A Lightweight Software Stack and Synergetic Meta-Orchestration Framework for the Next Generation Compute Continuum

D3.1 Initial Release of VOStack Layers and Intelligence Mechanisms on IoT Devices

Document Identification			
Status	Final	**Due Date**	30/11/2023
Version	1.0	**Submission Date**	22/12/2023

Related WP	WP3	**Document Reference**	D3.1
Related Deliverable(s)	D2.1, D2.2, D3.2, D6.1	**Dissemination Level (*)**	PU
Lead Participant	SIEMENS	**Lead Author**	Darko Anicic
Contributors	NTUA, CNIT, SIEMENS, ATOS, INRIA, UOM, ODINS, SMILE, ININ, WINGS, IBM, ERCIM, ZHAW	**Reviewers**	Adriana Arteaga (INRIA), Anastasios Zafeiropoulos (NTUA)

Keywords:
Virtual Object, VOStack, Computing Continuum, IoT Interoperability, IoT Virtualization, CEP, TinyML, Semantic Model, W3C WoT, OMA LwM2M, NGSI-LD

Document Information

List of Contributors	
Name	Partner
Symeon Papavassiliou, Anastasios Zafeiropoulos, Eleni Fotopoulou, Nikolaos Filinis, Dimitrios Spatharakis, Ioannis Tzanettis, Ioannis Dimolitsas, Constantinos Vassilakis	NTUA
Giacomo Genovese, Antonella Molinaro, Antonio Iera, Alessandro Carrega, Luigi Rachiele	CNIT
Darko Anicic, Mahda Noura, Haoyu Ren Kirill Dorofeev, Nilay Tuefek Oezkaya, Martin Oestreicher, Ege Korkan	SIEMENS
Guillermo Gomez	ATOS
Nathalie Mitton, Adriana Arteaga Arce, Carol Habib, Hazem Chaabi	INRIA
Lefteris Mamatas, Georgios Papathanail, Angelos Pentelas, Maria Drintsoudi, Panagiotis Papadimitriou, Ilias Sakellariou	UOM
Rafael Marin Perez, Alejandro Arias Jiménez	ODINS
Rudolf Sušnik, Janez Sterle	ININ
Marco Jahn	ECL
Konstantinos Almpanakis, Konstantinos Lessis	WINGS
Sofiane Zemouri	IBM
François Daoust	ERCIM
Giovanni Toffetti, Leonardo Militano	ZHAW

Document History			
Version	Date	Change editors	Changes
0.1	15/02/2023	Darko Anicic, Mahda Noura, Haoyu Ren, Kirill Dorofeev	ToC preparation and first draft, Editing of Sections 1 Editing of Sections 4, 9, and 11
0.2	30/03/2023	Giacomo Genovese	Editing of Sections 2, 3 and 7
0.3	30/04/2023	Anastasios Zafeiropoulos, Dimitrios Spatharakis	Editing of Sections 8 and 10 Extending Section 3 and 4
0.4	30/05/2023	Giacomo Genovese	Extending Section 4
0.5	30/06/2023	Panagiotis Papadimitriou	Editing of Sections 5
0.6	30/07/2023	Rafael Marin Perez, Adriana Arteaga Arce	Extending Section 5, editing of Sections 6
0.7	15/11/2023	Darko Anicic	Full version of the deliverable available for internal review
0.8	25/11/2023	Adriana Arteaga	Comments by the internal review
0.9	15/12/2023	Mahda Noura, Haoyu Ren, Darko Anicic, Kirill Dorofeev, Giacomo Genovese, Anastasios Zafeiropoulos, Panagiotis Papadimitriou, Rafael Marin Perez	Updated version with revisions
1.0	21/12/2023	Darko Anicic	Final version to be submitted

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	Darko Anicic (SIEMENS)	15/11/2023
Internal reviewers	Adriana Arteaga (INRIA), Anastasios Zafeiropoulos (NTUA)	15/12/2023
Project Coordinator	Symeon Papavassiliou (NTUA)	21/12/2023

Table of Contents

List of Figures

List of Tables

List of Acronyms

Abbreviation /Acronym	Description
2D	Two dimensional
3D	Three dimensional
5G	Fifth Generation
AI	Artificial Intelligence
API	Application Programming Interface
ARE	Ambulance in a Rural Environment
ASP	Application Service Provider
ASV	Application Service Vendor
BAS	Building Automation System
CCNM	Computing Continuum Network Manager
CD	Continuous Delivery
CFS	Container Freight Stations
CI	Continuous Integration
CICD	Continuous Integration and Continuous Delivery
CPU	Central Processing Unit
cVO	Composite Virtual Object
DID	Distributed Identifier
DL	Deep Learning
DLT	Distributed Ledger
DPR	Data Processing Requirements
DT	Digital Twin
E2E	End-to-End
EHR	Electronic Health Record
ERP	Enterprise Resource Planning
FR	Functional Requirement
FRM	Federated Resource Manager
GB	Gigabyte
GNSS	Global Navigation Satellite System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
GW	Gateway
HDA	Hyper Distributed Application
HDAR	Hyper Distributed Application Repository
HVAC	Heating, Ventilation and Air Conditioning
HW	Hardware
IaaS	Infrastructure as a Service
IoT	Internet of Things
KPI	Key Performance Indicator
LP	Local Processing
MB	Megabyte
mHWDev	Minimal HW Device
ML	Machine Learning
MQTT	MQ Telemetry Transport
NB-IoT	Narrowband Internet of Things
NFR	Non-Functional Requirement
OBU	On Board Unit

Abbreviation /Acronym	Description
OS	Operating System
PIS	Port Information Systems
PS	Primary Screen
QoS	Quality of Service
ROS	Robot Operating System
SLA	Service Level Agreement
SLAM	Simultaneous Location and Mapping
SMO	Synergetic Meta-Orchestrator
SR	System Requirement
SW	Software
TD	Touchscreen Display
TDD	Test Driven Development
TRL	Technology Readiness Level
TSN	Time Sensitive Networking
UC	Use Case
UHD	Ultra-High Definition
VO	Virtual Object
XACML	eXtensible Access Control Markup Language
ZSM	Zero Touch Network and Service Management

Executive Summary

Deliverable 3.1 is the first iteration of VOStack and intelligence mechanisms on IoT Devices in the NEPHELE project. VOStack, which stands for Virtual Object Stack, is a software stack that provides virtualization of physical devices via a concept of Virtual Object. Virtual Object (VO) is the virtual counterpart of an IoT device. It provides a set of abstractions for representing and managing any type of IoT device. Intelligence mechanisms on IoT Devices aim to facilitate on-device intelligence. Deploying TinyML and Complex Event Processing (CEP) techniques, IoT devices can enable on-device, real-time, context-aware decision-making in response to streaming data generated by these devices. Deliverable 3.1 (D3.1) provides the concept of intelligence on IoT Devices, semantic representation of IoT Devices via VOs, as well as a set of functionalities such as management, authorization, security, discovery, and orchestration as provided by VOStack. D3.1 also contributes to autonomous networking functions in the NEPHELE project, aiming to provide the capacity of a network to self-configure, self-monitor, and self-optimise without human intervention.

With this, the goal of D3.1 is to contribute to the convergence of different IoT Technologies, thereby guaranteeing continuous and seamless openness and interoperability in the NEPHELE project. This deliverable includes the current outcomes of Work Package 3, i.e., tasks T3.1 - T3.5. The document will be revised and updated in month 24 of the project, in deliverable D3.2, when the final release of VOStack Layers and intelligence mechanisms on IoT Devices will be provided.

1 Introduction

NEPHELE is a Research and Innovation Action (RIA) project funded by the Horizon Europe programme under the topic "Future European platforms for the Edge: Meta Operating Systems". NEPHELE vision is to enable the efficient, reliable, and secure end-to-end orchestration of hyper-distributed applications over a programmable infrastructure that is spanning across the compute continuum from IoT-to-edge-to-cloud.

The next generation Internet of Things (IoT) and Edge Computing technologies are evolving at a rapid pace and the global number of connected IoT devices continues to increase. This trend occurs in parallel with the increase in the heterogeneity of the IoT technologies and standards. In order to create the value, users of these technologies and standards have few challenges to tackle. For example, the heterogeneity of diverse types of intelligent IoT devices needs to be harnessed, diverse communication protocols need to be supported, the complexity of various information models for semantic representation of IoT assets must be resolved, and so forth. To address these challenges, the NEPHELE project introduces the concept of Virtual Object (VO). A VO is the virtual counterpart (digital twin) of an IoT device. It provides a mechanism to virtually represent and manage any type of IoT device. Virtual Object Stack (VOStack) is a software stack, which can be used for creating, discovering, orchestrating, and consuming VOs. Furthermore, the VOStack provides mechanisms for managing the interaction among IoT devices and VOs. VOStack resides at the edge and is a crucial component to efficiently exploit resources in the continuum from Cloud-to-Edge-to-IoT-Device.

This deliverable reports on the activities of Work Package 3 (WP3). WP3 is devoted to the topic of convergence of different IoT Technologies to guarantee continuous and seamless openness and interoperability.

WP3 in the NEPHELE project has the objective to develop the VOStack. It includes: the development and release of the software VOStack; the development of appropriate abstractions and translation mechanisms supporting semantic interoperability in terms of IoT devices and related VOs management; the development of a set of intelligent IoT devices management mechanisms; the definition and development of a set of generic/supportive functions and IoT device virtualized functions; and the definition and development of orchestration management interfaces for the VOs.

In this deliverable, we first describe the NEPHELE vision scoped around Virtual Object and the Virtual Object Stack (Section 2). Section 2 also provides relations of this deliverable to other project tasks. We continue with the state of the art related to this work (Section 3). We present the status of Task 3.1: “Intelligent IoT devices modelling, management and interoperability” (Section 4). Subsequently, the status of Task 3.2: “Autonomic functionalities and ad-hoc clouds” is presented in Section 5. The report on security functionality, as a part of Task 3.1, is provided in Section 6. The status of Task 3.3: “IoT device virtualized and supportive functions” can be found in Section 7. Activities in Task 3.4: “Orchestration management interfaces” are reported in Section 8. The work in Task 3.5: “Intelligence on IoT devices and interplay with VOs” is presented in Section 9. We provide the status of VOStack implementation and discuss our open-source activities in Section 10. Finally, we conclude this deliverable with Section 11.

2 Virtual Object and Virtual Object Stack

2.1 VO Concept

Nowadays, billions of interconnected devices constantly produce data which cross the network to be digested and processed by Cloud platforms or other IoT devices. IoT scenarios have evolved over time providing increasingly complex and integrated services as well as a growing development of heterogeneous sensors and devices with different resources capabilities, ranging from single-board computers such as *Raspberry Pi*, with Gigabytes of memory, to microcontrollers with a few kilobytes of memory. Furthermore, the interest that has developed around the IoT has driven the research and development of new standards which, depending on the case, sought to respond to the needs of the reference application context. With the passage of time, therefore, an increasingly heterogeneous context has been created, constrained by the limited availability of device resources, in stark contrast to the need for interoperability and scalability of new smart scenarios such as urban environments, factories, agriculture, logistics, etc.

Virtual Object (VO) is a software service that extends the properties, attributes, and functionalities of real-world physical devices within the digital infrastructure of the network. The VO serve several important purposes which help to overcome limitation described above:

- **Semantic abstraction**: It provides a standard representation of physical device for easier management, monitoring, and discovery of device resources.
- **Interoperability**: It can be used in heterogeneous scenario to bridge communication between different standards and protocols.
- **Data modelling**: It enables the use of different data modelling for different purpose leveraging physical device from complex data model transmission.
- **Simulation and testing**: In its Digital-twin extension, VO can be used to simulate the physical device behaviour in a virtual environment before deployment.
- **Remote management and accessibility:** VO enables continuous data access and remote management where physical device access can be limited or impractical.
- **Resource consumption**: It can leverage the physical device.

Virtual objects offer several advantages over physical objects, and they are an essential part of modern computing. They are easier to create, store, and share, for example, a virtual object can be resized, reshaped, or even migrated to a different location. Each VO is a part of a distributed application in the edge-cloud infrastructure, and it is orchestrated as a containerized stateless micro-service. So that, NEPHELE project aims to realize the VO as an extension of the IoT physical device in the virtualized infrastructure of the Edge-Cloud continuum and it is positioned as a men-in-the-middle between the physical device and the virtualized environment ad described in Figure 2-1.

Moreover, the project proposes the development of a new microservice, the composite **VO (cVO),** which composes and aggregates interfaces from multiple VOs to provide advanced service for specific clients' application. The cVO is an entity of the application graph providing VO client interfaces to the application meanwhile is connected to multiple VOs from which it acquires IoT data. Thus, a **cVO** is not just an abstract way to describe the communication of several VOs. It is a collaboration of several VOs towards the production and exposure of a combined set of measures/outputs. A cVO is not an application expressed as a service graph incorporating the several VOs as nodes of the graph. A cVO should be seen as a single VO consuming the output of several VOs and exposing a single output set following the definition of a VO. In the case that the cVO is linked with one VO, it can provide advanced functionalities (e.g., application specific, digital twin) for this VO. Thus, a cVO is a virtual object itself that:

- maintains the relationship among the participating VO(s),
- consumes the output of the participating VO(s),
- illustrates a logic processing several inputs,
- exposes a new output set regarding the coalition of VOs.

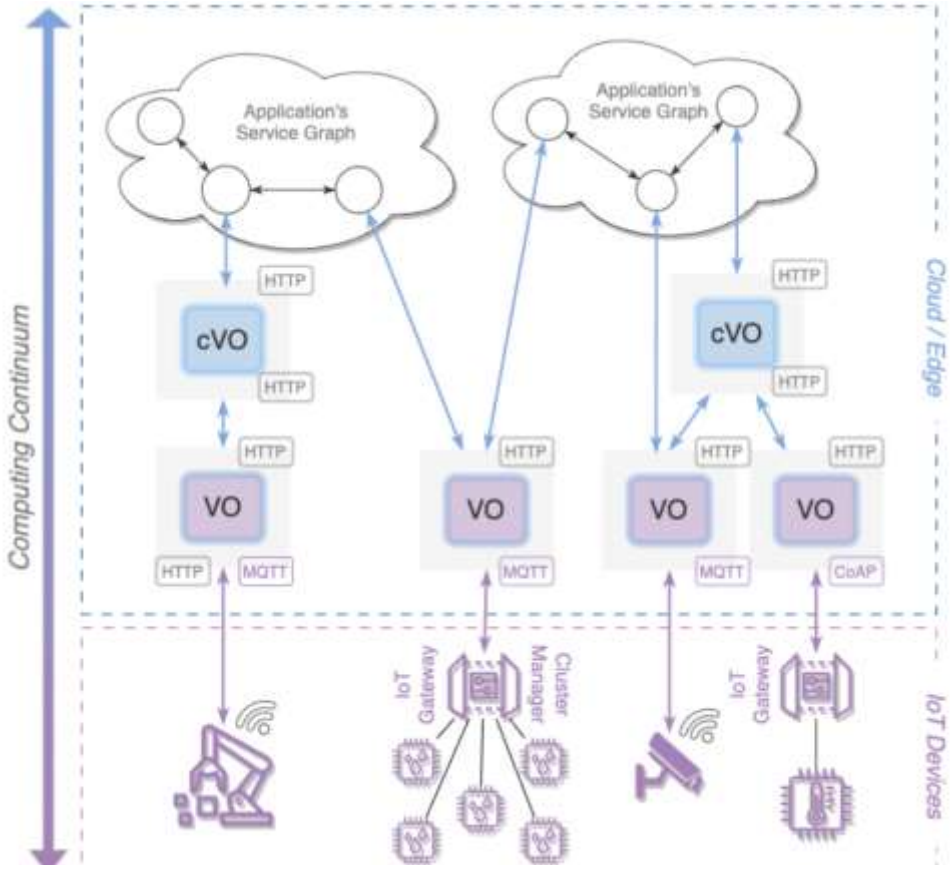


Figure 2-1: VO as an extension of the IoT physical device

2.2 VOStack

In order to support the features described above, NEPHELE project aims to develop a complete software stack, VOStack, which is divided in three levels:

- Edge/Cloud convergence,
- Backend logics (for IoT functionalities),
- Physical convergence.

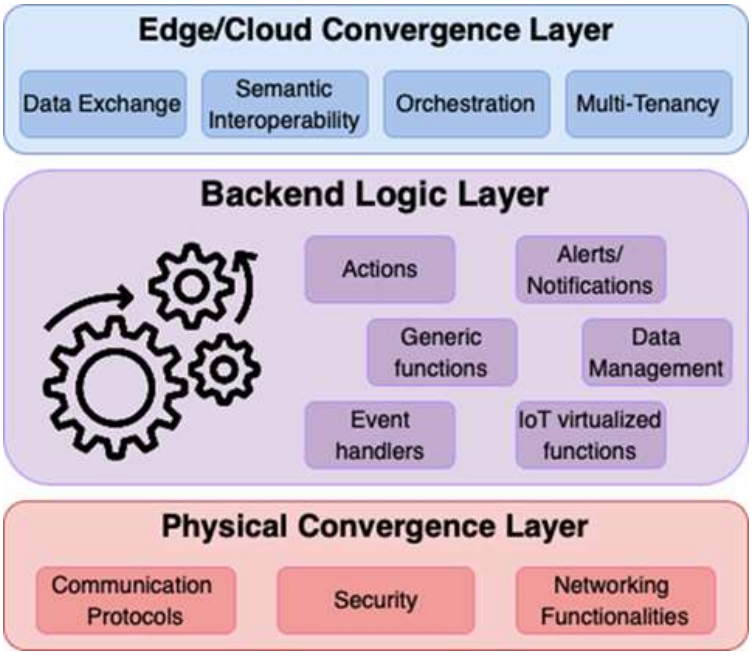


Figure 2-2: Virtual Object Stack (VOStack) Layers

The stack implementation is focused on development of specified functionalities to support IoT interoperability and cooperation in the context of edge-cloud continuum in a device-independent way. Such implementation will support computing and network function virtualization at the edge, and IoT application scenarios like Artificial Intelligence (AI), and real-time rescue operations. It is defined per layers, and requirements have been defined in WP2 in deliverable D2.1 and listed in two different tables. For convenience, we present below the Functional Requirements (FR) of the VOStack from D2.1 in Table 2.1, and the non-functional requirements supported by the multiple-layer VOStack in Table 2.2. The implementation of these requirements will depend on the characteristics and needs of each use case.

Table 2.1: VOStack Functional Requirements

ID	Description	VOStack related feature	Associated Interface(s)	Difficulty	Priority
FR_VOS_001	The VOStack shall provide interfaces to connect heterogeneous IoT devices directly or through an IoT gateway	Interoperability, Security, and IoT Device Management	VO-to-Device	High	High
FR_VOS_002	The VOStack shall provide linking and collaboration mechanism between VOs across the compute continuum	Autonomicity and Ad-Hoc Networking	VO-to-VO	Medium	High

FR_VOS_003	The VOStack shall provide multi-tenant access to IoT devices	Interoperability, Security, and IoT Device Management. Autonomicity and Ad-Hoc Networking. Generic Functions	VO-to-VO-to-Device. VO-to-VO	High	High
FR_VOS_004	The VOStack shall provide offloading functions between VO across the compute continuum	IoT Device Virtualization Functions	VO-to-Application. VO-to-VO-to-Device	Medium	Medium
FR_VOS_005	The VOStack shall allow the integration of resources across the compute continuum to enhances the capabilities of the IoT device	IoT Device Virtualization Functions	VO-to-Application	Medium	Medium
FR_VOS_006	The VOStack shall provide mechanisms for the VO migration across the compute continuum	Orchestration Management	VO-to-Orchestration	High	High
FR_VOS_007	The VOStack shall allow multiple instances of a VO across the compute continuum, and support linking and collaboration between them	Orchestration Management	VO-to-VO. VO-to-Orchestration	High	High
FR_VOS_008	The VOStack shall provide a proxy service for IoT devices	Generic Functions	VO-to-Application	Medium	Medium

FR_VOS_009	The VOStack shall provide security for the connection of the IoT device and data management	Interoperability, Security, and IoT Device Management. Generic Functions	VO-to-Device. VO-to-Application	High	High
FR_VOS_010	The VOStack shall define device management premises for configuration and control functions, monitoring and diagnostics, software maintenance and updates	Interoperability, Security, and IoT Device Management. Autonomicity and Ad-Hoc Networking. Orchestration Management	VO-to-Device; VO-to-Orchestration	High	High
FR_VOS_011	The VO Stack shall enable the computation of time-sensitive networking (TSN) schedules for prioritization of traffic between IoT devices and (c)VOs	Autonomicity and Ad-Hoc Networking	VO-to-Device	Medium	High
FR_VOS_012	The VO Stack shall enable the population of TSN schedule configurations into TSN bridges using a technology-specific SouthBound API (e.g., NETCONF/RESTCONF).	Autonomicity and Ad-Hoc Networking. Interoperability, Security, and IoT Device Management	VO-to-Device	Medium	High

FR_VOS_013	The TSN control plane in the VO Stack shall provide a generic NorthBound API using a well-defined JSON schema for application configuration and requirements processing	Autonomicity and Ad-Hoc Networking. Orchestration Management	VO-to-Orchestration	Medium	High
FR_VOS_014	The VO shall expose a special type of SouthBound interface that can support SDN-based IoT devices	Autonomicity and Ad-Hoc Networking. Interoperability, Security, and IoT Device Management	VO-to-Device	Medium	High
FR_VOS_015	Clustering capabilities shall be supported at the cVO level or Compute Continuum Network Manager, for associating IoT nodes with VOs, configuring node-specific protocol settings, and implementing proactive routing.	Autonomicity and Ad-Hoc Networking. Orchestration Management	VO-to-Orchestration; VO-to-Device	Medium	High

Table 2.2 VOStack non-Functional Requirements

ID	Description	NEPHELE related feature	Difficulty	Priority

NFR_VOS_01	The system should enable low latency and high bandwidth communications, and high computational power for rapid response on data processing	Cloud and Edge Synergetic Orchestration. Computing Continuum Network Management. Federated Resource Management	High	High
NFR_VOS_02	The system should guarantee data security and privacy in transmission and storage	Security and IoT Device Management. VO Storage Space	High	High
NFR_VOS_03	The system should support and store various IoT data sources with varying workloads	Generic/Supportive Functions. Interoperability. VO Storage Space. Computing Continuum Network Management	High	High
NFR_VOS_04	The system must be able to represent IoT devices as extended Digital Twins offering additional features and functionalities	IoT Device Virtualized Functions. IoT Device Management	Medium	Medium
NFR_VOS_05	The system must be able to receive and process data from IoT devices and the environment	Generic/Supportive Functions. Interoperability	Medium	High
NFR_VOS_06	The system must be able to store IoT data	VO Storage Space	Medium	High
NFR_VOS_07	The system must be able to monitor devices and networks to trigger alerts when an error on a task occurs or a specific event is detected	Generic/Supportive Functions IoT Device Management	High	High
NFR_VOS_09	The system must be able to detect objects and humans and predict future values of associated risks/motion/condition	Generic/Supportive Functions AI models	Medium	Medium
NFR_VOS_10	The system should be able to monitor devices and networks to deploy additional elements when needed	Generic/Supportive Functions IoT Device Management	Medium	High

The WP3 will release the VOStack, it will make available a set of open-source libraries and tools including the specified functionalities per layers and, in accordance with the VOStack specifications, the WP3 will specifically release two VO software implementations. Both implementations have to adhere to the same principles and core Application Programming Interfaces (APIs), however being

aligned with specifications coming from different standardisation groups. These groups regard the W3C Web of Thing[1] (WoT) and the Open Mobile Alliance (OMA) Lightweight M2M (LwM2M), see LwM2M – OMA SpecWorks specifications[2]. Both implementations utilize only the (c)VO Descriptor to configure (c)VOs. The (c)VO Descriptor is a YAML[3]file that represents all different options and mappings for a user-defined (c)VO. The descriptor is parsed during instantiation time and manages the initialization of the Web of Things runtime and the deployment of the Virtual Object itself.
Moreover, VO will have four main interactions within the deployment environment as depicted in the Figure 2-3, which are then implemented into the VOStack layers:

- **VO-to-IoT Device Interaction**: to link the physical world of IoT devices to the edge-cloud continuum virtualized environment.
- **VO-to-Applications (prosumers) Interaction**: to enable data exchange between cooperative services and to create services' chains.
- **VO-to-Orchestration Interaction**: to manage VO microservice lifecycle.
- **VO-to-Storage Entity Interaction**: for the allocation of data in an efficient and scalable manner.

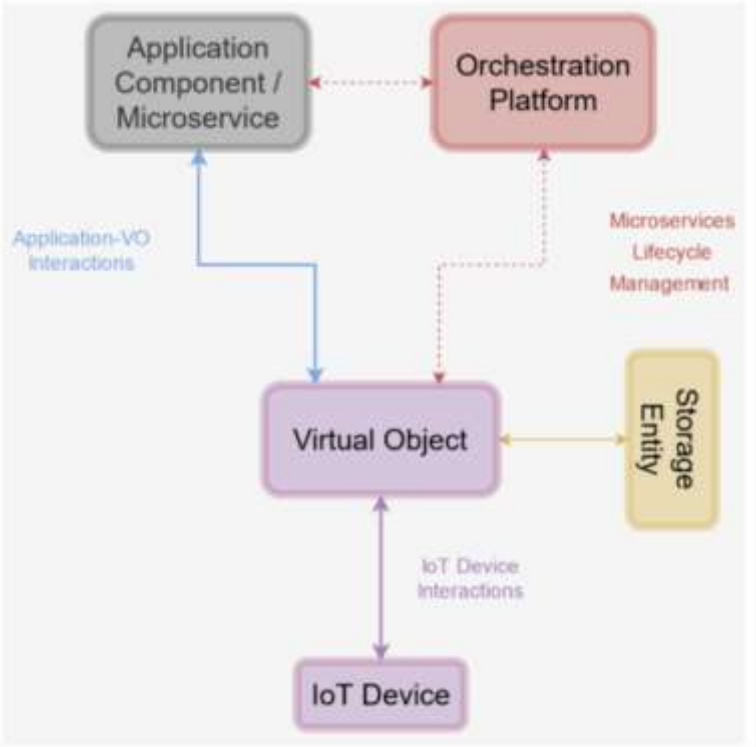


Figure 2-3: Virtual Object Interactions

2.2.1 Edge/Cloud Convergence Layer

This layer of the VOStack is application-oriented and it oversees the management of all interfaces that interact in communications with other edge-cloud entities present in the NEPHELE platform like applications and orchestrators that intend to interact with the IoT physical domain and virtualized services. Such Northbound interfaces, for instance, must be implemented to enable telemetry real-time data transmission using standard semantic protocols like W3C and OMA-LwM2M as well as security access. Generic and supportive functions will be exposed to consumer by this layer using standard protocols. As defined in the deliverable D2.1, this layer must enable the interaction between (c)VO an

[1] https://www.w3.org/WoT/
[2] https://omaspecworks.org/what-is-oma-specworks/iot/lightweight-m2m-lwm2m/
[3] https://yaml.org/spec/1.2.2/

Application, VO-to-Application Interaction, and between VO and Orchestrator, VO-to- Orchestration Interaction. In the first case, interfaces will rely on web-oriented protocols (i.e., Hypertext Transfer Protocol and Hypertext Transfer Protocol (HTTP) and HTTP Secure (HTTPs)).

2.2.2 Backend Logics Layer

This layer is internal to the VO, and it does not expose interface to prosumers. The layer deals with implementing the enhancement of VO functionalities with respect to the physical device and interacting with the support "sidecar" services. This layer implement generic/supportive functions like data management, authentication, authorization, telemetry, etc.
For instance, the datastore can be both internal, through the implementation of lightweight solutions (i.e., SQLite) for the historicization of data less linked to frequent temporal updating, and external, through the implementation of more high-performance solutions suitable for the processing of large quantities of time-related data (timeseries) with high update frequency (i.e., InfluxDB).
VO backend logics will help to develop more efficient security capabilities, authentication, and authorization functionalities compared to the ones implemented into physical devices with restricted computational resources in order to ensure secure communications for VO-to-X (where X could be an application, VO, or a physical device) interactions.

2.2.3 Physical Convergence Layer

Due to complex heterogeneity in IoT device scenario, the layer of convergence with the physical world must face important challenges to allow the interconnection, interoperability, and device management of as many as possible, if not all, constrained and non-constrained IoT devices. This task is therefore mainly delegated to the southbound interfaces of the VO service which has the task of supporting several protocols used within the IoT domain. For instance, it must be able to expose interfaces that allow the acquisition of data through application-level protocols such as MQTT and CoAP and use different meta-data for the semantic representation of the device and its resources according to standards such as, for example, OMA-LwM2M and W3C-Web of things. The choice of interfaces and protocols to enable has to be configurable in the VO during deployment by orchestrator with respect the virtualized physical device. The VO configuration is going to be defined in the ***Descriptor*** file.

3 State of the Art

In this section, we review the state-of-the-art topics related to NEPHELE. In particular we provide the following details, Section 3.1 explains the different IoT application protocols relevant to NEPHELE, device virtualization techniques in Section 3.2, device interoperability and management between different IoT protocols in Section 3.3, IoT networking requirements and ad-hoc cloud networking mechanisms in Section 3.4 and finally device intelligence in Section 3.5.

3.1 IoT Application Protocols

IoT devices shall be enabled with additional functionality, both in terms of event processing rules as well as capability of creating a chain of services together with other entities/services present in the continuum. The management of these "chains" requires a communication interface on constrained devices.

3.1.1 CoAP

The Constrained Application Protocol (CoAP) is a REST (REpresentational State Transfer) protocol based on the notion of resource to model all client/server interactions as an exchange of resource representations. Using CoAP is possible to create a remote resource management infrastructure through simple access and interaction functions similar to those of the HTTP protocol (PUT, POST, GET, DELETE). Each Client/Server resource is assigned to a universal identifier, called a URI, which can be used on the Web to obtain a representation of the resource. One of the main objectives of the CoAP protocol is to create a Web protocol suitable for the needs of devices with limited resources in terms of computation and energy. The way in which this protocol is to be implemented does not consist in a simple compression of the HTTP protocol, but rather in the implementation and optimization of a subset of the features offered by the REST architecture, in common with HTTP, in the perspective of machine-to-machine (M2M) communication applications. The protocol standardisation conducted by the Constrained Representational State Transfer Environments (CoRE) Working Group of the Internet Engineering Task Force (IETF) [1].
The key features of the CoAP protocol are:

- web protocol for network nodes with limited resources,
- transport on datagram protocol as User Datagram Protocol (UDP) with optional reliability,
- asynchronous exchange of messages,
- low overhead and low header parsing complexity,
- support for Uniform Resource Identifiers (URI) resources and content-type information of the payload,
- simple creation of intermediaries (proxies),
- ability to cache responses to reduce response times and bandwidth occupation,
- translatability into the protocol without HTTP states with the possibility of creating proxies to guarantee HTTP nodes access to CoAP resources and vice versa.

The interaction between CoAP nodes occurs similarly to the client/server model of the HTTP protocol. However, the nature of machine-machine interactions that occur between remote devices in IoT suggests an implementation of the CoAP protocol in which each node acts as both client and server. Such a node is called an endpoint. A CoAP request is equivalent to an HTTP request: it is sent from a client to a server to request the server to perform an action (via a method code) on a resource (identified by URI). The server then sends the original client a response code containing a response code and a representation of the requested resource, if any. Unlike HTTP, CoAP performs these request/response exchanges asynchronously over a datagram transport protocol such as UDP. This is achieved through a protocol layer of messages that can support optional reliability through exponential backoff algorithm. CoAP messages can be of four types: Confirmable (CON), Unconfirmable (NOT), Acknowledgement (ACK) and Reset (RST). The method and response codes included in some of these messages specify that it is a request or a response.
CoAP protocol as composed of two sublayers:

- a messaging sublayer that deals with the management of the exchange of messages which, as mentioned before, is asynchronous and bound to UDP.
- a request/response interaction sublayer that uses Method and Response codes to process the request or response.

The methods supported by the protocol are a subset of the HTTP protocol request methods: **GET**; **POST, PUT**, **DELETE.**

CoAP supports four security modes: 1) unsecured, 2) pre-shared key with AES ciphers, 3) raw public keys using DTLS, AES ciphers and Elliptic Curve algorithms for key exchange, and 4) DTLS together with X.509 certificates. Application layer security is possible using RFC 8613, which defines Object Security for Constrained RESTful Environments (OSCORE), a method for application-layer protection of CoAP using CBOR Object Signing and Encryption (COSE). RFC 9203 specifies a profile for the Authentication and Authorization for Constrained Environments (ACE). It uses OSCORE to provide communication security and proof-of-possession for client keys bound to OAuth 2.0 access tokens.

CBOR

The Concise Binary Object Notation (CBOR) is a binary serialisation format loosely based upon JSON and often used with CoAP to compress messages. CBOR supports integers, floats, strings and arrays and maps of name/value pairs where names are represented as semantic tags. IANA maintains a CBOR tags registry that maps semantic tags to a URL (i.e., web address) for a resource that describes the semantics. CBOR is specified in RFC 8949 Concise Binary Object Notation (CBOR)[4].

3.1.2 MQTT

The Message Queue Telemetry Transport (MQTT) protocol [80], was invented by Andy Stanford-Clarck (IBM) and Arlen Nipper (Arcom) and is designed for machine-to-machine (M2M) applications in an IoT environment. This application layer protocol was designed primarily with the aim of creating a defined publish/subscribe mechanism that would allow devices to exchange messages through an extremely lightweight protocol. This happens through the use of topic, each client publishes a message *m* specifying the topic *t* to which it is associated so, thanks to the use of a broker, which acts as a server within the network, all clients who have subscribed to topic *t* will receive the message *m*. Basically, in MQTT, each node (client) can be publisher when it produces (publish) data, and subscriber when it wants to consumes (subscribe) data, even its own data.

As in HTTP, MQTT uses the Transmission Control Protocol (TCP) and IP protocols as underlying layers, optimising their overhead. Then, the devices communicate with their applications using TCP through an MQTT message broker that manages message forwarding between publishers and subscribers while also ensuring secure information exchange. Publishers and subscribers are therefore both customers of the system and can usually perform both functions.

The reliability of the protocol is entrusted to three levels of Quality of Service (QoS):

- QoS 0- at most once delivery- No response and no new forwarding attempts. It is the minimum level of QoS provided by the protocol and is also the fastest and lowest cost.
- QoS 1- at least once delivery-Duplicates, resubmissions, of the message are allowed, a confirmation request is expected.
- QoS 2-exactly once delivery-the protocol ensures that no duplicates are forwarded and that it is sent once and only once to the application subscribed to the topic. It includes a four-step handshake.

To date, many brokers have been developed in different programming languages. Some examples are Mosquitto (developed by the Eclipse foundation in C language), Mosca (developed in node.js) and RabbitMQ (developed in Erlang language).

[4] https://www.rfc-editor.org/rfc/rfc8949.html

3.1.3 HTTP

HTTP, or Hypertext Transfer Protocol, serves as the bedrock of communication on the World Wide Web. Conceived by Tim Berners-Lee in 1989, it is an application layer protocol meticulously designed to facilitate the exchange of data between a client, typically a web browser, and a server. At its core, HTTP operates on a request-response model. When a user, through their web browser, initiates an action such as requesting a web page, the browser sends an HTTP request to the server. This request specifies the desired action, and the server, in turn, processes the request and issues an HTTP response. This response contains the requested data or signals any encountered errors.

One of the defining characteristics of HTTP is its stateless nature. Each client-server interaction is independent, and the server does not retain information about prior requests from a specific client. While this simplicity aids implementation, it necessitates additional mechanisms, often in the form of cookies or sessions, for managing user states.

Uniform Resource Identifiers (URIs) play a pivotal role in HTTP. Resources on the web, whether they be web pages, images, or other content, are identified by URIs. Commonly, URLs (Uniform Resource Locators) serve as a subset of URIs, providing the specific address for accessing a resource.

HTTP communicates using a variety of methods, or verbs, which convey the desired action on a resource:

- GET method retrieves data,
- POST submits data,
- PUT updates a resource,
- DELETE removes a resource.

Headers are integral to HTTP messages, enriching them with additional information. Whether conveying details about content type, length, caching directives, or other specifics, headers enhance the understanding of the message, or the resource being transferred.

In the pursuit of enhanced security, HTTPS (HTTP Secure) encrypts the exchanged data between client and server using SSL/TLS (Secure Socket Layer/Transport Layer Security) protocols. This encryption ensures the confidentiality and integrity of the communication, a crucial aspect in today's digital landscape.

In the context of the IoT, HTTP plays a crucial role in facilitating communication between IoT devices and servers. IoT involves a network of interconnected devices that collect and exchange data. HTTP provides a standardized and widely adopted protocol for these devices to communicate with central servers, transfer sensor data, and receive commands.

However, as IoT devices often operate in resource-constrained environments with limited power and bandwidth, there is a growing trend towards lightweight protocols like MQTT and CoAP in IoT applications. These protocols are more efficient for constrained devices, offering reduced overhead and improved scalability compared to traditional HTTP.

In summary, while HTTP is foundational for web communication, its role in IoT depends on the specific requirements and constraints of the devices and the overall architecture. For resource-intensive applications, alternatives like MQTT and CoAP may be more suitable, but HTTP remains a viable option for certain IoT use cases, especially those requiring interoperability with web-based systems [2].

HTTP has evolved through several versions, including HTTP/1.0, HTTP/1.1, HTTP/2, and HTTP/3. Each iteration brings improvements in performance, efficiency, and additional features, adapting to the dynamic needs of the ever-evolving web. As the backbone of the internet, HTTP orchestrates the seamless exchange of information, defining the user experience in browsing websites and interacting with web applications.

