DOCTORAL SCHOOL
MEDITERRANEA UNIVERSITY OF REGGIO CALABRIA

DEPARTMENT OF INFORMATION ENGINEERING, INFRASTRUCTURES AND SUSTAINABLE ENERGY (DIIES)

PHD IN
INFORMATION ENGINEERING

S.S.D. ING-INF/03
XXXVI CYCLE

Physical devices virtualization in edge computing environments

CANDIDATE
Giacomo GENOVESE

ADVISOR
Prof. Antonella MOLINARO

COORDINATOR
Prof. Antonella MOLINARO

REGGIO CALABRIA, OCTOBER 2023

Finito di stampare nel mese di **XXX 2024**

Edizione

Quaderno N. XX

Collana *XXXXXXXX*
Curatore *XXX*

ISBN XXX-XX-XXXXX-XX-X

Università degli Studi *Mediterranea* di Reggio Calabria
Via dell'Università 25. Reggio Calabria

Giacomo GENOVESE

Physical devices virtualization in edge computing environments

”Ubiquitous computing will enable nothing fundamentally new, but by making everything faster and easier to do, with less strain and mental gymnastics, it will transform what is apparently possible”

(Mark Weiser, The Computer for the 21st Century,1991)

Abstract

Internet of Things (IoT) has revolutionized the use of the network and it is one of the main drivers of new-generation networks, such as 5G and 6G. New applications require ever-increasing resource sharing and real-time analysis of large amounts of small but significant data. However, challenges persist due to the use of billions of IoT constrained devices with limited computational and energy resources which produce continuous traffic in the network using often non-interoperable protocols and solutions. The major issues include:

- *Integration*: IoT solutions find application in various fields, each with specific requirements. The use of different hardware and communication standards poses integration difficulties, especially with existing solutions or those from other manufacturers, unless specific interventions are made, requiring the creation of dedicated gateways and/or middleware.
- *Energy Consumption Optimization*: IoT devices are often battery-powered, and saving energy, including optimizing computational and transmission processes, may extend the device's lifetime.
- *Semantic Interoperability*: The heterogeneity of solutions also involves using different definitions for data, semantic structures, even if they actually refer to the same sources. For example, in two distinct solutions, a temperature sensor might be defined as *temp* in the first and *Temperature* or T in the second. This situation could be enough to prevent the direct use of data independently of the source device, requiring the use of a standardization level for the data to have the same meaning reciprocally in both solutions.
- *Security and Accessibility*: Most security systems rely on high computational complexity necessary to support it. Implementing this on IoT devices is challenging and comes at the expense of other functionalities, such as device accessibility to third parties. However, interconnected objects manage important aspects of our daily lives and cannot do without security.

To overcome these challenges, current research is developing new techniques and paradigms to converge IoT technologies with the Cloud-Edge computing continuum, creating seamless interaction between IoT devices and hyper-distributed applications. Efforts have focused on the virtualization of both IoT devices and networks to create a single hyper-distributed and interoperable environment for implementing new and more efficient services.

Virtual Objects (VOs), or Digital Twins (DTs), are software components representing the counterparts/extensions of physical objects deployed within the virtualized network infrastructure like the Multi-Access Edge Computing (MEC) defined by the European Telecommunications Standards Institute (ETSI). VOs, freed from the constraints of the physical world, provide greater interoperability, better access to resources, improved device computational capabilities, data historization, and contextual awareness offering new functionalities for new applications. In virtualized environments, collaboration between different devices can be further facilitated by introducing an additional virtual component, the Composite VO (cVO). The cVO aggregates multiple VOs to develop composite services in support of specific applications.

This thesis aims to address modern IoT challenges, focusing on the development of VOs and cVOs using standard protocols and semantics to enhance interoperability, and on the creation of a software stack, the VOStack, which facilitates the convergence between IoT and the virtualized infrastructures of Edge and Cloud computing. The result is the development of a microservice, the VO, based on the VO model. This microservice is designed as software for a specific service, i.e., the virtualization of physical devices, and it is independent of a particular device. The designed VO utilizes the Open Mobile Alliance Lightweight Machine-to-Machine (OMA-LwM2M) semantic standard and, at startup, initializes and shapes its functionalities based on the properties of the corresponding physical counterpart it represents. It addresses challenges related to semantic interoperability, communication protocol heterogeneity, energy savings, enhanced functionalities, scalability, and orchestration.

Abstract

L'Internet of Things (IoT) ha rivoluzionato l'utilizzo della Rete ed è uno dei principali motori delle reti di nuova generazione, come 5G e 6G. Le nuove applicazioni richiedono una condivisione sempre maggiore di risorse e l'analisi in tempo reale di grandi quantità di piccoli ma significativi dati. Tuttavia, non sono cambiate le problematiche dovute all'utilizzo di miliardi dispositivi IoT con limitata disponibilità di risorse computazionali e di energia, definiti "constrained", che producono continuo traffico nella rete utilizzando protocolli e soluzioni spesso non interoperabili tra loro. Le maggiori problematiche sono:

- *Integrazione.* Le soluzioni IoT trovano applicazione nei campi più disparati ed ognuna di esse ha requisiti specifici da soddisfare. L'utilizzo di hardware e di standard di comunicazione differenti comporta difficoltà di integrazione tra sistemi, specie con soluzioni pre-esistenti o di altri produttori, a meno di specifici interventi che richiedono la creazione di appositi gateway e/o middleware.
- *Ottimizzazione del consumo energetico.* I dispositivi IoT spesso sono alimentati a batteria e risparmiare energia, anche attraverso l'ottimizzazione dei processi computazionali e di rice-trasmissione, significa estendere la vita del dispositivo.
- *Interoperabilità semantica.* L'eterogeneità delle soluzioni comporta anche la scelta di utilizzare diverse definizioni per i dati, strutture semantiche, anche se questi in realtà sono associabili a fonti identiche. A esempio, in due distinte soluzioni, un sensore di temperatura potrebbe essere definito nella prima con "temp" e nella seconda con "temperature". Questa situazione potrebbe essere sufficiente ad impedire l'utilizzo diretto dei dati indipendentemente dalla fonte e richiede l'utilizzo di un livello di omologazione affinché il dato possa assumere lo stesso significato in entrambe le soluzioni.
- *Sicurezza e Accessibilità.* La maggior parte dei sistemi di sicurezza si basa sull'alta complessità computazionale necessaria a risolverla. Con questa premessa è ovvio che l'implementazione su dispositivi IoT risulta difficoltosa e a totale discapito di altre funzionalità quali, a esempio, l'accessibilità dei dispositivi a terze parti. Tuttavia, gli oggetti interconnessi gestiscono importanti aspetti della nostra quotidianità e non possono prescindere dalla sicurezza.

Per superare le problematiche appena elencate, oggigiorno, la ricerca sta sviluppando nuove tecniche e paradigmi per far convergere le tecnologie IoT con il continuum di Cloud-Edge computing creando una interazione senza soluzione di continuità tra i dispositivi IoT e le applicazioni iper-distribuite. In particolare, gli sforzi si sono concentrati verso la virtualizzazione sia dei dispositivi IoT che delle reti nel tentativo di creare un unico ambiente iper-distribuito ed interoperabile per l'implementazione di nuovi servizi.

I Virtual Object (VO), o Digital Twin (DT), sono componenti software che rappresentano le controparti/estensioni degli oggetti fisici implementati all'interno

dell'infrastruttura di rete virtualizzata come il Multi Access Edge Computing (MEC) previsto dall' European Telecommunications Standards Institute (ETSI). I VO, liberandosi dei vincoli appartenenti al mondo fisico, sono in grado di fornire maggiore interoperabilità, migliore accesso alle risorse, migliorare le capacità del dispositivo dal punto di vista computazionale, storicizzare i dati, avere consapevolezza del contesto, fornendo così nuove funzionalità al servizio di nuovi applicativi. Inoltre, in ambienti virtualizzati, la collaborazione tra diversi dispositivi può essere ulteriormente favorita attraverso l'introduzione di un ulteriore componente virtuale, il Composite VO (cVO). Il cVO ha il compito di aggregare più VO al fine di sviluppare servizi compositi a supporto di specifici applicativi.

Il lavoro presentato in questa tesi si è posto come obiettivo quello di affrontare le moderne problematiche del mondo IoT e, nello specifico, si è concentrato sullo sviluppo di VO e cVO attraverso l'utilizzo di protocolli e semantica standard, al fine di favorire l'interoperabilità e la creazione di un software stack, il VOStack, che favorisca la convergenza tra l'IoT e le infrastrutture virtualizzate di Edge e Cloud computing. Il contributo del lavoro svolto può essere sintetizzato con lo sviluppo di un microservizio, il VO. Il VO sviluppato si basa sul VO model. Ovvero, il microservizio è progettato come software per un servizio specifico (la virtualizzazione di dispositivi fisici) ed è indipendente da un dispositivo particolare. Il VO sfrutta lo standard semantico Open Mobile Alliance Lightweight Machine-to-Machine (OMA-LwM2M) e all'avvio, letto il file di configurazione, si inizializza e plasma le sue funzionalità in base alle proprietà del corrispondente omologo fisico che rappresenta, fornendo una risposta alle problematiche relative a interoperabilità semantica, eterogeneità dei protocolli di comunicazione, risparmio energetico, miglioramento delle funzionalità, scalabilità, ed orchestrazione.

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
3GPP	Third Generation Partnership Project
HC	HTTP-to-CoAP
AMQP	Advanced Message Queuing Protocol
IETF	Internet Engineering Task Force
CoAP	Constrained Application Protocol
ETSI	European Telecommunications Standards Institute
API	Application Programming Interface
AR	Augmented Reality
CBOR	Concise Binary Object Notation
cVO	Composite Virtual Object
DDF	Device Description File
DIDs	Decentralized Identifiers
DT	Digital Twin
DHCP	Dynamic Host Configuration Protocol
EMS	Element Management Systems
XMPP	eXtensible Messaging and Presence Protocol
HTTP	Hyper Text Transfer Protocol
IaaS	Infrastructure as a service
IDE	Integrated Development Environment
IoT	Internet of Things
IPSO	Internet Protocol Security Option
IANA	Internet Assigned Numbers Authority
IT	Information Technology
JSON	Java Script Object Notation
LAN	Local Area Network
LD	Linking Data

LwM2M	Lightweight Machine-to-Machine
MANO	NFV Management and Orchestration
MaaS	Mobility as a Service
M2M	machine-to-machine
MEC	Multi-acces Edge Computing or Mobile Edge computing
MQTT	Message Queuing Telemetry Transport
MTC	Machine Type Communication
NEF	Network Exposure Function
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
NFVO	NFV Orchestrator
NG	New Generation
NGSI	Next Generation Service Interface
OBU	On Board Unit
OF	OpenFlow
OMA	Open Mobile Alliance
OOP	Object-Oriented Programming
OSCORE	Object Security for Constrained RESTful Environments
OSM	Open Source Mano
OSS	Operation Support System
PaaS	Platform as a service
PoC	Proof of Concept
POM	Project Object Model
QoE	Quality of Experience
QoS	Quality of Service
RAN	Radio Access Network
REST	Representional State Transfert
RGW	Residential Gateway
SaaS	Software as a service
SLA	Service Level Agreement
SDN	Software-defined Networking
SDO	Standard Development Organization
SMF	Session Management Function
SQL	Structured Query Language
TLS	Transport Layer Security
TLV	Type–length–value

TD	Things Description
TSDB	Time Series Database
UDP	User Datagram Protocol
UPF	User Plane Function
VCs	Verificable Credentials
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Function
VNFM	VNF Manager
VO	Virtual object
VOStack	Virtual Object Stack
W3C	World Wide Web Consortium
WoT	Web of Things
WSN	Wireless Sensor Network
WWW	World Wide Web
XML	eXtensible Markup Languag
YAML	YetAnother Mark-up Language

1 Introduction

Nowadays, the world around us is becoming increasingly interconnected. The advancements in next-generation Internet of Things (IoT) and Edge Computing technologies are manifest. Already in 2021, the global number of connected IoT devices was expected to increase by 9% and reach 12.3 billion active endpoints by 2025; the latest forecasts show that such number expects to grow to 16.7 billion[1]. These technologies are not only reshaping businesses and individuals' daily lives but also generating solutions tailored for various industrial sectors. They serve as the cornerstone for creating a fully interconnected world [11].

