8th 2018).

[30] IBM, "Nabla Containers," https://github.com/nabla-containers/runnc, (Accessed July 3rd 2019).

[31] Ixia, "IxANVL," https://ixia.keysight.com/resources/ixanvl-overview, (Accessed Sep 14th 2018).