3.2 Device Virtualization Techniques

The Virtual Object (VO) as a digital counterpart of a physical IoT device has experienced an evolution of its functionality over the years. Since its introduction, in most of its deployments, the VO concept has been commonly intended to promote the interoperability of heterogeneous devices, facilitate the deployment of new services, improve reachability, and achieve self-management of devices [3]. Several other features further enhanced the VO as part of a variety of fit-for-purpose introduced management frameworks: common semantic representation of the device's data and functionalities for enhanced

interoperability, device augmentation with compute and storage capabilities, device augmentation with context awareness and cognitive management, device offloading and energy consumption optimization, are just a few examples. As a further step, a more effective collaboration of several physical devices is enabled in the virtual world by the introduction of the Composite Virtual Object (cVO) as an aggregation of trusted VOs illustrating a new set of functions out of the interaction of several member devices through their virtual counterparts. Deployments presented so far do not always consider a single corresponding VO to a physical device but have also introduced solutions, adapted to their reference scenario, where a single VO may correspond to multiple physical devices, each of them performing distinct functions/services, or multiple VOs correspond to a single physical device [4]. Furthermore, the combination of several VOs and cVOs along with other services, results into a new higher level of IoT services and applications, while their orchestration and execution in the cloud and/or edge have triggered the introduction of several methods and frameworks often targeted to a specific application area [5].
The concept of Digital twin (DT) also bases its definition on the mapping of a physical object onto a virtual space and builds upon it to illustrate a synchronous bidirectional data exchange to monitor, simulate, predict, diagnose, and control the state and behaviour of the physical object within the virtual space [6]. One could think of a DT as a VO or cVO with a set of advanced features and tight synchronicity and state matching with the physical object. It is believed that an open VO design establishing it as a potential building block of a DT or even future cyber-physical systems would promote significant advances in the area. Emergent applications developed in a cloud-native/microservices-based fashion, as service chains of scalable components-microservices, leverage a hyper-distributed execution of their interconnected components over a computing continuum of orchestrated resources in different network domains (IoT, edge, cloud) [7]. Considering these new requirements and potential, the VO design should be revisited to set the VO as a facilitator for: (i) a unified devices management, overcoming interoperability issues; (ii) the development of computing continuum native IoT applications where convergence aspects with edge and cloud computing technologies are tackled; and (iii) the development of new cyber-physical paradigms and new IoT-driven business models.

3.3 Device Interoperability and Management

3.3.1 NGSI-LD

The NGSI-LD (Next Generation Service Interface - Linked Data) standard[5] along with its associated API (Application Programming Interface) and broker, represents a framework for managing and exchanging information in a Linked Data format, particularly in the context of the IoT and smart applications. The NGSI-LD standard is defined by the ETSI CIM working group for Context Information Management (CIM). The standard includes NGSI-LD API used for the data interoperability in IoT, edge and cloud ecosystem to consume and update context information.
The concept of Linked Data based upon JSON-LD payloads is fundamental to effective data exchanges. The NGSI-LD standard sets rules and conventions for how entities and their context information should be structured and represented using linked data principles. This standardisation ensures that data can be easily understood and processed by IoT-edge-cloud entities, making it highly interoperable.
The NGSI-LD specification is regularly updated by ETSI. The latest specification is version 1.7.1 which was published in June 2023. Currently, several implementations of NGSI-LD context brokers are being performed such as Orion-LD, Scorpio, Stellio.
The NGSI-LD API and data-model provide a basis for unified interaction to existing IoT devices, edge nodes and cloud platforms. We plan to make use of available open-source implementations of this standard to enable the registration, discoverability, application management and data interaction using this specification. Below, NGSI-LD API and data models are described.

[5] https://ngsi-ld-tutorials.readthedocs.io/en/latest/index.html

3.3.1.1 NGSI-LD API and Broker

The NGSI-LD API defines how IoT-edge-cloud applications can communicate and exchange information using the standard of context information. It specifies the endpoints, methods, and data formats for retrieving and updating context information about IoT-edge-cloud entities. The NGSI-LD API enables semantic interactions, meaning that applications can make requests and receive data in a format that carries semantic meaning. This allows applications to understand the context and relationships between IoT-edge-cloud entities and their properties. In NGSI-LD, context information refers to the data about entities and their properties. This information can include real-time sensor readings, metadata, and other relevant details.

In particular, the NGSI-LD API can be implemented by means of Context Brokers such as Orion-LD[6] [7]. Orion-LD is a context broker developed by FIWARE as an open-source framework that supports the development of smart solutions. This context broker can run independently without requiring additional or extra components, being lightweight and efficient to manage the data exchange. The Orion-LD implements the NGSI-LD API including creation and servicing of contexts that are necessary when inline contexts are used. Context information provides entity types, attribute names, and attribute values (if applicable). The real name of an attribute (or entity type) is the expanded name, and that is what is stored in the NGSI-LD broker. Attribute values (only string values or string values inside arrays) are implemented if the context says that they should be[8].

3.3.1.2 Entities/Things in NGSI-LD API

An entity represents an object/thing that exists in the real world. In the creation of the entity, the endpoint */ngsi-ld/v1/entities* is accessed, and it is important to define the *fiware-service* header.

```
curl --location --request POST 'NEPHELE.odins.es:1026/ngsi-ld/v1/entities' \
--header 'Content-Type: application/json' \
--header 'fiware-service: test' \
--data-raw '{
  "@context": "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld",
  "id": "urn:ngsi-ld:Device:Room005Temperature",
  "type": "Room",
  "temperature": {
      "value": 21,
      "type": "Property"
   },
   "pressure": {
      "value": 803,
      "type": "Property"
   }
 }'
```

In the example above, the fiware-service header has been set to a test value. It serves as a name-space differentiator that allows several applications in a single server. Next, the data-raw field specifies the body of the query in JSON-LD format.

Inside the body of the request, there is the ID of the entity that is going to be created, the type of the entity and at least one variable, which represents the different real measures that the object stores.

All entities will be stored in the Context Broker. Once the entity is created, it can be checked from the Context Broker by using the *ngsi-ld/v1/entities* endpoint and asking for retrieving an object with a specific ID like in the following example:

```
curl --location --request GET 'http://NEPHELE.odins.es:1026/ngsi-ld/v1/entities?id= urn:ngsi-
```

[6] https://github.com/FIWARE/context.Orion-LD

[7] https://github.com/FIWARE/tutorials.Linked-Data

[8] https://github.com/FIWARE/context.Orion-LD/blob/develop/doc/manuals-ld/progress.md

```
ld:Device:Room005Temperature' \
--header 'fiware-service: test'
```

Please note that the entity type, as well as other parameters such as the context are returned in the body of the answer. When querying entities, it is also possible to get all the entities that have a specific type:

```
curl -G -X GET                                          \
        "http://NEPHELE.odins.es/ngsi-ld/v1/entities"                   \
        -H 'Accept: application/ld+json'                        \
        -H "fiware-service: NEPHELE"                            \
        -H "fiware-servicepath: /"                              \
        -d 'type=https://uri.fiware.org/ns/data-models%23DeviceMeasurement'
```

To update entities, it is discouraged to employ the *entityOperations* feature offered by the Orion-LD due to the loss of control about what is going to be updated — i.e., it is error-prone and may change entities without the user noticing. To avoid this, it is encouraged that the update of an attribute in an entity would be done in the following way:

```
curl -iX PATCH                                  \
        "http://NEPHELE.odins.es/ngsi-ld/v1/entities/urn:ngsi-ld:DeviceMeasurement:
Room005/attrs/numValue"         \
        -H "fiware-service: NEPHELE"                    \
        -H "fiware-servicepath: /"                      \
        -H 'Content-Type: application/json'               \
        -d "{\"value\":\"23.5\",\"observedAt\":\"2023-01-02T12:34:56Z\"}"
```

3.3.1.3 Smart Data Models

In NGSI-LD, the common representation of the raw-data captured from different IoT-edge-cloud entities is enabled with Smart Data Models[9] [10] [11] [12] which is an initiative by a consortium of relevant organizations in the IoT sector[13] [14] [15] [16] that aims at encouraging the interoperability of applications and services in several Smart verticals — e.g., Smart Cities, Smart Building, Smart Energy, Smart Agriculture, Smart Health. The motivation to employ these SmartData models is that they are compatible with the openly standardized NGSI-LD API. They include the technical representation of the model (schema) and the human-readable document (specification). These models are free and open-source software that does not require the payment of royalties. For example, these models define the Device Data Model that can be also found[17]. The following table shows a sample of Device Model with NGSI-LD API.

```
{
   "id": "urn:ngsi-ld:DeviceModel:myDevice-wastecontainer-sensor-345",
   "type": "DeviceModel",
   "category": {
      "type": "Property",
      "value": ["sensor"]
   },
```

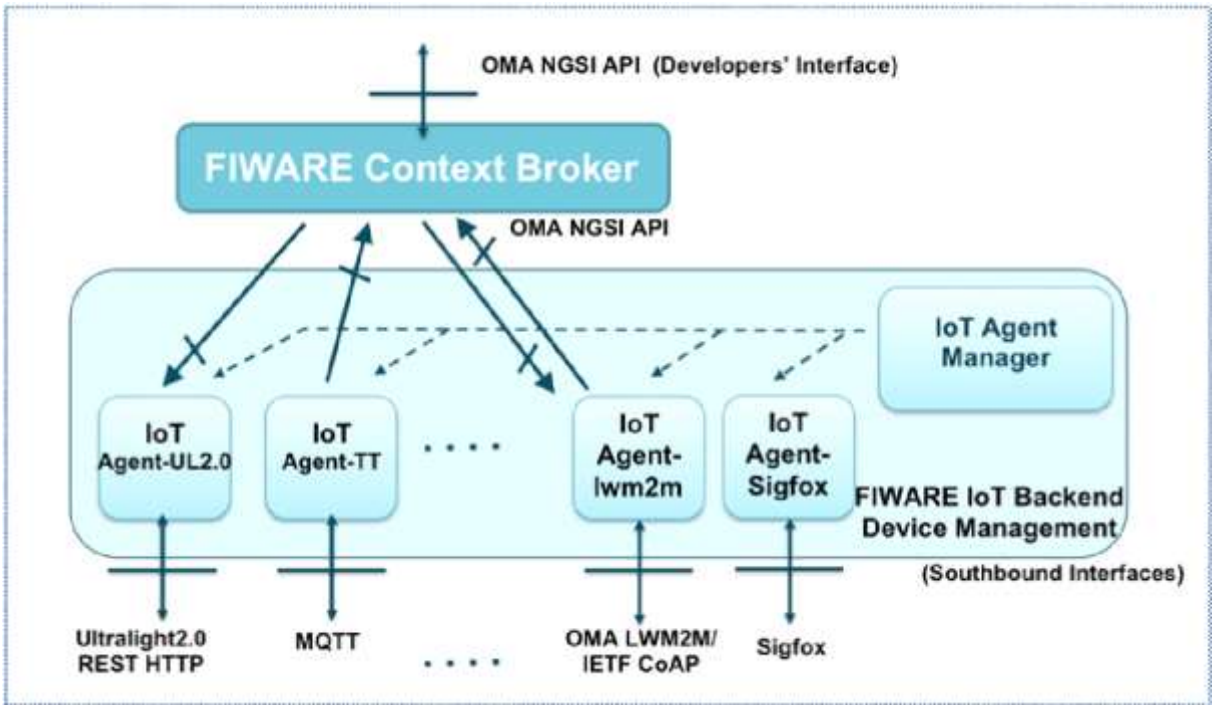
[9] https://smartdatamodels.org/
[10] https://github.com/smart-data-models/dataModel.Building/blob/master/Building/doc/spec.md
[11] https://github.com/smart-data-models/dataModel.Device/blob/master/Device/doc/spec.md
[12] https://github.com/smart-data-models/dataModel.Device/blob/master/DeviceMeasurement/doc/spec.md
[13] https://www.fiware.org/
[14] https://www.tmforum.org/
[15] https://iudx.org.in/
[16] https://oascities.org/
[17] https://github.com/smart-data-models/dataModel.Device

```
  "function": {
    "type": "Property",
    "value": ["sensing"]
  },
  "modelName": {
    "type": "Property",
    "value": "S4Container 345"
  },
  "name": {
    "type": "Property",
    "value": "myDevice Sensor for Containers 345"
  },
  "brandName": {
    "type": "Property",
    "value": "myDevice"
  },
  "manufacturerName": {
    "type": "Property",
    "value": "myDevice Inc."
  },
  "controlledProperty": {
    "type": "Property",
    "value": ["fillingLevel", "temperature"]
  },
  "@context": [
    "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld",
    "https://schema.lab.fiware.org/ld/context"
  ]
}
```

3.3.1.4 Interoperability of NGSI with other protocols

The interoperability between IoT devices and NGSI ecosystem contains two major software components:

- NGSI: It is the primary point of access for developers utilizing the NGSI Interface. An IoT device's current context can be retrieved by developers as a set of entity attributes. If they have access privileges for certain activities, developers can also update attributes connected to commands to send commands to devices.
- These components manage northbound communications from the device-specific protocol into NGSI instructions and southbound communications from the NGSI Context Broker to IoT devices. IoT integrators may now connect devices, send commands, and receive measurements thanks to this.



Any IoT standard or proprietary protocol (e.g., COAP, MQTT, Ultralight, Long Range Wide Area Network (LoRaWAN[18]) can be connected to FIWARE via the IoT Agent components. Currently FIWARE provides IoT Agents for:

- IoT Agent for JSON[19] - a bridge between HTTP/MQTT messaging (with a JSON payload) and NGSI
- IoT Agent for LwM2M[20] - a bridge between the Lightweight M2M protocol and NGSI
- IoT Agent for Ultralight[21] - a bridge between HTTP/MQTT messaging (with an UltraLight2.0 payload) and NGSI
- IoT Agent for LoRaWAN[22] - a bridge between the LoRaWAN protocol and NGSI
- IoT Agent for OPC-UA[23] - a bridge between the OPC Unified Architecture protocol and NGSI
- IoT Agent for Sigfox[24] - a bridge between the Sigfox protocol and NGSI
- The OpenMTC[25] Incubated Generic Enabler brings an open-source implementation of the OneM2M standard. A northbound interface with the NGSI Context Broker is implemented as part of the product.
- There is also an IoT Agent library[26] for developing your own IoT Agent to cover any other possible IoT Standard not covered by the existing enablers.

An open-source implementation of the OneM2M is provided by the OpenMTC Incubated Generic Enabler. Included in the offering is a northbound interface with NGSI Context Broker. To create your own IoT Agent to address any other IoT Standard that may not be supported by the current enablers, there is also an IoT Agent library.

[18] https://www.thethingsnetwork.org/docs/lorawan/
[19] https://github.com/telefonicaid/iotagent-json
[20] https://github.com/telefonicaid/lightweightm2m-iotagent
[21] https://github.com/telefonicaid/iotagent-ul
[22] https://github.com/Atos-Research-and-Innovation/IoTagent-LoRaWAN
[23] https://github.com/Engineering-Research-and-Development/iotagent-opcua
[24] https://github.com/telefonicaid/sigfox-iotagent
[25] https://github.com/OpenMTC/OpenMTC
[26] https://github.com/telefonicaid/iotagent-node-lib/

3.3.1.5 NGSI-LD API and W3C Web of Things Device Representation

NGSI-LD context management supports many ontologies like W3C SSN, SAREF, etc. In particular, the NGSI-LD device model appears complementary to the W3C Web of Things [12] device representation. It is planned to find a way to incorporate the Web of Things device representation into the NGSI-LD API specification. Furthermore, the Thing Description standard [13] adds the possibility to describe the device protocol description, which we plan to use for device on-boarding/registration of devices into the unified NGSI-LD layer. The combination of NGSI-LD and WoT device representation will be possible because both standards are based on JSON-LD format. Recently, some relevant works [8][9][10][11] have studied how to achieve the interoperability between W3C Web of Things and NGSI-LD Management. More details about WoT device representation are included in the next subsection.

3.3.2 W3C Web of Things

There has been a growing emphasis on addressing the issues related to the lack of IoT device interoperability and the presence of incompatible data models through the adoption of open standards. The integration of IoT and the World Wide Web (WWW), called Web of Things (WoT) [12] proposes to integrate the existing Web ecosystem with Things to provide an interoperable infrastructure which goes beyond basic network connectivity. Similar to how the Web functions on top of the Internet's application layer, enabling users an easy and secure means to interact with web resources through web browsers, the World Wide Web Consortium (W3C) endeavours to establish a similar synergy between the IoT and WoT.

The TD [13] is the main building block of the WoT standard, which is a model for describing the capabilities of Things and network interfaces like CoAP, Modbus, or MQTT to consumers. In analogy to how web browsers commonly access websites by using an *index.html* file on a web server as an entry point, in WoT the TD also serves as the primary entry to a Thing. The TD was first published as a W3C Recommendation standard in 2020 and recently its version 1.1 has been published with improvements to the standard. The TD describes Things capabilities in terms of their human-readable general metadata, *interaction affordances*, communication-related metadata (referred to as Protocol Bindings) for accessing the interaction affordances, security definitions, and Web links. According to the RFC, an affordance refers to the "*perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used*" [13]. The interaction affordances defined by W3C WoT are *properties*, *actions*, and *events*, which offer a model for consumers to interact with Things through abstract operations rather than specific protocols or data encodings. The protocol bindings, on the other hand, provides details required for accessing each interaction affordance on the network with a particular protocol. A single Thing can expose each interaction affordance with various protocols and is not restricted to one. The security definitions encompass the mechanisms deployed to govern secure access to a Thing and its interaction affordances. Finally, the Web links provide a hypermedia control scheme that links the Thing with other Things, documents, or representations.

The TD is a JSON-LD [14] based representation, which provides knowledge about Things in a machine-readable representation. The TD context currently includes the following standard vocabularies: *TD core*, *data scheme*, *WoT security*, and *hypermedia controls*. Semantic interoperability is achieved by extending the TD with JSON-LD-based context, allowing the incorporation of domain-specific semantic models. These models can enrich TD instances by using domain-specific vocabularies for additional Protocol Bindings or introducing new security schemes. Furthermore, the TD specification provides a JSON Scheme definition that can be given as input to JSON Schema validators to validate whether a TD instance corresponds with the TD specification.

Figure 3-4 shows an example of a TD instance for a smart lamp with HTTP bindings for reference. The TD includes the most essential elements required for describing a Thing, including the JSON-LD @context, human-readable metadata (*title* and *ID*), security definitions, as well as interaction affordances comprising the property *status*, action *toggle*, and the event *overheating* coupled with protocol bindings. In this example, all three affordances employ the HTTP protocol for accessing the interaction affordances and are structured as members within the *form's* element. The *@type* vocabulary term within the status and toggle is adopted from the JSON-LD working group and can be employed to specify a range of diverse data types, primarily inspired by JSON data types. Additional vocabulary

terms can be used to impose further restrictions on the valid data, such as specifying the minimum or maximum numeric values.

```
{
    "@context": "https://www.w3.org/2022/wot/td/v1.1",
    "id": "urn:uuid:0804d572-cce8-422a-bb7c-4412fcd56f06",
    "title": "MyLampThing",
    "securityDefinitions": {
        "basic_sc": {"scheme": "basic", "in": "header"}
    },
    "security": "basic_sc",
    "properties": {
        "status": {
            "type": "string",
            "forms": [{"href": "https://mylamp.example.com
              /status"}]
        }
    },
    "actions": {
        "toggle": {
            "forms": [{"href": "https://mylamp.example.com
              /toggle"}]
        }
    },
    "events": {
        "overheating": {
            "data": {"type": "string"},
            "forms": [{
                "href": "https://mylamp.example.com/oh",
                "subprotocol": "longpoll"
            }]
        }
    }
}
```

Figure 3-4: TD Instance for a Sample Lamp

The TD version 1.1 specification [13] defines a reusable model for representing Thing *class* definitions, called Thing Model (TM), that can be used for generating TD *instances*. As an analogy to the concept of abstract classes or interface definitions in object-oriented programming, the abstract class or interface serves as a blueprint (TM) for creating instances or objects (TDs). One of the primary objectives of a TM is to address situations in specific application scenarios such as mass production of IoT devices, or where a fully comprehensive TD is either unnecessary or impractical to provide. TMs are considered a superset for TD's allowing the omission of instance-specific information such as security schemes and partial protocol bindings. Instead, TMs incorporate placeholders for metadata such as *title*, *id*, *baseURI*, and so on. The specification also provides a process for deriving valid TDs from the corresponding TMs. During the transformation process, these placeholders are subsequently substituted with the correct values.

Figure 3-5 presents the equivalent TM for the TD illustrated above. TM definitions are distinguished from a TD by their *"@type": tm:ThingModel"*. As illustrated below, they do not include details about instance information, such as protocol bindings and security metadata. Additionally, a placeholder (indicated by a pair of double curly braces) is employed for the ID metadata. These placeholders are intended to be substituted during the conversion process to enable the utilisation of any data type as a replacement value within a predefined *placeholder map*.

```
{
    "@context": ["https://www.w3.org/2022/wot/td/v1.1"],
    "@type": "tm:ThingModel",
    "title": "Lamp Thing Model",
    "id": "urn:dev:{{ IDENTIFIER }}",
    "properties": {
        "status": {
            "description": "current status of the lamp (on|off)",
            "type": "string",
            "readOnly": true
        }
    },
    "actions": {
        "toggle": {
            "description": "Turn the lamp on or off"
        }
    },
    "events": {
        "overheating": {
            "description": "Lamp reaches a critical temperature
              (overheating)",
            "data": {"type": "string"}
        }
    }
}
```

Figure 3-5: Thing Model for the TD of the Lamp

TMs are equipped with a specialised vocabulary that facilitates to enhance the capabilities of an existing TM through the utilisation of the *"rel":* "*tm:extends"* relation type in the link section. This way, the TM inherits all the definitions from the extended part. Additionally, TMs allow importing only specific pieces of definitions from existing TMs rather than a complete set by leveraging JSON pointers and the keyword "*tm:ref"*. This specifies the location of an existing sub-definition for reuse. The TM to TD conversion process also provides mechanisms for resolving the references and extensions. The specification also defines techniques for nesting TM and TD by employing the "*tm:submodel"* link, which also allows for the reuse of the same TM for a different TD instance. In this case, the relation-type item is intended to demonstrate the presence of sub-links within a TD. The WoT TD specification also outlines a procedure for generating TDs from TMs, resolving all references and extensions, effectively creating instances of them.

TM *composition* is a term used in the W3C WoT specification to use existing TM definitions and integrate them into a new IoT system. For instance, one might design a new smart ventilator by combining two sub-TM definitions: a ventilation TM offering *on/off* and *adjust room temperature* functionalities, and an LED TM offering *dimmable* and RGB features, as illustrated in Figure 3-6. This can be achieved using the links container with *"rel": "tm:submodel"* keyword specifying the child TMs. An *instanceName* is optional and assigns a name to the composed TM sub-childs.

```
{
    "@context": "https://www.w3.org/2022/wot/td/v1.1",
    "@type": "tm:ThingModel",
    "title": "Smart Ventilator Thing Model",
    "version" : { "model": "1.0.0" },
    "links": [
      {
        "rel": "tm:submodel",
        "href": "./Ventilation.tm.jsonld",
        "type": "application/tm+json",
        "instanceName": "ventilation"
      },
      {
        "rel": "tm:submodel",
        "href": "./LED.tm.jsonld",
        "type": "application/tm+json",
        "instanceName": "led"
      }
    ],
    "properties" : {
        "status" : {"type": "string", "enum": ["on_value",
          "off_value", "error_value"]}
    }
}
```

Figure 3-6: Thing Model for a dimmable lamp

Another important building block of the WoT architecture is a *Servient* which represents a software stack responsible for implementing the core WoT elements. A WoT Servient has the capability to both host and expose Things, as well as consume Things. Depending on the specific protocol binding in use, Servients can perform in either a server or a client role.
In NEPHELE, the W3C WoT, specifically the TD and TM concepts are used to describe the capabilities of the VO and their functionalities to enable communication between devices and the edge layer, unifying the interaction and representation with and of devices. We describe the data layer as well as the application management layer using this description language.

3.3.3 OMA-LwM2M

OMA LwM2M [15] is a device management protocol designed for sensor networks and machine-to-machine (M2M) environment. LwM2M standard continues the work of the OMA towards developing a common standard for managing constrained and heterogeneous devices on a variety of networks necessary to accomplish the potential of IoT environment. The protocol is designed not only for remote device management, but it allows the enablement of related service too. The standard is built on REST architecture using CoAP protocol and it defines an extendible and scalable resource data model, and it has also begun to integrate the MQTT protocol recently. The data model is used to define a LwM2M client device as a composition of Resources organized in Objects. An Object can contain an infinite number of Resources and each Resource is identified by its URI.
LwM2M defines the application layer communication protocol between a Server and a Client, which is located in a LwM2M Device. Four interfaces are designed for the communications between the LwM2M Server and the LwM2M Client:

- Bootstrap
- Client Registration
- Device management and service enablement

- Information Reporting

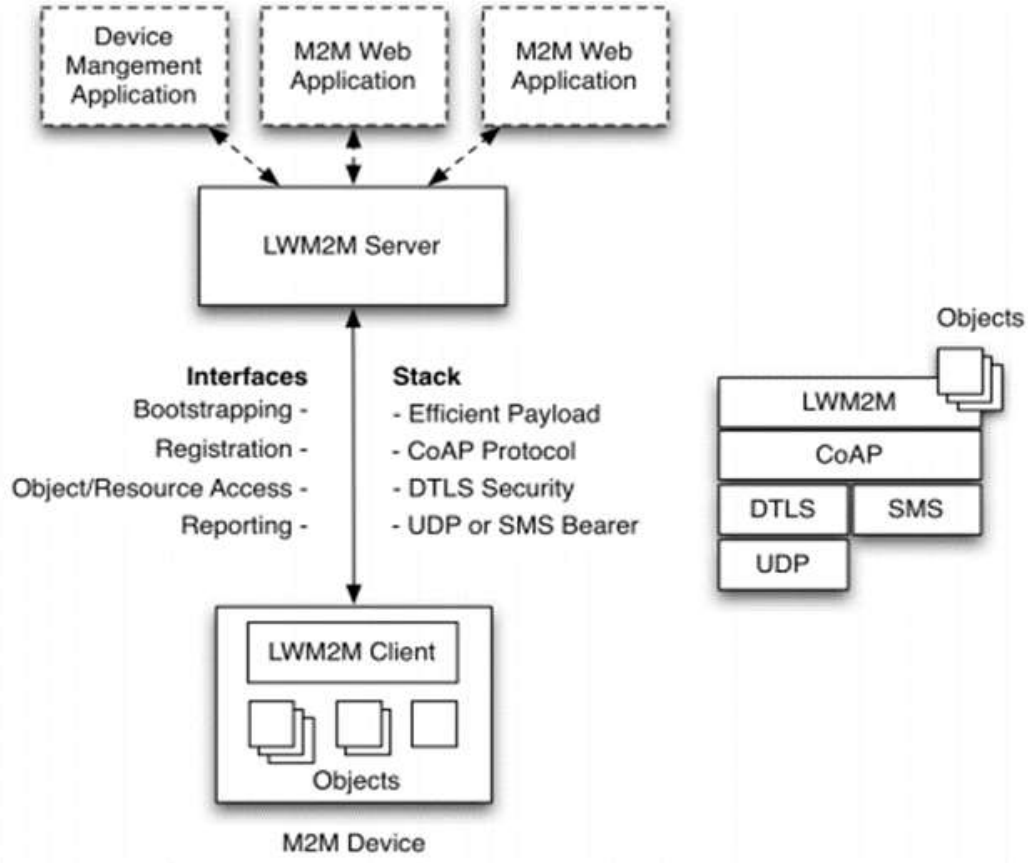


Figure 3-7: The overall architecture of the LwM2M Enabler [5]

Following, we describe the important components.

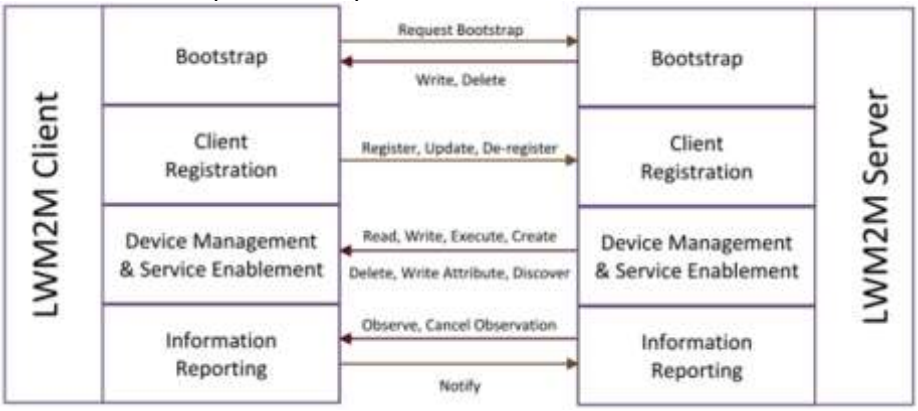


Figure 3-8: LwM2M Server Client interactions

Bootstrap - The interface is used at bootstrapping when the device wakes up for the first time and needs to initialize its Object(s) for the LwM2M client to register with one or more Server. A dedicated and separated LwM2M Server is used for this specific interface.

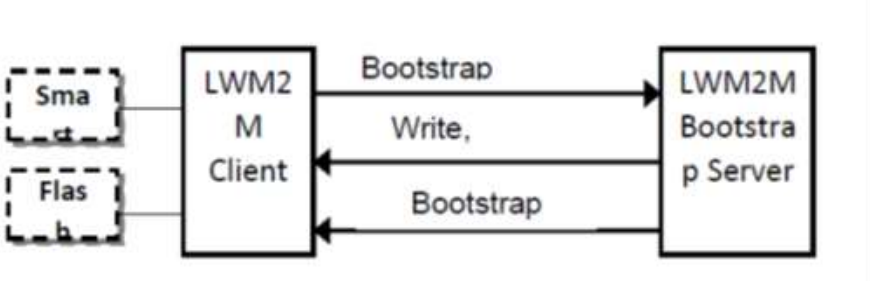


Figure 3-9: LwM2M bootstrap

Registration - This interface manages the Registration, Update (Keep alive like), and De-Registration of a Client to a Server. Towards this interface, the Client send to the server information about the object it contains and how they can be reachable.

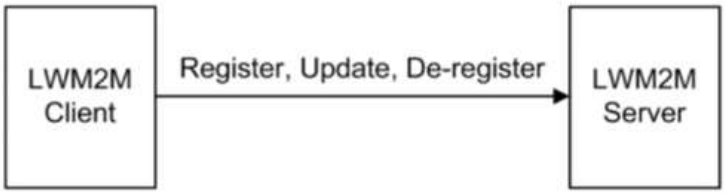


Figure 3-10: LwM2M Registration.

Device Management and Service Enablement - For this interface, the operations are downlink operations named "Read", "Create", "Delete", "Write", "Execute", "Write Attributes", and "Discover". These operations are used to interact with the Resources, Resource Instances, Objects, Object Instances and/or their attributes exposed by the LwM2M Client. The "Read" operation is used to read the current values; the "Discover" operation is used to discover attributes and to discover which Resources are implemented in a certain Object; the "Write" operation is used to update the values; the "Write Attributes" operation is used to change attribute values and the "Execute" operation is used to initiate an action. The "Create" and "Delete" operations are used to create or delete Instances.

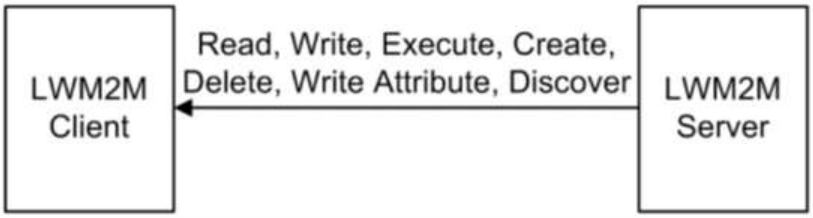


Figure 3-11: LwM2M operations

Information Reporting - This interface provides both uplinks, Notify, and downlink operations like Observe or Cancel Observation. This interface is used to send the LWM2M Server a new value related to a Resource on the LWM2M Client.

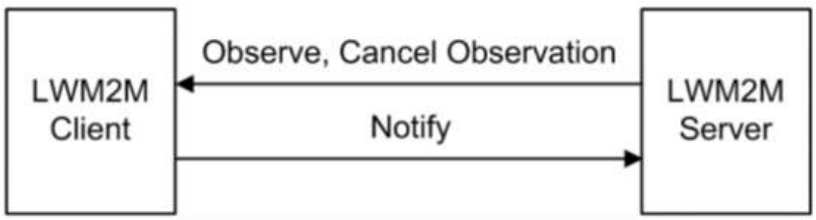


Figure 3-12: LwM2M information reporting

Table 3.1 lists the relationship between Operations and Interfaces.

Table 3.1 Operations and Interfaces relationship.

Interface	Direction	Operation
Bootstrap	Uplink	Request Bootstrap
Bootstrap	Downlink	Write, Delete
Client Registration	Uplink	Register, Update, De-register
Device Management and Service Enablement	Downlink	Create, Read, Write, Delete, Execute, Write Attributes, Discover
Information Reporting	Downlink	Observe, Cancel Observation
Information Reporting	Uplink	Notify

LwM2M operations for each interface are mapped via CoAP methods. In particular, each operation, except Notify, is encapsulated in a Confirmable message (CON) CoAP type and the ACK is used to provide the payload response too. Notify, on the other hand, can be both Confirmable and Non-Confirmable (NON).

3.3.3.1 Resource model

In the OMA-LwM2M proposed model, the basic information that an LwM2M client transmits is a Resource data, while Objects are composed of a set of Resources. An object is used to describe and control a specific software/hardware component of the device (such as sensors, antennas, or device firmware) with associated resources (e.g., value, unit, max value, min value). Depending on the characteristics of the Object, we may have one or more instances of the same object on the device. For example, that we have two temperature sensors (internal sensor and external sensor), then we will have two instances of temperature object that describe the device. The two Objects will be distinguished within the URI by the Object instance level. The figure below represents an example of the resource model used in OMA-LwM2M.

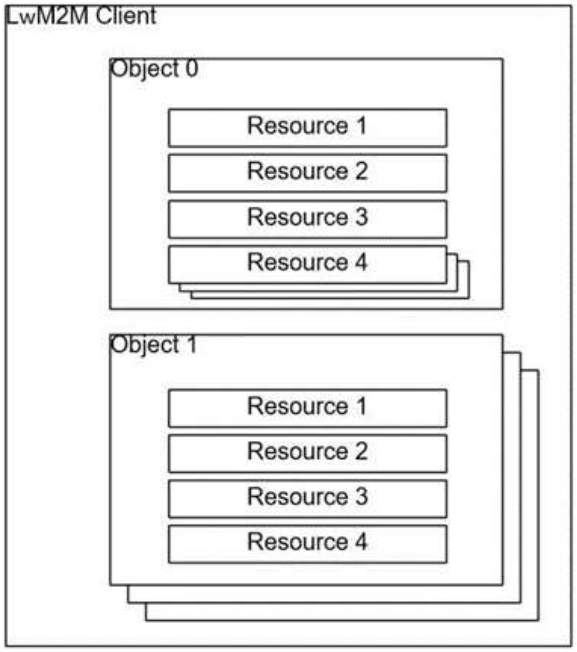


Figure 3-13: LwM2M resource model

The CoAP URI path is defined by objectID/InstanceObjectID/ResourceID. Following the standard, the Object Temperature sensor is defined by ID 3303 and the value of its sensor, resource sensor value, is defined by id 5700. Using this ID is possible to build the URI, in this case 3303/0/5700. A second sensor of temperature in the same device can be identified using a different InstanceObjectID, the URI will be 3303/1/5700. Object and their resources are defined using meta-model declared and shared between client and server. An object can be defined using an XML file like described in the following figure.

```xml
<?xml version="1.0" encoding="utf-8"?>
<LWM2M xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://openmobilealliance.org/tech/pr
    <Object ObjectType="MODefinition">
        <Name>Temperature</Name>
        <Description1>Description: This IPSO object should be used with a temperature sensor to report a temperature measurement.
        <ObjectID>3303</ObjectID>
        <ObjectURN>urn:oma:lwm2m:ext:3303</ObjectURN>
        <MultipleInstances>Multiple</MultipleInstances>
        <Mandatory>Optional</Mandatory>
        <Resources>
            <Item ID="5700">
                <Name>Sensor Value</Name>
                <Operations>R</Operations>
                <MultipleInstances>Single</MultipleInstances>
                <Mandatory>Mandatory</Mandatory>
                <Type>Float</Type>
                <RangeEnumeration></RangeEnumeration>
                <Units>Defined by "Units" resource.</Units>
                <Description>Last or Current Measured Value from the Sensor</Description>
            </Item>
            <Item ID="5601">
                <Name>Min Measured Value</Name>
                <Operations>R</Operations>
                <MultipleInstances>Single</MultipleInstances>
                <Mandatory>Optional</Mandatory>
                <Type>Float</Type>
                <RangeEnumeration></RangeEnumeration>
                <Units>Defined by "Units" resource.</Units>
                <Description>The minimum value measured by the sensor since power ON or reset</Description>
            </Item>
            <Item ID="5602">
                <Name>Max Measured Value</Name>
                <Operations>R</Operations>
                <MultipleInstances>Single</MultipleInstances>
                <Mandatory>Optional</Mandatory>
                <Type>Float</Type>
                <RangeEnumeration></RangeEnumeration>
                <Units>Defined by "Units" resource.</Units>
                <Description>The maximum value measured by the sensor since power ON or reset</Description>
            </Item>
            <Item ID="5603">
                <Name>Min Range Value</Name>
                <Operations>R</Operations>
                <MultipleInstances>Single</MultipleInstances>
```

Figure 3-14: LwM2M resource model in XML

Each Object and Resource is defined to have one or more operations that it supports. The LWM2M standard support different data formats for data transmission like JSON or TLV.

The OMA LwM2M standard provides a public registry of "Standard Objects"[27] but each developer can create their own object from OMA objects and the provided resource models.

Moreover, the LwM2M defines access control mechanism per Object entity based on associated Access Control Object Instance. An Access Control Object Instances contains Access Control Lists (ACLs) that define which operations on a given Object Instance are allowed for which LWM2M Server(s). For instance, a server could be authorized to perform all operations but a different one could be authorized to perform only Read operations.

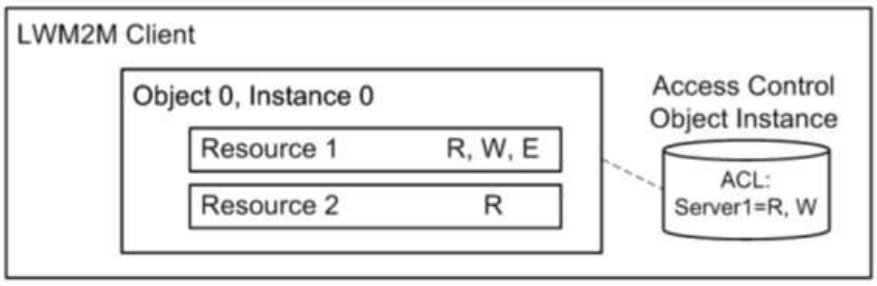


Figure 3-15: LwM2M access control object instance

[27] http://www.openmobilealliance.org/wp/OMNA/LwM2M/LwM2MRegistry.html

3.4 Networking

This section focuses on networking requirements at the IoT level and the different techniques for ad-hoc cloud networking.