The IoT evolution is closely linked to the growing heterogeneity of IoT devices and technologies [12] [13]. This heterogeneity manifests in the production of diverse intelligent IoT devices, the support for various communication protocols, and the conceptualization of different information models for semantically representing entities within the IoT landscape. These trends inherently highlight the necessity for innovative architectural approaches designed to facilitate complete convergence and integration among existing and evolving IoT and edge computing technologies.

Simultaneously, the landscape of data processing and analytics, currently concentrated in centralized computing facilities (such as cloud data centers) at 80%, is expected to undergo a significant shift toward *edge* computing facilities [14]. Moreover, principles governing distributed computing are undergoing profound changes in their lifecycle orchestration paradigms, aiming to efficiently leverage resources across the *continuum*, from cloud to edge to IoT. To effectively navigate data management and analysis across such a distributed environment, emerging hyper-distributed applications increasingly adopt *microservices*-based and cloud-native computing technologies.

[1] IoT Analytics (2023). https://iot-analytics.com/number-connected-iot-devices/

During this transition, two primary challenges come to the forefront. The first challenge involves the imperative for *convergence* of IoT technologies, facilitated by innovative architectural approaches. These approaches must ensure continuous and seamless openness and interoperability across the diverse arrays of existing and emerging IoT solutions, models, and devices. Simultaneously, they should facilitate analytics to gauge the lifecycle costs of these technologies, considering factors such as time and resource consumption (ranging from seconds or watts to CO_2 emissions). The second challenge revolves around establishing an integrated *meta-orchestration* environment tailored for hyper-distributed applications. This entails fostering synergy between cloud and edge computing orchestration platforms to optimally handle end-to-end deployment and data provision for applications across the continuum.

To meet the above-mentioned challenges, this thesis aims to contribute to the development of a fundamental innovation, which can be considered the core of the future IoT and edge computing software stack. This innovation revolves around leveraging the virtualization of IoT devices within the edge infrastructure and supporting openness and interoperability aspects in a device-independent manner. This key component is referred to as the Virtual object (VO). *The VO is the physical IoT device extension into the digital virtualized network environment.* Through a service-oriented architecture, specifically through microservices, the unified management of a diverse range of IoT devices and platforms can be achieved. Simultaneously, edge computing functionalities can be dynamically provided on demand, ensuring efficient support for the connection and management of IoT devices in the course of application operations. This approach has the further advantage of eliminating the need for intermediary middleware platforms.

The VO plays a pivotal role as a key component in ~~a~~ collaborative meta-orchestration framework designed to oversee the coordination between orchestration platforms in both cloud and edge computing. It is required for enabling the synergy among different orchestration systems/platforms by generalizing and modelling their orchestration modules. Tailored for hyper-distributed applications on the next-generation compute continuum, such frameworks achieve their objectives through high-level scheduling supervision and definition, employing a *system-of-systems* approach. In this scenario, the VO becomes instrumental in facilitating seamless coordination and efficient management across diverse computing environments.

The research work in this thesis focuses on a general-purpose VO design and its deployment and customization in different vertical scenarios. Specifically, the VO design has been adapted by the creation of a semantic description for IoT devices within many different contexts, such as Mobility-as-a-Service (MaaS) applications [15] (e.g., bus, vehicle virtualization), Smart Factories (e.g., a brewery's equipment virtualization), and Aerospace (e.g., CubeSat virtualization). Also a characterization of the VO in an edge Artificial Intelligence (AI) general context has been developed [16]. This thesis work has contributed to the development of some national P.O.N. 2014-2020 research and innovation projects: *eBrewery* "Virtualization, sensing and IoT for innovation of the industrial production process of beverages" (ARS0100582), *PM3* "Multi-Mission Modular Platform" (ARS0101181), *MyPasS* "Mobility for Passengers as a Service"(ARS0101100), and within the European project NEPHELE "A Lightweight Software Stack and Synergetic Meta-Orchestration Framework for the Next Generation Compute Continuum" (Horizon-CL4-2021 Proposal number 101070487).

This thesis is organized as follows:

- **Chapter 2** presents an overview on the heterogeneous and complex IoT world, setting the background and motivations of this thesis work. A short overview of the shift from cloud to edge computing in the IoT context is presented, with related advantages and challenges. The main application protocols for communications with IoT devices are detailed in this chapter, with special focus on Constrained Application Protocol (CoAP) and Message Queuing Telemetry Transport (MQTT). Attention is also dedicated to solutions for semantic interoperability in the heterogeneous IoT context, such as Open Mobile Alliance (OMA)-Lightweight Machine-to-Machine (LwM2M), Web of Things (WoT), and Next Generation Service Interface (NGSI)-Linking Data (LD).
- **Chapter 3** deals with IoT device virtualization and presents the concept of VO as a means to overcome the scarcity of IoT resources and achieve IoT interoperability and scalability, by creating a digital extension of the physical device, with which consumers and third parties can interact. The VO can be hosted at the edge premises' infrastructure and be orchestrated as a microservice. The designed VO stack is introduced and its main layers are detailed: *Physical convergence* layer; *Edge/Cloud convergence* layer, and *Backend logic* layer.
- **Chapter 4** focuses on the softwarized and virtualized environment where deploying and orchestrating VOs, considering the recent standardization efforts within the European Telecommunications Standards Institute (ETSI) and Third Generation Partnership Project (3GPP) organizations, with emphasis on Multi-acces Edge Computing or Mobile Edge computing (MEC), Software-defined Networking (SDN) and Network Function Virtualization (NFV).

- **Chapter 5** presents details of the VO design and developmemt platform: its model for semantic interoperability based on OMA-LwM2M; its datastores choices (SQLite and InfluxDB), and the northbound and southbound interfaces respectively towards the applications and the IoT devices. In this chapter, also the concept of Composite Virtual Object (cVO) is introduced.
- **Chapter 6** describes the customization of the developed VO concept within different vertical contexts, and presents the main contributions to the research activities within the eBrewery and MyPasS national projects.
- **Chapter 7** summarizes conclusions and open issues.

2

Internet of Thing scenario

This Chapter introduces the thesis background and motivations by providing a broad description of the Internet of Things (IoT) scenario and its evolution, together with the main protocols for communications with IoT devices, and solutions for semantic interoperability.

2.1 IoT Overview and Challenges

A long time has passed since, in 1982, an automatic machine for dispensing cold drinks was connected to transmit updated information on the list of available drinks and warned when the beverages had reached the right temperature. This innovation marked the beginning of a new era, characterized by the idea of having connected intelligent objects and smart devices. Many years later, in 1999, the word "Internet of Things" was first defined by Kevin Ashton, the executive director of the Auto-ID center. Since then, the IoT scenario has deeply evolved and nowadays billions of connected devices expose their services to other devices in a machine-to-machine (M2M) oriented communication, to applications, and to humans.

In simple terms, the IoT paradigm enables communication among physical objects, connected to a global infrastructure, and between objects and people. Billions of heterogeneous devices, in terms of application scenario, hardware, and software features, are able to communicate autonomously. These smart objects interact with the environment around them through the use of sensors and actuators, generating a huge amount of data, transmitted remotely and transformed into information. This creates a digital representation of the physical world, which intends to be available and reachable always (every time) and from every part of the world (everywhere).

On the one hand, this facilitates the creation of a rich set of services and applications, offers new opportunities, experiences and business logic, giving rise to the revolution in the field of efficient energy management (Smart Grid), cities (Smart

City), buildings and infrastructures (Smart Building), factories (Smart Factory), personal care (eHealth) and many other sectors. On the other hand, it presents important challenges due to the limited availability of computational and energy resources of IoT devices, defined as "constrained". These devices send data across the network to their respective remote Clouds, and use protocols and solutions which are often not interoperable. The main issues related to the IoT are highlighted in [17] [18] and described in the following.

Interoperability

IoT solutions find application in the most disparate fields, and each of them has specific requirements to satisfy. The use of different hardware and communication standards entails difficulties in integrating systems, especially with pre-existing solutions or those from other manufacturers, unless specific interventions require the design of specific gateways and/or middleware. The heterogeneity of the solutions also involves the choice of using different definitions for the data, semantic structures, even if these in reality can be asserted from identical sources. For example, in two distinct solutions, a temperature sensor could be defined in the first as "temp" and in the second as "temperature". This situation is sufficient to prevent the direct use of the data regardless of the source, and requires the use of a level of approval so that the data can assume the same meaning reciprocally in both solutions. It is clear that interoperability becomes fundamental to enable cooperation between IoT devices and applications.

Security and Accessibility

The IoT offers significant potential for managing all the devices we use daily. However, it also poses a substantial risk for cybercriminals who can exploit the routers, televisions, refrigerators, and other Internet-connected devices in our homes to execute widespread and distributed attacks. The threat posed by Internet-enabled devices is substantial due to their susceptibility to infiltration, the limited motivation for consumers to enhance their security, the ability of the rapidly increasing number of devices to transmit malicious content with minimal detection, the lack of proactive measures by many vendors to counter this threat, and the inadequacy of the existing security model in addressing this issue. Most security systems rely on the high computational complexity needed to solve it. With this premise, it is obvious that implementation on IoT devices is difficult and to the total detriment of other functions, such as the accessibility of devices to third parties. However, interconnected objects manage important aspects of our daily life and cannot ignore security.

Privacy

The notion of privacy is deeply ingrained in our societies, acknowledged in the legislations of civilized nations, and, as previously mentioned, concerns regarding its safeguarding have proven to be a significant hindrance to the widespread adoption of technologies associated with the IoT. People's anxieties about privacy are indeed well-founded. The methods through which data collection, mining, and provisioning will occur in the IoT differ significantly from what we currently understand, providing numerous opportunities for the collection of personal data. As a result, individual control over the disclosure of personal information will become practically impossible for people. It becomes particularly difficult especially considering the permeability of IoT devices and their Information Technology (IT) weakness, due to the limited computational capabilities of relatively cheap hardware.

Scalability

IoT systems are often distributed across multiple devices and applications, and re-modulate and scale services is quite difficult in this case. Indeed, the rapid growth of IoT systems, accompanied by the increasing complexity of service compositions, underscores the criticality of scalability. To effectively manage and handle the expanding volume of data and interactions, IoT systems need to seamlessly scale to accommodate the growing demands [19]. Scalability challenges encompass all IoT aspects, some of them already described in this section:

- *Data Volume*: the sheer volume of data generated by IoT devices poses a significant scalability hurdle. As the number of connected devices increases, the data stream also expands, demanding efficient data processing and storage capabilities, which cannot be allocated to IoT constrained physical devices.
- *Real-time Processing*: IoT systems often require real-time processing of data to enable timely responses and decision-making. This real-time constraint adds another layer of complexity to scalability, as systems need to handle data streams without performance degradation.
- *Geographical Distribution*: IoT systems often span large geographical distances, with devices distributed across different locations. This geographical distribution imposes scalability challenges in terms of network bandwidth, latency, and fault tolerance.

- *Resources capability*: IoT systems and devices needs to scale according to task activity and external infrastructure requirements.