3.4.1 Networking Requirements at IoT Level

In this section, we focus on the following three IoT concepts that define how end devices in the network operate and collaborate among each other: self-organising networks, mobile edge computing and multi-radio technologies.

We start by providing the definition and an explanation of these concepts in IoT. For each concept, we provide multiple references of existing works in the state-of-art. Then, we highlight their impact on time sensitive and critical IoT applications such as post-disaster management.

Self-organised networks in IoT refer to decentralised and adaptive communication infrastructures where devices autonomously organise and optimise their connectivity without centralised control [16]. In these networks, IoT devices collaborate to form dynamic and efficient connections, adapting to changes in the environment or network conditions. The self-organising nature eliminates the need for a central coordinator and allows devices to join or leave the network seamlessly. This autonomy enhances the scalability, flexibility, and resilience of IoT deployments, as devices can efficiently share information, coordinate tasks, and collectively respond to emerging challenges. Self-organised networks in IoT leverage principles such as distributed decision-making, intelligent routing algorithms, and collaborative protocols to achieve efficient and adaptive communication among a diverse set of interconnected devices. In [17], the authors propose a cross layer solution named self-organising and self-configuring algorithms for efficient data communication for IoT (2SAEC-IoT). The approach aims at ensuring the continuity of services for IoT applications, dealing with sensitive data, by tolerating failures that can occur on communications and devices. It is based on distributed algorithms and considers communication parameters related to MAC, network, and transport layers.

Mobile edge computing (MEC) in IoT refers to a paradigm where computation and data processing capabilities are brought closer to the edge of the network, typically within the proximity of IoT devices, on mobile units such as drones or robots. Unlike traditional cloud-centric models, MEC distributes computing resources to the edge of the network, enabling faster and more responsive processing of IoT data. In this context, edge servers, deployed on mobile units at the edge of the network infrastructure, manage computational tasks locally, reducing latency and bandwidth usage. MEC enhances the efficiency of IoT applications by enabling real-time data analysis, rapid decision-making, and reducing the need for extensive data transmission to centralised cloud servers. By leveraging the computational power at the edge, MEC contributes to improved scalability, reduced latency, and increased autonomy for IoT devices, making it a key enabler for applications demanding low-latency responses and efficient use of network resources. In [18], a complete overview of multi-access edge computing solutions is provided. The overview focuses on mission-critical applications, resource allocation schemes and deployment of these solutions on mobile resources. In [19], the authors provide a detailed survey on the problem of combining Edge and Cloud to manage task offloading. The survey focuses on existing works in literature that propose either optimization techniques, or artificial intelligence techniques or control theory to perform task offloading. Many aspects are taken into consideration in this study: the objective function, the granularity level, the use of Edge and/or Cloud infrastructures and the integration of mobile edge devices. For instance, in [20] the authors introduce the DRUID-NET framework. It incorporates analytic dynamical modelling of resources, workload, and networking environments in wireless communications and mobile edge computing. It introduces new estimators for time-varying profiles. The goal is to develop innovative resource allocation mechanisms that consider service differentiation and context-awareness to ensure well-defined QoS metrics. The approach combines tools from Automata and Graph theory, machine learning, modern control theory and network theory.

Multi-radio technologies in IoT involve the integration of multiple communication interfaces within IoT devices to enhance connectivity and flexibility. These devices are equipped with multiple radios, such as Wi-Fi, Bluetooth, Zigbee, and cellular technologies, allowing them to adapt to diverse communication environments and requirements [21]. The use of multi-radio capabilities in IoT facilitates seamless communication across various networks, enabling devices to dynamically switch

between different radio interfaces based on factors like data volume, energy efficiency, and network availability. This approach enhances the reliability and robustness of IoT deployments, enabling devices to optimise their communication strategies for different use cases and scenarios. By leveraging multi-radio technologies, IoT devices can efficiently interact with a variety of networks, supporting applications ranging from short-range sensor networks to long-range cellular connectivity, contributing to the overall flexibility and adaptability of the IoT ecosystem. To analyse the advantages of using multi-radio technologies, [22] presents an energy-monitoring platform to collect per-packet energy consumption, packet delivery ratio (PDR) and other parameters of LoRaWAN, NBIoT, and Sigfox. Results showed the comparative energy saving when adopting multi-radio technologies in static and mobile nodes for indoor and outdoor scenarios. [23] presents a lightweight method for selecting the optimal communication technology in a multi-radio sensor network based on the environment and the data requirements the nodes must fulfil, such as energy, delay, and cost, using a low computation time. RODEN [23] enables dynamic (re)selection of the best route and radio access technologies based on the data type and requirements that may evolve over time, potentially mixing each technology over a single path.

Integrating self-organised networks, mobile edge computing, and multi-radio technologies schemes is essential to provide solutions to mission-critical scenarios such as post-disaster management. In the aftermath of a disaster, traditional communication infrastructure may be compromised, making it challenging for first responders and emergency personnel to coordinate efforts. Self-organising networks enable IoT devices to autonomously establish communication links, facilitating swift and decentralised collaboration among rescue teams, drones, and sensor networks. Mobile edge computing brings computational capabilities closer to the disaster site, allowing real-time data analysis for informed decision-making without relying heavily on distant cloud resources. Additionally, multi-technology support equips IoT devices with diverse communication interfaces, ensuring robust connectivity even in heterogeneous and dynamic post-disaster environments. This comprehensive integration enhances the efficiency, reliability, and adaptability of post-disaster management systems, enabling rapid response, resource optimization, and effective coordination in the face of challenging conditions.

3.4.2 Ad-hoc Cloud Networking

3.4.2.1 Time-Sensitive Networking

Time-Sensitive Networking (TSN) encompasses a set of standards developed by the Time-Sensitive Networking task group within the IEEE 802.1 working group. These TSN standards specify methods for transmitting data with high time-sensitivity over Ethernet networks that are deterministic. The majority of TSN projects extend the IEEE 802.1Q – Bridges and Bridged Networks standards, which deal with Virtual Local Area Networks (VLAN) and network switches. These extensions aim at data transmission with bounded latency and high reliability. TSN mechanisms are particularly relevant to areas, such as automotive and industrial control, where real-time Audio/Video Streaming and control streams are used in converged networks.

Numerous IEEE 802.1 specifications are available, including 802.1Qbv -- Time Aware Shaper [24], IEEE 802.1 Qbu Preemption, and IEEE 802.1AS Timing and Synchronisation. These specifications provide support for various features and functionalities for network communication. For instance, 802.1Qbv is associated with the Time-Aware Shaper (TAS) mechanism for controlling latency, whereas 802.1Qbu offers the Preemption feature for interrupting and resuming frame transmission. In addition, 802.1AS focuses on timing and synchronisation within the network. These IEEE 802.1 standards are integral components of modern networking, providing essential tools for the transmission of data over networks with different performance requirements.

In the context of IEEE 802.1 standards, 802.1Qbv [24] introduces a transmission gate operation concept for each traffic class queue, as depicted in Figure 3-16. At the egress port of a TSN switch, outgoing frames go through a Traffic Classification block that categorises different streams to their respective traffic classes. This classification process is vital in preventing traffic overload, which could otherwise affect the switches. At T1, the traffic class 0, which is assigned to traffic priority 0, is open for transmission. The packets are then queued into various traffic classes based on the state of the transmission gates. These gates are either open or closed, and their status is controlled by a Gate Control

List (GCL). A GCL contains multiple schedule entries for each output port, allowing selected traffic to pass through open gates to the transmission selection block, which provides access to the medium. For a TSN switch with IEEE 802.1Qbv schedules to function as expected, the clocks of all switches should be synchronised. This would ensure that all TSN switches reference the same cycle base time in their schedules. Such synchronisation can be attained via the Precision Time Protocol (PTP). IEEE 802.1Qbv coupled with PTP provides the means for the implementation of appropriate GCL schedules and time synchronisation, which can guarantee the transmission of high-priority critical traffic without interference from other best-effort traffic. This eventually can ensure bounded delay and low jitter of scheduled traffic, as well as protection of high-priority flows.

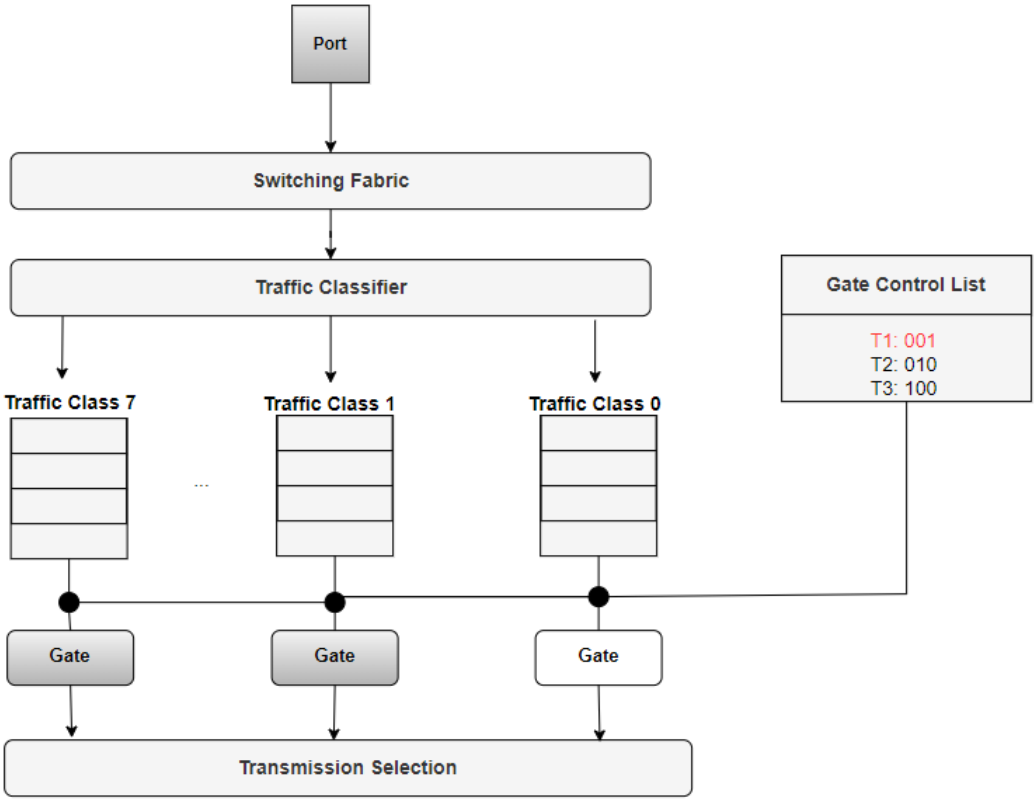


Figure 3-16: TSN bridge internals based on IEEE 802.1Qbv

3.4.2.2 Software-Defined Wireless Networking

Software-Defined Wireless Networking (SDWN) can support intelligent, programmable, and logically centralized control mechanisms that dynamically adjust protocol functionalities to achieve improved performance and resource utilization, addressing specific performance demands from IoT applications. They adopt the typical Software-Defined Networking (SDN) architecture, where the application plane is responsible to communicate application requirements, the control plane to implement control mechanisms and the data plane to communicate the data.

However, wireless IoT deployments are often affected by radio signal issues, e.g., due to mobility or interference, which can impair control communication. Furthermore, the latter may also be characterized by increased overhead. To tackle the challenges of intermittent connectivity with the controller, control message scalability, and mobility, several SDWN solutions have been proposed in the literature.

For example, they target control channel issues, such as (i) SDN-WISE [25] introducing stateful routing tables and proactive routing decisions to reduce the number of interactions with the controller; (ii) TinySDN [26] implementing a distributed control plane architecture based on multiple controllers; and (iii) Atomic-SDN [27] proposing a time-sliced mechanism that separates the SDN control from the WSN data plane messages using designated flooding periods for the control messages. Application-aware service provisioning is investigated in Soft-WSN [28], implementing topology control, device, and network management to meet run-time and application-specific requirements.

CORAL-SDN [29] and its evolution VERO-SDN [30] solutions adopt a modular controller architecture, multiple configurable topology discovery and flow establishment mechanisms and an out-of-band approach to handle control overhead, i.e., utilize a long-range low-rate control interface and a short-range high-rate data interface. Furthermore, SD-MIoT [31] augments VERO-SDN with a better support of mobile IoT nodes, through: (i) mobility-aware topology control, utilizing a hybrid of global and local topology discovery algorithms; (ii) routing mechanisms adapted to mobility employing a hybrid combination of reactive and proactive flow establishment methods; and (iii) an intelligent algorithm that detects passively in real-time the mobile network nodes.
Lastly, MINOS [32] is a multi-protocol SDN platform that implements: (i) logically centralized network control of diverse and resource-constraint IoT environments; (ii) multiple protocols that are deployable and configurable on-demand; (iii) individual protocol configurations per node; and (iv) flexibility to accommodate new protocols and control algorithms.

3.5 Device Intelligence

This section introduces the recent advancements as well as challenges in on-device intelligence in IoT. As we are using machine learning (ML) and complex event processing (CEP) for on-device intelligence in this project, we consider research work on ML and CEP.
Over the past decade, the advancement of ML applications has been propelled by the emergence of big data and enhanced computational capabilities. This has led to a surge in large-scale AI models, such as ChatGPT [33], which demands extensive resources and significant power consumption. The growing awareness in the ML community emphasises the escalating resource requirement and environmental unsustainability associated with big AI models. Tiny Machine Learning (TinyML) has emerged as a powerful paradigm that bridges the gap between ML and embedded systems. It brings real-time AI capabilities closer to the edge, shifting data processing from data centres to IoT devices. TinyML offers sustainability, data privacy, and efficiency advantages by minimising the need for cloud data transmission. The current estimate suggests that over 250 billion IoT devices are in active use today [34], with continual rising demand, particularly in the industrial sector.
TinyML is surging across research and industrial communities. Recently, a survey [35] summarised breakthroughs and challenges for promoting embedded ML. Better algorithms [36], [37] are designed to use resources, improve model performance, and optimise the deployment in a real-world environment more efficiently. Also, ultra-low-power AI chips [38] and accelerators [39] have been proposed to support always-on ML capability for an extended period by a battery. However, a joint design of hardware and algorithm [40], [41]is required to squeeze the performance since TinyML delivers ML solutions to constrained devices with limited resources. Moving a step forward, frameworks for characterising and assessing ML deployment on the edge [42], [43] help us systematically tackle potential issues. Many platforms and resources, such as open online courses [44], X-CUBE-AI [45] from STM, Apache's TVM [46], and TensorFlow Lite for Microcontrollers [47] from Google, are made available to accelerate TinyML. Besides, remarkable applications of TinyML show up across all fields, see [19] – [21].
However, traditional TinyML solutions assume models are trained in powerful machines or the cloud, and it is afterward uploaded to the edge device. The microcontroller unit (MCU) only needs to perform inference. This strategy treats models as static objects, which may cause performance degradation in real-world deployment environments due to evolving input data distributions [48]. To learn from new data, the model must be retrained from scratch and re-uploaded to MCUs, making the deployment of TinyML in the industry environment a challenging task.
Online learning can be a promising solution to this problem [49]. An ML system has two tasks: inference and training. Online ML involves performing these tasks online, i.e., processing data sequentially without revisiting previous samples, saving memory resources, and ideally not sacrificing model performance. However, less attention is paid to online learning [36], [42] compared with batch/offline learning because most ML engineers tend to assume that devices and their data are always available as a batch. A few works [50], [51] discuss how to prevent catastrophic forgetting problems of incremental online learning in NN. River is a popular Python library for online ML [49]. The neural network (NN) training at the edge is much less common than inference because of limited resources and reduced

numerical precision. According to [52], good performance can only be guaranteed when training and inference run on the same dataset. Otherwise, the accuracy of the model will decrease over time. Therefore, the author suggests retraining the network on the device regularly, in their case - Raspberry Pi. In [53], a shallow NN is implemented at the edge for inference, powered by an NVIDIA Jetson. The edge node is connected with a deep NN that resides in the server. To save energy, deep NN is only used to transfer knowledge to shallow NN when a significant drop of performance at the edge is detected. Similarly, a near real-time training concept is proposed in [54] for computer vision tasks on NVIDIA Jetson. Whenever a new object class is detected, a new NN model is initialised using the old model's weights, retrained at the fog, and be ported back to the edge. In another work [55], the checkpointing strategies are adopted to save training consumption at the edge, whereas in [56], incremental learning is applied to support the training of the k-nearest-neighbour model on Raspberry Pi. In [57], expert feedback is provided as labels to an unsupervised NN on the fly, thus turning unsupervised learning into semi-supervised learning. We believe that online learning is a perfect fit for TinyML applications. The concept helps us develop algorithms with lower memory footprints and keep models up to date in changing environments. Thus, we built a TinyOL system [58] which can fine-tune the pre-trained model directly on constrained devices using field streaming data by leveraging online learning settings.

CEP is a widely deployed technique to process events and detect patterns from multiple heterogeneous streaming sources. The user can define logic rules to describe the desired sequence and pattern of primitive events. The CEP system then runs reasoning on arriving events against the rules and produces complex (derived) events upon matching. The system can address sensor networks' ubiquity by efficiently matching input streaming events against a pattern, where irrelevant incoming data can be discarded immediately. A survey [59] discussed the early works of research, implementation, terminology, applications, and open issues in CEP. Several commercial and open-source event processing tools have been developed in recent years [60], [61]. Work [62] studied state of the art on CEP mechanisms and presented their drawbacks in heterogeneous IoT environments. A survey [59] discussed the techniques, the opportunities, and the challenges of CEP in the big data era. While most CEP tools are designed for centralised cloud analysis, works like [63] try to extend the CEP to mobile devices to leverage decentralised architectures. In [64], a by-partitioning CEP pipeline is proposed to use both edge and cloud resources for stream processing. A micro-service-based method is introduced in [65] to manage CEP in IoT systems. A few works [66], [67] focus on a hierarchical complex event model to adapt CEP systems in distributed sensing environments. Although these solutions proved to work in IoT systems, they do not work on the sensor networks composed of constrained MCUs. In their works, the IoT devices are usually Raspberry Pi-class with far more available resources. This class of devices does not fit into the scope of constrained IoT devices. Besides this, they assume the logical patterns do not need to be updated after deployment and are thus flashed to the devices prior to the runtime as static rules, which restrict the CEP engine's versatility in the context of IIoT applications. To address the issue, we designed the *Micro Complex Event Processing* µCEP [68] engine following the same language introduced in the reasoning system ETALIS [69], which can continuously match incoming events against user-defined patterns directly at the embedded devices with low latency and high throughput.

Combining ML with CEP can be a promising solution to circumvent some technical limitations in the current IoT system. In work [70], the author surveyed the synthesis of both paradigms and their transferability to intelligent factory use cases. Some researchers tried to derive CEP reasoning rules using ML methods [71]–[73]. A Bayesian network is used to predict upcoming online events in work [74]. Many CEP and ML-based solutions have been applied in the real world. A CEP and ML-based approach to support fault-tolerance of IoT systems is proposed in work [75]. In [76], a framework based on CEP and deep learning is implemented, and an unattended bag computer vision task is illustrated to evaluate its feasibility. Unlike other works, CEP is used to schedule distributed ML training on Raspberry Pi in [77]. Nevertheless, many implementation designs depend on the usage of the cloud for communication. Very few touch the area of constrained devices and evaluate their approaches under industry settings.

4 Intelligent IoT Devices Modelling, Management, and Interoperability

The following sections provide information regarding the VO descriptor which is a declarative way to define a VO based on the related semantics and modelling, i.e., (a) W3C WoT and (b) OMA-LwM2M. Following we provide analytical information regarding the interoperability and relevant solutions with W3C WoT and with OMA-LwM2M. Finally, we discuss the semantic interoperability between WoT and NGSI-LD.

The overall goal of Task 3.1 is to develop the set of features that will be supported by the VOStack layers, which includes *interoperability*, *IoT device management* and *security*. To reach this goal, this task designs and develops the semantic models required for effectively managing IoT devices and their VOs for enabling semantic interoperability in the VO. Also, we provide bi-directional translation mechanisms between two different well-known semantic models used to describe VOs for enabling semantic interoperability between them. Furthermore, the interaction with IoT devices using different communication protocols, as well as a set of basic management functionalities is enabled. This task additionally delivers security by-design mechanisms for ensuring the data protection and General Data Protection Regulation (GDPR) compliance privacy preserving in all data exchanges between constrained IoT devices and VO Edge. Finally, this task integrates digital twinning as a 3D representation for the various IoT attributes to provide an intelligible way to assert multi-states, monitoring, and analytics.

In this task, we have used the following technologies:

- W3C Web of Things technologies such as TD and TM [13] for describing VOs and their functions.
- W3C semantic web technologies such as RDF, OWL, SPARQL, SHACL shapes, ontologies, and RDF Mapping Language (RML) for semantic data translation between device descriptions provided in NGSI-LD [78]and W3C WoT TD [13].
- Different communication protocols such as HTTP [79], MQTT [80], and CoAP [81]to support the interaction with IoT devices.
- Web technologies such as REST [82] for defining the management interface of the virtual object functions.
- On-Dev-App Management with OMA LwM2M .
- Security technologies such as TLS, DLT, DID, XACML, OpenID at the VO layer.
- Internet-based open standardised protocols defined by IETF for authentication, authorization, key establishment, channel protection, integrity, confidentiality (e.g., EDHOC, OSCORE, ACE-Oauth) to guarantee the security in data exchanges between IoT constrained devices and VO Edge nodes.
- IoT security assessment and testing methodology developed in the ARMOUR H2020 project.

This task contributes to the deliverable by providing:

- A tool that can automatically create TDs based on provided TMs for devices and virtual object functions to further enhance the capabilities of the IoT devices with additional properties and exposes the device properties to the Web via a WoT Servient to enable interoperable data consumption/control by other applications.
- A REST API for virtual object function management that provides rule management, compatible IoT devices for rules, and deploy/unload rules from devices at runtime according to their properties.
- Application of the tool and API to manage rules on a Siemens thermostat that is rule-enabled and publishes sensor data via MQTT.
- A tool that can automatically provide semantic translations between NGSI-LD and W3C WoT TD.

In this section, we address the functional requirements FR_VOS_001 from Table 2.1 and non-functional requirements NFR_VOS_05 and NFR_VOS_06 from Table 2.2.

4.1 VO Descriptor

In the context of NEPHELE, we aim to utilize only the (c)VO Descriptor to configure (c)VOs. The (c)VO Descriptor is a YAML file that represents all different options and mappings for a user-defined (c)VO. The descriptor is parsed during instantiation time and manages the initialization of the runtime and the deployment and self-configuration of the Virtual Object. In Table 4 we provide information about all configurations (one at each index) of the YAML Descriptor along with the available values and the relative description of each key-value pair.

Table 4.1: Lists of all configurations of the VO Descriptor

index	Dictionary	Item or list		Available values	description
1		name		"String"	Name of the VO
2		type		[VO, cVO]	Type of the Virtual Object. Can be one of: VO/cVO
3	resourceType	specification		[WoT, omaLwM2M]	Protocol
		version		1.1	Version of VO
4		Deployment type		[A, B]	Deployment Type. Can be one of: A (the associated device is running the WoT runtime), B (not running the WoT runtime)
5		Catalogue port		9090	Catalogue port (port of an HTTP server) of the VO from where it is model can be consumed
6	bindingNB	bindingModeNB		[H, M, U]	Values: H (HTTP), M (MQTT), U(CoAP/UDP)
		hostname		"string"	Hostname for the DNS record of the VO
		ports	HttpPort	int	Port for Http server
			CoAPport	int	Port for CoAP server
		brokerIP		"mqtt://localhost:1883"	IP of MQTT Broker
		serverCert		"string"	Path to a certificate to instantiate an HTTPS server
		serverKey		"string"	Path to a private key to instantiate an HTTPS server
		mqttCAFile		"string"	Path to the certificate file of an MQTT NorthBound broker

		OSCORECredentialsMap		“string”	Path to a JSON Credentials Map for OSCORE (CoAP server)
		Security NB	securityScheme	[nosec,basic,bearer]	The security scheme that will be used.
			username	“string”	If `securityScheme` is set to basic, set the desired username-password
			password	“string”	
			token	“string”	If `securityScheme` is set to bearer, set the desired token
7	bindingSB	bindingModeSB		[H, M, U]	Values: H (HTTP), M (MQTT), U(CoAP/UDP)
		mqttCAFile		“string”	Path to the certificate file of an MQTT NorthBound broker
		OSCORECredentialsMap		“string”	Path to a JSON Credentials Map for OSCORE (CoAP server)
		Security SB	securityScheme	[nosec,basic,bearer]	The security scheme that will be used.
			username	“string”	If `securityScheme` is set to basic, set the desired username-password
			password	“string”	
			token	“string”	If `securityScheme` is set to bearer, set the desired token
8	databaseConfig	timeseriesDB	influxDB	[“enabled”,”disabled”]	Flag that enables/ disables the InfluxDB connection. Can be one of: enabled, disabled
			address	"http://wotpy-influxdb-cvo:8086"	URL of where the InfluxDB is deployed
			dbToken	“string”	Token used from the VO to access the database
		persistentDB	SQLite	[“enabled”,”disabled”]	Flag that enables/ disables the SQLite database. Can be one of: enabled,disabled
			dbFilePath	“string”	Optional field that sets

					the path to save the database
9	genericFunction			[forecasting, mean_value, vo_status, device_status]	List of generic functions that will be made available to the user-defined scripts.
10	periodicFunction	example_function		1 sec	User-defined functions that will be executed periodically Is a map of function names and their periodicity in seconds
11	consumedVOs	example_VO_name	url	"http://vo1:9090/vo_name"	Catalogue endpoint of the Virtual Object to be consumed in case of a cVO
			events	[“string”]	List of events whose subscriptions need to be mapped to user-defined functions
			propertyChanges	[“string”]	List of properties whose subscriptions to changes need to be mapped to user-defined functions

In the following sections we discuss the interoperability and relevant solutions of different semantic models. Also, the main goal of the VO is to provide a proxy service of the devices towards the application layer while also storing the data in databases. Specifically, the enhanced functionalities of the VO and all device management can be accessed through the VO.

4.2 Interoperability and Relevant Solutions with W3C WoT

The VO is described semantically according to the W3C Things Model. As explained in section 3.3.2, the W3C Thing Model is a TD that does not contain Thing instance-specific information, such as concrete protocol usage, or name. In the following listing, a TM describing a Siemens *thermostat* device is shown in Figure 4-17, providing common metadata, and describing the interaction affordances such as Properties, Actions and Events. As can be seen, the ID of the device contains a placeholder called *THERMOSTAT_NUMBER* which will be instantiated during the TD creation. Also, we use the @type metadata of the TM to annotate the device with additional semantic vocabularies: the *brick* ontology to specify the device is a thermostat and the µ*CEP* vocabulary [69] to indicate its support for lightweight CEP processing. µ*CEP* embedded devices are equipped with a lightweight engine that enables them to effectively process raw data streams, infer complex events using *rules*, and derive *intelligent insights* from the data [35], [36].

```
{
  "@context": [
    "https://www.w3.org/2022/wot/td/v1.1",
    {
      "brick": "https://brickschema.org/schema/1.0.3/BrickFrame#",
      "btas": "http://bt.schema.siemens.io/shared/btas#",
      "btzf": "http://bt.schema.siemens.io/shared/btzf#",
      "mcep": "http://mcep/shared",
      "unit": "http://qudt.org/vocab/unit/"
    }
  ],
  "id": "urn:node:knx:ThreadRoomSensor:{{THERMOSTAT_NUMBER}}",
  "title": "Thread based KNX room sensor No.{{THERMOSTAT_NUMBER}}",
  "btas:model": "QFA2890/WI",
  "version": {
    "model": "1.0.0"
  },
  "btas:manufacturingYear": "2023",
  "@type": [
    "tm:ThingModel",
    "brick:Thermostat",
    "mcep:Device"
  ],
  "base": "{{MQTT_BROKER_ADDRESS}}:{{MQTT_BROKER_PORT}}",
  "description": "Wireless room sensor for temperature & humidity",
  "securityDefinitions": {
    "basic_sc": {
      "scheme": "basic"
    }
  },
  "security": "basic_sc",
  "properties": {
    "deviceLightStatus": {
      "title": "Reads the status of the device (e.g. the status is
        'red' when the device has connection loss or low battery)",
      "type": "string",
      "enum": ["1", "2", "3", "4"],
      "btzf:hasEnumMap": {
        "btzf:hasMapMember": {
          "1": {
            "ref": "off"
          },
          "2": {
            "ref": "connected"
          },
          "3": {
            "ref": "noConnection"
          },
          "4": {
            "ref": "lowBattery"
```

```
            }
          }
        },
        "writeOnly": false,
        "observable": true,
        "forms": [
          {
            "href": "/ThreadRoomSensor/{{THERMOSTAT_NUMBER}}/lightStatus",
            "op": ["readproperty", "observeproperty"]
          }
        ]
      },
      "temperature": {
        "description": "temperature measurement",
        "@type": "brick:Temperature_Sensor",
        "type": "number",
        "minimum": 0.0,
        "maximum": 50.0,
        "unit": "unit:DEG_C",
        "readOnly": true,
        "observable": true,
        "forms": [
          {
            "op": ["readproperty", "observeproperty"],
            "href": "/ThreadRoomSensor/{{THERMOSTAT_NUMBER}}/temperature"
          }
        ]
      }
    }
  }
}
```

Figure 4-17: Thing Model Describing a Siemens Thermostat

Annotating the thermostat in this way plays a fundamental role in ensuring semantic clarity and interoperability, enabling other software components in our solution the capability to *reason* on it and automatically infer insights based on these capabilities. In the next section, we provide details on the semantic model that we designed to describe the functionalities of a VO using µ*CEP rules.*

4.2.1 Semantic Model for VO Functions

The existing solutions developed for IoT environments often assume a fixed functionality for devices, which means once they are deployed in the environment their capabilities remain *static*. For instance, assume a smart building that is equipped with several thermostats, each configured with a similar set of functionalities e.g., regulating the temperature on a preferred value. However, smart environments are not static and *dynamically evolve* based on situations like change in user requirements, energy efficiency requirements or performance optimization. Therefore, the motivation behind our approach is the evolving nature of requirements and the need for deployed devices to update and extend their functionality dynamically at runtime. However, *manually* customising the behaviour of these devices is tedious. Therefore, our solution to this challenge is utilising µ*CEP rules*, which can be dynamically pushed to the deployed WoT VO's for extending their capabilities at runtime. This approach empowers VO devices to update their behaviour without the need for manual intervention or reconfiguration and enhances the flexibility and adaptability of devices.

To integrate µ*CEP* rules with the W3C WoT architecture and therefore provide interoperability among these rules and with the VO stack, we also model them semantically according to the TM. In Figure 4-18, we demonstrate our TM model for a sample µ*CEP* rule that can calculate the *average temperature*. Similar to a TM of a VO device, this model also includes an additional @type modelled as "*mcep:ThingFunction"* to indicate its capability to change the functionality of a device. Moreover, the µ*CEP* rule is modelled as a property named *"mcep:rule"*, allowing to define the rule itself as a string.

```
{
 "@context": [
  "https://www.w3.org/2022/wot/td/v1.1",
  {
   "brick": "https://brickschema.org/schema/1.0.3/BrickFrame#",
   "mcep": "http://mcep/shared"
  }
 ],
 "id": "urn:mcep:rule:averageTemperature",
 "@type": [
  "tm:ThingModel",
  "mcep:ThingFunction"
 ],
 "properties": {
  "averageTemperature": {
   "title": "Virtual function calculates average temperature on a thermostat",
   "mcep:rule": "a_std[_,_](Y) :- aggr {e[_,_](X), *, Y := avg(X)} [count 3]$",
   "mcep:operates_on": "temperature",
   "mcep:deployable": [
    "mcep:Device",
    "brick:Thermostat"
   ],
   "type": "number",
   "readOnly": true,
   "observable": true,
   "forms": [
    {
     "href": "/ThreadRoomSensor/{{THERMOSTAT_NUMBER}}/
       urn:mcep:rule:averageTemperature",
     "op": [
      "readproperty",
      "observeproperty"
     ]
    }
   ]
  },
  "averageTemperatureStatus": {
   "@type": "mcep:status",
   "description": "Rule Status Values",
   "observable": true,
   "type": "string",
   "enum": [
    "Invalid",
    "Inactive",
    "Activation Triggered",
    "Active",
    "Deactivation Triggered",
    "Deactivated",
    "Failed"
   ],
```

```
    "forms": [
     {
      "href": "/uCEP-status",
      "op": [
       "readproperty",
       "observeproperty",
       "unobserveproperty"
      ]
     }
    ]
   },
   "averageTemperatureStart": {
    "@type": "mcep:start",
    "title": "start the virtual function deployment on device",
    "writeOnly": true,
    "description": "Activate deployed virtual function on device",
    "type": "string",
    "readOnly": false,
    "observable": true,
    "forms": [
     {
      "href": "/uCEP-rule",
      "op": [
       "unobserveproperty",
       "observeproperty",
       "writeproperty"
      ]
     }
    ]
   }
  }
}
```

Figure 4-18: A Sample µCEP Rule Modelled as a Things Model

We also model additional knowledge within the properties of the *µCEP* rule such as the *environmental parameters* that a rule operates on with the "*mcep:operates_on"* field and the "*mcep:deployable"* field indicating the device types that this rule can be deployed on. In our example, the rule is restricted to being deployed on a device type t*hermostat* and that it supports the *µCEP* rule engine software stack.
This modelling ensures that only devices capable of changing the specified environmental properties (e.g., temperature, humidity, lighting) are eligible for rule deployment. It prevents the deployment of rules to arbitrary devices that cannot affect the desired environmental changes, for instance a *pressure sensor* that has no control on the parameter *temperature* or that is not enabled with the *µCEP* software stack is ruled out of the automatic reasoning.
Additionally, a TM of a VO function, models properties such as e.g., "*averageTemperatureStart"* and "*averageTemperatureStatus"* to deploy the *µCEP* rule on a compatible *µCEP* device. For instance, the "*averageTemperatureStart"* property indicates how to start a rule on a VO at runtime and the "*averageTemperatureStatus"* models the interface for retrieving the current status of a deployed rule on a VO.

4.2.2 Semantic Model for Device Intelligence

TinyML has gained widespread popularity where machine learning is democratised on ubiquitous IoT devices, processing sensor data everywhere in real-time. As the TinyML development progresses, we need to manage heterogeneous resources in TinyML systematically, including devices and TinyML models. However, managing TinyML, especially in the industry, where mass deployment happens, presents various challenges, including hardware and software heterogeneity, non-standardized representations of ML models, device, and ML model compatibility issues. IoT devices are typically tailored to specific tasks and are subject to heterogeneity and limited resources. Moreover, TinyML models have been developed with different structures and are often distributed without a clear understanding of their working principles, leading to a fragmented ecosystem. Questions to be addressed

include: How to find which IoT devices in the inventory have the necessary sensors and capability to support a given ML model without examining each device individually? How to discover existing NN models to be executed on a given embedded device without spending time reinventing a new model? How to manage and reuse TinyML components at scale? How to orchestrate the discovery result into an IoT application? Considering these challenges, we propose to use Semantic Web technologies to enable the joint management of TinyML models and IoT devices at scale, from modelling information to discovering combinations and benchmarking, and eventually facilitate TinyML component exchange and reuse. Here, we propose an information model, TM, like the TM for the CEP rules in the last section. This TM is designed to describe NN models aligned with the W3C WoT, therefore providing interoperability among these models and with the VO stack. Similar to a TM of a VO device and a CEP rule, this model also includes an additional @type modelled as *nnet:ThingFunction* to indicate its capability to change the functionality of a device.

Figure 4-19 shows the TM for TinyML models. We use the prefix *nnet:* to identify the namespace of our NN ontology: *https://w3id.org/tinyml-schema/neural-network-schema*. Here, metadata can be added to enrich an ML model. For instance, ID is used to identify a model uniquely. The property *title* is the name of the model. Similarly, a ML model can have a human-readable *description*. Furthermore, a neural network consists of various *layers* with the *input layer* being the first layer and *output layer* being the last layer. Each layer contains individual information, e.g., *input shape* and *output shape*. Also, model-specific information is covered in the ontology, which is essential for gaining insight into an ML model and relating the model to hardware. For example, the *RAM* and *Flash* requirements are critical to assess whether a model fits into an IoT device and are described as subclasses of *s3n:Memory* to conform to the TD. Additionally, different *sensors* and relevant information are assigned to a model for imposing sensor requirements. The *input* and *output* properties supply knowledge about the model usage.

```
{
  "@context": [
    "https://www.w3.org/2022/wot/td/v1.1",
    {
      "ssn": "http://www.w3.org/ns/ssn/",
      "nnet": "https://w3id.org/tinyml-schema/neural-network-schema",
      "cep": "https://w3id.org/tinyml-schema/cep-rule-schema#",
      "om": "http://www.ontology-of-units-of-measure.org/resource/om-2",
      "s3n_extend": "http://tinyml-schema.org/s3n_extend/",
      "schema": "https://schema.org/",
      "sosa_extend": "http://tinyml-schema.org/sosa_extend/"
    }
  ],
  "id": "urn:nnet:02ed1b17-de4f-4239-8812-392917a954dc",
  "description": "This is a fully quantized version (asymmetrical int8) of the MicroNet Small model developed by Arm",
  "title": "ARM_MicroNet_Small_anomaly_detection",
  "@type": [
    "tm:ThingModel",
    "nnet:NeuralNetwork",
    "nnet:Unsupervised"
  ],
  "properties": {
    "": {},
    "s3n_extend:flash": {
      "title": "Flash requirement of the model",
      "schema:minValue": "1.8712e+01",
      "schema:unitCode": "om:kilobyte",
      "readOnly": true
    },
    "s3n_extend:RAM": {
      "title": "RAM requirement of the model",
      "schema:minValue": "8.08e+00",
      "schema:unitCode": "om:kilobyte",
      "readOnly": true
    },
    "nnet:Layer": {
      "nnet:shapeIn": "[1 1960]",
      "nnet:shapeOut": "[1 4]",
      "nnet:hasType": "nnet:Reshape",
      "readOnly": true
    },
    "nnet:NetworkOutput": {
      "enum": [
        "0",
        "1",
        "2",
        "3"
      ],
      "btzf:hasEnumMap": {
        "btzf:hasMapMember": {
          "0": {
            "ref": "silence"
          },
          "1": {
            "ref": "unknown"
          },
          "2": {
            "ref": "connected"
          },
          "3": {
            "ref": "no"
          }
        },
```

```
      "readOnly": true,
      "observable": true,
      "forms": [
        {
          "href": "/urn:tinyml:02ed1b17-de4f-4239-8812-392917a954dc/networkOutput",
          "op": [
            "readproperty",
            "observeproperty"
          ]
        }
      ]
    },
    "schema:Sensor": {
      "description": "the sensors that the TinyML model supports",
      "@type": "sosa_extend:Microphone",
      "sosa_extend:hasSensorInfo": "tested on MP34DT05 20000 sps"
    }
  },
  "actions": {
    "uploadTinyML": {
      "title": "upload a TinyML model on the device with given uuid",
      "writeOnly": true,
      "type": "string",
      "uriVariables": {
        "thing": {
          "type": "string",
          "format": "uuid"
        }
      },
      "forms": [
        {
          "href": "/urn:tinyml:02ed1b17-de4f-4239-8812-392917a954dc/upload",
          "op": "invokeaction"
        }
      ]
    },
    "stopTinyML": {
      "title": "stop the deployed TinyML model on the device with given uuid",
      "writeOnly": true,
      "type": "string",
      "uriVariables": {
        "thing": {
          "type": "string",
          "format": "uuid"
        }
      },
      "forms": [
        {
          "href": "/urn:tinyml:02ed1b17-de4f-4239-8812-392917a954dc/deploy",
          "op": "invokeaction"
        }
      ]
    },
    "deleteTinyML": {
      "title": "delete a TinyML model from the device with given uuid",
      "writeOnly": true,
      "type": "string",
      "uriVariables": {
        "thing": {
          "type": "string",
          "format": "uuid"
        }
      },
```


Figure 4-19: TinyML Modelled as a Thing Model

4.2.3 Overview of VO Descriptor Based on W3C WoT

Until now, we discussed the individual artefacts and the semantic models that are required for changing the behaviours of devices dynamically at runtime. In this section, we define how each of the artefacts are used and fit in the overall architecture of the solution.

Figure 4-20 shows a high-level conceptual architecture of our proposed solution for extending the behaviours of VO devices dynamically at runtime.

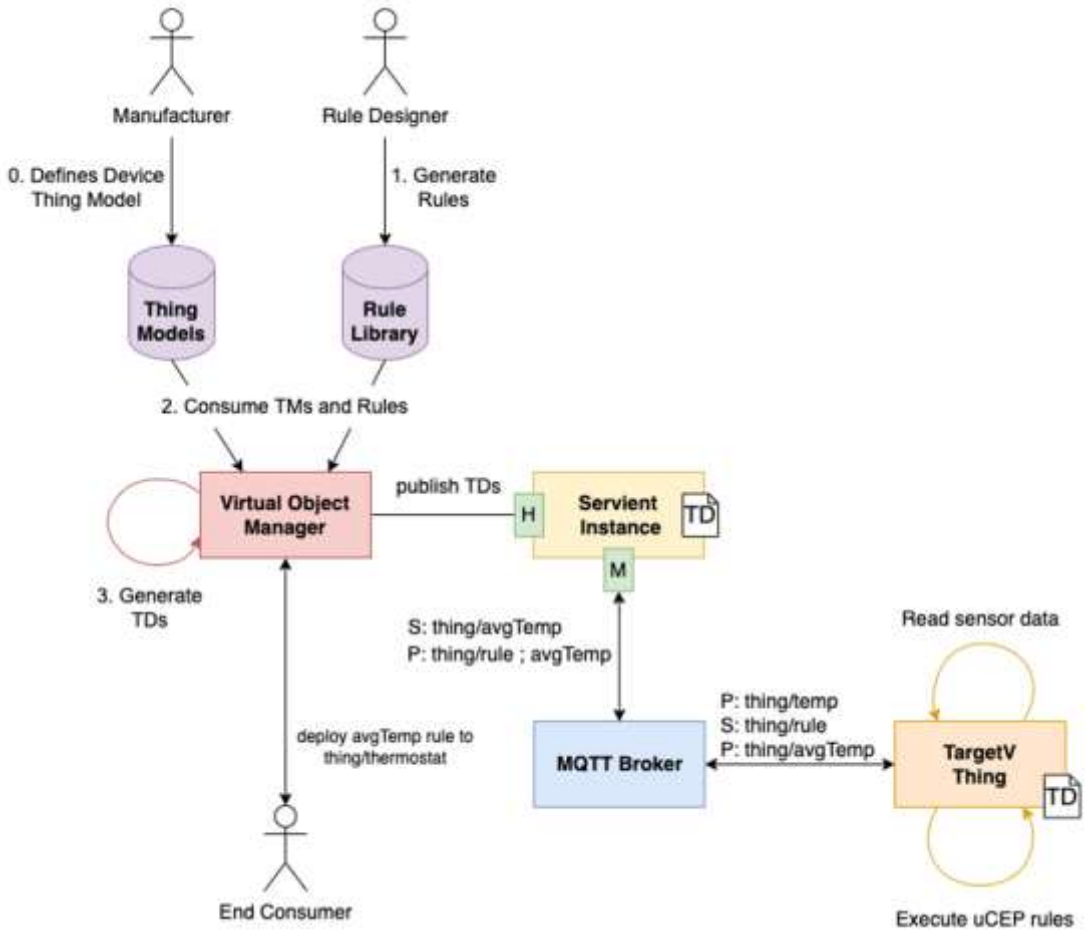


Figure 4-20: Overview of architecture for extending VO behaviour at runtime.

In our solution, three *human actors* are involved at various stages of the process:

- The *device manufacturer* is the actor responsible for providing the descriptions of the VO embedded devices using the W3C TM as JSON-LD document described in section 4.2. The WoT *Editor tool* can be used to simplify the process.
- The *rule designer* is the actor providing a semantic description of the µ*CEP* rules using a visual interface according to the model described in section 4.2.1.
- The *end consumer* is the actor who interacts with the system for manipulating the behaviour of deployed VO devices dynamically at runtime.

The models provided by the device manufacturers and rule designers are first stored in a knowledge base. The *Virtual Object Manager* (VOM) is a software system role representing our toolchain for supporting end consumers to manage their virtual object functions. It provides a management interface to the stored descriptions and the rules as well as communicating with the WoT servient for retrieving device properties or controlling the devices. We use the sayWoT servient in our solution, which receives the TD of a device from VOM and automatically connects with the device using the communication interfaces described within the TD as well as exposing the device properties over HTTP and WebSocket protocol. In particular, it subscribes to the Things using their native southbound protocols and bridges the communication to Web-based protocols as its northbound interface. In the case of this project, the embedded devices support the MQTT protocol and publish their data to an MQTT broker.

By interacting with VOM API, the end consumer can then perform the following:

- Rule management,
- Identify compatible VO devices for a rule,
- Deploy /unload rules from the VO devices at runtime according to their properties,
- Enhance the capabilities of the devices with additional properties.

Figure 4-21 provides a detailed overview of our proposed development process as a Business Process Modelling Notation (BPMN) diagram, with the main actors represented in different lanes and artefacts represented with the shape of a paper. The diagram shows the three main stages of the life cycle of our solution. The upper part is already explained, and it is for creating the models by human roles, which is fed to VOM. The third stage is our proposed toolchain and represents VOM for managing and extending the behaviours of devices at runtime.

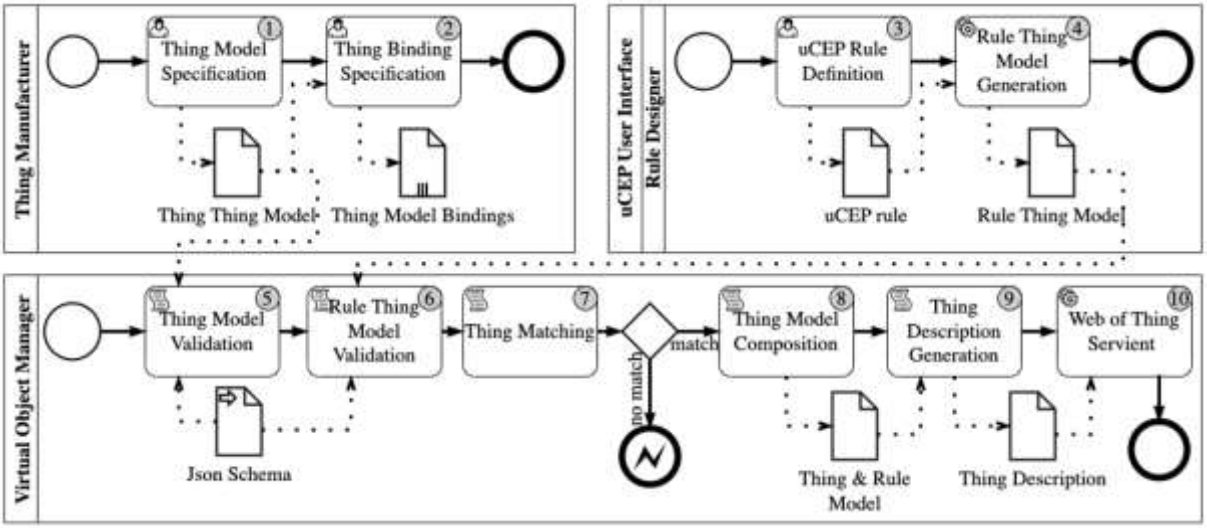


Figure 4-21: Overview of our solution as BPMN diagram

Below we describe each step involved in this process.

Thing Model and Rule Model validation - VOM can consume the provided device models and rules as TMs. The process begins with the validation of the models using JSON schemas. JSON schemas provide a structured approach to verify the syntactical correctness of these models and the adherence to the W3C WoT TM specification.

Thing Model Composition - Once the models are validated, VOM implements the algorithms defined within the W3C WoT specification for integrating a specific device with a rule. Each TM is used and integrated into one combined TM. This is achieved using the *links* container with *"rel": "tm:submodel"* keyword specifying the child TMs. An *instanceName* is optional and assigns a name to the composed TM sub-childs. The output is an artefact representing a composed TM.

Thing Description Generation - The TM to TD conversion process provides mechanisms for resolving the references and extensions provided in the link container. The conversion process begins by resolving the extensions and imports, applying additional metadata and binding information. To manage the additional information, we used the JSON *Merge-Path* algorithm. Then any placeholders are replaced using the placeholder map. The output is a TD which combines the initial capabilities of the device with an additional property that describes the rule functionality. This TD is then passed to the WoT Servient service to expose the device capabilities.

The sequence diagram represented in Figure 4-22, illustrates the detailed interaction between various components of our solution. It demonstrates the process through an example request sent to VOM, aiming to generate a TD for a particular device and rule.

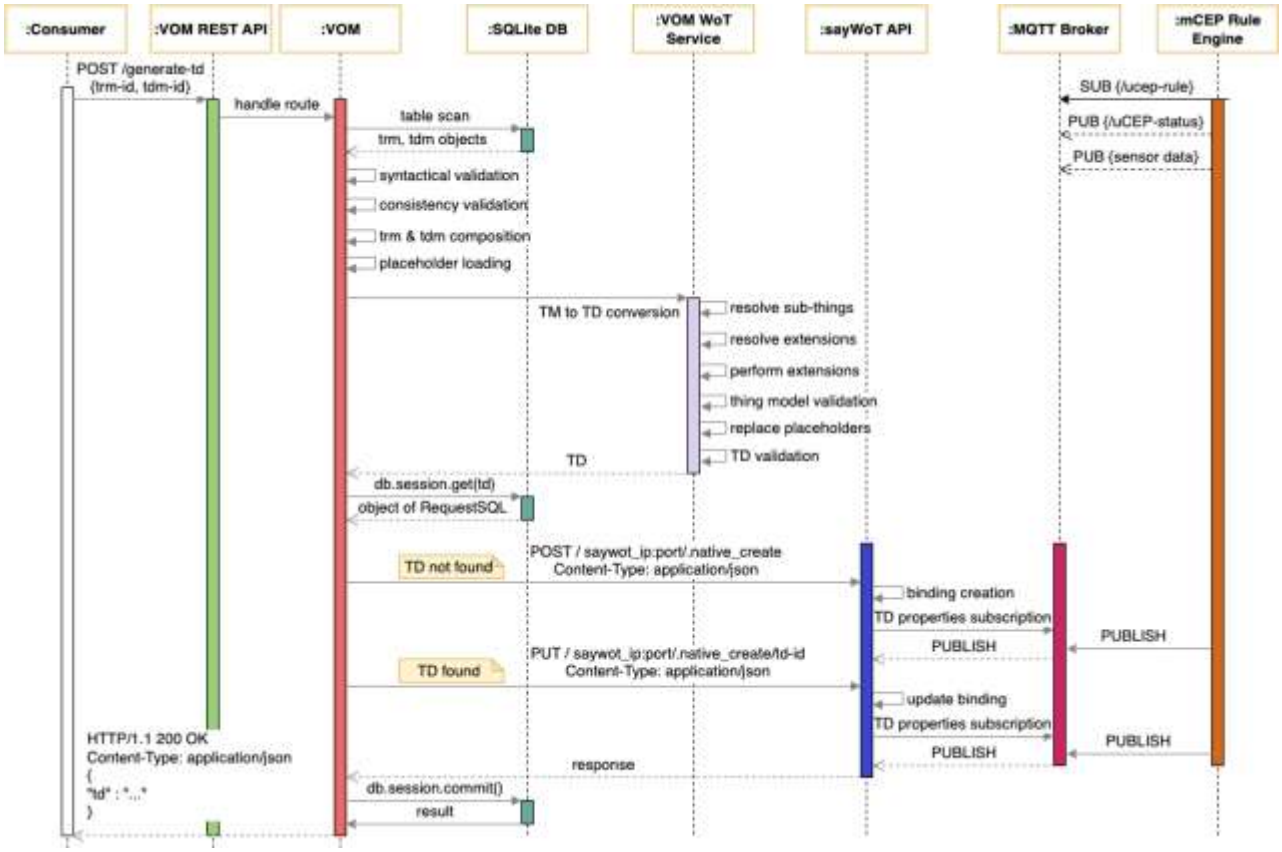


Figure 4-22: VO Descriptor Based on W3C WoT as Sequence Diagram

4.3 Interoperability and Relevant Solutions with OMA-LwM2M

The issue of semantic interoperability affects all the Information Technology IT systems and the way for approaching it depends on the specific field of interest. The semantic interoperability in web services focuses on the ontology organization of the web world, its rules, entities, and scenarios through the creation of complex meta-data not always related to the real context. Existing solutions for IoT build upon such concepts but depart from them to specifically support IoT peculiarities and, in particular, the abstraction of a physical device. Some of the most significant approaches are Semantic Sensor Networks (SSN) and TD, both recently proposed by World Wide Web Consortium (W3C) and extending the Semantic Web standard Web Ontology Language (OWL). In this project, among the approaches addressing semantic interoperability in IoT, we refer to the one proposed by the Open Mobile Alliance (OMA) [15] with the Lightweight Machine to Machine (LwM2M) standard building upon several IoT protocols and standards, like CoAP and IP for Smart Objects (IPSO) Alliance semantics. It encompasses IPSO semantics specifically focused on M2M device management through the organization of device resources. The most appropriate manner to represent IoT devices is by using semantic technologies [3]. Hence, the VO provides the semantic enrichment of data and functionalities provided by IoT device. The result of the semantic description is the VO model which includes, for instance: objects' characteristics, objects' location, resources, services, and quality parameters provided by objects. The VO model, intended as a software built for such a service, is independent from a specific device; it is initialized at startup according to the properties of the physical homologous it is going to represent thanks to a configuration file built on-purpose. The semantic description copes with heterogeneity and provides interoperability in the IoT domain eliminating vertical silos. In addition, it is immensely powerful in supporting search and discovery operations. Indeed, search and discovery mechanisms allow to find the device that is most appropriate to perform a given application's task.

In OMA-LwM2M provides Device Description File (DDF) of Objects by an eXtensible Markup Language (XML) configuration file, which defines the object structure and its resource data. The data producer, which hosts objects and resources, is defined as the OMA-LwM2M client, whereas the data

consumer is the OMA-LwM2M server. The client and the server just need to own the same configuration file in order to serialize/de-serialize the information. The public registry[28] provides objects defined by OMA and standard objects produced by third-party organizations. Moreover, developers can define customized objects following the technical specifications.
VO based on OMA-LwM2M presence targets the following crucial objectives: (i) overcoming platform heterogeneity, (ii) ensuring interoperability, (iii) improving search and discovery, and (iv) reducing the pressure on constrained devices. It provides the semantic description of the physical counterpart so to ensure a common understanding of its features and capabilities among all potential consumer applications. Specifically, it describes the embedded components by abstracting the specific hardware and software platform implementation. Hence, the VO exposes the capabilities of the relevant physical device to interested applications, managing transparent access to the intelligent heterogeneous resources. Such a feature is particularly beneficial for sophisticated applications relying, for instance, on AI inference capabilities. Indeed, the semantic description, in general and specifically of AI-empowered IoT devices, facilitates search and discovery procedures in order to identify the components that are the most appropriate, according to the demands of the requesting application (e.g., in terms of accuracy, expected inference latency), to perform a given task. Moreover, in so doing, the conceived abstraction of the capabilities of IoT devices makes the latter ones available to all interested applications in an interoperable manner, by overcoming fragmentation. Moreover, it acts as a proxy between the physical device and the consumer applications, it ensures interoperability by replying to the requesting applications, on behalf of the physical device, using dedicated interfaces developed with more web-oriented protocols like HTTP.
OMA-LwM2M is used in solutions proposed by FIWARE[29], AVSystems[30], Eclipse Foundation in (i) Leshan project[31] and Wakaama project[32], ARM[33]. The VO implemented bases its interoperability on backend logics which relies on interfaces and stored an information according to OMA-LwM2M standard. Interfaces implemented allows Device Bootstrapping, Registration, Management and service enablement, and Information Reporting using most of all LwM2M operations in uplink and downlink like Read, Write, Observe, Execute, Delete. Data is organized following entities defined by LwM2M: Device, Objects, Observables, Observers, Resources, and Values.
Thanks to the adoption of a semantic standard specifically developed for interoperability between heterogeneous devices and their management, the developed VO allows self-configuration at startup with respect to any physical device as long as the resources owned by it are described through OMA-objects LwM2M according to standard. Consequently, any entity interacting with the VO will be able to use the same semantic models to understand the device's meta-information.

4.4 Semantic Interoperability between WoT and NGSI-LD

To enable information exchange between WoT and NGSI-LD-based platforms, it is essential to provide semantic interoperability. Therefore, an architecture that enables this information exchange is required as well as semantic mapping between the two specifications. This section outlines the proposed architecture and the mappings between WoT TD and NGSI-LD model, which serves as the foundation of the semantic transformer that will be implemented in the future of this deliverable.
The NGSI-LD Meta-Model and TD serve as different purposes. WoT TD is tailored for describing capabilities and interfaces of device instances, while NGSI-LD is used for sharing context information about entities and their attributes. The following figure illustrates a conceptual diagram showing the

[28] www.openmobilealliance.org/wp/OMNA/LwM2M/LwM2MRegistry.html

[29] https://fiware-iotagent-lwm2m.readthedocs.io/en/latest/userGuide/index.html

[30] https://www.avsystem.com/products/anjay/

[31] https://www.eclipse.org/leshan/

[32] https://github.com/eclipse/wakaama

[33] https://github.com/PelionIoT/java-coap

interoperability problem between two VO's that are based on different standards. It considers the two mapping variants that are required in the context of the NEPHELE project. In the first case (left part of the diagram), *Device 1* is based on a WoT virtual object, while *Device 2* provides an NGSI-LD virtual object. In this case, the composite VO can access the data if it is of type WoT. Similarly, on the right-hand side of the diagram, the other way round is considered where an NGSI-LD composite VO is required.

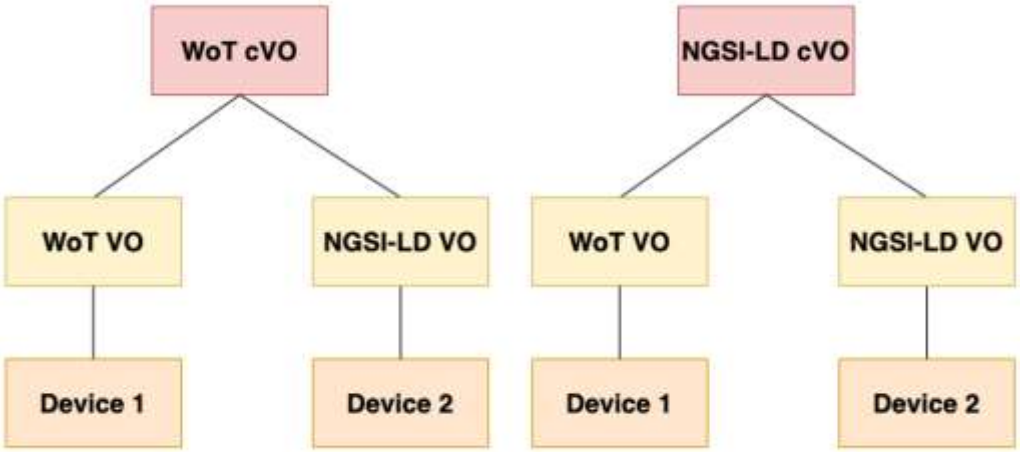


Figure 4-23: Semantic interoperability challenge between WoT and NGSI-LD composite virtual objects.

Figure 4-24 illustrates a potential solution for a bi-directional interworking between WoT and NGSI-LD aimed to bridge semantic interoperability between NGSI-LD and the W3C WoT specification. On the one hand, NGSI-LD Context Broker offers applications an NGSI-LD interface to publish, consume and subscribe to information (e.g., Properties or Events) offered by Things or abstracted as NGSI-LD Entities. On the other hand, WoT Servient, plays both the role of a client and a server in the WoT domain, i.e., allowing applications to interact with Things (executing Actions) associated with NGSI-LD Entities, and to consume the functionalities provided by Things, exposing them as/to NGSI-LD Entities.

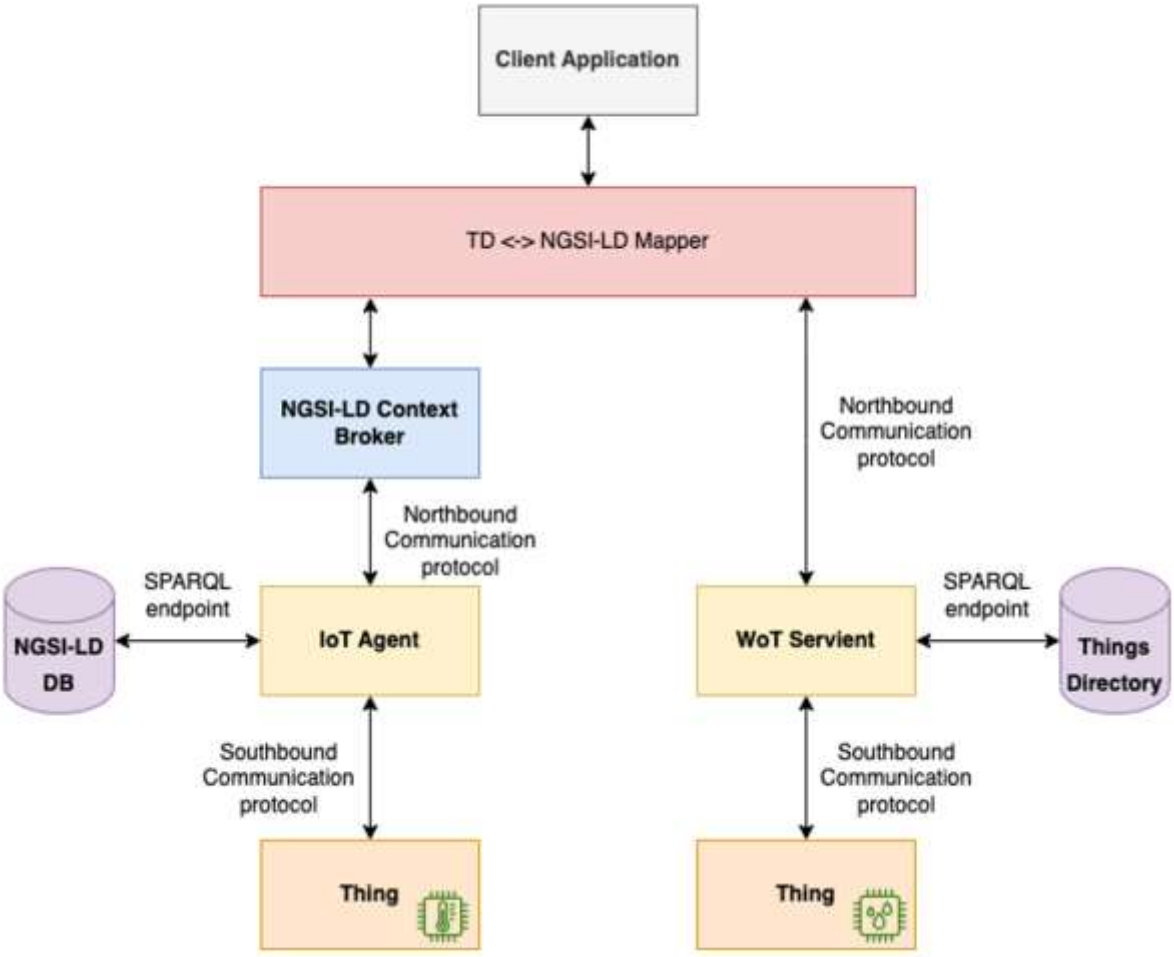


Figure 4-24: High-level overview of interworking between NGSI-LD and WoT

Providing a bi-directional semantic mapping between NGSI-LD and TD is challenging due to the inclusion of specific information, such as security schemes and binding protocols, which cannot be directly expressed in NGSI-LD. Consequently, when converting NGSI-LD models to TDs, some mandatory fields might be omitted, rendering the resulting TDs invalid. To address this challenge, one solution is to use TMs, as an intermediary for the mapping between NGSI-LD to TDs and vice-versa. Unlike TDs, TMs do not require certain-specific information and have a more limited set of mandatory fields. In particular, a JSON-LD @Context importing the semantic vocabularies and an @type tm:ThingModel are the only mandatory fields.
Therefore, the TD<->NGSI-LD Mapper component requires a mapping between the three models: one between NGSI-LD and TM and the other between TM and TD. Figure 4-25 shows a high-level conceptual view of the transformation process, illustrating the role of TMs as an intermediary between the two models. The TMs can be enriched with the following information: protocol bindings, metadata, and placeholder map to facilitate the conversion to a valid TD.

Figure 4-25: Conceptual overview of the semantic mapping between TDs and NGSI-LD models

Current status: For this sub-task, we have provided an initial semantic mapping between the TM and NGSI-LD semantic models and vice-versa. Also, we have implemented the process of converting TMs to TDs. Furthermore, the implementation for converting between an NGSI-LD model and TM and vice-versa is ongoing. For this, we are using an open-source project Chimera [38], which is a framework that offers components to define schema and semantic conversion pipelines based on standard Semantic Web techniques.

4.4.1 Interoperability between VO Descriptor and models based on W3C TD and LwM2M

To evaluate the interoperability between a VO based on the VO Descriptor and the W3C TD, we aim to create a VO with each of the provided models. In particular, there will be two MQTT endpoints one offering temperature data and the other presence data, where both are acting as servers. Then, we will create a composite application that acts as a consumer of the two modelled VOs, requiring the use of sensor data provided by both devices. The composite application acts as the consumer of the data. It is essential to acquire the sensor data from each of the VOs without any additional effort, and in a uniform way using HTTP as the northbound protocol, regardless of the southbound interface used by each individual VO.

5 Autonomic Functionalities and Ad-hoc Clouds Management

5.1 Autonomic Networking Functionalities at IoT Level

Autonomous networking functions refer to the capacity of a network to self-configure, self-monitor, and self-optimise without human intervention. These functions are relevant because they allow automatic network adaptation when there are changes in topology, network traffic, and demand, ensuring the availability of resources when required. Autonomous networks can detect and respond to improve network failures in real time, improving response times and considering different optimization schemes in decision-making. One optimization process that benefits from autonomous functions is allocating network resources such as bandwidth and computing capacity based on real-time demand and application requirements. As the network increases in complexity with the adoption of innovative technologies such as SDN and cloud computing, autonomous functions help keep network management manageable. As technology evolves, autonomous networking functions can adapt to new networking paradigms and emerging technologies, ensuring that the network remains relevant and efficient in the face of change.

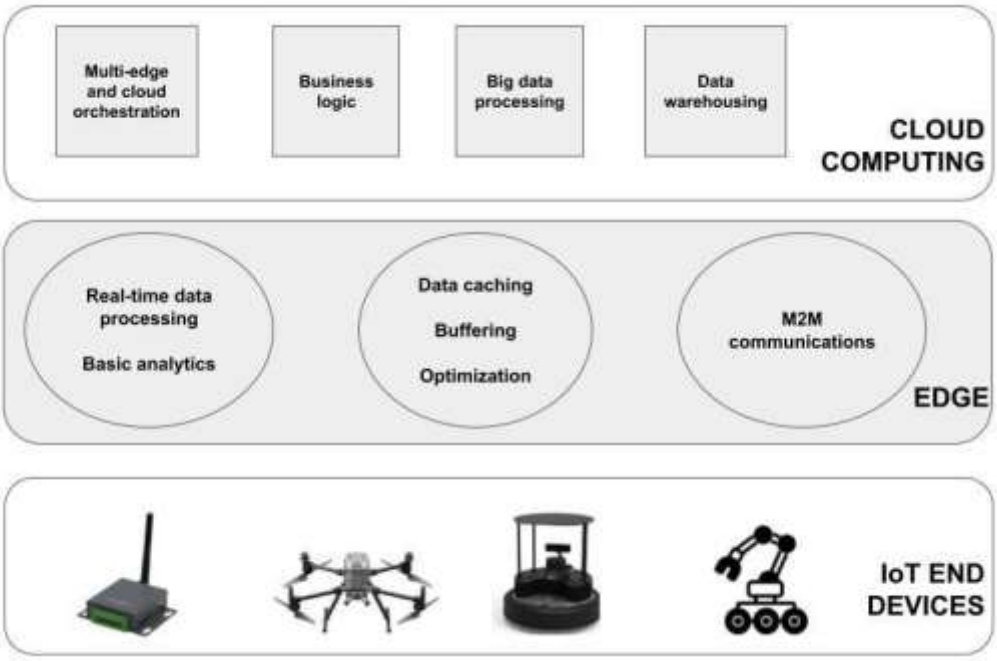


Figure 5-26: Functionalities distribution along the compute continuum

In the IoT context, one of the challenges in autonomous networks is enabling agility, mobility, scalability, and resilience between end devices and Edge/Cloud computing services through different communication technologies. One of the components that facilitates the adoption of IoT solutions in the computing continuum is the deployment of Edge computing close to the source of information since IoT applications often require real-time or near-real-time data processing. Edge computing filters and processes data locally, sending only relevant or aggregated information to the cloud, thereby reducing the volume of data that needs to be transmitted to the cloud servers. The availability of edge servers through mobile agents such as robots and drones allow flexible and quick-reaction solutions in dynamic and unpredictable scenarios. Edge devices can continue operating even when the network connection is lost, allowing critical functions to be still performed at the edge, ensuring the reliability of IoT applications. Figure 5-26 shows the distribution of some functionalities through continuous computing, where services such as real-data processing, data caching, and M2M communications are available through edge servers. Thus, the data sent by IoT end devices through the VO or the (c)Vo will be processed at the edge level, reducing latency, and using available resources better.

End devices, such as sensors, robots, drones, and gateways must be accessible over the network to send their data and for remote monitoring and management. Combining multiple wireless technologies such

as Wi-Fi, Bluetooth, Cellular (4G, 5G), and LoRa can significantly enhance network reliability and robustness, particularly in critical applications like emergency services or industrial IoT, where latency, bandwidth, and throughput requirements are crucial to maintaining operation. Multi-radio technologies devices can switch between the technologies or use them concurrently, optimising for variables such as data rate, coverage, and power consumption. For example, a device could use Wi-Fi for high-data-rate transmissions but switch to LoRa for long-range, low-power communication. The integration of technologies requires seamless handover algorithms to ensure smooth transitions between technologies, preventing communication disruptions. In a previous work [83] a Network Interface Selection (NIS) technique which is been adjusted to run over updated hardware to provide network robustness and flexibility supporting several technologies. It consists of having multi-technology wireless sensor nodes coupled with autonomous and adaptive algorithms. The latter would allow the nodes to dynamically select the communication technology that fits best the environment, application, and data requirements/constraints. In [84] we propose a lightweight Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS)-based method for NIS in WSN. The proposed technique shortens computation time by 38% compared to classic TOPSIS with 82% similarity in terms of obtained decision-making results. In the context of the post-disaster in a container port case study, the robots exploring the region would be able to communicate using different technologies and deploy multi-technology sensor nodes. This may allow the network devices to adapt to heterogeneous data rates (video, image, time-series, audio, numerical, geospatial, etc.) and to the dynamic nature of the network impacting coverage, communication distances, and throughput caused by the mobility of robots and the dynamic deployment of sensor nodes.

Another benefit of multiple technologies integration is loading balancing for distributing network traffic across multiple channels, reducing congestion, and improving performance. When needed, data offloading strategies can optimise network usage by shifting data transfer to less congested or more reliable technologies. Redundancy and fail-over mechanisms allow the network to maintain functionality even if one technology fails or experiences high latency. The availability of technologies with different data rates and coverage enables the network to support QoS management, ensuring that critical data gets priority and sent over the most reliable and fastest available channel.

Finally, routing is a critical autonomous function in the cloud-to-edge-to-IoT continuum to ensure that data and communication flow efficiently and securely across the computing continuum. In the IoT, data can originate from various sources, including IoT devices, edge computing systems, or cloud services. Effective routing ensures data packets reach the appropriate destination, whether it is another IoT device, a local edge server, or a cloud service. Routing mechanisms must be optimised to minimise latency, especially for real-time or time-sensitive applications using edge computing resources to process data closer to the source. Effective routing enables seamless integration between edge and cloud resources, which means routing decisions may adapt dynamically based on network conditions, device availability, and data processing requirements. So far, the preliminary design of the routing protocol for supporting an adaptive wireless sensor network in NEPHELE considers the connection of the sensor nodes through VO or (c)VOs implemented in an IoT gateway. This configuration may facilitate the management of constraint devices, such as temperature or humidity sensors. Different protocols may be used depending on the context, considering that lightweight and low-power protocols such as MQTT or CoAP are most common at the edge and in IoT devices.

The autonomic networking functionalities at the IoT level addressed in this section would support the requirements FR_VOS_011, FR_VOS_012, FR_VOS_014, FR_VOS_015 in Table 2.1, and NFR_VOS_01 in Table 2.2.

5.2 Networking Functionalities in the VOStack

The networking functionalities supported by the Virtual Object (VO) are accommodated within the ***Physical Convergence*** layer of the VOStack. For instance, in the context of a smart port use case, a network of IoT devices is deployed in the area, ranging from cameras to sensors monitoring storage parameters (*e.g.*, temperature, humidity) mounted on cargo containers. In the former case of IoT devices, minimal latency and jitter are crucial metrics, whereas in the case of cargo container sensors, dynamic routing is crucial, since cargo container stacking, a usual practice in large ports, can create signal

interference that may lead to low quality or even connectivity loss, in a rather dynamic mode as these containers are being constantly relocated. A plethora of network-oriented functionalities are provided to support uninterrupted connectivity with the devices, manage dynamic routing protocols, implement TSN techniques, and also address mobility considerations.

The internal structure of the networking functionalities of the VO is depicted in Figure 5.2. These functionalities are organised as two distinct elements, specifically the TSN Control plane and Reactive Routing. Figure 5-27 illustrates that the internal networking components in the (c)VO may consist of either components exclusive to the cVO (*e.g.*, Clustering and Schedule Engine) or components that are deployed either to VO and cVO (e.g., Network Model and Flow & Path Model).

By utilising TSN, we will ensure low latency and jitter in the communication between IoT devices and their associated VOs. The deployment of these VOs can be conducted as containers on servers that are positioned at the network's edge, or on network devices (*i.e.*, IoT Gateways) at the far-edge of the compute continuum. TSN will assign greater precedence to the transmission of data pertaining to high-priority traffic communication. This communication may be related to tasks, such as the streaming of videos captured by cameras, the transfer of data collected by sensors, and the dissemination of instructions to control robotic arms. In contrast, other types of traffic, such as best-effort, will be assigned with a lower level of priority. The Qbv-based TSN scheduler, specifically the TAPRIO qdisc, is employed on IoT Gateways that are placed between IoT devices and (c)VOs.

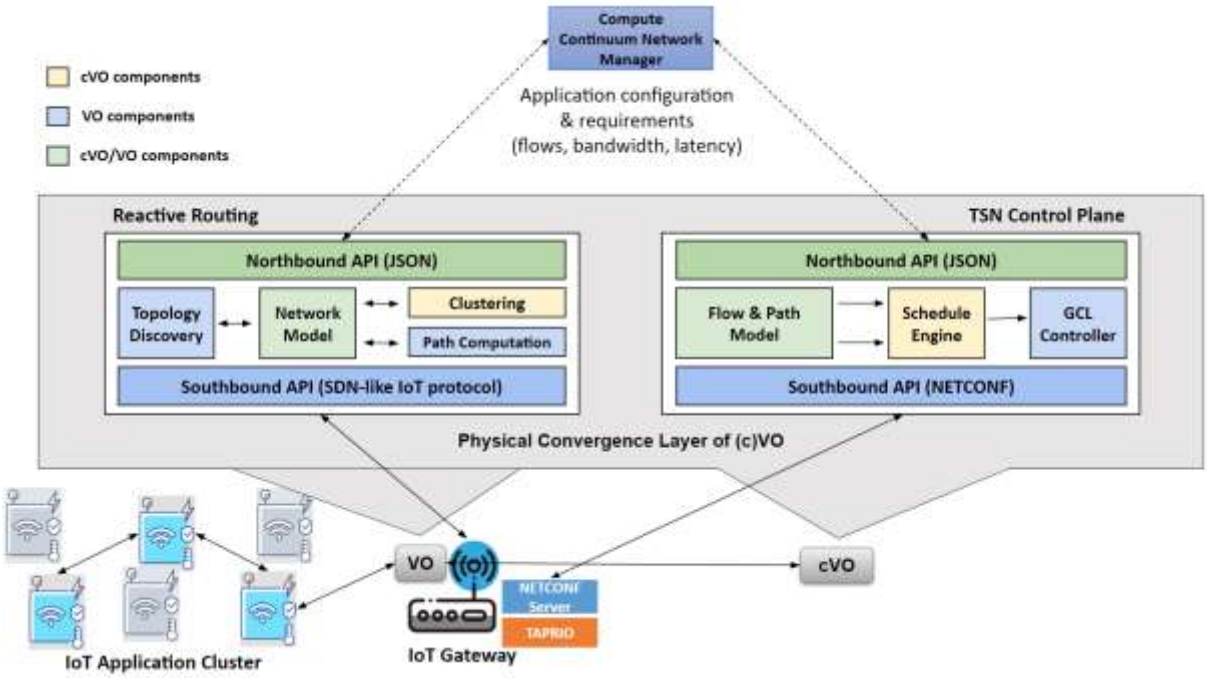


Figure 5-27: Ad-hoc cloud and networking functionalities at the VO stack

On the other hand, Reactive Routing maintains network connectivity between wireless IoT nodes and the IoT gateway. The former typically collect measurements from their environment and communicate them periodically to a VO through the IoT gateway. There are also cases of deployments with both sensors and actuators, where the central system does not have the role of measurement collection but is instead notified of specific events or has a coordinating role with the nodes. The network paths between the wireless sensor nodes and the IoT gateway (or the wireless sensor nodes) are occasionally multi-hop, as their distance may extend beyond their individual network coverage. While traditional IoT network protocols can be employed in such scenarios (*e.g.*, RPL), our primary focus lies on Software-Defined Wireless Sensor Networking (SD-WSN) approaches, which offer several distinct advantages. SD-WSN enables dynamic routing adjustments based on a global view of the network, *e.g.*, managing routing changes due to mobility or signal interference. Furthermore, they provide logically centralised and

programmable routing control, allowing easy integration with the distributed control across the compute continuum, fostering enhanced network intelligence and monitoring capabilities.