Network requirements

Current mobile technologies, exemplified by Long Term Evolution (LTE) and LTE-Advanced (LTE-A) or fourth-generation (4G) systems, were primarily designed to handle High Throughput Computing (HTC) traffic. Consequently, the impending arrival of fifth-generation (5G) networks necessitates a redesign of mobile system transmission procedures to seamlessly accommodate both human-type communication and Massive Machine Type Communication (MTC) traffic characterized by small packet size, high frequency and low latency requirements, ensuring compliance with the distinct requirements of these heterogeneous traffic types. The standardization efforts for 5G systems are geared towards facilitating a multitude of business cases emerging from the IoT [20]. Taking it a step further, the disruptive technologies driving the deployment of 5G systems aim to introduce flexibility, customization, and re-configurability in both the radio and core segments of the network. This approach is poised to provide enhanced IoT services that connect people and everything [17]. Beyond simply connecting devices to the Internet, the natural progression involves remotely controlling these devices through the Internet. Notably, 5G not only facilitates communication between machines but also emphasizes enabling innovative industry-specific IoT applications in both consumer and business environments. Examples include enhancing industry automation, enabling remote control, and supporting tactile Internet applications [21].

Context awareness

Our world is transitioning into an era where billions of sensors will be accessible for utilization through scalable and reconfigurable services. [22] This new scenario needs IoT context awareness, the capability of IoT systems, device and platforms, to understand and respond to the context in which they physically and virtually operate. IoT systems need to automatically gather and interpret information about the environment, users, and connected devices, enabling the IoT system to make informed decisions and provide more relevant and personalized services. They have to sense and understand the world around them also with the help of new technologies such as AI. For instance, in the IoT paradigm, choosing the most suitable sensors that can offer pertinent sensor data to address specific issues, amidst billions of possibilities, presents a challenge but it can ensure more efficient system and reduce the number of IoT resources deployed. Moreover, for instance, in critical operations, it can allow rescuers' equipment to immediately interconnect with the systems (network infrastructure, platforms, geolocation service, etc) and IoT devices present on site to acquire all the information available to optimize intervention operations.

2.2 The IoT and Cloud/Edge computing convergence

The long wave of cloud computing, having radically changed the way in which relevant services are built and provided in the Internet, has affected also the IoT domain. The interest on the Cloud-IoT paradigm was the result of the blending of IoT and cloud computing technologies together. The reference applications of this model are substantially inherited by the IoT domain, as the end-devices' technologies do not change and still represent the information sources that originate and feed the applications [23].

Cloud computing technologies make the resulting system powerful by complementing the more traditional IoT technologies for communication, computing, and storage. Indeed, the big amount of data generated by physical devices badly matches the limited local memory capacity and the cost of adding local memory. Sending data to a sink, gateway, or local server is a viable solution, but the best option is definitely to send them to the cloud where additional and powerful functionalities can be linked to such data, e.g., encryption, authentication, duplication, annotation.

Besides, for a device to take part to the broad IoT application portfolio the first requirement is to speak different protocols and support different interaction mechanisms, loosely or tightly coupled, synchronized or asynchronous for complex event processing. Implementing these protocols and procedures in the things is often prohibitive, given the mismatch between available and required resources. The cloud, instead, can provide different views of the service/data offered by a thing and relieve this latter from the burden of responding to repetitive requests from different clients for the same data.

To date, the growing number of IoT-based solutions, such as home automation, wearables, smart mobility and interconnected vehicles, is exponentially increasing the amount of connected devices and data transmitted over the network. As reported in the Cisco annual Report [24], in 2023 approximately 30 billion devices will be connected, of which 50% is M2M devices. This awareness is bringing an important architectural change in the IoT field, from Cloud Computing to Edge Computing. Many IoT applications, indeed, must guarantee adequate Quality of Service (QoS) and Quality of Experience (QoE), mainly in terms of analysis and real-time response to events (e.g., autonomous driving, remote surgery). In such cases, the Cloud could not suit the applications requirements in terms of latency, data privacy, and network burden caused by the large amount of data traversing the core network to the remote Cloud. In this context, the MEC paradigm promotes the use of nodes at the edge of the network (Edge nodes) composed of specific services, wireless and non-wireless access points. From an IoT point of view, this solution guarantees both faster access to resources and the distribution of analytical loads in the various nodes, and the advantage of being able to manage and pre-process the traffic generated towards the core of the network. It therefore allows address the main problems of Cloud-based architectures.

2.3 IoT virtualization and edge computing

The resource limitations of IoT devices highlighted the need *to expand the capacity of IoT devices beyond their physical limits*, by using the digital space, or cyber-space, available to applications in the Cloud/Edge. The creation of *virtualization* levels in the IoT architectures has led to the birth of *virtual counterparts of physical objects*, the so-called Virtual Objects (VOs) [25] or, more simply, Digital Twins (DTs).

VOs are becoming a key component capable of acquiring, analyzing, interpreting information relating to the context, improving the security of devices and addressing problems related to the interoperability of IoT solutions. From this perspective, in particular, the VO can allow the management of multiple standards and models and facilitate cooperation between platforms by encouraging the sharing of resources. Moreover, VOs can lighten the workload of physical devices by limiting access to them, thus substantially extending the life of battery-powered devices and reducing the wear and tear on electronic components.

In the new proposed solutions, architectural elements such as VOs are instantiated close to end users, at the network edge, to increase the responsiveness of applications.

In the subsequent sections, the major application protocols for communication with IoT devices, and semantic description solutions are summarized, as clear examples of the high heterogeneity in the IoT world.

2.4 IoT Standard developing organizations

Prominent Standard Development Organizations (SDOs), including 3GPP, ETSI, Internet Engineering Task Force (IETF), and oneM2M, advocated for a unified, cost-effective, easily deployable M2M service layer that can be integrated into diverse hardware and software environments to facilitate communications between devices and their interoperability. However, the proliferation of distinct standards has given rise to a significant challenge of interoperability. Recognizing this limitation, various SDOs have collaboratively addressed the need for common standardized solutions, introducing multiple layers of interoperability [26]. A short overview is outlined below delineating the key organizations that have actively contributed to the semantic dimensions of interoperability concerning IoT devices.

Third Generation Partnership Project (3GPP). The standardization efforts have been carried out for the network domain to the creation of the Mobile Broadband Standard, with an increasing emphasis towards connecting the IoT – whether the need is for ultra-reliable low latency communications at one end of the scale or for energy efficient low-cost, low-power sensors and devices at the other [27].

European Telecommunications Standards Institute (ETSI). ETSI works to establish efficient telecommunication systems to protect people in an emergency situation, as well as on security issues in next generation networks, M2M, intelligent transport systems (ITS), and among others, also development of standards for IoT easy data access through network infrastructure (cellular or fixed) and providing end-to-end service capabilities [28].

oneM2M. OneM2M is a global organization, its task is to create a world of interoperable and secure IoT services where market adoption is easy and delivers benefits to society. The standard works on technical specifications addressing the requirement of a common M2M service layer between the network and the application domains, and to develop cooperative optimization standards. These specifications consider the requirements, functional architecture, service layer core protocol specifications, security solutions and mapping to common industry protocols such as CoAP, MQTT and HTTP [29].

Internet Engineering Task Force (IETF). IETF Working Groups, spanning multiple areas are developing protocols and best common practices that are directly relevant to the communication and security aspects of IoT. Groups collectively facilitate the IP-based integration of constrained devices into the Internet in a standardized way. [30]

Organization for the Advancement of Structured Information Standards (OASIS). OASIS plays a pivotal role in the IoT domain for the creation and maintenance of standards across diverse domains, from security and privacy to cloud computing and web services. It released several IoT standards, like the MQTT protocol, which plays a pivotal role in enabling efficient communication between IoT devices and platforms [31].

Open Mobile Alliance (OMA). The OMA is a consortium that collaborates on developing open standards for the mobile industry, ensuring interoperability and a harmonious mobile ecosystem with any cellular network technologies being used to provide networking and data transport, and defines the requirements for objects like device, network, etc. Some standards produced by OMA are OMA-Device Management (OMA-DM) and OMA-LwM2M [32].

Internet Protocol Security Option (IPSO). IPSO primarily supports IP networked devices to be used in healthcare, energy, and industrial applications. The aim of IPSO alliance is to create a better understanding of IP and its role in connecting smart objects. Actually, IPSO merged to OMA forming a OMA SpecWorks [33].

World Wide Web Consortium (W3C). W3C aims to ensure that the evolving Web platform and Web technologies improve in integrity, security and privacy. Moreover, it is focused on deployment of the extensible web architecture to empower industries and individuals to address the evolving user's needs [34].

2.5 IoT Application Protocols

IoT protocols were created to serve fairly simple network architectures, as in Figure 2.1. However, over the years, with the increase in scenarios, the protocols evolved trying to satisfy increasingly stringent requirements in terms of QoS and, specifically, latency, computational cost, and energy consumption. There have been various solutions proposed like Advanced Message Queuing Protocol (AMQP) or eXtensible Messaging and Presence Protocol (XMPP) but, to date, two protocols in particular stand out joining the undisputed protagonist protocol of web traffic, the Hyper Text Transfer Protocol (HTTP). These protocols are CoAP, a server-client protocol based on Representional State Transfert (REST) specifications, and MQTT, a publisher-subscriber scheme based on message labeling.

The main differences amomg such protocols are summarized in Table 2.1.

2.5.1 Constrained Application Protocol (CoAP)

CoAP follows a REST architecture similar to the HTTP protocol. A resource is an abstract concept that is assigned to a universal identifier, called Uniform Resource Identifier (URI), which can be used on the Web. Access to a URI provides a representation of the resource as a response.

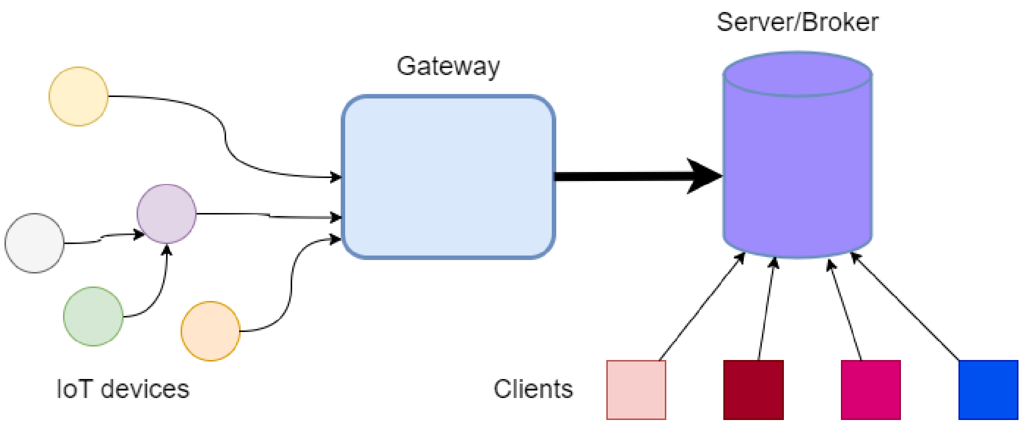


Fig. 2.1. IoT Architecture

Protocol	Transport	QoS options	Architecture	Security
CoAP	UDP	YES	Request/Response	DTLS
MQTT	TCP	YES	Publish/Subscribe	TLS/SSL
XMPP	TCP	NO	Request/Response, Publish/Subscribe	TLS/SSL
AMQP	TCP	YES	TLS/SSL	TLS/SSL

Table 2.1. Major differences among protocols [10]

One of the main objectives of CoAP 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 of a subset of the features offered by the ReST architecture in common with HTTP. The main features of CoAP are:

- web protocol for network nodes with limited resources;
- transport on User Datagram Protocol (UDP) with optional reliability;
- asynchronous exchange of messages;
- low overhead and low header parsing complexity;
- support for 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 end-point.

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 containing a response code and a representation of the requested resource, if any. Unlike HTTP, CoAP performs these request/response exchanges asynchronously over UDP. This is achieved through a protocol layer of messages that can support optional reliability through exponential backoff algorithm.

CoAP messages can be of 4 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 is 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.

Communication between CoAP hosts takes place according to a request/response pattern similar to HTTP: a client sends one or more requests to a server that processes the request and sends a response. Unlike HTTP, requests and responses are not sent over a previously established connection but are exchanged asynchronously through the CoAP message sublayer. A CoAP request consists of a method to be applied to a resource, the URI identifier of that resource, a possible payload with a Content-Format indicator, and any additional metadata. A confirmable or non-confirmable request message is created by specifying in the Code field of the header the Method code of the request (codes 1-31) together with other additional information included in the message. A host that receives a request with unrecognised or unsupported method code must send a piggy-backed response with Method Not Allowed (Response code) 4.05.

The methods supported by the protocol are a subset of the HTTP protocol request methods that are as follows:

- GET: it requests a representation of the information corresponding to the resource identified by the URI option included in the message. It can include an Accept option that specifies the preferred format (Content-Format) for the response, or an ETag option that requires you to send a response confirming the validity of the stored corresponding response, if any, or to send the resource representation if the validity has not been confirmed. The response codes can be 2.05 (Content) or 2.04 (Valid) if the saved answer corresponding to the indicated ETag is confirmed valid. The GET method is safe and idempotent.

- POST: it requires that the representation of the resource identified by the URI field be processed through a function performed by the server where that resource resides. This function is established by the server itself and is dependent on the indicated resource. Typically, the result is to update the indicated resource or to create a new resource if it is not present on the server. The response code is 2.01 (Created) if a new resource has been created (in which case you can include one or more Location-Path and/or Location-Query options related to the new resource), 2.04 (Changed) if an existing resource has been modified without creating a new one, or 2.02 (Deleted) if the indicated resource has been deleted. The POST method is neither secure nor idempotent.
- PUT: it requests that the resource identified by the URI field indicated in the message be updated or created with the representation included in the request. The impersonation format can be specified by including a Content-Format option. If the resource indicated by the URI exists, the representation included in the message is considered a modified version of the resource itself and must be answered with a code 2.04 (Changed) if the change has been successful. If the indicated resource does not exist on the server, the server can create a new one identified by the indicated URI by responding with code 2.01 (Created). If the indicated resource cannot be modified or created, the server must respond with an appropriate error code. Additional restrictions on the PUT method can be imposed through the If-Match or If-None-Match options. The PUT method is not safe but it is idempotent.
- DELETE: it requests that the resource identified by the URI field indicated in the message be deleted. If the resource delete operation is successful or the indicated resource did not exist before the request, the response message must be coded 2.02 (Deleted). The DELETE method is not safe but it is idempotent.

CoAP supports four security modes: 1) unsecured, 2) pre-shared key with Advanced Encryption Standard (AES) ciphers, 3) raw public keys using Datagram Transport Layer Security (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 Concise Binary Object Notation (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. The CBOR [35] is a binary serialisation format loosely based upon Java Script Object Notation (JSON) [36] 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. Internet Assigned Numbers Authority (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).

CoAP-HTTP Proxing One of the advantages of using the ReST CoAP protocol is the relative simplicity with integrating it with other ReST protocols such as HTTP. In fact, CoAP and HTTP share the set of basic requests methods which are implemented in a not too dissimilar way. IETF RFC 8075 [1] defines the guidelines for mapping implementations of HTTP [RFC 7230] to CoAP through an intermediary proxy that performs cross-protocol conversion. This will enable an HTTP client to access resources on a CoAP server through the proxy. The RFC 8075 describes how requests are mapped between HTTP and CoAP and how the response is mapped back including status code, URI, and media type mappings, as well as additional interworking advice. HTTP-to-CoAP (HC) proxy, specifically, acts as an HTTP server and a CoAP client accomplishing the role of forward, reverse, or interception Proxy. The scenario in Figure 2.2 describes an HC proxy situated at the boundary of constrained CoAP domain acting as gateway with respect the web HTTP domain.

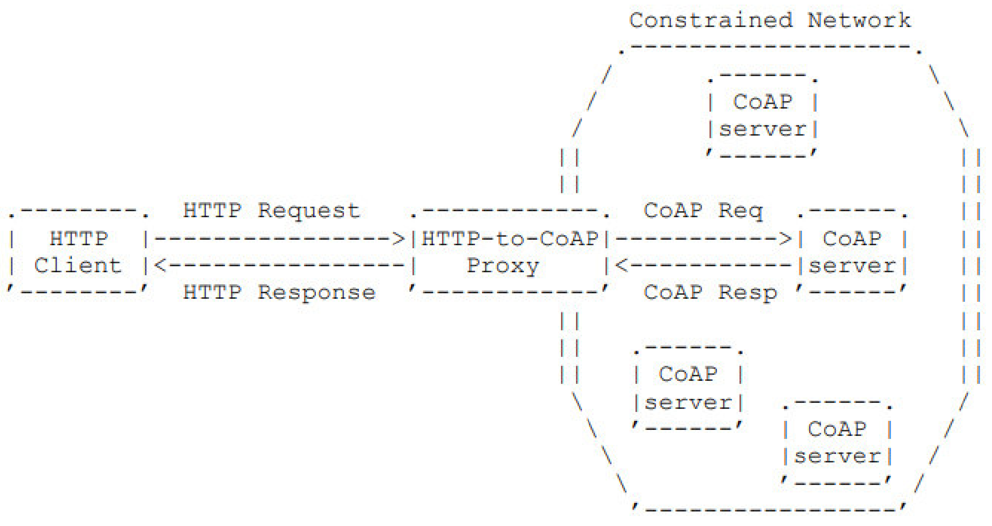


Fig. 2.2. HTTP-To-CoAP Proxy Deployment Scenario [1])

The described scenario can be implemented in different use cases which can be grouped into three macro cases:

- Legacy industrial application without CoAP: Industrial IoT systems that uses HTTP can interact with CoAP devices by HC in trasparent way.
- Web integration: HC can be used to extend web world connectivity to constrained device for Web Applications interoperability.
- Networks integration: HC can be used for integrating differente IoT Local Area Network (LAN)s

2.5.2 Message Queue Telemetry Transport (MQTT)

The evolution of IoT systems and, in particular, the need to have continuous data monitoring, has led to the development of protocols that distinguish themselves from **ReST!** (**ReST!**)-based solutions typical of the web-oriented Internet. Among these protocols, the MQTT protocol certainly stands out and enjoys growing success today.

MQTT [37] has been designed for resource-constrained devices, making it a lightweight protocol based on the concept of publishing and subscribing to specific information labeled by a designated *Topic*. The protocol is connection-oriented and the two connected peers, the client and the Broker, base their communication on a Transmission Control Protocol (TCP) session using a publisher/subscriber mode with low overhead (2-byte header) for applications with limited bandwidth.

MQTT has been developed for M2M communications, where all entities involved in data production and consumption are clients communicating through a common server known as the Broker. Due to the presence of the Broker, clients do not communicate directly with each other, instead, the exchange of information always occurs through the common Broker creating a star network architecture (Figure 2.3). The client can perform as *Publisher* and/or *Subscriber*, and it can simultaneously function as both. In the first case, the client publishes the data to the Broker, and in the second case, the client receives the requested data on subscribed *topic* from the Broker.

Topics are a kind of *labels*. They are at the core of the protocol and are structured hierarchically, similar to file system paths, where levels are separated by the "/" character. Topics are self-descriptive and they:

- are case-sensitive;
- use UTF-8 strings;
- must contain at least one character (a topic level cannot be empty).

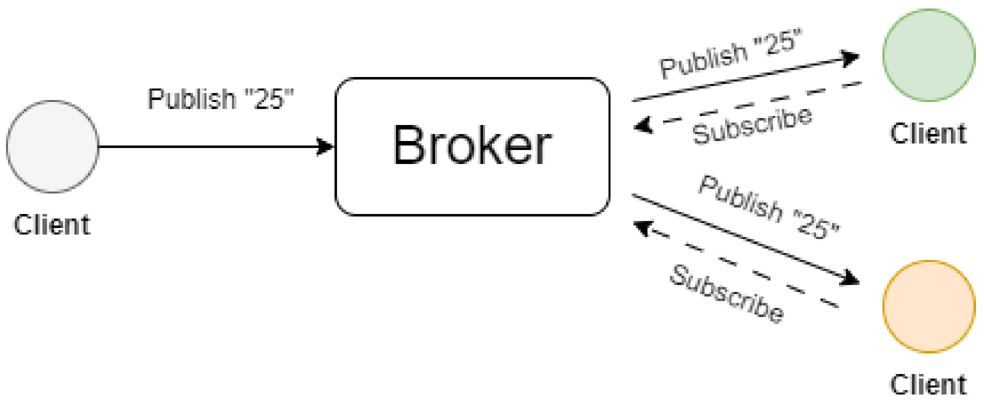


Fig. 2.3. MQTT Architecture

A topic is not generated in the broker, it doesn't exists, until someone subscribes to it or a client publish a message with retain option enabled. A topic has no sense to exist if there are no subscribed clients except for the *$SYS* topic (System's Topic).

According to the idea of spreading messages to as many nodes as possible, within different topic levels, MQTT includes special/wildcard characters:

- '#' → allows the topic to be valid for all sub-levels.
- '+' → allows skipping a single level.

A client can subscribe to single topics like *""*, *"/house"*, *"/house/garage"*, and *"/house/garage/light"*, or client can use wildcard to subscribe to all sublayers *"/house"* topic using subscription to topic *"/house/#"*. For instance, subscription to topic *"/house/#"* covers:

- *"/house/garage"*.
- *"/house/room"*
- *"/house/garage/light"*
- *"/house/garage/door"*
- *"/house"*
- *"/house/room/light"*
- etc.

Subscribing to topic *"/house/+/light"* covers:

- *"/house/garage/light"*
- *"/house/room/light"*
- *"/house/kitchen/light"*
- etc.

A message can be received by a group of clients if they subscribe to the same topic, However, a client can only publish messages to a single topic, and cannot publish to a group of topics (Figure 2.4).

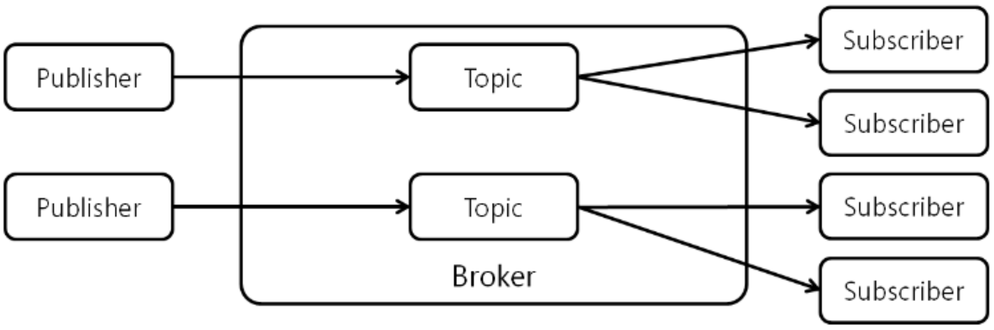


Fig. 2.4. MQTT Topic architecture

MQTT uses TCP protocol where sessions and subscriptions play a key role in management of connection and messages exchange between Broker and clients. The session creates connection and it will remain open until one of the two peers decides to terminate it. Therefore, every time communication begins between a client and a server, a session opens, and there will be as many sessions as there are clients communicating with that server, the Broker. Without an active session via the TCP protocol, unlike what happens in CoaP using UDP, the Broker cannot send data to the client. The session starts with a handshake procedure, which is always initiated by the client trying to connect to the known IP address of the Broker. It is preferable for the client to initialize the session since the reverse is not always possible, because they could reside in a protected LAN behind some local router or firewall.