5.3 SDN-based Reactive Routing

SDN can support intelligent, programmable, and logically centralized control mechanisms that dynamically adjust protocol functionalities to achieve improved performance and resource utilization, addressing specific performance demands from IoT applications. They adopt the typical SDN architecture, where the application plane is responsible to communicate application requirements, the control plane to implement control mechanisms and the data plane to communicate the data.
However, wireless IoT deployments are often affected by radio signal issues, *e.g.,* due to mobility or interference, which can impair control communication. Furthermore, the latter may also be characterized by increased overhead. To tackle the challenges of intermittent connectivity with the controller, control message scalability, and mobility, several SD-WSN solutions have been proposed in the literature.
In our SDN controller architecture, the Compute Continuum Network manager implements high-level network management functionalities (*e.g.,* communicates application or flow requirements), the cVO translates such requirements into a customized protocol configuration for a number of VOs and the latter implement autonomous network control features, *e.g.,* topology discovery or flow establishment. As shown in Figure 5-27, the (c)VOs implement Reactive Routing through the following SDN control facilities: (i) Topology Discovery, (ii) Network Model, (iii) Clustering, and (iv) Path Computation, which we detail below. The implementation is modular enough to accommodate new mechanisms or extend the existing ones, in a straightforward manner.

- **Topology Discovery (TD)**. We currently support three topology discovery mechanisms, *i.e.,* the Node's Advertisement Flooding (TC-NA), the Node's Neighbours Requests from the Controller (TC-NR), as well as their hybrid combination. TC-NA is an epidemic algorithm inspired by the topology discovery mechanism, employed by the state-of-the-art non-SDN IoT routing protocol IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL). In TC-NA, TD periodically communicates topology discovery control packets to its assigned border router, triggering the broadcasting of ''Neighbours' Discovery'' short-range beacon message from the latter. Each receiving neighbour creates a response message that informs TD for any link existence between the beacon node and itself. TC-NR is a centralized topology discovery algorithm better aligned to SDN paradigm and dynamic topologies, *i.e.,* collects topology information through individual requests to the nodes from the TD. It is flexible enough to send targeted topology requests on specific nodes or parts of the network without overloading the rest of the topology. Lastly, we also utilize a hybrid combination of TC-NA/TC-NR for heterogeneous topologies that consist of both mobile and fixed nodes, for maximum adaptation to the dynamic characteristics of the network. TD is implemented in each VO and can have bespoke configuration, depending on the deployed IoT application and the network characteristics (*e.g.,* level of dynamicity). The results of TD are also being forwarded to the cVO, *i.e.,* building up the global picture of all deployments.
- **Network Model (NM).** NM is responsible for topology maintenance and representation, *i.e.,* the former ensures that the latter is up to date. Although IoT protocols such as RPL use distributed mechanisms residing at the nodes (*e.g.,* trickle timer), our topology maintenance is fully coordinated from the VO, allowing an easy integration of context-sensitive solutions. For example, different static topology refresh periods can be assigned to fixed (*i.e.,* higher values) compared to mobile nodes (*i.e.,* lower values), regulating control overhead and achieving efficient topology maintenance. The topology representation assigns frequently updated link quality values to each edge of the graph, which can be selected from the administrator (*e.g.,* Link Quality Indicator (LQI) or Received Signal Strength Indicator (RSSI)). NM resides both at VOs, *i.e.,* maintaining the topology graph of IoT nodes assigned to it, and the cVO (*i.e.,* keeping track of all IoT nodes in the area).
- **Path Computation (PC).** PC specifies the end-to-end paths from source to destination nodes (*i.e.,* the latter could be the IoT gateway passing the data to the VO). These paths should be aligned to the requirements of the IoT application, *e.g.,* reduce delay, achieve reliable

communication, avoid loops or deadlocks, as well as construct alternative paths. PC determines these paths that are being translated to flow rules in tuples of Destination and Next Hop node addresses, being stored to the individual flow table of each node. In constrained and dynamic IoT environments the flow rule expiration mechanisms are also important. In our case, the flow rule expiration is being managed entirely by the PD, so it can be aligned to other important network control decisions (*e.g.*, topology discovery or maintenance parameters). We currently support four types of flow establishment processes (*i.e.*, being responsible to construct and maintain the forwarding tables of the nodes), the Next-Hop Only (NHO), the Complete Path (CP), the NHO-CP combination as well as the Proactive Flow Establishment (PFE) mechanisms. NHO communicates to the nodes their own forwarding rule only, *i.e.*, to reach the next node, where the CP informs all the intermediate nodes participating in the routing path. An NHO-CP combination is being employed in the case of topologies consisting of both mobile and fixed nodes, where the fixed part employs CP and the mobile the NHO, allowing the maintenance of the dynamic parts of the paths only. Lastly, PFE employs clustering to classify links based on their connectivity quality history, which guides proactive flow establishment rules.

- **Clustering.** Clustering capabilities are being supported at the level of cVO or Compute Continuum Network Manager for various reasons, including: (i) the appropriate association of IoT nodes with particular VOs, which may also involve a network slicing activity at the IoT network level; (ii) the association of bespoke protocol configuration per node type or characteristic, *e.g.*, mobile nodes may be configured by TC-NR topology discovery and refresh their connectivity status more frequently; and (iii) implement proactive routing by classifying links based on their connectivity quality history. At this point of investigation, we experimented with the K-means algorithm for IoT mobility and a combination of partitional clustering with similarity-based measures (*i.e.*, based on dynamic time warping and k-medoids algorithm) to implement proactive SDN-based flow establishment.

As a bottom line, each VO receives high-level configuration options and directives being decided from the components above them (*i.e.*, cVO or Compute Continuum Network Manager), configuring or adapting its autonomic mechanisms, so they perform in alignment to the requirements of the IoT application.

5.3.1 SDN Control Plane

We assume IoT nodes that support two radio interfaces: a long-range interface for the SDN control channel and a short range for data communication, thus implementing out-of-band SDN control. Although we assume one border router (BR) is located at the IoT Gateway, the approach allows for multiple BRs, thus supporting elaborate partition of the IoT infrastructure.

The protocol assumes a control and a data network stack installed in each IoT node, with the former catering for the long-range communication and SDN control messages and processes, while the latter handling low power short range wireless communication and the forwarding layer of the SDN protocol. The current implementation is based on a Contiki-OS fork to support the dual network stack, with improved network core modules and Zolertia RE-mote devices, upgraded to enable activation of both radio interfaces.

The southbound API manages control messages exchanged between the SDN Controller and the IoT nodes in order to support the functionality described previously. More specifically, API messages fall under the following categories: topology control and routing control and device control, as indicated in Table 5..

The *topology control* messages concern the basic functionalities of topology discovery and maintenance. Thus, this message class includes Border Router (BR) related messages (registration, solicitation, new BR), and messages related to node discovery; for instance, new node response messages, or messages related to the initiation of neighbourhood discovery.

The *routing control* message group contains messages related to the establishment of paths among node devices and include all the forwarding rule management actions; for instance, adding forwarding rules to nodes.

Finally, the *data delivery control* message group includes messages related to IoT data delivery, *i.e.*, add/remove subscription messages of nodes to data generated by the node, since the VO will be in charge of controlling IoT data delivery, as well.

Table 5.1 SDN Southbound API

Message Group	**Message Type**
Topology Control	New Bridge Router Solicitation Request Bridge Registration New Node Solicitation Request New Node Response Neighbour Request (TC-NA/TC-NR) Neighbour Response Update Protocol Parameters
Routing Control	Missing Forwarding Rule Add Forwarding Rule Remove Forwarding Rule Replace Forwarding Rule Remove All forwarding Rules
Data Delivery Control	Add Subscription Remove Subscription

The SDN-based reactive routing codebase for the VOStack is available here[34].

5.4 Time-Sensitive Networking

5.4.1 TSN Control Plane

Regarding the control plane, we use a hybrid TSN implementation, as described in the IEEE 802.1Qcc standard, in order to automate the TAPRIO configuration process. More specifically, we implemented a prototype Central Network Controller (CNC) that has the capability to compute TSN 802.1Qbv schedules and populate these schedules into TSN-enabled switch data paths. The CNC consists of three internal modules: (i) Flow & Path Model, (ii) Schedule Engine, and (iii) GCL Controller. Below we briefly describe these modules:

- **Flow & Path Model.** The Path and Flow Model module has two main functionalities. The first one is the categorization of incoming flows into distinct traffic classes, such as high-priority and best-effort. This is achieved based on some predefined rules that can match applications' network requirements (*e.g.*, latency less than 1 ms) to traffic classes and determine whether the request should be categorized as critical or non-critical. The second functionality of this module is the path configuration and by path, we define the fixed network between the IoT Gateway and the VO. This path will be used as an input to the Schedule Engine module.
- **Schedule Engine.** The Schedule Engine module is implemented as a cVO component and is responsible for implementing a scheduling model to determine a scheduling pattern for the incoming flows in order to satisfy their latency requirements. In this implementation, the scheduling model is based on constraint programming, since the latter offers versatility in problem modelling, and efficient heuristic search in combination with powerful constraint propagation techniques. It should be mentioned that the TSN scheduling problem is classified as an NP-complete problem and various strategies have been proposed to address this problem in the literature, including Satisfiability Modulo Theories (SMT), Constraint Programming, Heuristics, and Genetic Algorithms.
- **GCL Controller.** The GCL Controller module receives the output of the Schedule Engine as its input and is responsible for configuring time intervals on the Gate Control List, and

[34] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-sdn

determining the duration over which each queue is open for transmission. The GCL Controller uses in order to send the GCL configuration to the IoT Gateway.

Furthermore, the TSN Control Plane incorporates two application programming interfaces (APIs). The proposed system includes a Northbound API, implemented using a well-defined JSON schema, which is capable of processing requests related to application configuration and requirements from the Compute Continuum Network Manager. Additionally, a technology-specific Southbound API, utilising NETCONF, is responsible for transmitting the GCL configuration to the IoT Gateway through the use of Remote Procedure Calls (RPC).

5.4.2 TSN Schedule Engine

As mentioned, the TSN scheduling problem is considered a NP-complete problem and several approaches have been proposed towards its solution, such as Satisfiability Modulo Theories (SMT) [85], Constraint Programming [86], Heuristics [87], and Genetic Algorithms [88]. A recent systematic review can be found in [89]. In terms of TSN schedule computation, the main objective is to determine a feasible scheduling pattern for incoming flows with respect to their specified requirements (e.g., latency, jitter), based on a defined model and constraints. The scheduling model is built on constraint programming, harvesting on its flexibility in terms of problem modelling and exploiting the power of efficient general heuristic search combined with robust constraint propagation techniques for domain reduction, in the form of global constraints.

1. Problem Formulation

We follow a similar approach to modelling the TSN scheduling problem in constraint programming, as those in [85], [86], [90]. Thus, we assume the usual graph representation found in previous work, i.e., we consider a graph $G = (V, E)$, where vertices (nodes n_i) are either switches or end-points, whereas edges are links $(l_{ij} \in E)$ connecting the former, i.e. link l_{ij} connects nodes i and j respectively. Each link l_{ij} is characterized by its propagation delay ld_{ij}, its speed sp_{ij}, and a number of queues Q_{ij} that are at most 8 according to IEEE 802.1 Qbv. Switches also have a processing (or fabric) delay sd_i which is constant.

In the current setting, we consider a set of periodic flows F of single packets (frames), i.e., the payload of each flow can fit into a single Ethernet frame, where each flow $f_k \in F$, has a deadline df_k, a packet size p_k, a period T_k that determines the frequency of the transmission, and is associated with a valid path P^k, that consists of an ordered list of links $[l_{ti}, l_{ij}, \dots l_{nl}]$ from the transmitting end-point n_t, i.e., the talker, to the receiving end-point n_l, i.e., the listener of the flow.

The problem can be considered as a classic job-shop scheduling problem, where each flow transmission is considered as a task, consisting of as many transmission operations as the links it traverses along the path to the listener, with a number of additional constraints that will be discussed below. Thus, a flow f_k on a path P_k of length N can be modelled as a set of operations 1..N, each having a start time (offset) relative to the start of the schedule (time point 0), i.e., the set $S^k{}_{1,i,j}, S^k{}_{2,j,k}, \dots, S^k{}_{N,k,l}\}$ where each variable $S^k{}_{x,i,j} \vee x \in 1..N$ represents the start time of the transmission of the packet of flow k on the link l_{ij}. We define this set as the primary flow. Additionally, for each flow f_k we define a queue Q_k, which is considered to be the same for all links of the flow.

Scheduling of flows that have different periods occurs in a time domain defined by the hyper period [91], where the latter is defined as least common multiple of all flow periods, i.e., $HP = lcm(T_k \vee f_k \in F)$. Flows may occur multiple times in a hyperperiod and in the later case, each flow f_k that occurs $M = HP/T_i$ times in the hyperperiod, consists of N*M start times, as indicated in the following equation:

$$f_k = \{S^k{}_{1,i,j}, \dots, S^k{}_{N,k,l}, S^k{}_{N+1,i,j}, \dots S^k{}_{2*N,k,l}, \dots S^k{}_{N(M-1),i,j}, \dots, S^k{}_{N*M,k,l}\}$$

We define sub flows $S^k{}_{x,i,j} \vee x > N$ as secondary flows; we make this distinction since primary flow start time variables and those of the secondary flow, participate in different constraints in the model. In the CP model, the decision variables are the start times in both primary and secondary flows and the flow's queue, i.e., $\{S^k{}_{o,i,j} \in [0..HP] \vee o \in 1..N * M, Q_k \in 1..7 >$. Since queue 0 is reserved for unscheduled best-effort traffic, the domain of Q_k is restricted to the range [1.7]. The requirement for

zero jitter, enforces that decision variables in the primary and secondary flows are linked by the following constraint (zero jitter):

$$\forall n \in 1..N, \forall m \in 2..M, \forall l_{ij} \in P_k, S^k{}_{n,i,j} + (m-1) * T_k = S^k{}_{N*(m-1)+n,i,j}$$

The first scheduling constraint imposes an ordering among the start times of each operation in a task, ensuring that a packet on link l_{ij} has arrived on node n_j, before being transmitted over the link l_{jl}. There is no need to define the constraint for secondary flows, due to the equality constraints of zero jitter constraint mentioned above, which ensures that the correct time distance is maintained among operations of each subflow. Thus, enforcement of the constraint occurs only on decision variables of the primary flow and is depicted below:

$$\forall f_k \in F, \forall o \in 1..(N-1), S^k{}_{o.i.j} + sd_j + ld_{ij} + \lceil \frac{p_k}{sp_{ij}} \rceil \leq S^k{}_{o+1,j,l}$$

Equation above considers the propagation delay ld_{ij} of the link, the transmission delay computed as the ceiling of the packet size over the speed of the link $\lceil \frac{p_k}{sp_{ij}} \rceil$, and the switch delay sd_j of the receiving node.

Given that flows have a deadline, the flow's packet must arrive to the listener node n_l over a link l_{ml} before the deadline df_k:

$$\forall f_k \in F, S^k{}_{N,l,m} + ld_{ml} + \lceil \frac{p_k}{sp_{ml}} \rceil \leq df_k$$

Once more, the deadline constraint is imposed only on the primary flow variables, and equality constraints of the zero-jitter constraint ensure that is enforced for all flows in the hyper period.

Each egress link transmits a single packet at each time point, thus, no two transmissions on the same link may overlap, yielding the constraint:

$$\forall l_{ij} \in E, f_k, f_r \in F \vee k < r, \forall S^k{}_{x,i,j} \in f_k, \forall S^r{}_{y,i,j} \in f_r,$$
$$S^k{}_{x,i,j} + ld_{ij} + \lceil \frac{p_k}{sp_{ij}} \rceil \leq S^r{}_{y,i,j} \vee S^r{}_{y,i,j} + ld_{ij} + \lceil \frac{p_r}{sp_{ij}} \rceil \leq S^k{}_{x,i,j}$$

The previous constraint is commonly found in scheduling problems and is managed by the global constraint disjunctive [92], with a plethora of dedicated algorithms to efficiently tackle it. Note this constraint is enforced in all decision variables of both primary and secondary flows.

Finally, since a frame is fully received before being copied to its destination queue and frames are transmitted according to their reception order, it must be ensured that packets addressed to the same egress link, arrive in the correct order when placed in the same egress queue or are forced to be placed in different queues. The latter is referred to as the frame isolation constraint, and is modelled by the disjunction depicted below:

$$\forall l_{ij} \in E, f_k, f_r \in F \vee k < r, \forall S^k{}_{x,i,j} \in f_k, \forall S^r{}_{y,i,j} \in f_r,$$
$$\vee Q_k \neq Q_r$$

where $A^k{}_{x,i}$ is the arrival time of the packet of flow f_k at node n_i, and is given by:

$$A^k{}_{x,i} = S^k{}_{x-1,a,i} + ld_{ai} + \lceil \frac{p_k}{sp_{ai}} \rceil$$

The definition of $A^r{}_{y,i}$ is similar. The constraint ensures that the reception of one of the frames is completed before the other, and thus, maintains the order of placing the frames in the queue. This constraint can be easily implemented using reified constraints, offered in most CP solvers.

The current model's implementation relies on the ECLiPSe Constraint Logic Programming system [93]; the latter supports all constraints required by the model, offers a variety of search strategies and variable/value ordering general heuristics for finding solutions.

5.4.2.1 Evaluation

A preliminary validation of the TSN schedule engine is conducted on the OMNeT++ v6.0.1 simulation platform, using the INET 4.5.0 framework. To this end, we utilize the topology illustrated in Figure 5-28. This topology includes paths between IoT nodes to VOs, as well as additional communicating nodes responsible for cross traffic. The traffic is forwarded via TSN switches that utilize GCLs at their egress ports in order to manage traffic prioritization. The IoT nodes are responsible for generating time-sensitive traffic, which is treated as high-priority (also termed as scheduled traffic). The period of all these flows is set to 800 μs, which is also considered as the deadline for each flow, thus, ensuring that

all packets are delivered to the VO within a cycle. Furthermore, nodes tagged as BE serve the purpose of traffic interference, by injecting Best-Effort (BE) traffic towards sink nodes. BE nodes generate traffic at intervals of 10 μs.

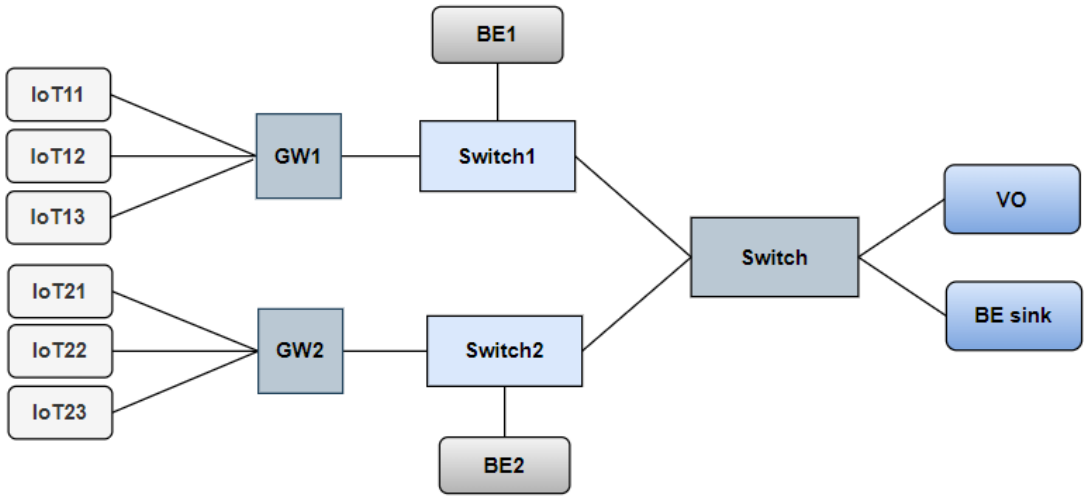


Figure 5-28: Evaluation topology

We utilize the aforementioned TSN schedule model to derive the GCL schedules and conduct a set of simulations in order to assess the efficiency of the computed schedules for high-priority traffic. The search is based on a value ordering heuristic, selecting the minimum value for the start time from each domain, thus, reporting a valid solution, which is not optimized according to some metric. This helps maintain a negligible runtime for schedule computation, (i.e., approximately 4 ms). As a baseline for evaluations, we rely on static schedules which have been employed by numerous studies for traffic prioritization [94], [95]. In such a static scheduling, the high-priority queue is associated with an 800 μs scheduling interval, whereas the best-effort queue is scheduled at 200 μs.

Figure 5-29 and Figure 5-30 illustrate the latency and jitter experienced by the scheduled traffic. The proposed scheduler (indicated as *"Dynamic_Scheduled"* in these plots) outperforms the static scheduler both in terms of latency and jitter. More precisely, our schedule model yields a median delay of 0.19 ms. The small size of the interquartile range indicates that a considerable number of latency values are tightly positioned around the median, which indicates a much lower degree of deviation from the median. This combined with the lack of outliers corroborates the superiority of the proposed scheduler, especially in terms of latency bounds which are more significant (than mean values) in the context of TSN. For instance, the latency with the proposed scheduler remains bounded below 0.25 ms (Figure 5-30), as opposed to the static schedules that may lead to delays in excess of 0.4 ms.

Similar observations can be drawn with respect to the jitter that the scheduled traffic experiences (Figure 5-30). Scheduling traffic with static intervals yields an increase by 80 in the median value of jitter (compared to our scheduler); however, the margin between the two scheduling techniques is even larger in terms of jitter bounds. While the deviation around the mean is barely perceptible for the proposed scheduler, jitter can exceed 0.1 ms with the static schedules.

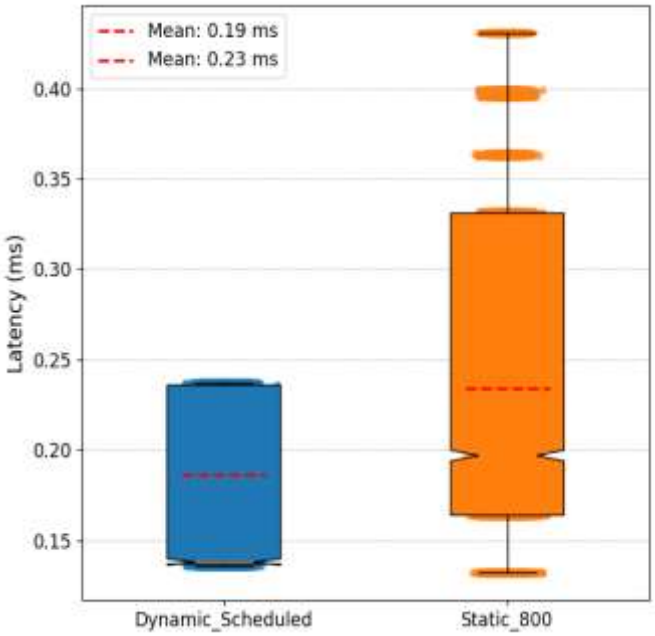


Figure 5-29: Latency of scheduled traffic

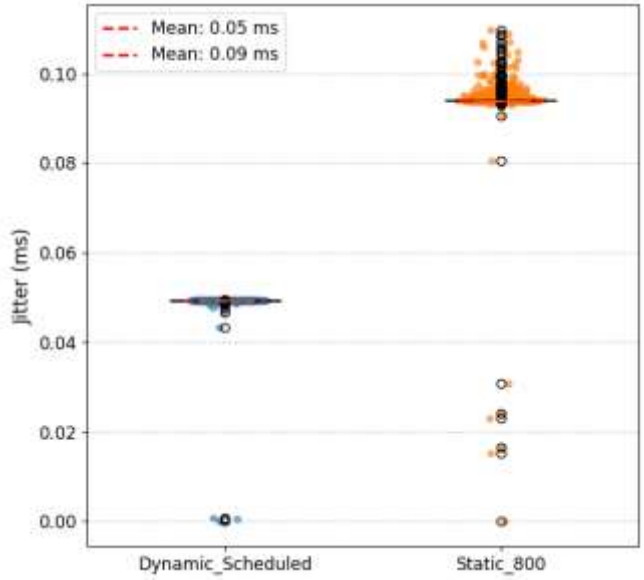


Figure 5-30: Jitter of scheduled traffic.

5.4.2.2 TSN Data Plane

For TSN data plane activation, we use TAPRIO (Time-Aware Priority Packet Scheduler) - a powerful queuing discipline available in the Linux kernel's traffic control (tc) tool. TAPRIO plays a crucial role in simulating the behaviour of IEEE 802.1Qbv, which is a standard for enhancing time-aware scheduling in Ethernet networks. By integrating TAPRIO, we allow the configuration of a series of gate states, each one responsible for enabling outgoing traffic for specific subsets of traffic classes based on the concept of time slices.

To ensure proper packet classification into the appropriate traffic class, TAPRIO uses the priority field of the socket buffer (*skb*) employed by the network stack of the Linux Kernel. This enables TAPRIO to effectively assign time-sensitive flows to their respective priority queues. In our implementation, we map traffic classes to queues by modifying the DSCP (Differentiated Services Code Point) field of the packet header. As such, we prioritise traffic based on specific service requirements and deliver the desired QoS to diverse types of data streams. To achieve the modification of the skb priority field before

packets are directed to the queuing discipline, we employ the use of *iptables*, a versatile packet filter tool operating at the IP layer. By incorporating the relevant classifier rules into iptables, we effectively manipulate the *skb* priority field with precision. As such, we establish the appropriate priority for the skb (socket buffer) as packets traverse the network. Through this comprehensive setup, we effectively integrate TAPRIO into the data plane of our IoT Gateway, enabling the timely delivery of data between IoT devices and (c)VOs.

More precisely, the workflow for packet handling and classification by TAPRIO is the following: (i) the incoming packet, which is tagged with a DSCP value of 0x40 indicating high priority, arrives at the ingress interface; (ii) the first step of classification involves using IPTables to set the skb priority field to 0x40; (iii) subsequently, the TAPRIO queuing discipline assigns the incoming packet to high-priority queues. The aforementioned steps are depicted in Figure 5-31.

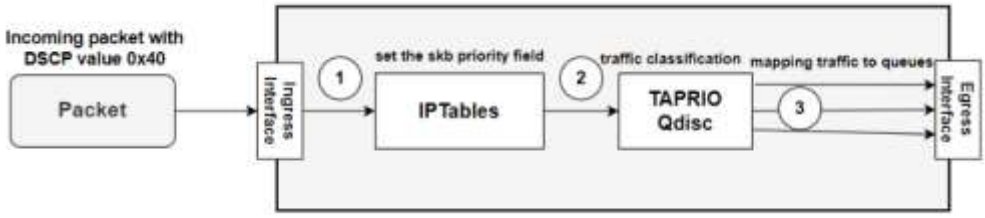


Figure 5-31: Packet handling workflow in TAPRIO

5.4.2.3 Interactions between TSN Control Plane and Data Plane

Leveraging on IEEE 802.1Qcc, we utilize a hybrid TSN implementation in order to automate the process of the TAPRIO configuration on the egress interface of the IoT gateway. In principle, CNC communicates with the TSN bridges via remote network management protocols such as NETCONF, RESTCONF and IETF YANG data models. In the case of a client/server-based network management protocol architecture, the TSN bridge acts as a management server, whereas CNC acts as a management client.

In the TSN architectural framework illustrated in Figure 5-32, a YANG Parser, deployed at the userspace of the TSN bridge, parses the YANG-TSN model to a set of actions that can be applied directly to the queuing disc layer of the Linux kernel. The CNC establishes communication through the NETCONF plugin by utilizing the YANG data model. The NETCONF plugin functions as a management client and establishes communication with the NETCONF server that is operational on each TSN bridge, such as an IoT Gateway. Following the completion of their computational process, the TSN schedules are transmitted to the IoT gateway.

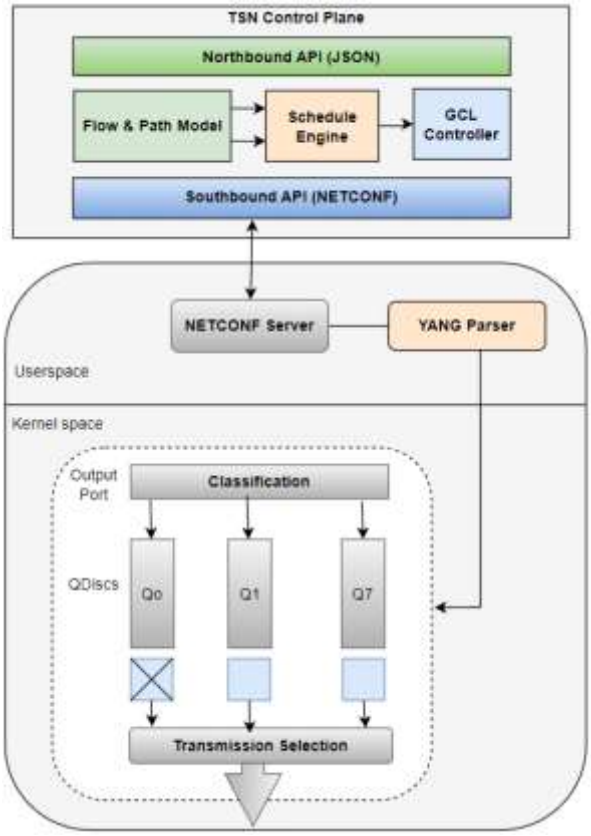


Figure 5-32: Interaction between the TSN data and control plane

An illustrative sample of this model appears in Figure 5-33. In particular, this figure illustrates the *TSN-TAPRIO* module for transmitting the TAPRIO configuration to the relevant network interface via an RPC. This module allows for the definition of the interface through which the TAPRIO configuration will be activated, as well as the specification of the number of traffic classes (referred to as *num-tc*) and the number of hardware queues to which the traffic classes will be mapped. Furthermore, the Gate Control List schedule can be defined using the *sched-entries* parameter, which allows for the configuration of time intervals. These intervals determine the length for which each scheduled entry will be active before transitioning to the next entry.
The TSN control plane codebase for the VOStack is available here[35].

[35] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-tsn

```
module: tsn-taprio
  +--rw tsn-taprio-structure
     +--rw interface* [dev]
        +--rw dev                string
        +--rw parent?            string
        +--rw handle?            uint32
        +--rw num-tc?            uint8
        +--rw map?               string
        +--rw queues
        |  +--rw queue* [id]
        |     +--rw id       uint32
        |     +--rw elem?    string
        +--rw base-time?         uint64
        +--rw clockid?           string
        +--rw sched-entries
           +--rw sched-entry* [id]
              +--rw id            uint32
              +--rw command?      string
              +--rw gatemask?     string
              +--rw interval?     uint64
  rpcs:
    +---x taprio-set
       +---w input
       |  +---w command?    string
       +--ro output
          +--ro result?    boolean
```

Figure 5-33: Example of TSN-TAPRIO module

6 Security Functionality

6.1 Decentralized Identifiers

Decentralized Identifiers (DIDs) [96] are unique identifiers that enable verifiable, self-sovereign digital identity. They are crucial components in a decentralized identity system, allowing individuals, entities, and things to have a persistent, globally unique identifier independent of any centralized authority. DIDs are foundational in enabling secure and private interactions in digital environments.
As shown in Figure 6-34, DIDs are divided in 3 parts:

- The **URI scheme** identifier: this is the "did:" at the beginning of the identifier that serves as a Uniform Resource Identifier (URI) scheme that indicates it is a Decentralized Identifier.
- The identifier for the **DID method**: this part specifies the method used to create, resolve, and manage the DID. It indicates the underlying decentralized network or system. Each method may have its own rules and processes for creating and managing DIDs.
- The DID method-specific **identifier**: This is a unique identifier assigned to a particular entity or individual within the chosen DID method. The format and length of the specific identifier may vary depending on the method. It uniquely identifies the subject of the DID within the chosen decentralized network or system.

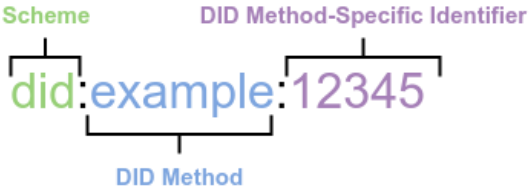


Figure 6-34: Example of a Decentralized Identifier (DID)

As depicted in Figure 6-35, DIDs resolve to DID documents, which are JSON-LD [97] representations associated with DIDs. They contain key information about the subject of the DID, including public keys, authentication methods, service endpoints, and additional metadata. DID Documents serve as a crucial component for verifying and interacting with decentralized identities within a given DID method. They enable secure and standardized management of identity-related information.

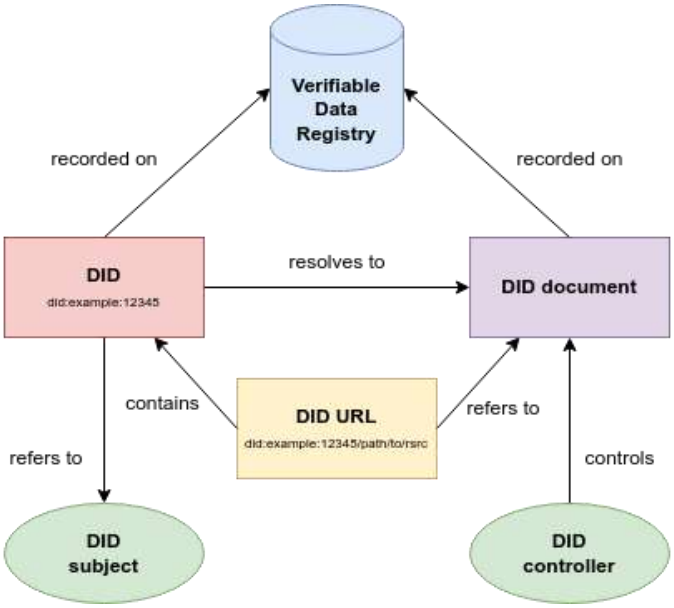


Figure 6-35: Overview of DID architecture.

6.2 Verifiable Credentials and Verifiable Presentations

Verifiable Credentials (VCs) [98] provides a standard way to express credentials on the Web in a way that is cryptographically secure, privacy respecting, and machine verifiable.
A VC can represent all of the same information that a physical credential represents. The addition of technologies, such as digital signatures, makes VCs more tamper-evident and more trustworthy than their physician counterparts.
Holders of VCs can generate Verifiable Presentations (VPs), which are digital artefacts that encapsulate VCs and information about a subject, such as an individual/entity or devices. They are often used in the context of decentralized identity systems and are cryptographically signed, allowing third parties to verify the authenticity of the presented information without needing to contact the original issuer. VPs enhance privacy and enable individuals to selectively disclose specific details about themselves while maintaining control over their personal data. They play a key role in the secure exchange and verification of credentials within decentralized identity ecosystems. Both VCs and VPs can be transmitted rapidly.

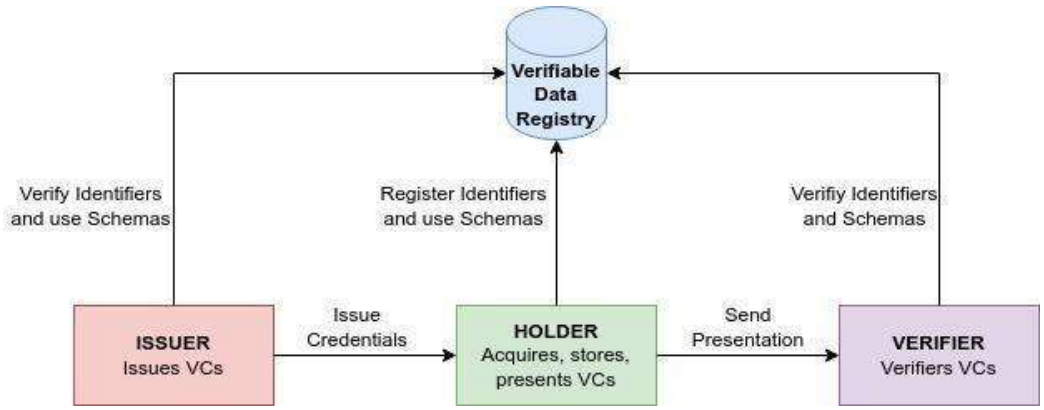


Figure 6-36: Verifiable Credential flows

6.3 Security Architecture

For the Southbound interface (VO-to-Device), security measures will be robustly implemented through the utilization of MQTT with TLS (Transport Layer Security) [99] and CoAP with OSCORE [100]. These protocols ensure a secure and encrypted communication channel between devices and systems. MQTT with TLS guarantees data integrity and confidentiality, while CoAP with OSCORE provides a secure framework for constrained environments, ensuring the protection of data exchanged over the network. This dual approach enhances the overall security posture, addressing the specific needs of diverse communication protocols in the southbound interface.
For the Northbound interface (VO-to-cVO, VO-to-App and cVO-to-App), the architecture is based on the utilization of **Holder, Verifier, Issuer and PEP-Proxy** components. These components are deployed within Kubernetes on every element (Applications, cVOs, and VOs). Additionally, there is the option of deploying them on IoT Devices if the limitations of the devices allow it.
The Application will only deploy the Holder component because the interaction flow is always VO-to-App, i.e., the App will never receive requests from VOs, so the App will never have to verify credentials to produce an Access Token. VOs and cVOs will deploy all 3 components because they will have to both Verify and Authenticate depending on the type of interaction that is taking place (Direct VO-to-VO, VO-to-IoT-Device(s), or VO-to-App). The IoT Devices, if the conditions are met, will deploy just the Holder component for the same reason as the Application does, but in this case the interaction is VO-to-IoT-Device(s).
The **Holder** is the component that will function as a wallet. The Holder functionality is to obtain the SIOP request, select the Verifiable Credentials indicated in the SIOP Request [101], create the VP and send it to the correspondent API (also indicated in the SIOP Request) to be verified and to obtain the

Access Token (OIDC4VP Protocol) [102]. As it acts as a wallet, the Holder also stores the VCs and the DIDs.
The **Verifier** is the component that will generate the SIOP Request and then verify the credentials to generate the Access Token. The SIOP Request is a request that is sent by the relying party (in this case, the Verifier) to an individual's self-issued OpendID [103] Provider asking for an OpenID Token. The request includes information about the relying party and the specific claims or information it requires from the Holder. The OpenID Token is composed of 2 parts: the ID Token and the Verifiable Presentation Token (VP Token), and both are generated by the Holder. The Access Token is a Bearer Token in JSON Web Token (JWT) [104] format and is signed with the private key of the device where the Verifier involved is deployed. This access token includes information about the authenticated entity or the context of the token such as the issuer, the subject, expiration time, etc.
The **Issuer** is the component that will function as the trusted entity for the generation of VCs. All Apps/(c)VOs that want to obtain Verifiable Credentials will need to execute the OpenID Connect for Credential Issuance (OIDC4CI) protocol [105] with the Issuer. Unlike the other components, the Issuer is not deployed in every App/(c)VO/Device. The Issuer will be deployed as an independent component.
The **PEP-Proxy** component acts as an entry point for the (c)VOs. This component will offer the same APIs (including Verifier/Holder APIs) as those offered by the (c)VOs on which the proxy is deployed, so the PEP-Proxy just redirects the request to the corresponding component, thus ensuring that the (c)VOs and all of its components are not exposed and can only be accessed through the PEP-Proxy. This component is also responsible for verifying the Access Token when receiving a request to a resource with the Access Token included in the 'x-auth-token' header.
All these addresses the security-related functional requirements FR_VOS_009 from Table 2.1 and non-functional requirements NFR_VOS_02 and NFR_VOS_06 from Table 2.2.

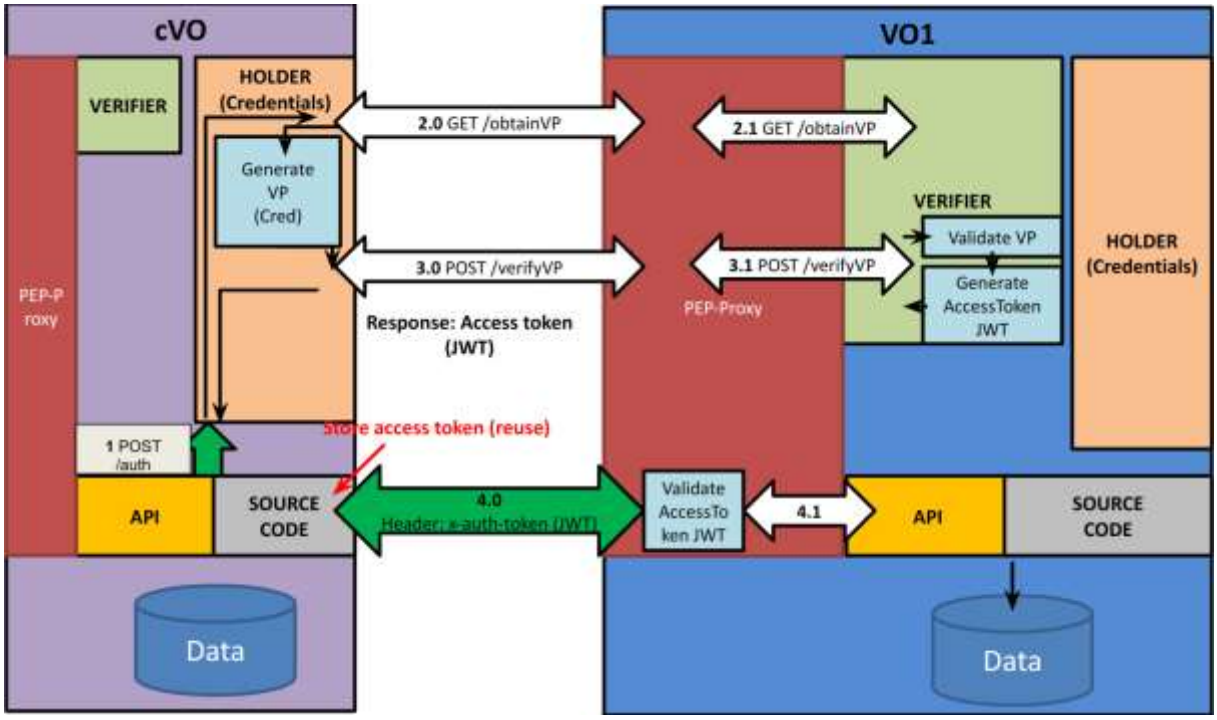


Figure 6-37: Security components deployed for an interaction cVO-to-VO.

7 IoT Device Virtualized and Supportive Functions

This task aims to develop the set of functions that will be supported by the "IoT Device Virtualized Functions" and the "Generic/Supportive Functions" layer of VOStack. The IoT Device Virtualized Functions Layer refers to functions that tackle part of the business logic of an application, while the Generic/Supportive Functions Layer considers a set of supportive functions that can be horizontally applied over all the instantiated VOs for an application. At the Generic/Supportive Functions Layer, a service mesh approach is going to be followed, where development of generic supportive functions (e.g., elasticity management, security, authentication, telemetry) is going to take place. Secure communication and authorization are going to be supported, as well as enforcement of authentication policies. Telemetry functions as well as elasticity management actions for IoT application components will also be made available. NEPHELE follows a microservices-based approach with cloud-native applications being represented as an application graph made of independently deployable application components. This guarantees modularity, openness, and interoperability with orchestration platforms in general and with the NEPHELE Meta-orchestrator in particular. The deployable components are application functionalities, i.e., supportive functions and IoT device virtualized functions, which can be deployed at the cloud or the edge part of the continuum. The VOStack offers them as generic functionalities that can be tailored to the specific application needs. Indeed, the components in the application graph are accompanied by a sidecar as per the service-mesh approach to be activated on demand. The VO representing an IoT device will offer the IoT Virtualized functions that it supports to the application, whereas other supportive functions can be activated as part of the application graph. The NEPHELE hyper-distributed applications (HDA) repository, will host a set of application graphs, application components, virtualized IoT functions and VOs to application developers. The NEPHELE meta-orchestrator is then responsible for activating the appropriate orchestration modules to efficiently manage the deployment of the application components across the continuum. The interplay among VOs and IoT devices will allow for exploitation functions even at the device level in a flexible and opportunistic fashion.

Drivers for the definition of the IoT virtualized functions and the supportive functions are the NEPHELE use cases. These cover a large spectrum of IoT devices, applications and services and will require distinct functions to be supported. Some of the IoT devices and functionalities are specific for a use case, whereas others show commonalities or even equalities over multiple use cases. For instance, ground robots and an ultrasound probe are specific IoT devices for a post-disaster use case and e-Health use case respectively (NEPHELE use case 1 and 4), whereas cameras and sensors see adoption in multiple of the NEPHELE use cases. Similarly, robot navigation and medical report production are supportive functions that are specific for NEPHELE use case 1 and use case 4 respectively, whereas object detection and data aggregation are adopted in multiple use cases. We will next elaborate on the IoT Virtualized Functions and Supportive functions identified by the NEPHELE use cases as representative examples. For a complete and detailed overview on the use case definition and requirements please refer to NEPHELE D2.1.

In Use case 1, a post-disaster scenario is envisaged, and the hyper-distributed application aims at enhancing the situational awareness for first responders. To reach this goal, sensing data collected from the IoT devices will be used for instance to enable, among others, AI-powered decision-making, path planning, and precise 2D/3D representations of emergency scenarios in real-time[1]. Several IoT device virtualized functions and supportive functions are identified for the post disaster use case where the following service will be enabled:

- Fleet management software component to enable and control multiple robots simultaneously.
- Navigation and mapping software component to enable mobile robots to navigate autonomously the area and map it using multiple robots and map merging functionalities.
- Software component for location and identification of victims in unknown areas, and assessment of victims' injuries.
- Risk detection component to identify areas with dangerous and risky elements using sensor data analysis and computer vision.

- Pick and place software components for a mobile robot to deploy sensor devices and take liquid samples.
- Interactive GUI component for the presentation tier including a mission control to list tasks based on data collected from the IoT devices.

Some Generic/Supportive functions that can be horizontally applied over all the instantiated VOs for an application will be:

- Security, authentication specifically described in section 6.
- Telemetry and monitoring components to verify the status of network connectivity, sensor, robots' status, and trigger actions in case of low energy or disconnections.
- Storage and replay software component for historical analysis of robot actions and performed tasks.
- Data aggregation.
- Elasticity management for VO deployment in a Kubernetes Cluster using CRDs and rules for horizontal/vertical scaling.
- Alarms.
- Image processing.
- Video streaming selection.

In Use case 2, containers routing optimization in a smart (sea) port is explored and the hyper-distributed application aims at automating the containers routing process. For this purpose, data collected from IoT sensors installed in port trucks/forklifts, video streams from cameras installed within the port area, and data collected from Port Information System will, supported by AI/ML powered image processing and decision-making, allow for optimizing containers routing process in real time. Several IoT device virtualized functions and supportive functions are identified to be utilized in the use case:

- supporting multiple types of sensors (data format, semantics, single vs. multiple sensors connected on single bus)
- manipulating sensors properties
- IoT device monitoring
- (AI/ML) image detection/processing (described in *Image processing* section of this chapter)
    - detecting obstacles (e.g., traffic jam)
    - detecting free parking spaces (load/unload area)
- simulation tools for ML algorithms training
- interfaces for collecting data from Port Information System
- container routing optimization
- dispatch decision making.

7.1 Telemetry

Leveraging on the Observe primitive defined by OMA-LwM2M, VO and cVO implement telemetry mechanisms in order to automatically notify resource's status change to the data consumer, the Observer. The VO receives, on the northbound interfaces, the OBSERVE request from an application to observe a Resource or Object instance. When an entity is under observation, the observer is registered in a list dedicated to entity observation, Resource, or Instance. The VO uses this list to notify the application of the new value from the observed entity. In this way, a new observer for the same entity will be added to the list without the need of further operations.

Data Storage

VO enables different type of data storage according to the nature of data. The datastore can be both internal, through the implementation of lightweight solutions (i.e., SQLite) for the historicization of data less linked to frequent temporal updating, and external, through the implementation of more high-performance solutions suitable for the processing of large quantities of time-related data (timeseries) with high update frequency (i.e., InfluxDB). Such scalable and stateless configuration, with respect the VO status, is a key feature for seamless orchestration and elasticity management of the VO microservice within the NEPHELE VOStack.

7.2 Data Aggregation

Through the NorthBound, the VO provides the interfaces for directly accessing the history of the data recorded in the chosen period. These interfaces, exposed on the api/data path, will allow:

- data extraction by type of resource, instance/object or object.
- data extraction by value (applicable only on resource).
- time frame definition for data extraction.

7.3 Elasticity Management

Elasticity is the ability to grow or shrink infrastructure resources dynamically as needed to adapt to workload changes in an autonomic manner, maximizing the use of resources. A VO can be dynamically created and destructed, may consist of information, services and is a dynamic object since it has to represent dynamically the changes from real world objects. Cognitive mechanisms at the VO level enable self-management and self-configuration of real objects, in fact, the introduction of cognitive mechanisms could lead to know how real-world objects react to specific situations and in this way the operations for controlling objects become more efficient. The VO level would also involve mechanisms that provide awareness about the physical objects' presence and relevance, a complete knowledge about physical objects is needed to keep the association between the VOs and the real objects. This knowledge is used to have a constant handling of physical devices without caring about if there's physical object mobility or failure of some of the links to them. In other words, the VO level comprises mechanisms for monitoring the status/capabilities of physical objects and controlling the various links with physical objects to make sure that the VOs are resilient, even when the associated physical objects might be temporarily unavailable [106].

7.4 Alarms

Through the use of semantic models like OMA-LwM2M and W3C-WoT, the VO exposes thresholds and related alarm resources. This allows, for instance, consumers to monitor the resource trend and, once the alarm resource has been monitored, receive notification of any anomalous trends. In order to make the alarm manageable, OMA-LwM2M objects must contain three specific OMA-LwM2M resources within their models:

- Alarm State, ID 6013, represents the True/False alarm status.
- Alarm Set Threshold, ID 6014, represents the threshold level to be set which will be used as a parameter for activating the alarm. This resource is enabled for both reading (READ) and writing (WRITE).
- Alarm Set Operator, ID 6015, is a readable and writable resource used in conjunction with the Set Threshold to represent when an alarm is triggered. This resource must be set to one of the following values:
    - Greater than or equal to, that is, the alarm will be triggered when the sensor value is greater than or equal to the threshold.
    - Less than or equal to, that is, the alarm will be triggered when the sensor value is less than or equal to the threshold.

7.5 Image Processing

Image processing for object and person recognition offers numerous benefits in various applications. It enhances automation and efficiency in industries like surveillance, healthcare, and autonomous vehicles by enabling real-time identification and tracking. This technology also aids in security and safety, as it can detect intruders, monitor access points, and identify missing persons. In medical imaging, it assists in diagnostics and treatment planning. Moreover, it improves user experiences in applications like augmented reality and gaming. Additionally, image processing for object/person recognition contributes to data analysis, enabling businesses to gain valuable insights from visual data, leading to better decision-making and improved customer experiences. Several approaches have been proposed by the literature [107]–[109]. The main categorization of these approaches is whether they fall into an

individual object detection category, where the objects are identified and localized separately, or into density estimation category, in which the texture of the frame depicting a group or a crowd is analysed to conclude an approximate density estimation, utilizing regression techniques. Most object-detection algorithms are based on feature extraction and identification. In particular, they utilize trained descriptors, which have been inferred using other images of similar objects as a training set as shown in the figure below. Later, during the inference phase, the algorithm looks at whether these features are present in the frame being examined. On the other hand, density estimation approaches are based on analysing the "microstructures" of the image to produce an estimate of the density of the people being depicted. The advantage of the later approach is that it can manage scenes, in which the individual objects are occluded from other ones. In such a scenario, classical image processing approaches, which try to localize them as a whole using pre-trained descriptors, fail, as the majority of the features are occluded. This is mostly the case, when we are dealing with crowded scenes with a lot of objects/persons occluding one another. As a result, we can follow a deep learning approach, which, based on convolutional features extracted from convolutional layers are used to render a density map the integration of which creates an estimate of the number of objects/persons apparent in the scene.

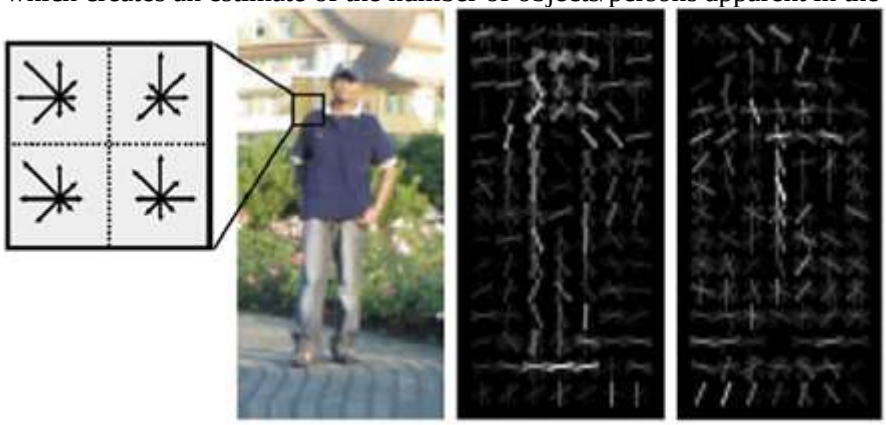


Figure 7-38: Descriptors extracted histogram of oriented gradients (Source: [26])

In conclusion, image processing for object and person recognition offers a wide array of benefits, from enhancing automation and security to improving healthcare and user experiences. Its applications span across various industries and contribute to data-driven decision-making, making it a valuable and versatile technology with significant potential for innovation and improvement in our increasingly visual world.

These features, together with SQL-like and non-relational access to the database and the implementation of more web-oriented protocols for data sharing and other implemented solutions, enable the VOS to be defined as an enhanced and enriched digital extension of the physical device to the internal network infrastructure, VOStack, capable of enabling innovative function management logics with a view to developing new future generation networks which will allow full integration of the virtual space of services with the physical space of devices.

8 Orchestration Management Interfaces

In NEPHELE, major importance is given on the convergence of IoT-based technologies with edge and cloud computing orchestration technologies. The objective is to support the end-to-end orchestration of distributed applications across the computing continuum in a unified way. Such applications are represented in the form of an application graph and may include application components as well as VOs and cVOs. All of them are represented in the form of microservices that are interconnected among each other. Thus, we can speak about an application graph with application components and dependencies among them. The dependencies are represented in the form of virtual links. In Figure 8-39, we depict a high-level representation of the proposed approach where application graphs may be interlinked with VOs and cVOs based on well-defined HTTP-based interfaces.

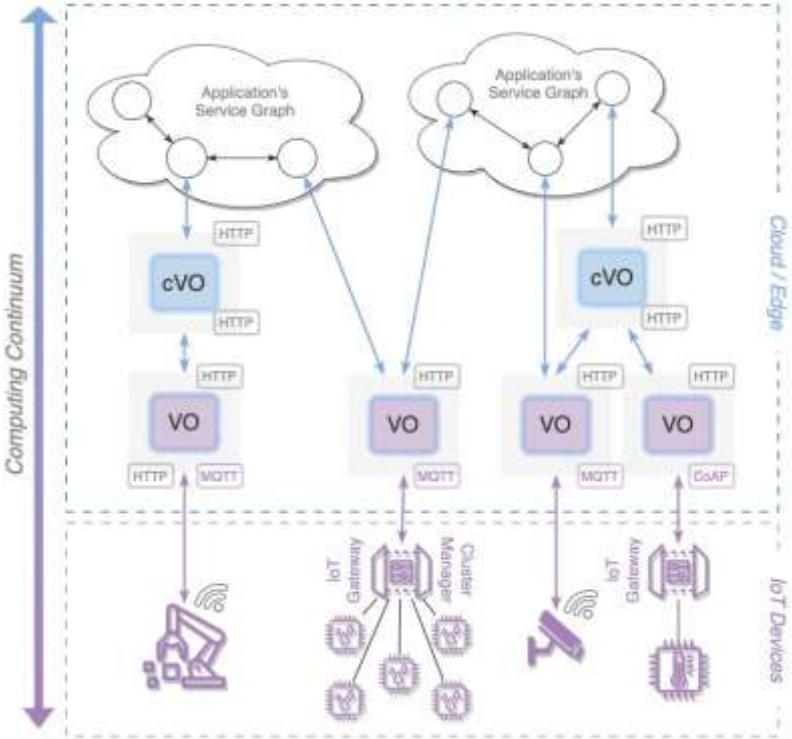


Figure 8-39: High-level view for the VO positioning in the computing continuum

In Figure 8-40, a more detailed representation of an indicative application graph is presented, where application components, VOs and cVOs are accompanied with a set of metadata for declaring deployment preferences and constraints.

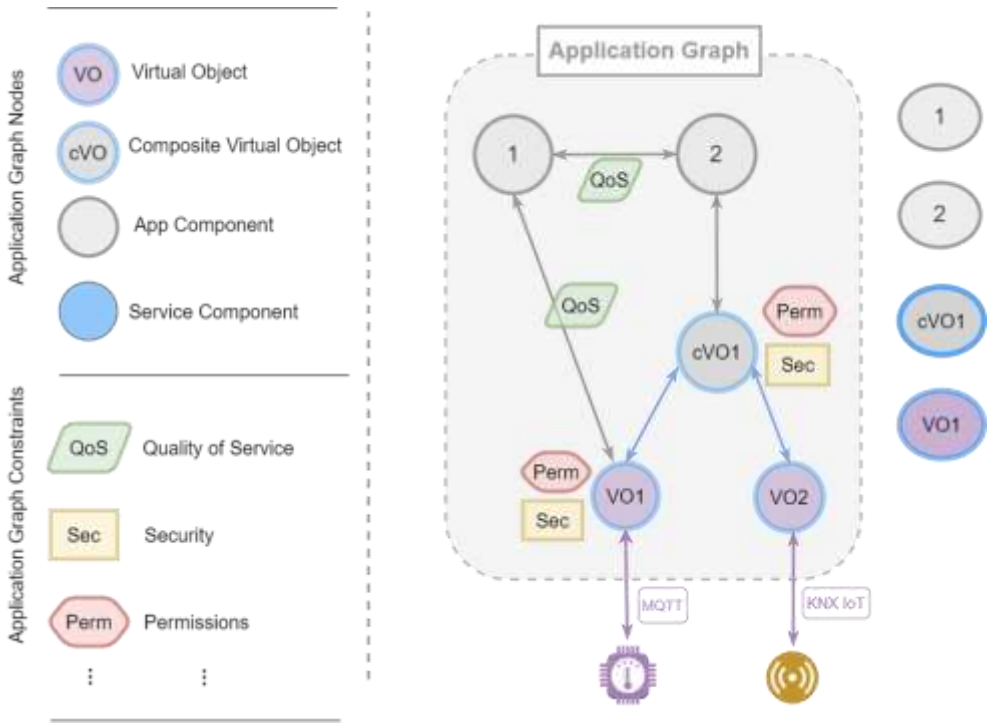


Figure 8-40: Application graph example with example constraints

Therefore, we consider that the (c)VO is an integral part of a distributed application graph and, thus, manageable by cloud/edge computing orchestration mechanisms. Each VO can be independently orchestrated as a part of a hyper-distributed application. As a result, the (c)VO interacts with applications that require services from the VO (e.g., APIs to support the interconnection of IoT application graph components with (c)VOs). Moreover, the (c)VOs interact with the respective Orchestration allowing basic operations i.e., (a) monitoring (e.g., status of a (c)VO), (b) scaling (e.g., assign more resources to a (c)VO), (c) lifecycle management (e.g., data required for the deployment of the (c)VO are stored in the VO database and exposed to the orchestration platform). Moreover, the Orchestration platform can execute health checks to the VO and the devices using the respective generic function for monitoring reasons, for example triggering alerts when needed. All these addresses the non-functional requirements NFR_VOS_07 and NFR_VOS_10 from Table 2.2.

9 Intelligence on IoT Devices and interplay with Virtual Objects

This task aims to facilitate on-device intelligence, which supports seamless interaction between the VOs and the physical IoT devices. In particular, the task involves implementing IoT functions using TinyML and complex event processing (CEP), extending existing VOs with additional features and services. Deploying CEP and TinyML on IoT devices can enable on-device, real-time, context-aware decision-making in response to the streaming data generated by these devices. Efficient deployment and execution of these IoT functions are the focus of the task. The close collaboration with Task 4.2 ensures that these developments align seamlessly with the VOStack, promoting consistency, synergy, and interoperability throughout the framework. In this section, we address the non-functional requirements NFR_VOS_07 and NFR_VOS_09 from Table 2.2.

An IoT system encompasses a vast network of equipment with sensors installed, from smart building to factory motors, connecting all these assets to the digital realm. Through the pervasive deployment of sensors, an extensive stream of data is continuously gathered and shared across the network, denoted as big data. However, the significance of big data is not solely a function of its volume; rather, its value lies in how effectively it is managed and harnessed. The information needs to be analysed and combined with background knowledge to make quick decisions continuously and quickly, allowing for real-time monitoring and in-depth analysis.

In the conventional compute-centric paradigm, raw data is continually transmitted from the IoT devices to the cloud for centralised processing. However, as IoT devices become more prevalent, concerns are raised. Transferring a large amount of data in this manner is not only energy-intensive but also susceptible to attack and subject to high latency. As sensors and IoT devices become more cost-effective and powerful, a shift towards edge computing is observed, where data processing occurs at the source, directly on edge IoT devices. This novel approach is often referred to as the "data-centric paradigm." By enabling intelligence at the edge, we process streaming data on IoT devices and send only necessary information to the cloud, reducing communication costs, minimising latency, and safeguarding data privacy. This approach offers real-time distributed data analysis, especially valuable in complex environments, opening new horizons for applications.

However, it is important to acknowledge the challenges posed by IoT devices, which are designed for longevity with limited resources and a focus on low power consumption. Many IoT devices operate even without an operating system, making deploying intelligence (so-called IoT functions) in decentralised sensor networks a complex endeavour. The situation is even more challenging when various on-device intelligence (TinyML and CEP in this task) are deployed across decentralised IoT networks. Key questions arise, such as how to deploy IoT functions to devices seamlessly, how to discover the deployed IoT functions in a unified way, how to retrieve the results from these functions, and how to coordinate services across the platform efficiently.

9.1 Approach

In response to these challenges, this task focuses on deploying intelligence on IoT devices with interoperability and management in mind, which is tightly coupled with Task 4.2. Specifically, we introduce two techniques, CEP and TinyML, and explore their powerful synergy, enabling on-device processing and effectively reducing the need for extensive data transmission. Furthermore, this task proposes a semantic-based workflow for the scalable management of on-device intelligence across distributed IoT devices within the VOStack, building upon Task 4.2.1 and 4.2.2.

9.2 Complex Event Processing

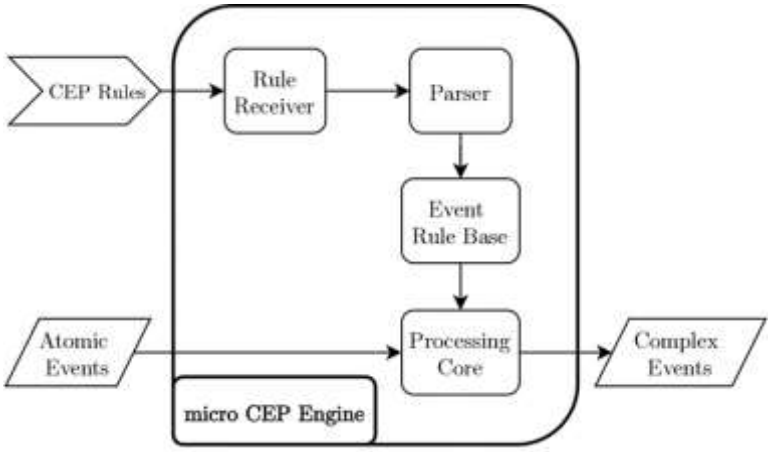


Figure 9-41: System Design of the µCEP Engine

CEP is a widely used method for analysing and identifying patterns within heterogeneous streaming sources. Users can define logical rules to specify the desired sequence and patterns in the streaming data. The CEP system then continuously monitors incoming data against these rules and generates notifications (complex events) when a match occurs, be it a special incident or an error in the system. This task develops an µCEP engine capable of processing streaming events directly on resource constrained IoT devices with high throughput and minimal latency. This empowers on-device monitoring, making it a valuable addition to IoT systems. The engine, depicted in Figure 9-41, requires as little as 20 KB of memory. It comprises four components: a rule receiver, a parser responsible for interpreting the rules, an event rule base that manages these rules, and a processing core.

Several types of data, including sensor readings and inference results, can be fed into the µCEP engine as "atomic events." Upon arrival, the engine autonomously analyses the temporal and spatial relationships between these atomic events. When incoming events match with predefined patterns, the engine instantly generates results in the form of complex events. To ensure efficient operation, the µCEP engine maintains a running buffer for events and automatically discards events that do not fit within the scope of the stored rules. The engine is designed to follow the language derived from ETALIS, outlined in [64]. More explanation and examples of the language can be found in [63].

A noteworthy feature of the engine is its ability to modify rules on the fly, allowing the system to update CEP rules during runtime. This empowers users to adapt reasoning logic on the fly, enhancing VOs with flexible monitoring functionalities. The µCEP engine not only enables on-device monitoring but also facilitates data management on IoT devices. With the µCEP engine, the massive data on IoT devices can be filtered, aggregated, and compressed based on user definition. This process extracts essential information before transmitting data to the cloud, reducing communication overhead, and improving application efficiency.

9.3 TinyOL (TinyML with Online Learning)

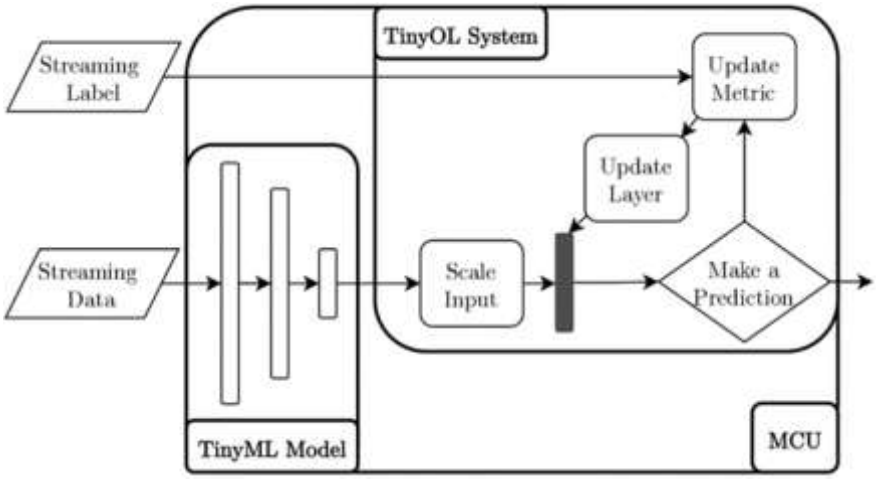


Figure 9-42: Building blocks of TinyOL

TinyML is an emerging AI field focusing on deploying NN at the edge. Currently, many TinyML solutions like TensorFlow Lite Micro and MicroTVM follow the traditional approach. They train models offline on powerful machines or in the cloud, which are then loaded onto IoT devices for inference. However, once deployed, these offline-trained models become static and may not maintain their effectiveness in changing real-world environments, a problem known as "concept drift and data drift." The situation implies that the statistical properties of field data can change over time, which the offline-trained models cannot foresee during their initial training. Consequently, these models may perform arbitrarily poorly after deployment.

This task proposes a solution called TinyML with online learning (TinyOL) that builds upon existing TinyML models [110]. TinyOL allows for incremental on-device post-training directly on IoT devices. The system, illustrated in Figure 9-42, is implemented in C/C++. The principal component of TinyOL is the layer highlighted in grey in the figure, which can be added to any existing NN as an additional layer in IoT devices. In addition to performing inference, TinyOL enables on-device training by leveraging online learning. Typically, offline-trained models remain fixed once uploaded in the Flash memory of an IoT device. However, TinyOL can train the added layer as it operates in RAM. With online learning, TinyOL enables NNs to learn from massive streaming data one piece at a time without retaining historical data. As a result, the models remain adaptable to changing conditions after deployment and can address "drifts" effectively.

New sample data flows through the existing NN at each step and is subsequently fed into TinyOL. Depending on the specific tasks, the system updates the accumulated mean and variance, offering the option to standardise the input data. Subsequently, the system performs an inference. If a corresponding label is available, the evaluation metrics and the additional layer's weights are adapted using online gradient descent algorithms, e.g., stochastic gradient descent (SGD). Thus, the training and prediction steps are interleaved. Once the neurons in the added layer are updated, the data pairs can be discarded effectively. Only one data pair resides in the memory at a time. This design minimises memory usage, making it well-suited for IoT devices with limited resources.

TinyOL empowers IoT devices to undertake AI/ML tasks directly within dynamic environments. This includes activities such as object and human detection (Use Case #1) and prediction of future values related to risks, motion (Use Case #2), or environmental conditions (Use Case #3). By computing data locally instead of streaming it to the cloud, TinyML offers computational efficiency, lower latency, reduced communication overhead, and improved privacy.

9.4 Interplay with VOs

In section 4, the W3C WoT TD and its TM are employed to create a universal framework for modelling and managing IoT devices and their on-device functions at scale. Heterogeneous knowledge about IoT components can be semantically rendered in a unified language and centrally hosted in a graph database. The Thing Model concept transforms VOs into digital twins, providing a simplified representation of

IoT devices and their numerous on-device functionalities. This permits easy discovery, offerings management, and the orchestration of existing components across the IoT network, enabling vendor-agnostic software-hardware co-management and reasoning over data. Furthermore, the knowledge about the IoT devices and their on-device functions, such as CEP rules and TinyML models, will be systematically refreshed within the knowledge database and exposed to VOs upon update. This streamlined process ensures that VOs are constantly up to date with the latest information about IoT devices and their embedded functionalities.

9.5 Current Status

We have developed the uCEP and TinyOL engines. The next step is already on the way, where we are going to improve the efficiency of both components, benchmark their performances on the TargetV devices, and integrate them into the use cases.

As shown in Figure 9-43, we have successfully ported the µCEP engine onto the TargetV device. Additionally, the processes including the integration of the TinyOL engine are under development, which is marked with dashed lines in the figure. Taking the project to the next level, we used the open source coreMQTT library that empowers the TargetV device with MQTT client functionality for seamless communication. With the coreMQTT library in place, the TargetV board gains the ability to acquire new rules and TinyML models during runtime and instantaneously dispatch the results over MQTT.

The TargetV device can now be connected to a WoT Servient, creating a bridge between the physical world and the digital realm. WoT Servient exposes a generic REST interface, allowing pushing the new rules and TinyML models and getting the generated values via HTTP. The ease of access to these values through HTTP enhances the system's versatility, making it easier to integrate with a wide array of external services and applications.

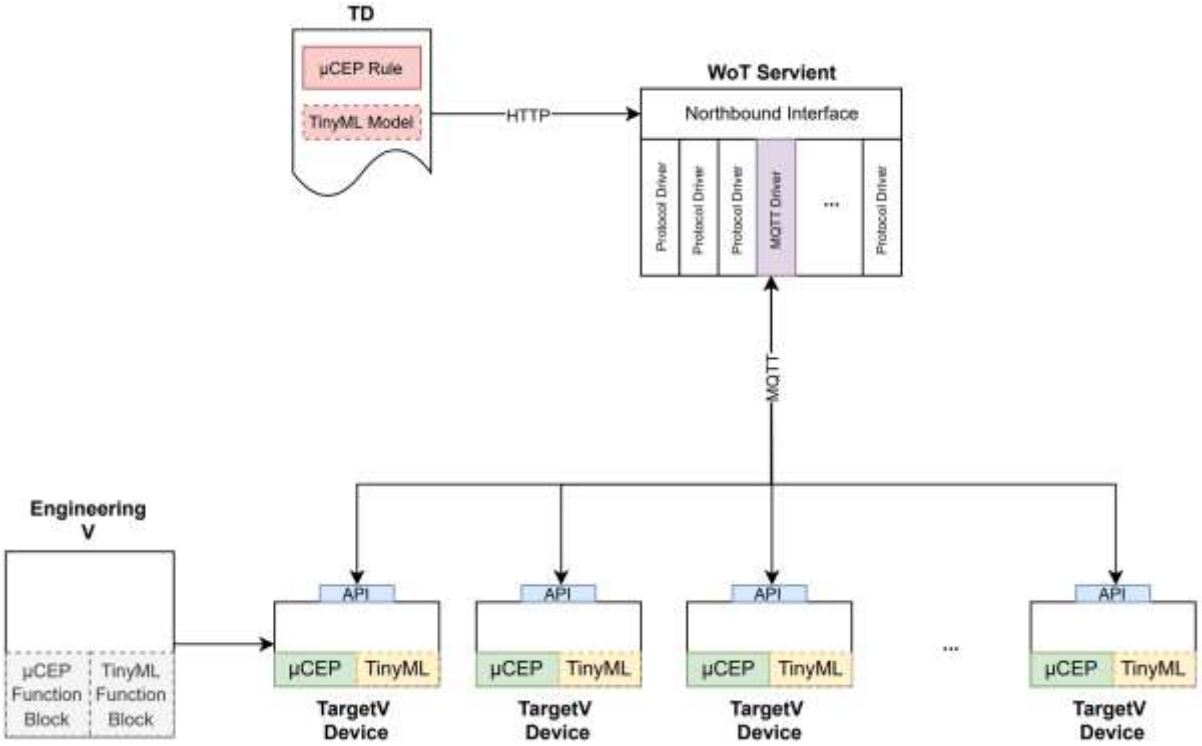


Figure 9-43: The overview of the current implementation status

10 VOStack Implementation and Open-Source Activities

The following sections describe the VOStack implementation based on W3C WoT and OMA-LwM2M models.

10.1 VO alignment with W3C

This section presents the alignment with the W3C Web of Things (WoT). The complete documentation of this implementation can be found here[36]. Also, the related codebase is provided here[37].

Based on the W3C WoT, the Exposed Thing (ET) is a software abstraction that represents a locally hosted Thing that can be accessed over the network by remote Consumers. Similarly, a Consumed Thing (CT) is a software abstraction that represents a remote Thing used by a local application.

In NEPHELE we consider two diverse types of deployments for the VOs according to the capabilities of the devices:

- In case of devices with some computing capabilities (e.g., drones, robots, raspberry PIs), we consider that the device can run a WoT runtime to communicate with the VO. As a result, the VO has both an ET and a CT to expose the virtualized device to cVOs or other Consumers as shown in Figure 10-44.
- In case of limited resources, we consider that the device directly communicates with the ET of the VO, as shown in Figure 10-45.

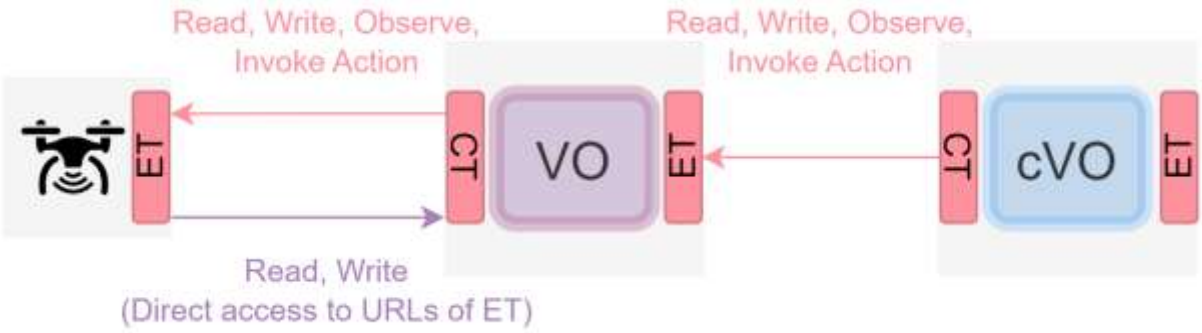


Figure 10-44: VO deployment based on W3C WoT in case of device with computing capabilities.