Subscription defines a logical link between a client and a topic. The first client that subscribes to a specific Topic essentially creates it. Subscriptions are registered by Broker for each client and they can be:

- *Persistent*: subscriptions persist in the broker's memory even when the session with the client is lost. As soon as the session becomes operational again, the buffered messages will be forwarded to the client. This option can be enabled using the flag option *Clean Session* to *false*.
- *Temporary (or transient)*: in the case of reconstructing the session, the subscription must be reissued. Therefore, for transient subscriptions, if the session falls, so do all the subscriptions. This option can be enabled using the flag option *CleanSession* to *true*.

The *CleanSession* option is one of several options provided by MQTT to optimize communications. Most used MQTT options are listed below:

- *Clean Session*: a clean session is one in which the Broker is not expected to remember anything about the client when it disconnects; with a non clean session the Broker will remember client subscriptions and may hold undelivered messages for the client depending on used QoS option.
- *Retain Flag*: it is used in publish messages and it is normally set to *False*, which means that the Broker does not keep the message in its queue. If retain flag is *True*, the message received will be memorized in queue. The main use of this option is for values that do not update very much over time, and publish their status infrequently. If device only publishes its status without the retain flag set to *True*, the client subscribed after the last publication would not know the status of the sensor until it will be published it again.
- *Last Will message*: the idea of the last will message is to notify a subscriber that the publisher is unavailable due to network outage. This message is set by the publishing client on a per topic basis, which means that each topic can have its own last will message. The message is stored on the Broker and sent to any subscribing client (to that topic) if the connection to the publisher fails. If the publisher disconnects normally, the last Will message is not sent.

MQTT messages can transport data of any type, respecting space limits, and can enjoy three levels of Quality of Service (QoS) [38] (Figure 2.5):

- Level 0 – *At most once delivery*: data is sent without the application-level concern of receiving confirmation of its delivery. It is used for non-critical data. Even with QoS 0, TCP continues to operate its control mechanisms at the transport level and sends the ACK. The two protocols, being on different levels, work independently of each other.
- Level 1 – *At least once*: the message is guaranteed to arrive, but duplicates can occur. One might not receive the ACK and send subsequent messages; the subscriber could then receive different replicas, which can be a problem if the sensor state changes rapidly.
- Level 2 – *Exactly once delivery*: the message is guaranteed to be received once and only once. In this case, the security level is the highest, but the process is slower because the exchange of multiple messages is planned. The delivery guarantee is provided by two request-response flows (4-way handshake) between the sender and the recipient. Both use the QoS 2 packet identifier to coordinate message delivery.

The different QoS use different type of messages to ensure communication and acknowledg transmissions. QoS 1 and QoS 2 use *Message ID number* to track the message. The schema below shows message flow between client and broker for the different type of QoS.

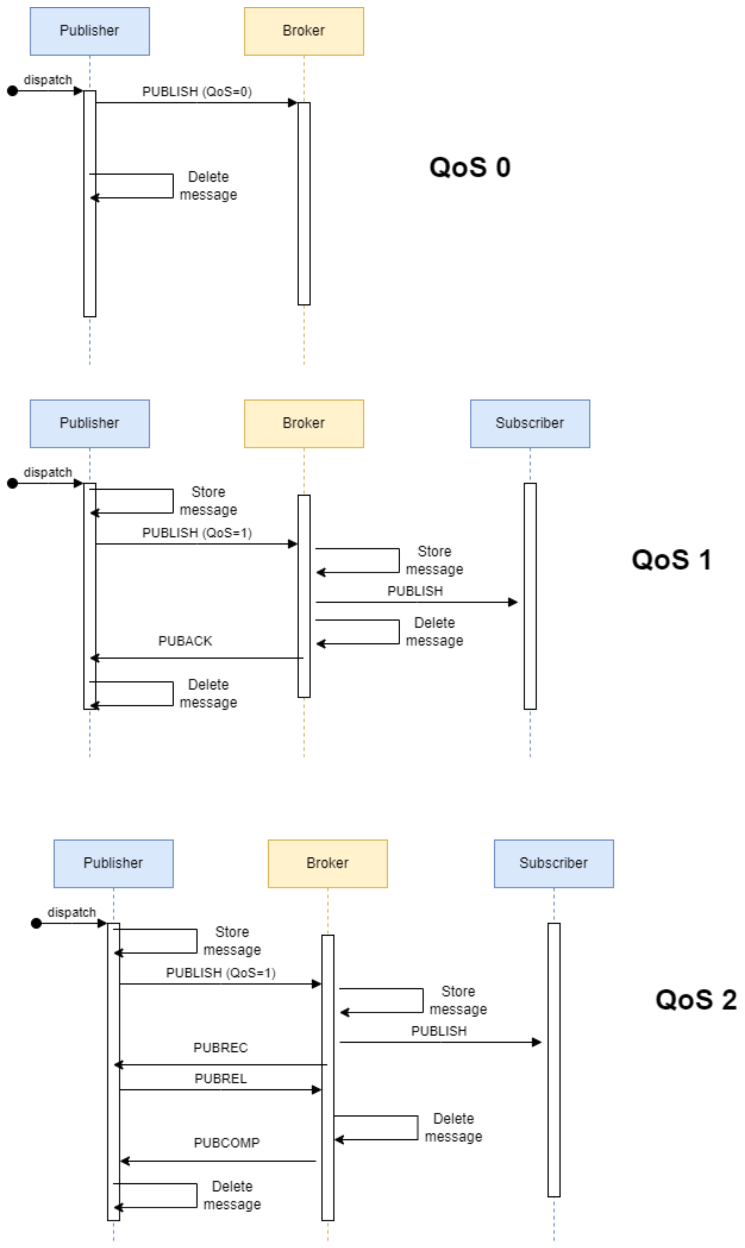


Fig. 2.5. MQTT messages flow schema

There is a variant of MQTT, called MQTT-Sensor Network (MQTT-SN), which includes various optimizations for even more constrained wireless devices that may not have an IP address, such as wireless sensors. The Wireless Sensor Network (WSN) typically does not implement a TCP/IP stack but employs its own stack with protocols such as ZigBee [39] and Bluetooth [40]. This design choice ensures lightweight and efficient operation for resource-constrained devices. Consequently, a direct implementation with MQTT brokers (servers) is not feasible. The solution involves introducing intermediate TCP/IP gateways, which will then communicate with the MQTT Broker. WSNs will connect to these gateways using MQTT-SN, leveraging TCP/IP network functionalities and establishing a connection to the broker via MQTT. The introduced innovations are as follows:

- Adoption of an ID for topics rather than a string.
- Predefined topic IDs that do not require registration.
- Dynamic broker discovery procedures, eliminating the need for static broker configuration.
- Reduction in payload size.
- Use of UDP instead of TCP, making the connection to the broker "virtual".
- Wake-up and reception of buffered messages for sleeping clients.

MQTT-SN gateways come in two types: (i) Transparent, where each MQTT-SN connection corresponds to a unique MQTT connection; (ii) Aggregation, where multiple MQTT-SN connections correspond to a single MQTT connection.

2.5.3 Advanced Message Queuing Protocol (AMQP)

AMQP is a standardized messaging protocol designed for reliable, high-performance message delivery between applications. It is a binary protocol that enables asynchronous communication between diverse systems, making it well-suited for IoT applications.

Key characteristics of AMQP for IoT are:

- *Reliability*: AMQP prioritizes message delivery assurance, employing various mechanisms to guarantee message persistence and prevent message loss, critical aspects for IoT applications where data integrity is paramount.
- *High performance*: AMQP is designed for high-throughput messaging, enabling efficient handling of large volumes of data generated by IoT devices, ensuring timely processing and analysis.
- *High security*: AMQP supports various security mechanisms, including authentication, authorization, and encryption, safeguarding sensitive data exchanged between IoT devices and central servers.

- *Remarkable flexibility*: AMQP offers message routing and filtering capabilities, allowing for targeted message delivery to specific consumers, enhancing data organization and utilization.
- *Scalability*: the AMQP architecture is designed to handle growing message volumes and accommodate new devices, making it suitable for large-scale IoT deployments.

In Device-to-Device Communication, AMQP enables reliable and secure communication between IoT devices, facilitating device coordination and data exchange within the IoT network. In Device-to-Cloud Communication, AMQP facilitates efficient data transmission from IoT devices to cloud platforms, enabling data aggregation, analysis, and visualization. In Cloud-to-Device Communication, AMQP enables cloud platforms to send commands and configuration updates to IoT devices, ensuring effective device management and control. Its Real-time Data Streaming supports high-throughput data streaming from IoT devices to enable real-time monitoring, analytics, and decision-making. As Event-driven applications, AMQP facilitates the implementation of event-driven applications in IoT scenarios, where devices can trigger actions based on specific events or data patterns.

2.5.4 Extensible Messaging and Presence Protocol (XMPP)

XMPP, known as Jabber until October 2002, is a quasi real-time communication protocol for exchanging structured data between network entities. It is an XML-based protocol that facilitates a near-real-time communication, making it conceivable for IoT applications. Key Characteristics of XMPP for IoT are mainly referred to its being open and standardized, as XMPP is an open standard, freely available for implementation, and enabling interoperability among devices from different vendors,those are crucial aspects for IoT ecosystems. A second fundamental characteristic is extensibility: XMPP allows for the introduction of new features and applications through XMPP Extension Protocols (XEPs), open-source specifications that define protocol extensions. This flexibility caters to the evolving needs of IoT applications. From an architectural view, XMPP is decentralized this eliminates the reliance on a single central server, enhancing resilience and fault tolerance, critical factors for IoT applications that demand continuous operation. Last, XMPP enables low lag message delivery, ensuring that messages reach their intended recipients promptly. This quasi real-time capability is essential for IoT applications that require a non mediate response and action.

In Device Management the XMPP streamlines device management, facilitating command transmission, device status retrieval, and configuration settings updates, effectively managing IoT devices remotely. As Data Streaming: XMPP enables real-time data streaming from IoT devices to central servers, enabling continuous monitoring and analytics for informed decision-making. Regarding the Notification Delivery, XMPP facilitates the delivery of notifications from IoT devices to users, providing timely alerts and updates on device status, anomalies, or critical events. XMPP Presence Information provides information for IoT devices, indicating whether a device is online, offline, or ready for communication, ensuring efficient device utilization and communication optimization.

2.6 Semantic interoperability Protocols

Technical-interoperability of IoT framework from different standards is achievable as long as standards abide by the concept of three layered architecture (sensor, core/backbone network, application/services) and conceptual IoT architecture model. In the previous section protocols for data transmission have been introduced. Such protocols facilitate the exchange of data, but these data need to be intelligible to those who receive them. In other words, the recipient must be able to understand what type of data has been sent in order to transform it into information useful for the provided service. This section is focused on standards which provide semantic and syntactic interoperability by performing mapping among different groups (like mandatory, optional) of attributes of interfaces and different Application Programming Interface (API)s, and transforming data from heterogeneous systems into information.

2.6.1 Open Mobile Alliance Lightweight Machine-to-Machine (OMA-LwM2M)

OMA LwM2M [2] is a device management protocol designed for sensor networks and 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, recently, it has also begun to integrate the MQTT protocol.

The semantic data model is used to define a LwM2M client device as a composition of Resources organized in Objects. An Object can contains 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 (Figure 2.6):

- Bootstrap
- Client Registration
- Device management and service enablement
- Information Reporting

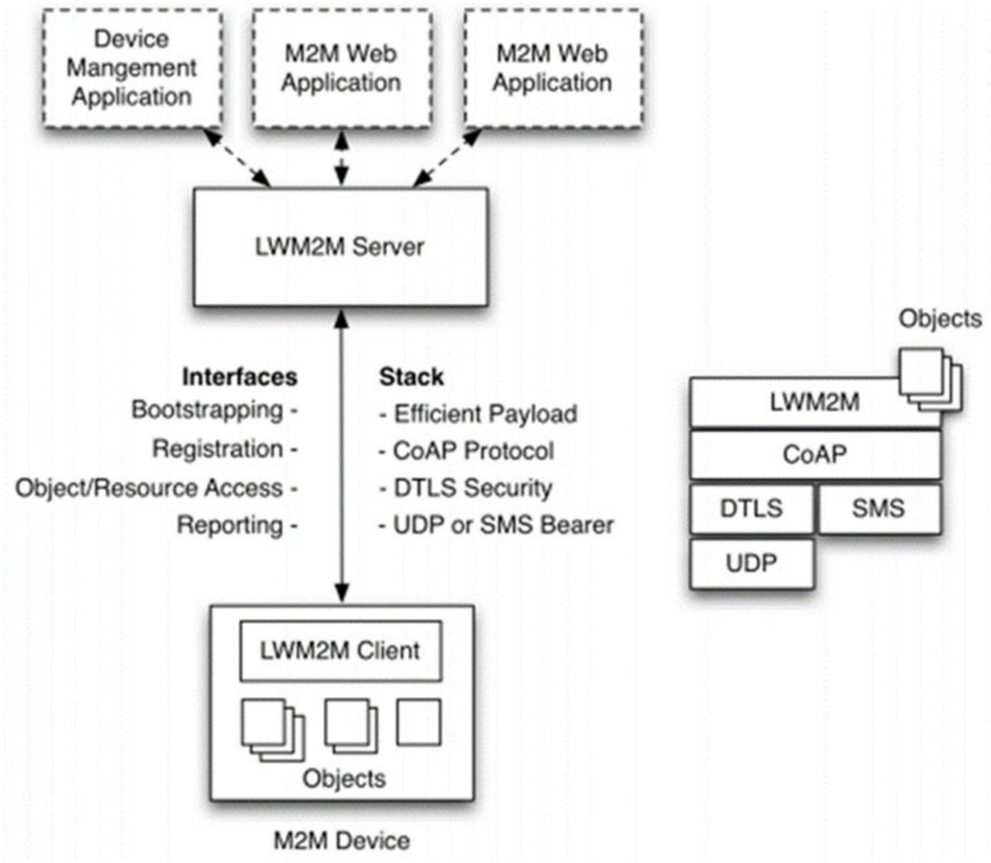


Fig. 2.6. LwM2M Enabler architecture [2]

Bootstrap Interface

The interface (Figure 2.8) 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

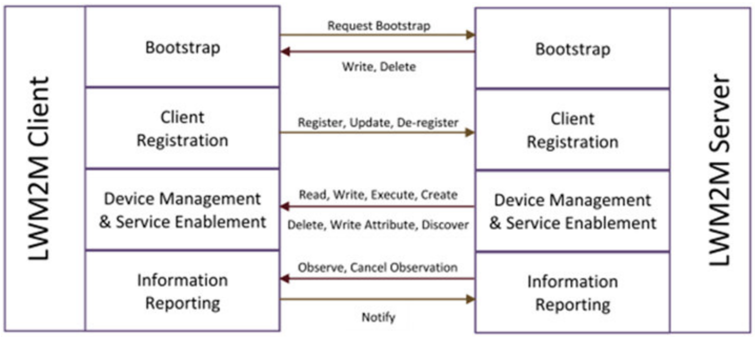

LWM2M Client
Bootstrap
Client Registration
Device Management & Service Enablement
Information Reporting
Request Bootstrap
Write, Delete
Register, Update, De-register
Read, Write, Execute, Create
Delete, Write Attribute, Discover
Observe, Cancel Observation
Notify
Bootstrap
Client Registration
Device Management & Service Enablement
Information Reporting
LWM2M Server

Fig. 2.7. LwM2M Server-Client interaction [2]

this specific interface. This interface has uplink operations named *Bootstrap-Request* and *Bootstrap-Pack-Request*, and downlink operations named *Bootstrap-Discover*, *Bootstrap-Write*, *Bootstrap-Read*,*Bootstrap-Delete* and *Bootstrap-Finish*.

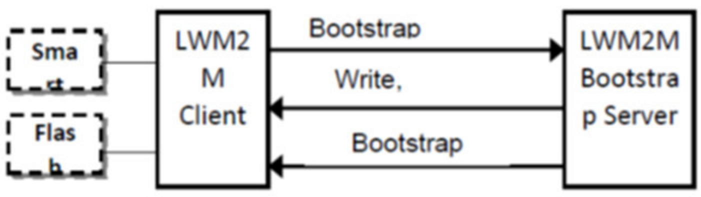


Fig. 2.8. LwM2M Bootstrap Interface [2]

Registration Interface

This interface (Figure 2.9) manages the Registration, the Update (Keep alive like), and the 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.

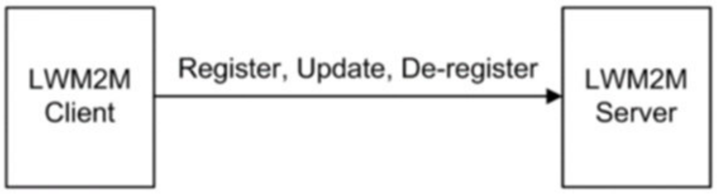


Fig. 2.9. Registration interface

Device Management and Service Enablement For this interface (Figure 2.10), 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.

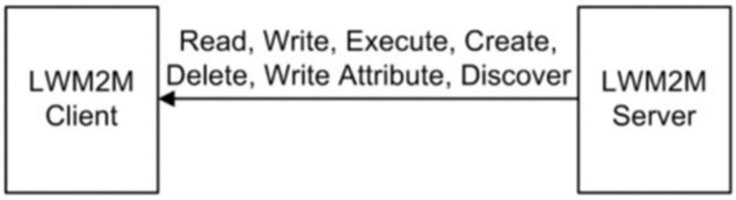

Read, Write, Execute, Create,
Delete, Write Attribute, Discover
LWM2M Client
LWM2M Server

Fig. 2.10. Device management and service enablement interface

Information Reporting This interface (Figure 2.11) provides both uplink, *Notify*, and downlink operations like *Observe or Cancel Observation.* This interface is used to send to the LwM2M Server a new value related to one or more Resources on the LwM2M Client.

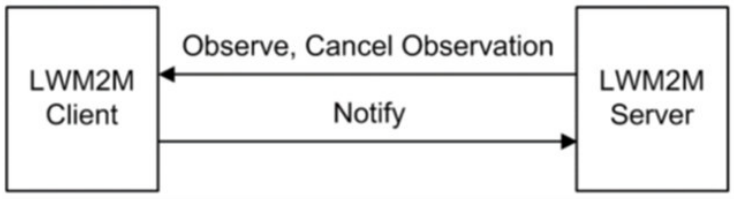


Fig. 2.11. Information reporting interface

Table 2.2 lists the relationship between Operations and Interfaces.

Client and server interaction are reported in Figure 2.7.

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)

LwM2M 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.

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	Uplink	Notify
Information Reporting	Downlink	Observe, Cancel Observation

Table 2.2. Operation and Interfaces relationship

Basically, 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. Figure 2.12 represents an example of the resource model used in OMA-LwM2M.

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 *33303/1/5700*. Object and their resources are defined using meta-model declared and shared between client and server. An object can be defined using a eXtensible Markup Languag (XML) file like described in Figure 2.13.

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 Type–length–value (TLV). The OMA LwM2M standard provides a public registry of "Standard Objects" [41] but each developer can create their own object from OMA objects and the provided resource models. Moreover, LwM2M defines access control mechanism per Object entity based on associated Access Control Object Instance (Figure 2.14). An Access Control Object Instances contains Access Control Lists (ACLs) that define which operations on a given Object Instance are allowed for

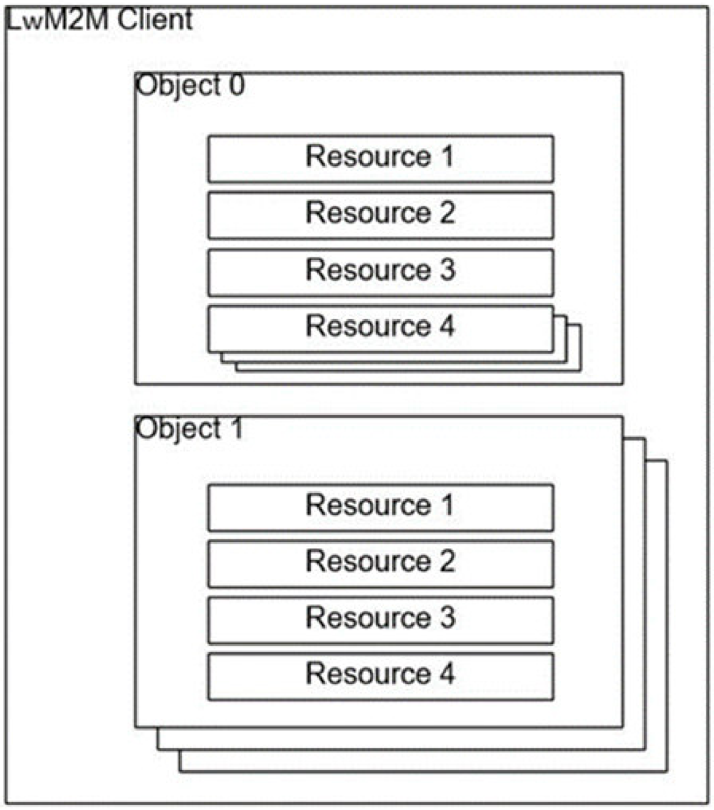


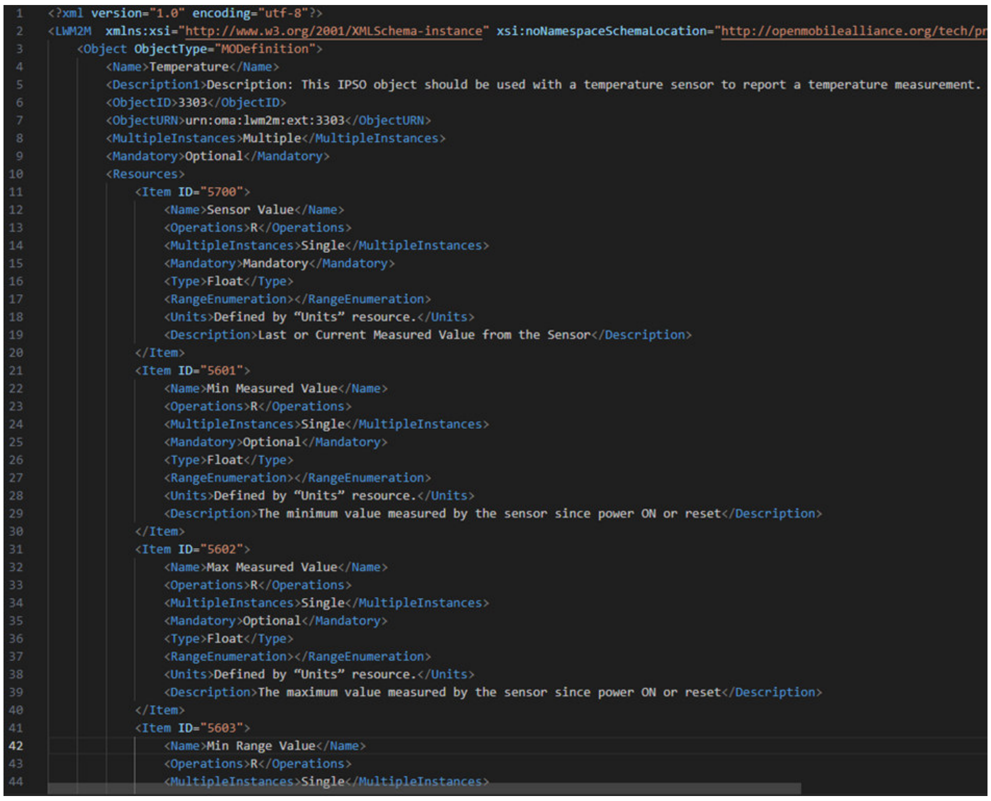
Fig. 2.12. LwM2M resource model

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.