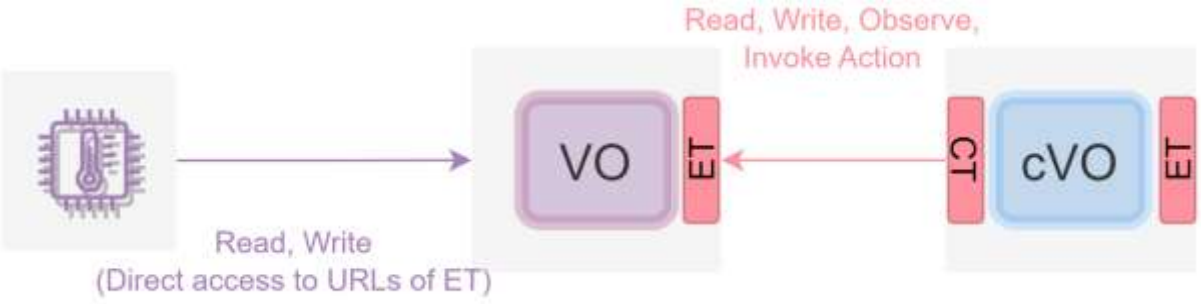


Figure 10-45: VO deployment based on W3C WoT in case of device without computing capabilities.

VO Technology stack

[36] https://netmode.gitlab.io/vo-wot/index.html

[37] https://gitlab.eclipse.org/eclipse-research-labs/NEPHELE-project/vo-wot

This section explains the technology stack of the VO. We are using (updating and expanding) the python implementation of the (Python WoT[38]). Concisely, the proposed technology stack is comprised of the following basic functionalities:

1. Protocol Bindings: HTTP, MQTT, CoAP
2. Security Protocols: Basic, OAuth2, TLS
3. Storage: SQLite, InfluxDB
4. Generic Functions: A set of functions to support the basic operations that the interplay between IoT and Applications require.
5. User-Defined Functions: A set of functions to extend the functionalities of the (c)VOs which are contained to a python script.
6. (c)VO Descriptor: A descriptor to summarize all necessary information for the (c)VO.
7. Automation Script Runner: A script to configure the (c)VO according to the Descriptor.
8. Virtualization: Helm Chart for ease of Deployment in Kubernetes

Python script

The runtime requires a Python script that declares the user-defined code that manages interactions with the Virtual Object. Apart from the ones mentioned in the Quick Start Guide the additional ones are:

1. Periodic Functions: snippets of code that need to run periodically.
2. Subscriptions to remote Events or Property Changes: snippets of code that need to run whenever an event is emitted or a property is changed on a remote Virtual Object that has been consumed.
3. Proxy functions: declaration of Properties, Actions or Events as proxies meaning that interaction with them is not managed locally but rather propagated to another Virtual Object

A full working example is located inside the examples/cli directory of the GitLab repository.

Setup

This section explains how to use the VO-WoT runtime to expose a device's information. The three-file format has been adopted to avoid boilerplate code and to simplify the development process. Three components need to be defined by the user:

- Thing Description: a JSON representation of a thing according to the Web of Thing specification specifying Properties, Actions and Events.
- (c)VO descriptor: a YAML descriptor containing initialization and configuration information.
- User-defined code: a Python script where the developer can define the code that will be run upon Action invocation, handling of Events, etc.

After populating these three files, the user needs to install the python package and execute the cli module as such:

A VO developer should provide a valid TD, a valid (c)VO descriptor and a valid python script with the User-Defined Functions, to have a custom version of a VO. These components are then given as input to a script runner component/cli that reads the Thing Description, the Virtual Object descriptor, and the python script to configure the Virtual Object. This more declarative approach of describing the Virtual Object was employed in order to minimize the boilerplate code a developer needs to write. Additionally, this configuration process for the Virtual Object facilitates the deployment in a virtualized environment such as Kubernetes.

Open-Source Activities

As said previously, the VO-WoT repository is a fork of the original WoTPy[39] repository. WoTPy is an experimental implementation of a W3C WoT Runtime[40] and the W3C WoT Scripting API[41] in Python, inspired by the exploratory implementations located in the Thingweb GitHub page[42]. In this repository we provide a clean and updated version of the original WoTPy repository, with the required new libraries and functionalities to align with the new versions of the WoT modelling.

[38] https://github.com/agmangas/wot-py/
[39] https://github.com/agmangas/wot-py
[40] https://github.com/w3c/wot-architecture/blob/master/proposals/terminology.md#wot-runtime
[41] https://github.com/w3c/wot-architecture/blob/master/proposals/terminology.md#scripting-api
[42] https://github.com/thingweb

Moreover, the VO-WoT repository is part of the w3c WoT Software and Middleware[43] solutions.

10.2 VO Stack Implementation Based on W3C

The VOStack implementation details based on W3C WoT provided in this section is relevant in the context of **Use Case #3** for personalized devices. An overview of the VO stack implementation based on the W3C architecture is shown in Figure 10-46. The target architecture facilitates the *modelling*, *orchestration*, and *management* of constrained IoT devices.

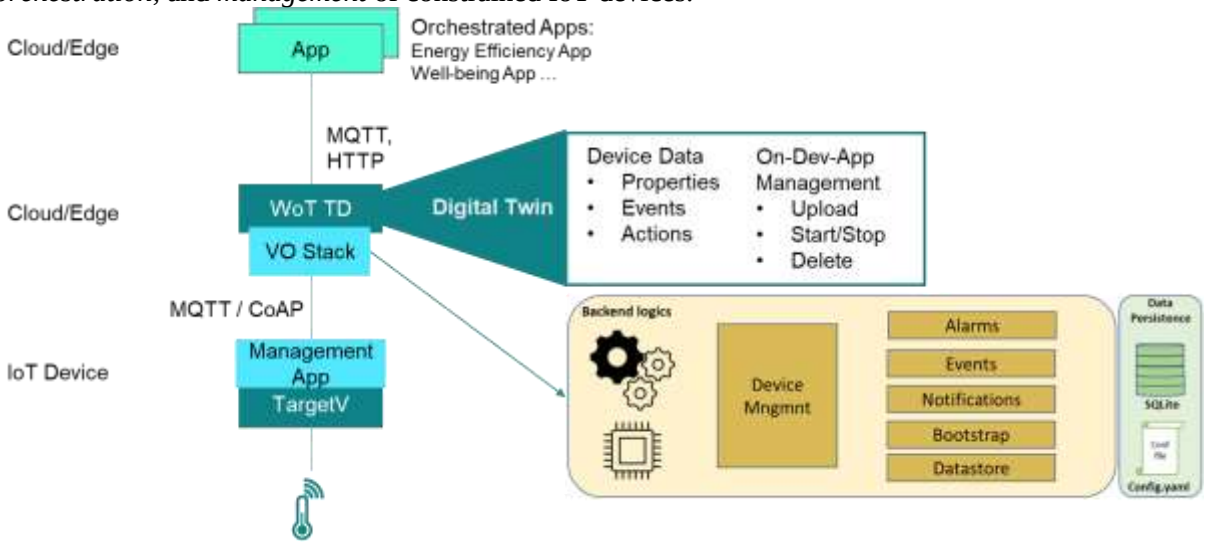


Figure 10-46: The overview of the framework: the modelling, management and interoperability of IoT Devices and Their Functions

The device layer is built with the Siemens proprietary PlatformV toolchain. PlatformV is an end-to-end development suite for cost-sensitive devices with limited computational resources. With its fundamental values - modularization, parallelization, and re-use - PlatformV brings speed, harmonization, and innovation to the products and fun to the development process. Three core components of PlatformV are used for the project implementation: EngineeringV, TargetV, and EmbeddedV. We introduce the functionalities of each component as follows:

- EngineeringV: a simple and intuitive engineering tool, browser-based and licence-free, making every developer productive from day one.
- TargetV: Commercially available target hardware lets developers get started with firmware development right away. There is no need to wait for the product hardware to be available.
- EmbeddedV: An extensive library of firmware assets ready for re-use boosts efficiency and lets developers focus on what is new and important about the products.

We engineer EmbeddedV firmware to be loaded onto TargetV devices using the EngineeringV tool. Figure 10-47 demonstrates a TargetV device that is equipped with EngineeringV. Also, the TargetV device incorporates the µCEP engine and TinyML engine.

[43] https://www.w3.org/WoT/developers/#software

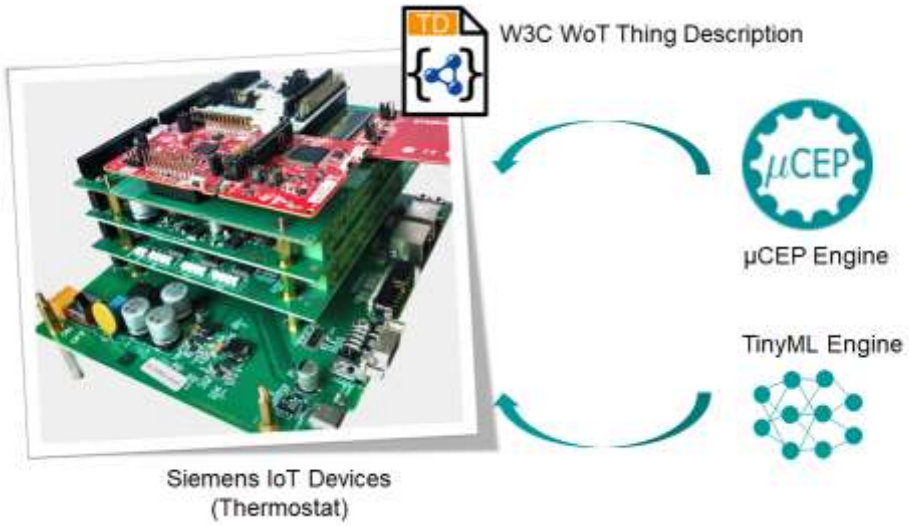


Figure 10-47: Deploying µCEP and TinyOL on Siemens IoT devices and implementing their digital twins using WoT Thing Description

Deployed CEP rules and TinyML models, alongside the TargetV devices, are described with corresponding W3C TMs and represented in VOs. This allows users to configure, interplay, and maintain running devices at runtime. The framework also enables end consumers to also manage and configure on-device functions (µCEP rules and models) on the fly using the VOM management API shown in Figure 10-48. The end consumer can use the following on device application management shown below to execute different API endpoints. This includes the following API requests: integration of device models with rules, identification of all the rules deployed within the system, identification of compatible devices and rules, as well as deploying, starting, and stopping and checking the status of rules on WoT devices.

The technology stack employed in the implementation of VO stack components based on W3C architecture includes the following:

- µCEP Engine: C, coreMQTT library
- TinyML Engine: C/C++
- Protocol Bindings: HTTP, MQTT

On-device function management: Python, SQLite database, SQLModel for Object-Relational Mapper, Uvicorn Web server, and FastAPI Web framework.

Figure 10-48: Virtual Object Manager REST API

10.3 VO alignment with OMA-LwM2M

The OMA-LwM2M standard offers a framework for monitoring and managing IoT devices, ensuring seamless integration across IoT systems. Within this framework, every IoT device is conceptualized as a collection of Objects, each characterized by its Resources. These Objects represent either hardware or software facets of the device, while Resources de- tail their attributes. Both are assigned distinct IDs and are structured using a specific meta-model [111]. By linking these IDs, a unique URI path is formed, facilitating operations on individual resources. In our VO design, which aligns with the LwM2M standard, we can outline and present a range of sensor resources, from temperature readings to software functionalities. This setup enables the generation of standardized URIs for each piece of VO-related

data, accessible to application modules through Representational State Transfer (RESTful) methods like Read, Write, and Delete. To bring the VO to life in line with OMA-LwM2M, every IoT device is portrayed using objectIDs, defined in a configuration file (the *Descriptor* file), along with associated protocols and security measures. In this architecture, the VO acts as the OMA-LwM2M Server, with the IoT device housing the client. This arrangement allows the VO to present two distinct interface sets, facilitating interactions between the VO and the IoT device (southbound) and between the VO and the Application (northbound), as illustrated in the Figure 10-49.

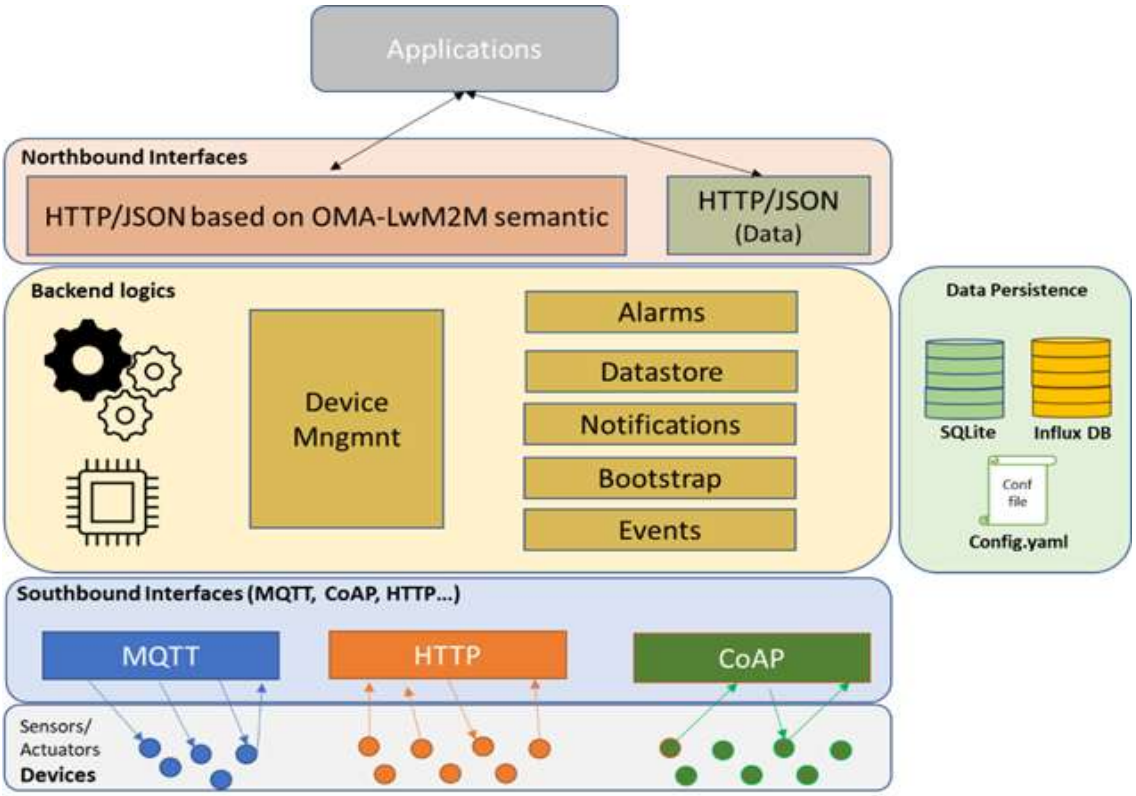


Figure 10-49: VO architecture in OMA-LwM2M standard

10.4 VO Stack implementation based on OMA-LwM2M

The environment surrounding a Virtual Object (VO) is multifaceted, encompassing various components that facilitate its operation and interaction within the computing continuum. This section delves into these primary components, shedding light on their roles and functionalities.

The VO is intended as a microservice composed by:

- Southbound Interfaces (to physical devices).
- Northbound interfaces (towards consumer applications and cVO).
- Management interfaces (to set up the connection with the respective physical devices).
- Device Abstraction Layer, for the semantic description of physical devices according to the OMA-LwM2M standard.
- Datastore: a relational, light, and fast database (SQLite) which will compose the data Volume of the Container and interfaces to a non-relational datastore (InfluxDB) deployed external to the VO container for high data rate resource values.
- Backend logic core for processing and enhancing functionality.

Southbound interfaces

This part of the service is dedicated to the interfaces that will communicate with physical devices for data acquisition and device management. Among the various IoT protocols examined for the implementation of these interfaces, it was decided to implement CoAP, MQTT/TCP and HTTP protocols previously described in the document. Regardless of the protocol used, to remain relevant to

the LwM2M standard, the payload is always defined according to the OMA-LWM2M model using the JSON format.

Resource	
Topic	stat/deviceID/objId/instId/resId
Payload	{"tmstp":"…","e":[{"v":"value"}]}
Instance	
Topic	stat/deviceID/objId/instId
Payload	{"tmstp":"…","e":[{"n":"resID","v":"value"},{"n":" resID","v":"value"},{"n":" resID","v":"value"}]}
Object	
Topic	cmnd/deviceID/objId
Payload	{"tmstp":"…","e":[{"n":"instId/resId","v":"value"},{"n":"instId/resId","v":"value"},{"n":"instId/resId","v":"value"},{…}]}

Figure 10-50: An example of MQTT southbound interface for READ operation.

Communications between physical devices and VOs are enabled by interfaces based on the URI path *ObjectID/ObjectInstanceID/ResourceID* relating to the owned objects declared by the physical device. The interfaces compliant with the standard are as follows:

- READ
- WRITE
- EXECUTE
- OBSERVE
- DELETE OBSERVE

CoAP and HTTP are ReST based protocol and resources addressing is easily mapped between them, it is not the same for the Publish/subscribe MQTT protocol. In MQTT, resources addressing take place using topics. In this implementation it will be distinguished:

- **Topic**: resource identifier (ie./*deviceID/objId/instId/resId*/).
- **FullTopic**: composed of a prefix (ie. *cmnd*).

The Topics are used following the standard modelling of the LwM2M protocol, which identifies a unique URI for each resource. For instance, the topic of the value resource 5700 of the instance 0 of the temperature object 3303 of the hot water device B01 will be made up as follows: B01/3303/0/5700.
FullTopics use, in addition to the topic, a prefix (e.g., cmnd). The prefixes are implemented to avoid the possibility of creating any loops between MQTT topics. Within this implementation, three distinct prefixes will be used:

- cmnd - prefix for giving commands or for asking status update.
- stat - reports the status or the configuration message.
- tele - reports telemetry information at given intervals.

Northbound Interfaces

The VO exposes these interfaces ready to other applications and/or services. The implementation takes advantage of RESTful protocol (Representational State Transfer), specifically HTTP. This interface level recalls the methods used in the OMA-LwM2M protocol to enable device management and information notification operations. The specifications of each interface (endpoint) are summarized later in the document.

- READ
- READ Realtime
- WRITE
- EXECUTE
- OBSERVE

READ

The VO receives a READ request from an application that wants to read information. It can be read:

- general information of the VO.
    - information coming from an object inside the VO.
    - information coming from an instance of an object.
    - information coming from a resource.

In the case of information coming from an object, it will be provided the data coming from all the instance of the object requested. The VO can manage two diverse types of READ request. The first one is a standard READ request (READ), that give back the last stored value by the VO. The second one is the real-time request (READ Realtime, described below), that is forwarded to the physical device to obtain an update value. The different use of the READ requests depends on the application requirements. Interface is defined below:

- HTTP method: GET.
- Interface: api/clients
- Resource path: /deviceId/objectID(opt.)/InstanceID(opt.)/ResourceID(opt.)
- Parameters: null or *?getRealtime=true* for Realtime request.
- Payload: null

Depending on the requested level of information, it is necessary to specify the specific URI path.

WRITE

The VO receives a WRITE request from an application that wants to write a value or several values to a specific resource or instance. The VO forwards the request to the device and stores the values. This function can be used for Resources and Instances. *Write* Interface is defined below:

- HTTP Method: PUT
- Interface: api/clients
- Resource path: /deviceId/objectID/InstanceID/ResourceID(opt.)
- Parameters: null
- Payload:

Instance:{"id":"InstanceID","resources":[{"id":resID,"value":"XXX"},{"id":resID,"value":"YYYY"}]}

Example for instance '0':
{"id":"0","resources":[{"id":14,"value":"+01"},{"id":15,"value":"Europe/Reggio Calabria"}]}

Resource: {"id":"resID","value":"+01"}

Example for resource '14': {"id":14,"value":"+01"}

EXECUTE

The VO receives an execution request. The "Execute" operation is used to start/trigger some actions and can only be performed on single resource. Interface is defined below:

- HTTP Method: POST
- Interface: api/clients
- Resource path: /deviceId/objectID/InstanceID/ResourceID
- Parameters: null
- Payload: null

OBSERVE

The VO receives the OBSERVE request from an application to track a resource, an object instance, or an object. When an entity is under observation, the observer is registered in a list. The VO uses this list to notify the application of the new incoming value of the observed entity. Doing so, a new observer for the same entity will be added to the observer list without forwarding new OBSERVE requests through the southbound interface and then to the device. Interface is defined below:

- HTTP Method: POST
- Interface: api/clients
- Resource path: /deviceId/objectID/InstanceID/ResourceID(opt.)/observe
- Parameters: null
- Payload: null

The observe request response will be:

Resource:{"event":"NOTIFICATION","data":{"ep":"VOid","res":"/3303/0/5700","val":{"id":5700,"value":55}}}
Instance:{"event":"NOTIFICATION","data":{"ep":"Device1","res":"/3303/0","val":{"id":0,"resources":[{"id":5601,"value":11.7}, {"id":5602,"value":24.0}, {"id":5700,"value":15.6},{"id":5701,"value":"cel"}]}}

DELETE

This functionality is used to delete a previous OBSERVE. Interface is defined below.

- HTTP Method: DELETE
- Interface: api/clients
- Resource path: /deviceId/objectID/InstanceID/ResourceID(opt.)/observe
- Parameters: null
- Payload: null

Moreover, through the NorthBound, the VO provides the interfaces for directly accessing the history of the data recorded in the chosen period. These interfaces, exposed on the api/data path, will allow:

- Data extraction by type of resource, instance/object or object.
- Data extraction by value (applicable only on resource).
- Time frame definition for data extraction.

Datastore APIs

The interfaces, exposed on the *api/data* path, enable data aggregation, and specifically allow the operations described below.

Data extraction by value (applicable only to a single resource)

This interface allows you to search for a specific value within the logged data of a specific resource, for example, search when the value (resId=5700) of the temperature sensor (objectId=3303) took on a specific value (val=18). In this request you can choose the type of operator to use in the search for the match:

- less ("<", operator=1),
- greater (">", operator=0),
- equal ("=", by default).

The API endpoint is defined as:

- HTTP method: GET.
- Interface: api/data
- Path: /deviceId/oggettoID/IstanzaID/ResourceID/value
- Parameters: ?value, operator (optional, default is "=")
- Payload: empty

For instance, the following request returns in chronological order all recorded values less than 18.1:

/deviceId/oggettoID/IstanzaID/ResourceID/value?value=18.1&operator=1

Extraction of the last n recorded values.

The interface in question allows you to extract a limited series of values (n) from the same resource.
The API endpoint is defined as:

- HTTP method: GET.
- Interface: api/data
- Resource path: /deviceId/objectId/instanceId/resourceId/limit
- Parameters: ?limit=n
- Payload: empty

For instance, the following request returns the last 10 values, in chronological order, recorded on that resource*: /deviceId/objectId/instanceId/resourceId/limit?limit=10*

Data extraction over a period of time

This interface allows consumer applications to extract a history of data for construction, for example, to be used in specific diagrams inserted in dedicated dashboards. The dates must be entered according to *SimpleData* format as described later on.
The API endpoint is defined as:

- HTTP method: GET.

- Interface: api/data
- Resource path: /deviceId/objectID/InstanceID/ResourceID/date
- Parameters: ?startDate=yyyy-MM-dd hh:mm:ss&endDate=yyyy-MM-dd hh:mm:ss
- Payload: empty

For instance, the following request returns the values recorded over a 30 second time interval:
/deviceId/objectID/InstanceID/ResourceID/date?startDate=2023-10-30 10:14:12&endDate=2023-10-30 10:14:42

DATASTORE

SQLite

In the SQL Lite, are implemented the table that are rarely changed, and more related to device description according to the semantic model.

Those tables are:

1. **Device:** this table contains the descriptive attributes of the device such as VO name (endpointname), lwm2m version, registrationId, etc.
2. **Object:** contains the LwM2M objects contained by the VO and their attributes as defined by OMA-LwM2M.
3. **Observable:** they are the entities (instance or resource) placed under observation by one or more observers.
4. **Observer:** contains the list of all external applications that have requested the observation of a resource.
5. **Resource:** stores the list of resources made available by enabled OMA-LwM2M objects.

InfluxDB

In influxDB, are implemented the table that changes more frequently.

- Measurement table, the Measurement table is used to store the values notified by the device, for each resource.
- Event table is used to register the row packet received by VO from the southbound interface using JSON. Each event arrives in JSON format, and it is saved in row in this table.

Coding

The code of the VO is built using Spring boot. Spring Boot is an innovative project within the larger Spring ecosystem, designed to simplify the process of building production-ready applications with minimal upfront configuration. It provides a set of default configurations, libraries, and frameworks to help developers create stand-alone, web-based applications with ease. One of the primary advantages of Spring Boot is its ability to automatically configure your application based on the libraries present in the project, eliminating the need for specifying beans in the configuration files. Within the Spring Boot framework, the architecture is typically divided into layers, with each layer having a specific role. Two of the most crucial layers are the *Service* and *Controller* layers.

Service Layer is responsible for the business logic of the application. It interacts with the data access layer and performs CRUD (Create, Read, Update, Delete) operations. The *'@Service'* annotation in Spring Boot is used to indicate that a class provides business services. These services are then injected into controllers or other services, promoting the principle of inversion of control, and ensuring a clear separation of concerns.

Controller Layer, annotated with '*@Controller*' or '*@RestController*', acts as an intermediary between the model and the view. It listens to the client's requests, processes them (with the help of services), and returns the appropriate view or data. In the context of RESTful web services, the controller is responsible for handling and responding to incoming HTTP requests, often returning JSON or XML data. Together, the service and controller layers form the backbone of a Spring Boot application, ensuring a modular and maintainable codebase while providing a seamless user experience.

For instance, to ensure a robust and maintainable implementation in line with Object-Oriented Programming (OOP) principles, a layered approach was adopted. Initially, a general interface for the *ClientService* was defined. This was followed by the creation of an abstract class that encapsulates the common code shared between the two services, MQTT and CoAP. Building upon this foundation, two concrete classes, *ClientServiceM* and *ClientServiceU*, were developed. These classes extend the abstract

class, ensuring a clear separation of concerns and promoting code reusability. A practical example illustrating this layered approach can be found at link[44]. To provide a visual representation of the VO's functioning and the inter-relationship between the classes, a class diagram was constructed, as depicted in Figure 10-51.

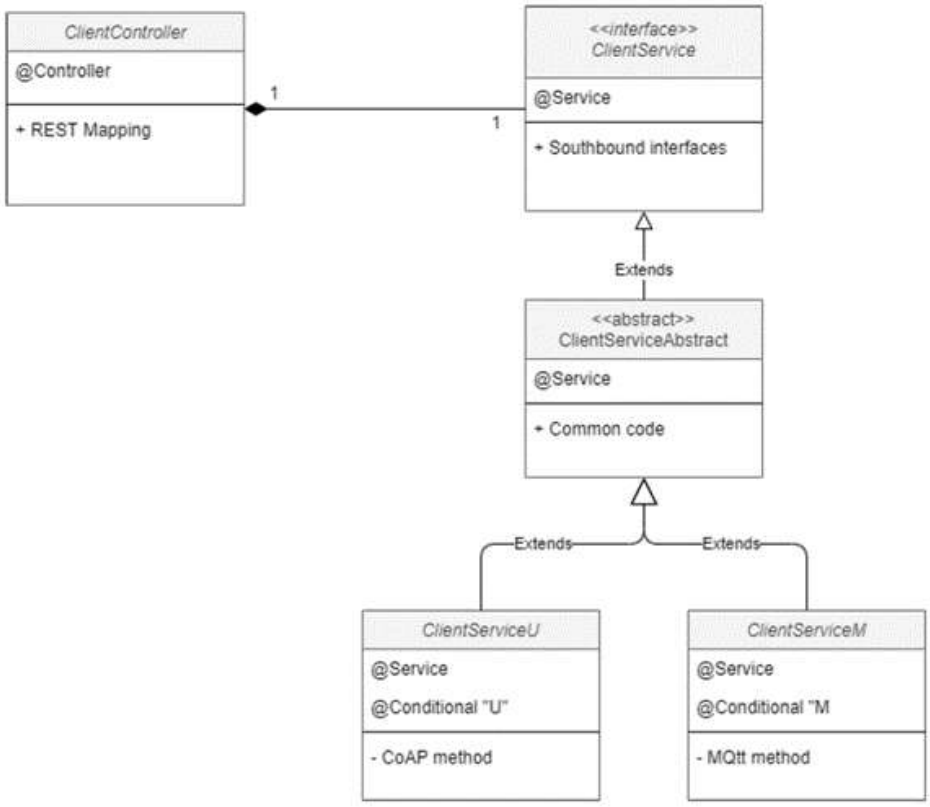


Figure 10-51: Class diagram describing ClientController and ClientService relationship.

To ensure the organization analysed in figure, the Spring Boot annotation @ConditionalOnProperty can be used. The property annotation allows to register beans conditionally depending on the presence of a configuration property. The configuration file (Descriptor) defines which communication protocol the VO utilizes at the VO instantiation using bindingMode parameter: M (MQTT), or U (UDP), or H (HTTP) So, the VO uses one single Spring Boot controller that, using the conditional on the configuration file, creates a ClientService (M, H or U) depending on the value of the bindingMode.
The MQTT implementation is performed by using a class called MqClient. This class is used by the ClientServiceM, the Spring boot service created when the VO is instantiated in MQTT mode. The ClientServiceM manages the Controller calls by using the MQTT Paho Async Client that permits to perform operations simultaneously with the connection or subscription.
Eclipse Leshan project provides ready-to-use classes for the implementation of a LwM2M CoAP server. To integrate these capabilities into the VO, specific classes from the Leshan project were imported and subsequently modified to fit the requirements of the VO.

10.5 VO-OMA-LwM2M Proof of Concept

This proof of concepts provides a tangible representation of the theoretical and practical work undertaken throughout this project.

[44] https://davidgiard.com/java-services-and-interfaces-in-a-spring-boot-application

10.5.1 Setup and Equipment

This section delves into the specific tools, both hardware and software, which were employed to bring the system to life. A comprehensive understanding of the setup is crucial, as it lays the foundation for the subsequent sections where the architecture and implementation details are discussed. The choice of each tool and equipment was influenced by their compatibility, efficiency, and the ease with which they could be integrated into the overall system.

This Demo is not involved in the demonstration of Orchestration environment. Anyway, VO and cVO has been deployed as containers using *Docker*. Docker is a platform used to develop, ship, and run applications inside containers. Containers allow developers to package up an application with all parts it needs, such as libraries and other dependencies, and ship it all out as one package. In the context of the demo, Docker played a pivotal role in orchestrating multiple instances of the VO. By leveraging Docker, multiple VOs could be activated simultaneously, ensuring scalability and flexibility. Furthermore, Docker was not just limited to the VO. The instances of InfluxDB, which are integral for real-time data management and storage, were also initiated and managed using Docker. This ensured a cohesive environment where all components, from the VO to the databases, were managed uniformly, reducing complexities and potential points of failure.

The Application layer of this demo focuses on the applications that transparently utilize the data generated by the physical devices trough the (c)VO for visualization and interaction: the Node-RED Dashboard.

Node-RED is a flow-based development tool for visual programming, initially developed by IBM for wiring together hardware devices, APIs, and online services. It provides a web browser-based editor that makes it easy to wire together flows using the wide range of nodes in the palette. Flows can be then deployed to the runtime in a single click. The light-weight runtime is built on Node.js, taking full advantage of its event-driven, non-blocking model, making it ideal for data-intensive real-time applications that run across distributed devices. In the context of our system, Node-RED was employed as a primary tool for the northbound interface due to its flexibility, ease of use, and extensive community support[45].

The physical device on ground utilized in the system is an ESP32[46] based board, a highly versatile and low-power system-on-chip (SoC) with integrated Wi- Fi and dual-mode Bluetooth capabilities. Developed by Espressif Systems, the ESP32 is designed for mobile, wearable electronics, and IoT applications. Its characteristics are: 520 KB SRAM, 448 KB ROM, 16 KB RTC SRAM. Moreover, the sensor integrated to the real device setup is the DHT11 sensor. The DHT11 is a basic, ultra low-cost digital temperature and humidity sensor. It uses a capacitive humidity sensor and a thermistor to measure the surrounding air and outputs a digital signal on the data pin. The DHT11 provides the ESP32 with real-time temperature and humidity readings, which are then relayed to the Virtual Object

10.5.2 PoC Architecture

The Proof-of-Concept system architecture, shown in Figure 10-52, is composed of three Virtual Objects, each serving a distinct role to ensure seamless integration and communication. These include two standard Virtual Objects, labelled as VO001 and VO002, and a Composite Virtual Object, CV001.

[45] nodered.org/

[46] https://www.espressif.com/en/products/socs/esp32

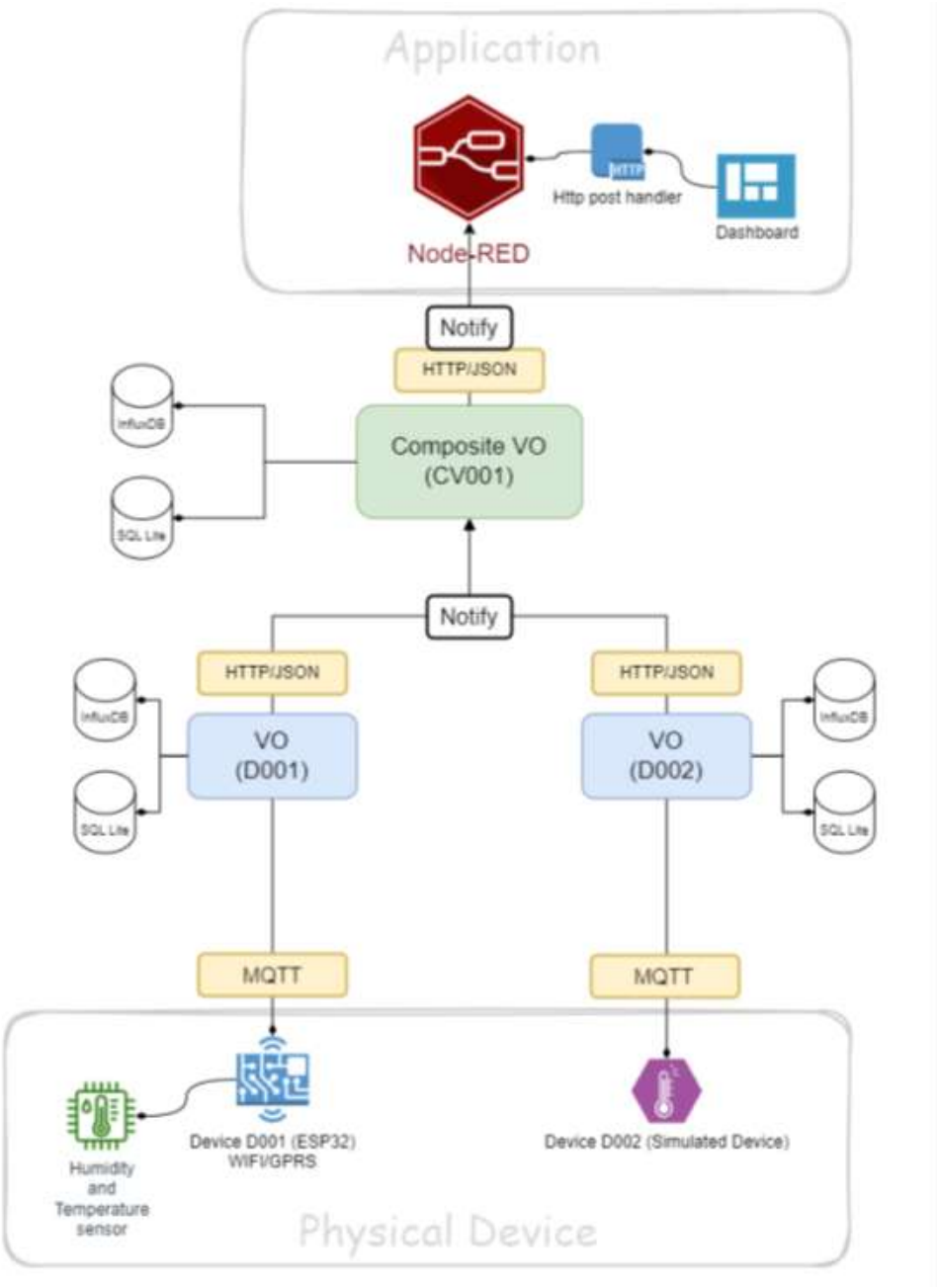


Figure 10-52: PoC architecture

Both VO001 and VO002 are equipped with their dedicated instances of SQLite and InfluxDB, ensuring data integrity and efficient storage. The Composite Virtual Object, CV001, is an aggregation of the two standard VOs, VO001 and VO002. This design allows for a hierarchical structure where the composite object can efficiently manage and relay information from the individual Vos.

On the *Southbound* side, VO001 is connected to a physical device, specifically an ESP32 integrated with a temperature sensor. This connection is facilitated using the MQTT protocol. In contrast, VO002 communicates with a simulated physical device, leveraging the CoAP protocol for data exchange.

The northbound communication is orchestrated between CV001 and VOs, and between CV001 and Node-Red Application. CV001 acts as the primary interface for external applications. Both VO001 and VO002 relay their data to CV001, which in turn communicates with a testing platform implemented using Node-RED. This layered approach ensures that the communication between the VOs and the application is streamlined and efficient.

The communication protocol between the cVO and VOs is designed to be user-transparent, relying on a ”Notify” mechanism. Whenever there is a new resource update from the southbound side, the respective VO updates CV001. The composite object then takes the responsibility of updating the Application, ensuring real-time data synchronization and minimal latency.
This architecture, with its clear division of roles and responsibilities, ensures that the system is scalable, adaptable, and efficient in handling real-time data from various sources.

10.5.3 Application

Every update that the cVO receives from the VOs is notified to the northbound application built on Node-RED. The data received are used to build a dashboard; one dashboard for each device. The user can easily switch from dashboard to dashboard by using the side menu. The first dashboard is built on the data received from the real device. The dashboard is shown in Figure 10-53.