2.6.2 World Wide Web Consortium Web of Things (W3C WoT)

The growing number of IoT devices communicating and exposing their services on the network is inevitably influencing the development of the World Wide Web (WWW). The World Wide Web Consortium (W3C) consortium, specifically established for web standardization, has thus directed its efforts to address this evolution, giving rise to the WoT initiative. WoT aims to create a standard that enables better integration of IoT devices into the WWW. 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 W3C endeavours to establish a similar synergy between the IoT and WoT. Key features of the W3C WoT initiative include:

- *Thing Description (TD)*: Thing Descriptions provide a standardized description of the capabilities of an IoT device, including details such as supported properties,



```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>
```

Fig. 2.13. LwM2M resource model in XML

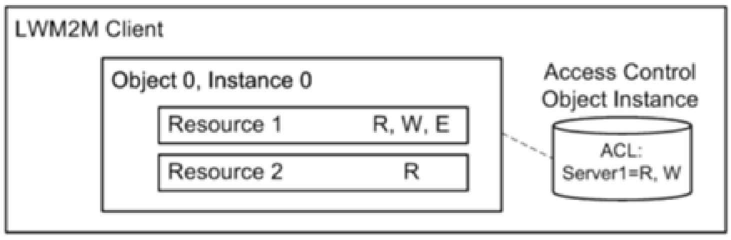


Fig. 2.14. LwM2M access control object instance

actions, and events. This description helps applications understand and interact with devices consistently.

- *Servient*: it represents a software stack responsible for implementing the core WoT elements. Servients have 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.
- *Interoperability*: W3C WoT aims to ensure interoperability between IoT devices and applications through clear and standardized specifications. This allows devices from different vendors to collaborate seamlessly.

- *Scripting APIs*: WoT includes a scripting API that simplifies the development of applications capable of interacting with IoT devices using scripting languages such as JavaScript.
- *Security and Privacy*: The W3C WoT initiative considers security and privacy as fundamental elements. Mechanisms are provided to ensure secure transactions and protect sensitive information exchanged between devices and applications.
- *Communication Protocols*: W3C WoT supports various communication protocols, including HTTP, CoAP, and MQTT, to enable communication between IoT devices on heterogeneous networks.

Things Description (TD)

The Things Description (TD) 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, MQTT, etc., to consumers. In analogy to how web browsers commonly access websites by utilising 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 standard, 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.* 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, provide 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.

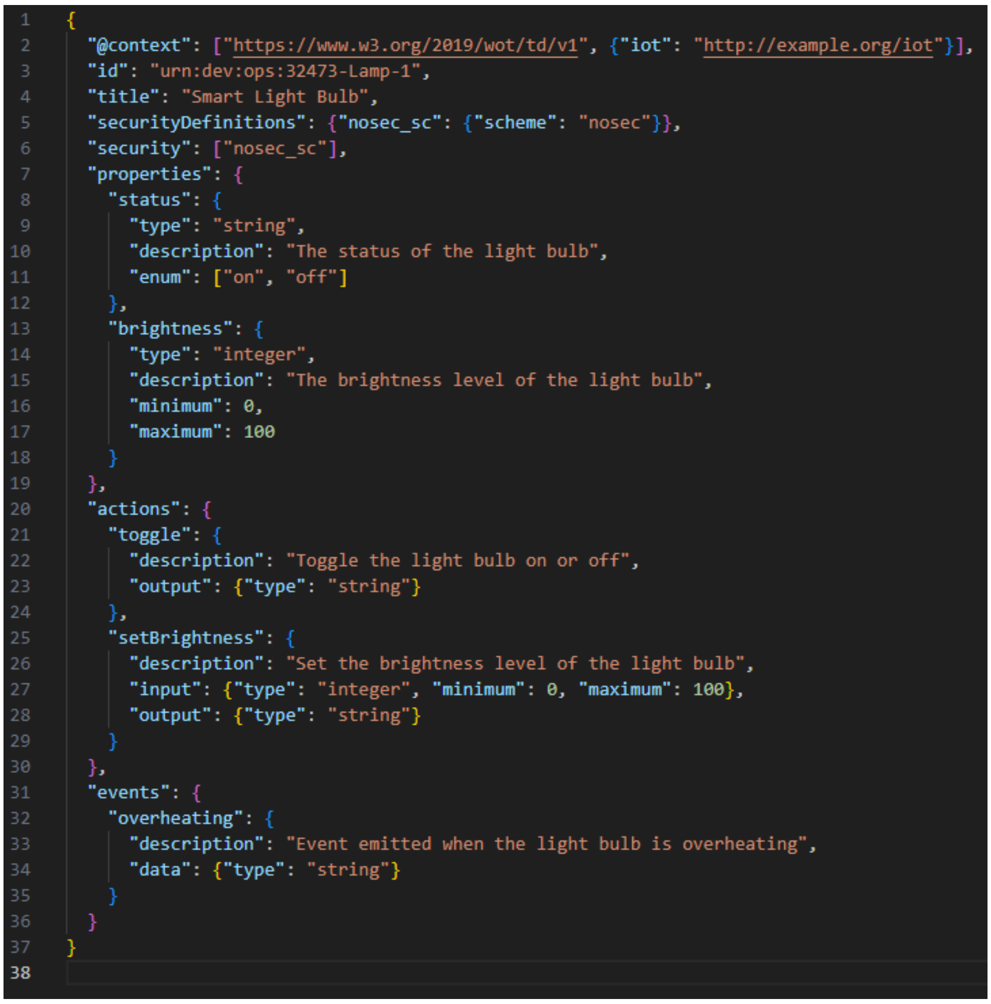
The TD is a JSON-LD based representation, which provides knowledge about Things in a machine-readable representation. The TD context currently includes the following standard vocabularies: *TD core*, *data schema*, *WoT security*, and hypermedia controls. Semantic interoperability is achieved by extending 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.

The listing in Figure 2.15 shows an example of a TD instance for a smart device 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. 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 utilised to impose further restrictions on the valid data, such as specifying the minimum or maximum numeric values.

The TD version 1.1 specification defines a reusable model for representing Thing class definitions, called Thing Model (TM), that can be mainly 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.

2.6.3 Next Generation Service Interfaces - Linked Data (NGSI-LD)

NGSI-LD is a standard specification developed by the ETSI Context Information Management (CIM) group for managing and exchanging information in the context of the IoT and smart cities along with its associated API and broker. NGSI-LD is part of the broader NGSI family of standards [42] [43].



```json
{
  "@context": ["https://www.w3.org/2019/wot/td/v1", {"iot": "http://example.org/iot"}],
  "id": "urn:dev:ops:32473-Lamp-1",
  "title": "Smart Light Bulb",
  "securityDefinitions": {"nosec_sc": {"scheme": "nosec"}},
  "security": ["nosec_sc"],
  "properties": {
    "status": {
      "type": "string",
      "description": "The status of the light bulb",
      "enum": ["on", "off"]
    },
    "brightness": {
      "type": "integer",
      "description": "The brightness level of the light bulb",
      "minimum": 0,
      "maximum": 100
    }
  },
  "actions": {
    "toggle": {
      "description": "Toggle the light bulb on or off",
      "output": {"type": "string"}
    },
    "setBrightness": {
      "description": "Set the brightness level of the light bulb",
      "input": {"type": "integer", "minimum": 0, "maximum": 100},
      "output": {"type": "string"}
    }
  },
  "events": {
    "overheating": {
      "description": "Event emitted when the light bulb is overheating",
      "data": {"type": "string"}
    }
  }
}
```

Fig. 2.15. WoT TD Example

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 [42].

NGSI-LD is based on the Resource Description Framework (RDF), a W3C standard for representing information in a machine-readable format. RDF allows for the creation of linked data, which is a way of representing information as a network of interconnected resources. This makes it possible to easily share and reuse information between different applications. The NGSI-LD information model defines three basic concepts:

- *Entities*: Entities are the basic building blocks of NGSI-LD. They represent real-world objects or concepts, such as a person, a device, or a location.
- *Properties*: Properties are used to describe the characteristics of entities. They can be simple data types, such as strings or numbers, or they can be more complex structures, such as arrays or objects.
- *Relationships*: Relationships are used to connect entities together. They can be directed or undirected, and they can have different types, such as parent-child, sibling, or association.

NGSI-LD APIs and Broker

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 [44]. 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 handle 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.

Entity and Things in NGSI-LD API

An entity represents an object/thing that exists in the real world. Entities are represented using JSON-LD, a JSON-based serialization format for Linked Data. This means that entities can be easily shared and reused between different applications and systems. Each entity has a unique identifier, which is used to reference the entity in other parts of the NGSI-LD data model. The identifier can be any type of string, but it is typically a URI. In addition to its identifier, an entity can have a number of other properties, such as:

- *Type*: The type of the entity. This is a string that identifies the class of the entity, such as ParkingSpot or Truck.
- *Attributes*: The attributes of the entity. These are the properties that describe the characteristics of the entity. Attributes can be simple data types, such as strings or numbers, or they can be more complex structures, such as arrays or objects.
- *Relationships*: The relationships of the entity. These are the connections that the entity has with other entities. Relationships can be directed or undirected, and they can have different types, such as parent-child, sibling, or association.

Such properties create the NGSI Entity-Attribute-Value (EAV). Entities can be created, read, updated, and deleted using the NGSI-LD API. The API also provides operations for subscribing to changes in context information. However, some Performance Considerations need to be done. EAV models may face performance issues, especially as the amount of data grows. Querying, indexing, and maintaining the integrity of the data can be more challenging compared to traditional models. It's important to note that while the EAV model offers flexibility, it also comes with trade-offs, and its suitability depends on the specific requirements of the application. In many cases, a balance between flexibility and query performance needs to be struck, and other data modeling approaches may be considered based on the nature of the data and the application's use cases.

Based on NGSI-LD, a consortium of relevant organizations in the IoT sector [1] [2] [3] [4], 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, propose semantic *Smart Data Model* [45].

[1] https://www.fiware.org/
[2] https://www.tmforum.org/
[3] https://iudx.org.in/
[4] https://oascities.org/

3

Virtualization of IoT devices

This Chapter focuses on IoT device virtualization and presents the Virtual Object (VO) concept, its interfaces, the basic interactions and the main layers in its protocol stack.

3.1 The Virtual Object (VO) concept

Virtualization commonly involves the abstract representation of underlying hardware devices through a software implementation or description. In the context of the IoT, it can impact either the network and its functions [46] or the devices themselves [47]. Device virtualization has notably become a fundamental component of various reference IoT platforms (e.g., iCore [48], IoT-A [49]) and commercial implementations (e.g., Amazon Web Services IoT). The objective is to enable plug-and-play functionality for heterogeneous objects, meaning that when a device joins a network, it can immediately engage with the external world [47].

The Virtual Object (VO) serves as the digital extension of the physical IoT device into the virtual environment. Semantic technologies are identified as the most suitable means to represent IoT devices [47]. Consequently, the VO enriches the data and functionalities provided by IoT devices through semantic descriptions. The resulting VO model encompasses various aspects, including object characteristics, location, resources, services, and quality parameters [16]. Semantic descriptions address heterogeneity and enhance interoperability in the IoT domain, eliminating vertical silos. Moreover, semantic technologies play a crucial role in supporting search and discovery operations. Search and discovery mechanisms enable context awareness, allowing the identification of the device most suitable for a given application's task.