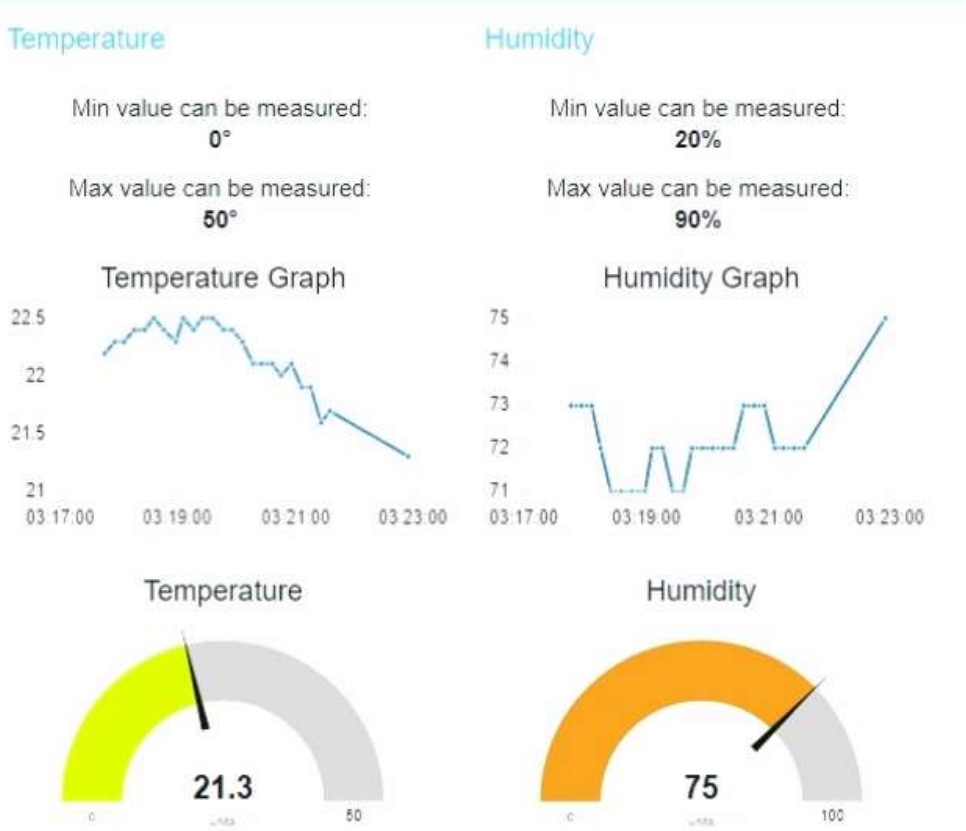


Figure 10-53: PoC dashboard snapshot of Device 001

The second dashboard is built on the data received from the simulated device based on CoAP. The dashboard is shown in Figure 10-54.

Figure 10-54: Device D003 dashboard snapshot

10.5.4 Performance Analysis

The final part of the Proof-of-Concept activity consists of a performance analysis through some tests, to better understand the improvement when adding InfluxDB to the VO stack.

The analysis performed is divided in two sections: Only store data (No *Observer* activation), Store and notify (Observervation activated by node-red application).

Two values are analyzed: percentage of messages lost, and delay on messages elaboration.

The *Percentage* of message lost graphic indicate in the X axis the frequency of the test and on the Y axis the percentage of message lost calculated like that:

$$Pktlost\% = (pktreceived)/(pkttotal) * 100$$

where received indicates the messages received correctly by the VO and total indicates the total messages sent by the device.

The delay on message elaboration is calculated as:

$$delayME\% = (elaborationdelay)/(sendingTime) * 100$$

The elaboration delay is the time that elapses from the last message sent by the device and the last message computed by the VO; the sending Time indicates the time that the device uses to send all the burst of messages. This data is calculated using:

$$SendingTime = messages * 1/(forwardfrequency)$$

where *# messages* indicate the number of messages sent in the burst.

This formula is used to give a more general value of the delay and not strictly restricted to the number of messages sent in the burst. Obviously more messages are sent in the burst higher is the flat delay. Sampling it on the number of messages multiply the period, it can be obtained a smaller number of messages dependent value.

Each analysis is conducted in the above-described system, considering 800 messages sent through MQTT, with a simulated python MQTT client. The different frequencies used are calculated by the time passing from a MQTT publish and another. Tests are conducted at the frequency of 20MHz, 10MHz, 5MHz, 2MHz, 1MHz, 0.5MHz.

Hardware environment

It is important to say that the machine computational power is provided by a standard PC, with 8 GB ram and 2.2 GHz quad core. It is not even comparable with a normal state orchestration HW, but it is enough for the Proof of Concept performed in this thesis aimed at testifying to the better performance of InfluxDB with respect to SQLite.

Only store data scenario

In this case the VO enter the database only to write data and does not need to be forwarded.

The first analysis is built on the data that the VO can save in the database. It can be seen in figure that for the lower frequency, both the databases perform sufficiently well. But at higher frequency SQL Lite loses a relevantly higher amount of data. Data integrity is a really important in IoT scenario and should be satisfied.

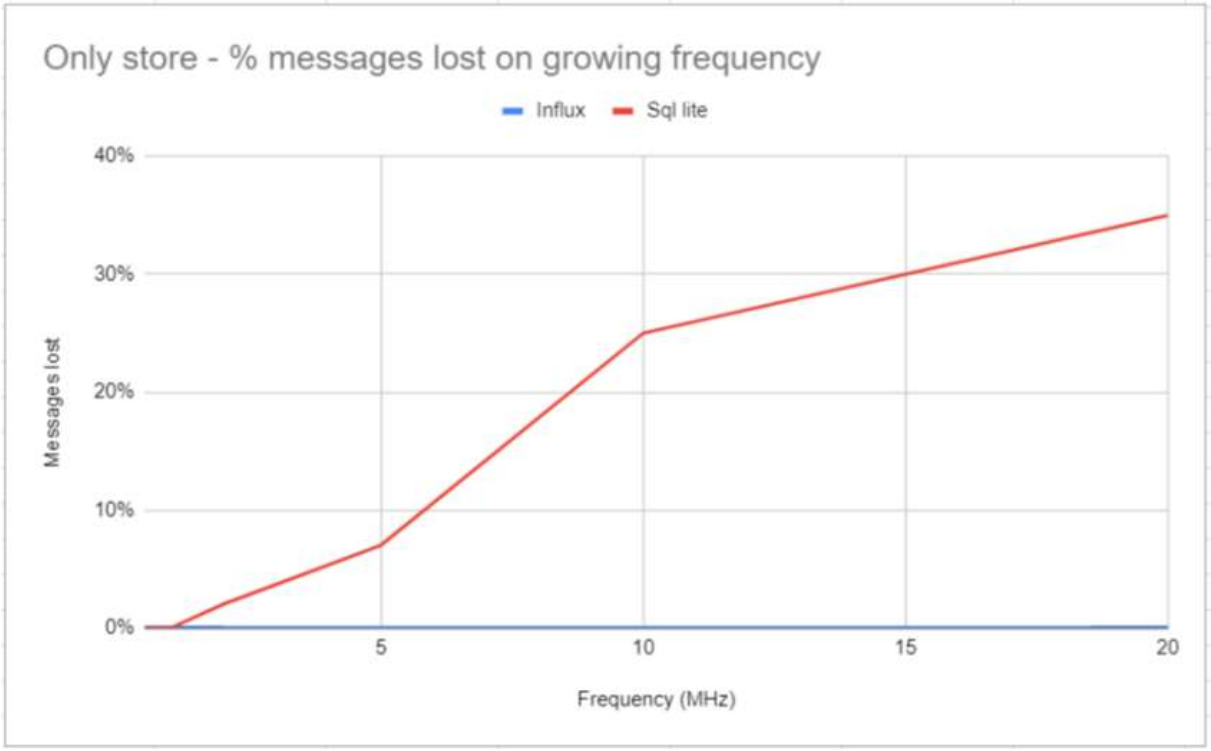


Figure 10-55: InfluxDB and SqlLite comparison on message lost on growing frequency.

The second analysis focuses on the delay with which the VO store the data in the database. The delay is calculated after all the 800 messages are sent and is calculated in percentage with respect of the time that all messages arrive. It can be seen in figure below how at low frequency both databases perform decent job while at a higher frequency the SQL Lite delays are manifestly high compared to the one of InfluxDB. At 20MHz the delay calculated by the formula is higher than the sending time (time that device uses to send all the messages). In real-time system, data cannot arrive with an excessive delay; and the delay observed for the SQL Lite could be inadequate.

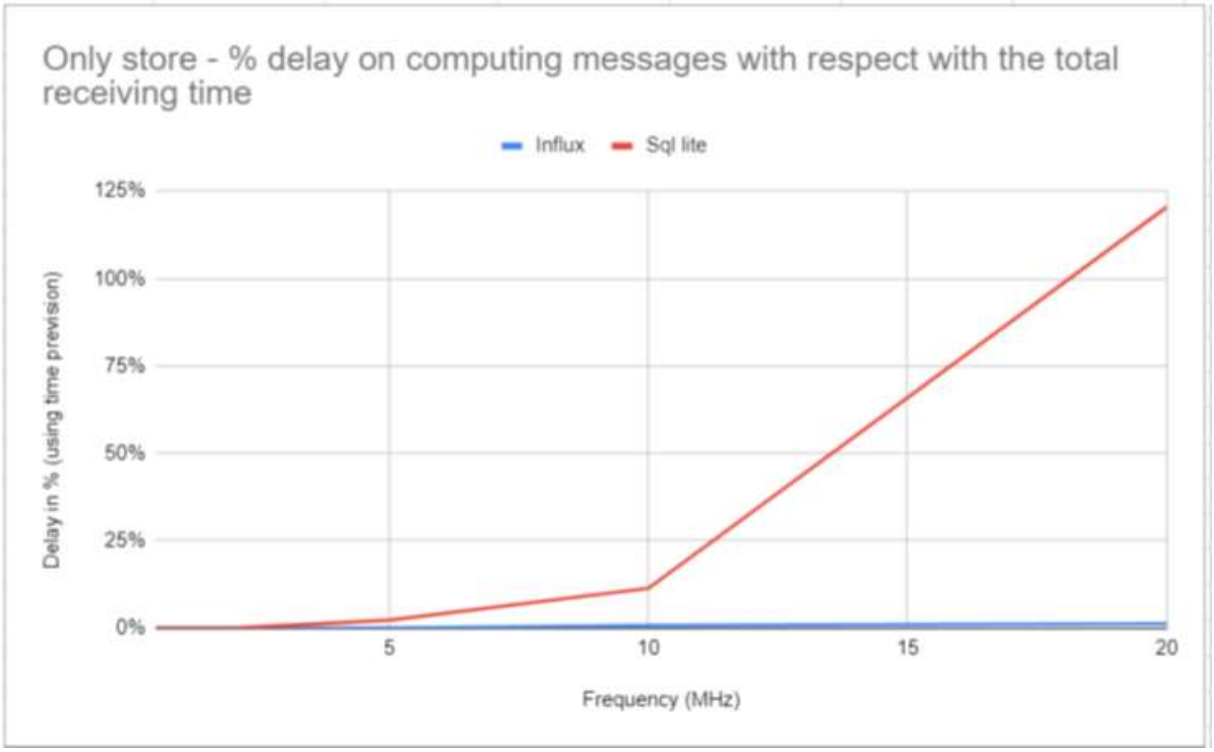


Figure 10-56: InfluxDB and SqlLite comparison on latency in message delivery on growing frequency

Store data and notify scenario.

In this case the VO enters for the first time the database to write data and for a second time to read data to forward in northbound. This means that a slow database will be conditioned in a greater extent. The figure below shows that in this case the amount of lost data is bigger.

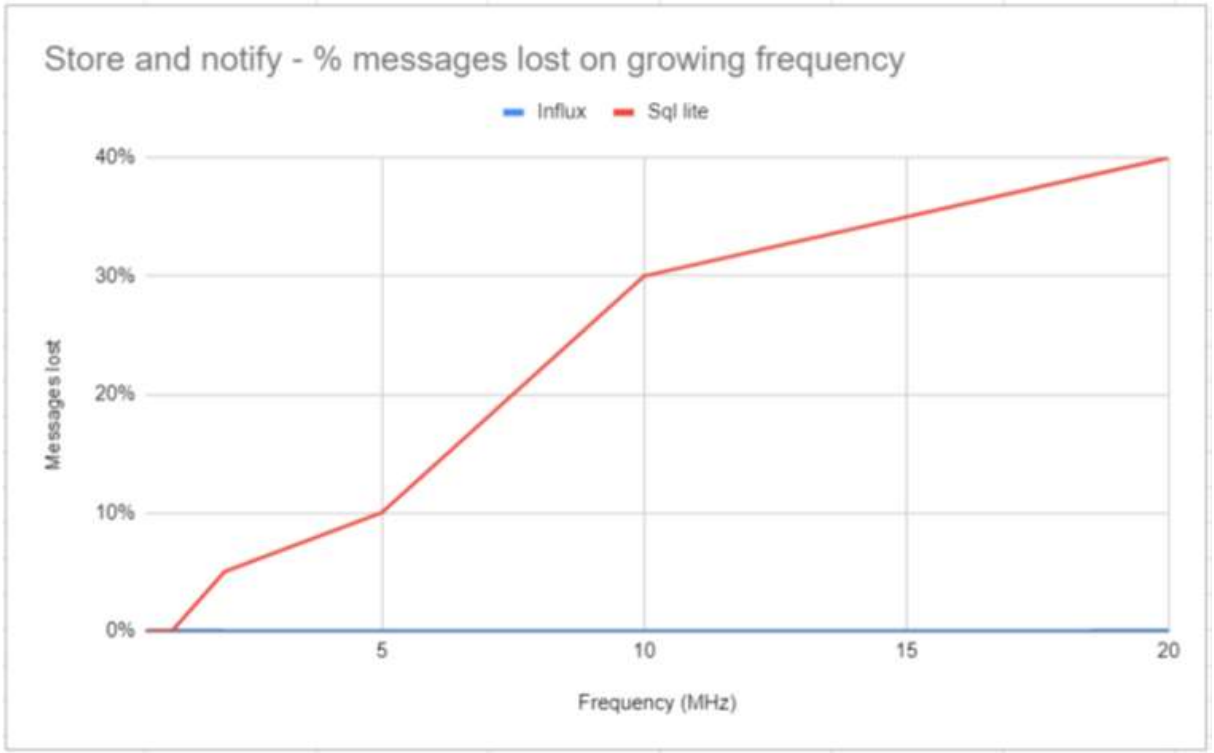


Figure 10-57: InfluxDB and SqlLite comparison on message lost to northbound delivery on growing frequency.

Also, in the case of Delay, InfluxDB has better performance. Almost reducing to 0 the delay for the messages to arrive.

In conclusion, the SQLite database suffers from concurrency access to its tables when there are different and simultaneous write operations. So, the PoC demonstrates the limitation of SQLite datastore for high frequency data transmission frequency higher than 2 Hz (0.5 seconds).

11 Conclusions

This document is the first report on the development of VOStack in the NEPHELE project. Virtual Object Stack (VOStack) is a software stack, which can be used for creating, discovering, orchestrating, and consuming Virtual Objects (VOs). The VOStack provides mechanisms for managing interactions among IoT devices and VOs. VOStack resides at the edge and is a crucial component to efficiently exploit resources in the continuum from Cloud-to-Edge-to-IoT-Device.

The VOStack and Virtual Objects are concepts introduced in the NEPHELE project to harness the complexity of heterogeneity of different IoT devices, diverse communication protocols, and the complexity of various information models for semantic representation of IoT assets. The ultimate purpose of the VOStack and VOs is to enable an easy creation of applications, which are based on a standard-conform and unified data access. This work has been conducted in Work Package 3, which is concerned with the topic of convergence of different IoT Technologies with the goal of guaranteeing continuous and seamless openness and interoperability.

In this document, we introduced concepts of VOs and VOStack in the NEPHELE project. The concepts enable the vision of Digital Twin, which is based on standards such as W3C Web of Things, Open Mobile Alliance (OMA) Lightweight M2M (LWM2M), and the Next Generation Service Interface - Linked Data (NGSI-LD). We presented the status on the work related to intelligent IoT devices modelling, management, and interoperability. We provided the semantic model for describing VO functions, which run on an intelligent IoT device or Edge, as well as the VO Descriptor, which is a declarative way to define a VO based on the related semantics and modelling. Security functionality at the device and Edge level have been also worked out. Subsequently, the status on activities in the area of autonomic functionalities and ad-hoc clouds is presented, too. Autonomic networking functionalities at IoT level, as well as SDN reactive routing and Time Sensitive Networking have been also reported. Further, we presented the development work on the set of functions that should be supported by the IoT device virtualized functions and the generic/supportive functions layer of VOStack (e.g., elasticity management, security, authentication, and telemetry functions). The preliminary work on the end-to-end orchestration of distributed applications across the computing continuum in a unified way has been presented too. The work will be extended in the next deliverable to support creation of application graphs based on microservice-based VOs and cVOs. We reported on the current implementation of intelligent IoT devices and their interplay with VOs. The first results based on TinyML and CEP mechanisms have been demonstrated. Finally, we provided the status of VOStack implementation as a software stack that will integrate all other developments in Work Package 3.

This deliverable has the focus on the definition of important concepts from NEPHELE project and the first implementations thereof. The successor of this deliverable, i.e., D3.2, will provide the final release of VOStack layers and intelligence mechanisms on IoT devices.

References

[1] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (CoAP)," 2014.

[2] M. A. A. da Cruz, J. J. P. C. Rodrigues, P. Lorenz, P. Solic, J. Al-Muhtadi, and V. H. C. Albuquerque, "A proposal for bridging application layer protocols to HTTP on IoT solutions," *Future Generation Computer Systems*, vol. 97, pp. 145–152, 2019.

[3] M. Nitti, V. Pilloni, G. Colistra, and L. Atzori, "The virtual object as a major element of the internet of things: a survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1228–1240, 2015.

[4] L. Atzori *et al.*, "SDN&NFV contribution to IoT objects virtualization," *Computer Networks*, vol. 149, pp. 200–212, 2019.

[5] M. A. Jarwar, S. Ali, and I. Chong, "Microservices model to enhance the availability of data for buildings energy efficiency management services," *Energies (Basel)*, vol. 12, no. 3, p. 360, 2019.

[6] M. Segovia and J. Garcia-Alfaro, "Design, modeling and implementation of digital twins," *Sensors*, vol. 22, no. 14, p. 5396, 2022.

[7] H. Kokkonen *et al.*, "Autonomy and intelligence in the computing continuum: Challenges, enablers, and future directions for orchestration," *arXiv preprint arXiv:2205.01423*, 2022.

[8] J. M. Cantera Fonseca and M. Bauer, "2nd W3C Workshop on the Web of Things." Nov. 01, 2018. Accessed: Oct. 02, 2023. [Online]. Available: https://www.w3.org/WoT/ws-2019/Papers/05%20-%20Fonseca%20+%20Bauer%20-%20ERCIM%20Position%20Statement.pdf

[9] J. M. Cantera, "Towards interworking between NGSI-LD and WoT (2 nd W3C Workshop on Web of Things) Public review," Jun. 2019. [Online]. Available : https://www.w3.org/WoT/ws-2019/Presentations%20-%20Day%202/Future%20Work/14_ETSI%20ISG%20CIM%20-%20Presentation_for_WoT_Workshop_Munich_05th_June_2019.pdf

[10] L. Frost, "HOW COULD WOT AND NGSI-LD FIT TOGETHER?," Aug. 2018.

[11] A. Abid, J. Lee, F. Le Gall, and J. Song, "Toward Mapping an NGSI-LD Context Model on RDF Graph Approaches: A Comparison Study," *Sensors*, vol. 22, no. 13, Jul. 2022, doi: 10.3390/s22134798.

[12] M. Lagally, R. Matsukura, M. McCool, and K. Toumura, "Web of Things (WoT) Architecture 1.1," W3C Proposed Recommendation, Jul. 11, 2023

[13] S. Kaebisch, M. McCool, and E. Korkan, "Web of Things (WoT) Thing Description 1.1," W3C Proposed Recommendation, Jul. 11, 2023

[14] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, P.-A. Champin, and N. Lindström, "JSON-LD 1.1 - A JSON-based Serialization for Linked Data," W3C Recommendation. Accessed: Nov. 29, 2023. [Online]. Available: https://www.w3.org/TR/json-ld11/

[15] S. Z. A. S. H. J. Klas G. Rodermund F., "OMA Whitepaper LightweightM2M," 2014.

[16] F. Ahmed, "Self-organization: a perspective on applications in the internet of things," *Natural Computing for Unsupervised Learning*, pp. 51–64, 2019.

[17] S. Hamrioui, J. Lloret, P. Lorenz, and J. J. P. C. Rodrigues, "Cross-Layer Approach for Self-Organizing and Self-Configuring Communications Within IoT," *IEEE Internet Things J*, vol. 9, no. 19, pp. 19489–19500, 2022.

[18] N. Santi and N. Mitton, "A resource management survey for mission critical and time critical applications in multi access edge computing," *ITU Journal on Future and Evolving Technologies*, vol. 2, no. 2, 2021.

[19] F. Saeik *et al.*, "Task offloading in Edge and Cloud Computing: A survey on mathematical, artificial intelligence and control theory solutions," *Computer Networks*, vol. 195, p. 108177, 2021.

[20] D. Dechouniotis, N. Athanasopoulos, A. Leivadeas, N. Mitton, R. Jungers, and S. Papavassiliou, "Edge computing resource allocation for dynamic networks: The DRUID-NET vision and perspective," *Sensors*, vol. 20, no. 8, p. 2191, 2020.

[21] N. Javaid, A. Sher, H. Nasir, and N. Guizani, "Intelligence in IoT-based 5G networks: Opportunities and challenges," *IEEE Communications Magazine*, vol. 56, no. 10, pp. 94–100, 2018.

[22] G. Leenders, G. Callebaut, L. der Perre, and L. De Strycker, "Multi-RAT IoT–What's to Gain? An Energy-Monitoring Platform," in *2023 IEEE 97th Vehicular Technology Conference (VTC2023-Spring)*, 2023, pp. 1–5.

[23] B. Foubert and N. Mitton, "Lightweight network interface selection for reliable communications in multi-technologies wireless sensor networks," in *2021 17th International Conference on the Design of*

Reliable Communication Networks, DRCN 2021, Institute of Electrical and Electronics Engineers Inc., Apr. 2021. doi: 10.1109/DRCN51631.2021.9477350.

[24] IEEE, "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic," *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q-2014/Cor 1-2015)*, pp. 1–57, 2016, doi: 10.1109/IEEESTD.2016.8613095.

[25] A.-C. G. Anadiotis, L. Galluccio, S. Milardo, G. Morabito, and S. Palazzo, "Towards a software-defined Network Operating System for the IoT," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, IEEE, Dec. 2015, pp. 579–584. doi: 10.1109/WF-IoT.2015.7389118.

[26] B. T. De Oliveira, L. B. Gabriel, and C. B. Margi, "TinySDN: Enabling multiple controllers for software-defined wireless sensor networks," *IEEE Latin America Transactions*, vol. 13, no. 11, pp. 3690–3696, 2015.

[27] M. Baddeley *et al.*, "Atomic-SDN: Is synchronous flooding the solution to software-defined networking in IoT?," *IEEE Access*, vol. 7, pp. 96019–96034, 2019.

[28] S. Bera, S. Misra, S. K. Roy, and M. S. Obaidat, "Soft-WSN: Software-defined WSN management system for IoT applications," *IEEE Syst J*, vol. 12, no. 3, pp. 2074–2081, 2016.

[29] T. Theodorou and L. Mamatas, "CORAL-SDN: A software-defined networking solution for the Internet of Things," in *2017 IEEE conference on network function virtualization and software defined networks (NFV-SDN)*, 2017, pp. 1–2.

[30] T. Theodorou and L. Mamatas, "A versatile out-of-band software-defined networking solution for the Internet of Things," *IEEE Access*, vol. 8, pp. 103710–103733, 2020.

[31] T. Theodorou and L. Mamatas, "SD-MIoT: A software-defined networking solution for mobile Internet of Things," *IEEE Internet Things J*, vol. 8, no. 6, pp. 4604–4617, 2020.

[32] T. Theodorou, G. Violettas, P. Valsamas, S. Petridou, and L. Mamatas, "A Multi-Protocol Software-Defined Networking Solution for the Internet of Things," *IEEE Communications Magazine*, vol. 57, no. 10, pp. 42–48, Oct. 2019, doi: 10.1109/MCOM.001.1900056.

[33] OpenAI, "ChatGPT." Nov. 2023. [Online]. Available: https://openai.com/blog/chatgpt

[34] Statista, "Microcontroller unit (mcu) shipments forecast worldwide from 2021 to 2027." Nov. 2023. [Online]. Available: https://www.statista.com/statistics/1327567/worldwide-microcontroller-unit-shipments-forecast/

[35] M. Shafique, T. Theocharides, V. J. Reddy, and B. Murmann, "TinyML: current progress, research challenges, and future roadmap," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 1303–1306.

[36] L. Ravaglia, M. Rusci, D. Nadalini, A. Capotondi, F. Conti, and L. Benini, "A tinyml platform for on-device continual learning with quantized latent replays," *IEEE J Emerg Sel Top Circuits Syst*, vol. 11, no. 4, pp. 789–802, 2021.

[37] H. Cai, C. Gan, L. Zhu, and S. Han, "Tiny transfer learning: Towards memory-efficient on-device learning," *arXiv preprint arXiv:2007.11622*, vol. 3, no. 4, p. 6, 2020.

[38] M. Giordano *et al.*, "CHIMERA: A 0.92 TOPS, 2.2 TOPS/W edge AI accelerator with 2 MByte on-chip foundry resistive RAM for efficient training and inference," in *2021 symposium on VLSI circuits*, 2021, pp. 1–2.

[39] B. Jiao *et al.*, "A 0.57-gops/dsp object detection pim accelerator on fpga," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, 2021, pp. 13–14.

[40] V. Jain, N. Jadhav, and M. Verhelst, "Enabling real-time object detection on low cost FPGAs," *J Real Time Image Process*, vol. 19, no. 1, pp. 217–229, 2022.

[41] B. Sudharsan, P. Yadav, J. G. Breslin, and M. I. Ali, "An sram optimized approach for constant memory consumption and ultra-fast execution of ml classifiers on tinyml hardware," in *2021 IEEE International Conference on Services Computing (SCC)*, 2021, pp. 319–328.

[42] H. Qiu *et al.*, "ML-EXray: Visibility into ML deployment on the edge," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 337–351, 2022.

[43] P. Corneliou, P. Nikolaou, M. K. Michael, and T. Theocharides, "Fine-grained vulnerability analysis of resource constrained neural inference accelerators," in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2021, pp. 1–6.

[44] V. J. Reddi *et al.*, "Widening access to applied machine learning with tinyml," *arXiv preprint arXiv:2106.04008*, 2021.
[45] STMicroelectronics, "Ai expansion pack for stm32cubemx." Nov. 2023. [Online]. Available: https://www.st.com/en/embedded-software/x-cube-ai.html
[46] T. Chen *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.
[47] R. David *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021.
[48] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under concept drift: A review," *IEEE Trans Knowl Data Eng*, vol. 31, no. 12, pp. 2346–2363, 2018.
[49] Ó. Fontenla-Romero, B. Guijarro-Berdiñas, D. Martinez-Rego, B. Pérez-Sánchez, and D. Peteiro-Barral, "Online machine learning," in *Efficiency and Scalability Methods for Computational Intellect*, IGI global, 2013, pp. 27–54.
[50] M. Andrade, E. Gasca, and E. Rendón, "Implementation of Incremental Learning in Artificial Neural Networks.," in *GCAI*, 2017, pp. 221–232.
[51] F. M. Castro, M. J. Mar\'\in-Jiménez, N. Guil, C. Schmid, and K. Alahari, "End-to-end incremental learning," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 233–248.
[52] R. Kolcun *et al.*, "The case for retraining of ML models for IoT device identification at the edge," *arXiv preprint arXiv:2011.08605*, 2020.
[53] M. Farhadi, M. Ghasemi, S. Vrudhula, and Y. Yang, "Enabling incremental knowledge transfer for object detection at the edge," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020, pp. 396–397.
[54] D. Li, S. Tasci, S. Ghosh, J. Zhu, J. Zhang, and L. Heck, "RILOD: Near real-time incremental learning for object detection at the edge," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 113–126.
[55] N. Kukreja *et al.*, "Training on the Edge: The why and the how," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 899–903.
[56] S. Disabato and M. Roveri, "Incremental on-device tiny machine learning," in *Proceedings of the 2nd International workshop on challenges in artificial intelligence and machine learning for internet of things*, 2020, pp. 7–13.
[57] T. Pimentel, M. Monteiro, A. Veloso, and N. Ziviani, "Deep active learning for anomaly detection," in *2020 International Joint Conference on Neural Networks (IJCNN)*, 2020, pp. 1–8.
[58] H. Ren, D. Anicic, and T. A. Runkler, "Tinyol: Tinyml with online-learning on microcontrollers," in *2021 International Joint Conference on Neural Networks (IJCNN)*, 2021, pp. 1–8.
[59] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. Garofalakis, "Complex event recognition in the big data era: a survey," *The VLDB Journal*, vol. 29, pp. 313–352, 2020.
[60] T. A. S. Fundation, "Apache flink." Feb. 2021. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.12
[61] TIBCO, "TIBCO." Nov. 2023. [Online]. Available: https://www.tibco.com
[62] M. D. de Assuncao, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, pp. 1–17, 2018.
[63] P. Graubner *et al.*, "Multimodal complex event processing on mobile devices," in *Proceedings of the 12th ACM international conference on distributed and event-based systems*, 2018, pp. 112–123.
[64] N. Govindarajan, Y. Simmhan, N. Jamadagni, and P. Misra, "Event processing across edge and the cloud for internet of things applications," in *Proceedings of the 20th International Conference on Management of Data*, 2014, pp. 101–104.
[65] M. Vrbaski, M. Bolic, and S. Majumdar, "Complex event recognition notification methodology for uncertain IoT systems based on micro-service architecture," in *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2018, pp. 184–191.
[66] C. Y. Chen, J. H. Fu, T. Sung, P.-F. Wang, E. Jou, and M.-W. Feng, "Complex event processing for the internet of things and its applications," in *2014 IEEE International Conference on Automation Science and Engineering (CASE)*, 2014, pp. 1144–1149.

[67] L. Lan, R. Shi, B. Wang, L. Zhang, and N. Jiang, “A universal complex event processing mechanism based on edge computing for internet of things real-time monitoring,” *IEEE Access*, vol. 7, pp. 101865–101878, 2019.
[68] H. Ren, D. Anicic, and T. A. Runkler, “The synergy of complex event processing and tiny machine learning in industrial IoT,” in *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, 2021, pp. 126–135.
[69] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, and R. Studer, “Etalis: Rule-based reasoning in event processing,” *Reasoning in event-based distributed systems*, pp. 99–124, 2011.
[70] J. Wanner, C. Wissuchek, and C. Janiesch, “Machine Learning and Complex Event Processing: A Review of Real-time Data Analytics for the Industrial Internet of Things,” *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, vol. 15, p. 1, 2020.
[71] N. Mehdiyev, J. Krumeich, D. Enke, D. Werth, and P. Loos, “Determination of rule patterns in complex event processing using machine learning techniques,” *Procedia Comput Sci*, vol. 61, pp. 395–401, 2015.
[72] R. Mousheimish, Y. Taher, and K. Zeitouni, “Automatic learning of predictive cep rules: bridging the gap between data mining and complex event processing,” in *Proceedings of the 11th ACM international conference on distributed and event-based systems*, 2017, pp. 158–169.
[73] R. Bruns, J. Dunkel, and N. Offel, “Learning of complex event processing rules with genetic programming,” *Expert Syst Appl*, vol. 129, pp. 186–199, 2019.
[74] Y. Wang, H. Gao, and G. Chen, “Predictive complex event processing based on evolving Bayesian networks,” *Pattern Recognit Lett*, vol. 105, pp. 207–216, 2018.
[75] A. Power and G. Kotonya, “Providing fault tolerance via complex event processing and machine learning for iot systems,” in *Proceedings of the 9th International Conference on the Internet of Things*, 2019, pp. 1–7.
[76] T. Xing *et al.*, “Deepcep: Deep complex event processing using distributed multimodal information,” in *2019 IEEE international conference on smart computing (SMARTCOMP)*, 2019, pp. 87–92.
[77] J. A. C. Soto, M. Jentsch, D. Preuveneers, and E. Ilie-Zudor, “CEML: Mixing and moving complex event processing and machine learning to the edge of the network for IoT applications,” in *Proceedings of the 6th International Conference on the Internet of Things*, 2016, pp. 103–110.
[78] ETSI, “GS CIM 009 - V1.7.1 - Context Information Management (CIM); NGSI-LD API,” 2023. [Online]. Available: https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx
[79] R. Fielding *et al.*, “Hypertext Transfer Protocol -- HTTP/1.1,” Jun. 1999. doi: 10.17487/rfc2616.
[80] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, “MQTT Version 5.0, OASIS Standard,” *OASIS Standard*. Mar. 07, 2019. Accessed: Nov. 29, 2023. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html
[81] Z. Shelby, K. Hartke, and C. Bormann, “The Constrained Application Protocol (CoAP),” Jun. 2014. doi: 10.17487/rfc7252.
[82] R. T. Fielding, “REST: architectural styles and the design of network-based software architectures,” 2000.
[83] B. Foubert and N. Mitton, “Lightweight network interface selection for reliable communications in multi-technologies wireless sensor networks,” in *2021 17th International Conference on the Design of Reliable Communication Networks (DRCN)*, 2021, pp. 1–6.
[84] B. Foubert and N. Mitton, “RODENT: a flexible TOPSIS based routing protocol for multi-technology devices in wireless sensor networks,” *ITU Journal on Future and Evolving Technologies*, vol. 2, no. 1, 2021.
[85] S. S. Craciunas, R. S. Oliver, M. Chmel\’\ik, and W. Steiner, “Scheduling real-time communication in IEEE 802.1 Qbv time sensitive networks,” in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, 2016, pp. 183–192.
[86] M. Vlk, Z. Hanzálek, and S. Tang, “Constraint programming approaches to joint routing and scheduling in time-sensitive networks,” *Comput Ind Eng*, vol. 157, p. 107317, 2021.
[87] F. Ansah, M. A. Abid, and H. de Meer, “Schedulability analysis and GCL computation for time-sensitive networks,” in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, 2019, pp. 926–932.

[88] M. Pahlevan and R. Obermaisser, "Genetic algorithm for scheduling time-triggered traffic in time-sensitive networks," in *2018 IEEE 23rd international conference on emerging technologies and factory automation (ETFA)*, 2018, pp. 337–344.
[89] T. Stüber, L. Osswald, S. Lindner, and M. Menth, "A Survey of Scheduling Algorithms for the Time-Aware Shaper in Time-Sensitive Networking (TSN)," *IEEE Access*, 2023.
[90] M. Vlk, Z. Hanzalek, K. Brejchova, S. Tang, S. Bhattacharjee, and S. Fu, "Enhancing Schedulability and Throughput of Time-Triggered Traffic in IEEE 802.1Qbv Time-Sensitive Networks," *IEEE Transactions on Communications*, vol. 68, no. 11, pp. 7023–7038, Nov. 2020, doi: 10.1109/TCOMM.2020.3014105.
[91] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner, "Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, New York, NY, USA: ACM, Oct. 2016, pp. 183–192. doi: 10.1145/2997465.2997470.
[92] P. Baptiste, P. Laborie, C. Le Pape, and W. Nuijten, "Constraint-based scheduling and planning," in *Foundations of artificial intelligence*, vol. 2, Elsevier, 2006, pp. 761–799.
[93] J. Schimpf and K. Shen, "ECLiPSe–from LP to CLP," *Theory and Practice of Logic Programming*, vol. 12, no. 1–2, pp. 127–156, 2012.
[94] G. N. Kumar, K. Katsalis, and P. Papadimitriou, "Coupling source routing with time-sensitive networking," in *2020 IFIP Networking Conference (Networking)*, 2020, pp. 797–802.
[95] G. Papathanail, L. Mamatas, and P. Papadimitriou, "Towards the Integration of TAPRIO-based Scheduling with Centralized TSN Control," in *2023 IFIP Networking Conference (IFIP Networking)*, 2023, pp. 1–6.
[96] D. Reed *et al.*, "Decentralized identifiers (dids) v1. 0," *Draft Community Group Report*, 2020.
[97] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström, "JSON-LD 1.1," *W3C Recommendation, Jul*, 2020.
[98] N. G. L. D. B. D. C. Z. B. Sporny M. and D. Chadwick, "Verifiable Credentials Data Model 1.1," 2022.
[99] O. Standard, "MQTT Version 5.0," *Retrieved June*, vol. 22, p. 2020, 2019.
[100] G. Selander, J. Mattsson, F. Palombini, and L. Seitz, "Object security for constrained restful environments (oscore)," 2019.
[101] M. B. Y. K. Microsoft Jones and T. Lodderstedt, "Self-Issued OpenID Provider V2," 2023.
[102] O. Terbu, T. Lodderstedt, K. Yasuda, and T. Looker, "OpenID for Verifiable Presentations - draft 18." Apr. 21, 2023. Accessed: Nov. 29, 2023. [Online]. Available: https://openid.net/specs/openid-4-verifiable-presentations-1_0.html
[103] B. J. J. M. de M. B. Sakimura N. and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1," 2014.
[104] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," May 2015. doi: 10.17487/RFC7519.
[105] Y. K. Lodderstedt T. and T. Looker, "OpenID for Verifiable Credential Issuance," 2023.
[106] V. Foteinos *et al.*, "A Cognitive Management Framework for Empowering the Internet of Things," in *The Future Internet*, A. Galis and A. Gavras, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 187–199.
[107] K. Khan, W. Albattah, R. U. Khan, A. M. Qamar, and D. Nayab, "Advances and trends in real time visual crowd analysis," *Sensors*, vol. 20, no. 18, p. 5073, 2020.
[108] J. Shao, C. Change Loy, and X. Wang, "Scene-independent group profiling in crowd," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 2219–2226.
[109] L. Al-Salhie, M. Al-Zuhair, and A. Al-Wabil, "Multimedia Surveillance in Event Detection: Crowd Analytics in Hajj," 2014, pp. 383–392. doi: 10.1007/978-3-319-07626-3_35.
[110] S. Herath, S. Irandoust, B. Chen, Y. Qian, P. Kim, and Y. Furukawa, "Fusion-dhl: Wifi, imu, and floorplan fusion for dense history of locations in indoor environments," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 5677–5683.
[111] O. M. Alliance, "OMA LightWeight M2M (LWM2M) Object and Resource Registry." 2014.