The VO can enhance the physical counterpart with storage and computing capabilities. It achieves this by offering caching and preliminary filtering/aggregation/processing of raw data streamed by the corresponding IoT device before feeding into IoT applications. Caching data from the physical device helps prevent overwhelming it with identical requests from multiple remote applications, which is particularly beneficial for resource-constrained IoT devices.

While VOs were initially designed for deployment in the remote cloud, recent literature has highlighted the advantages of edge networks in meeting latency constraints for pairing a physical device with its corresponding VO [50], [51], [52]. For instance, [51] considers a proxy Virtual Machine (VM) hosted at the edge, while [53] explores the use of containers to create virtualized cameras.

Emerging applications, developed in a cloud-native/microservices-based manner as service chains comprising scalable components (microservices), leverage a hyper-distributed execution of interconnected components across a computing continuum with orchestrated resources spanning different network domains (IoT, edge, cloud) [54]. In light of these evolving requirements and potentials, the VO design should be reconsidered to position the VO as a facilitator for: (i) unified device management, addressing interoperability challenges; (ii) the development of computing continuum-native IoT applications, addressing convergence aspects with edge and cloud computing technologies; and (iii) the establishment of new cyber-physical paradigms and IoT-driven business models.

VOs, agents, and Digital Twins

In the realm of IoT abstractions, alternative approaches, such as the agent concept, have been advocated, as extensively surveyed in [55]. Agents find utility in implementing vertical IoT solutions within the same specific domain, integrating multiple heterogeneous systems belonging to the same entity. However, agents currently appear more suitable for specific micro-operations or platform-to-platform interconnection and, unlike VOs, are not yet poised to enhance the connectivity and interoperability of service-oriented architectures (SOs) [55].

Moreover, the VO embraces the Digital Twin (DT) concept, which is also grounded in the mapping of a physical object onto a virtual space. It extends this idea to depict a synchronous bidirectional data exchange for monitoring, simulating, predicting, diagnosing, and controlling the state and behavior of the physical object within the virtual space [56]. One can conceptualize a DT as akin to a VO but endowed with advanced features and a close synchronicity and state alignment with the physical object. An open VO design, positioning it as a potential foundational element for a DT or even future cyber-physical systems, would foster significant advancements in the field.

3.2 The Composite VO (cVO) concept

As a further step, a more effective collaboration of several physical devices is enabled in the virtual world by the introduction of the 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. A single VO may correspond to multiple physical devices, each of them performing different functions/services, or multiple VOs correspond to a single physical device (Figure 3.1) [57].

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 trigger the introduction of several methods and frameworks often targeted to a specific application area [58].

A cVO virtually 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.

VO-Physical device interactions

Based on the development environment vision and purpose, VOs can vary in their specific characteristics and functionalities. However, they can generally be described as [47] : *"a digital representation, semantically enriched, of a real world object able to acquire, analyze and interpret information about its context, to augment the potentialities of the associated services"*. The nature of the correspondence between the Physical Device (PD) and its virtual counterpart, as well as the established relationship, can significantly vary based on the reference service scenario [57]. Consequently, the following associations are possible:

- **One-to-One**: A single physical object (or device) is linked to one VO, as in ETSI NFV architecutere (Release 1) [59] and in FIWARE project [1]. In this scenario, the VO is responsible for receiving and processing all requests for the PD.

[1] https://www.fiware.org/

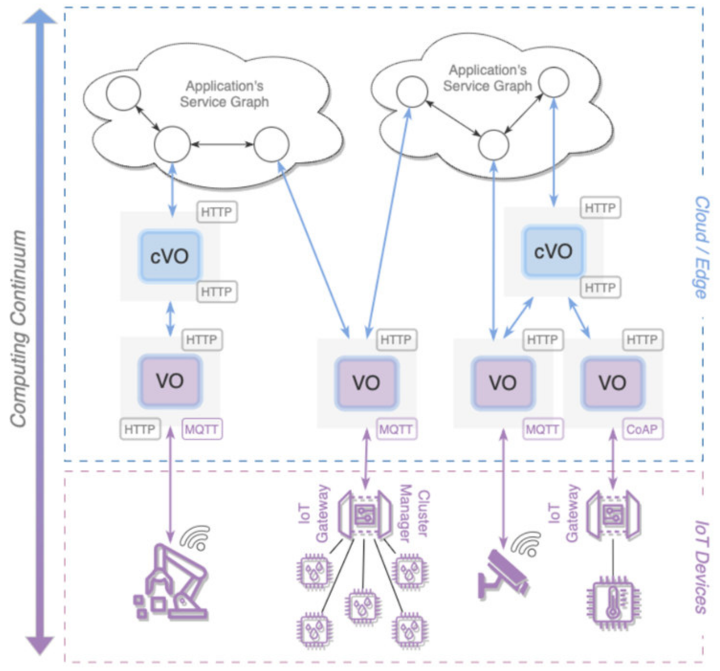


Fig. 3.1. VO in the computing continuum [3]

- **One-to-Many**: A single physical object is associated with multiple VOs [49] [60] [61]. The virtual entities connected to the same physical object are dedicated to providing different services.
- **Many-to-One**: Multiple elementary physical objects are linked to a single VO [62]. This VO is capable of integrating and processing data from diverse objects, presenting them through a unified framework for interaction with external services.
- **Many-to-Many**: Many physical objects are connected to many VOs through a double level of abstraction involving the use of both VO and cVO, as demonstrated in [48]. The aggregation of multiple VOs aims to meet the requirements of applications beyond the initial domain of these VOs.

3.3 The VO stack

As described in [3], the NEPHELE European project [4] is actively developing a comprehensive IoT and edge computing software stack to capitalize on the virtualization of IoT devices at the network edge. The primary focus is on supporting openness and interoperability aspects in a device-independent manner.

A VO software stack facilitates the unified management of a diverse array of IoT devices and platforms, eliminating the need for middleware platforms. Additionally, it enables on-demand provision of edge computing functionalities to efficiently support the operations of IoT applications. The IoT and edge computing software stack is structured as a multi-layered system addressing IoT interoperability and openness at two levels:

- *IoT Device Level*: This level centers on providing virtual counterparts for IoT devices. At this level, the VO concept is introduced. The VO enhances its functionalities through the implementation of a multi-layer software stack known as the Virtual Object Stack (VOStack). The VOStack is specifically designed to equip VOs with edge computing and IoT functions, including distributed data management and analysis based on machine learning (ML) and digital twinning techniques, security and trust mechanisms, autonomic networking, time-triggered IoT functions, service discovery, and load balancing.
- *Integration Level*: This level focuses on integrating IoT functions with edge and cloud computing applications. By leveraging the VOStack and adopting the microservice paradigm, the software stack enables the design of hyper-distributed applications. In this framework, components of IoT, edge, and cloud computing applications can be jointly represented in a unified application graph. IoT application components are represented as functions supported by the VO, while edge and cloud computing application parts are presented as pure edge/cloud-native functions.

This approach aims to achieve convergence in IoT technologies, addressing both protocol and semantic interoperability challenges. Furthermore, it enhances the interoperability of IoT technologies with emerging edge and cloud computing specifications. The NEPHELE project seeks to advance these innovations to contribute significantly to the evolving landscape of IoT and edge computing

As depicted in Figure 3.2, to encompass all the proposed capabilities, a VO engages in four key interactions with the computing continuum environment:

- **VO-to-IoT-Device Interaction**: This interaction aims to address interoperability and convergence challenges within the IoT ecosystem.
- **VO-to-Application Interaction**: This interaction enables communication between VOs and cVOs and facilitates interactions between cVOs and application components that contribute to the distributed application's business logic.
- **VO-to-Orchestration Interaction**: This interaction enables the development of edge/cloud computing distributed applications where cVOs become integral parts of a distributed application graph, making them manageable through orchestration mechanisms employed in cloud/edge computing environments.

- **VO-to-Storage Entity Interaction**: This interaction serves to maintain records of device metadata, status, and messages exchanged with other devices and applications. Additionally, the VO must be capable of supporting basic data management operations by integrating multiple data sources to produce contextual device information that can be utilized by clients.

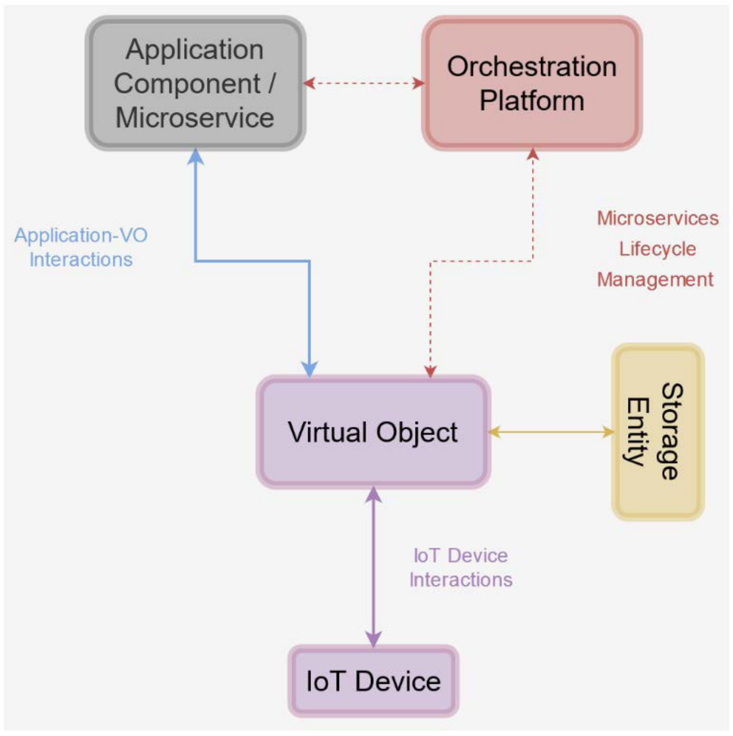

Application Component / Microservice
Orchestration Platform
Application-VO Interactions
Microservices Lifecycle Management
Virtual Object
Storage Entity
IoT Device Interactions
IoT Device

Fig. 3.2. VO interaction [4]

The VOStack is instantiated within the framework of stateless pluggable *microservices*. The primary motivation behind this design choice is to ensure that VOs remain lightweight and modular, while still providing essential functionalities that cater to the requirements of most devices and applications. As a result, the VOStack is organized into three main architectural layers [3]:

- **Physical Convergence Layer**: This layer is the foundational element of the VOStack. It deals with the direct interaction with physical devices, ensuring a seamless convergence between diverse device types. The main functionality of this layer includes device discovery, communication protocol adaptation, and the provision of a uniform interface for data exchange. Essentially, it bridges the gap between the physical IoT devices and the virtual representation.

- **Edge/Cloud Convergence Layer**: Positioned between the physical convergence layer and the backend logic layer, this layer focuses on enabling the convergence of data and functions between edge and cloud environments. It facilitates the transition of data from the edge to the cloud or vice versa. Key functionalities encompass data aggregation, preliminary processing, and secure transmission to ensure efficient communication between the edge and cloud components.
- **Backend Logic Layer**: At the topmost layer of the VOStack, the backend logic layer is responsible for handling more complex operations and functionalities. It includes features such as distributed data management, analysis based on machine learning and digital twinning techniques, security protocols, blockchain mechanisms, autonomic networking, and time-triggered IoT functions. This layer encapsulates the intelligence and advanced capabilities required to enhance the overall functionality of VOs.

Figure 3.3 present the VOStack in the IoT to Edge to Cloud Continuum architecture with respect the physical domain of devices, edge domain of VOs and IoT services, and cloud orchestration and big data management domain.

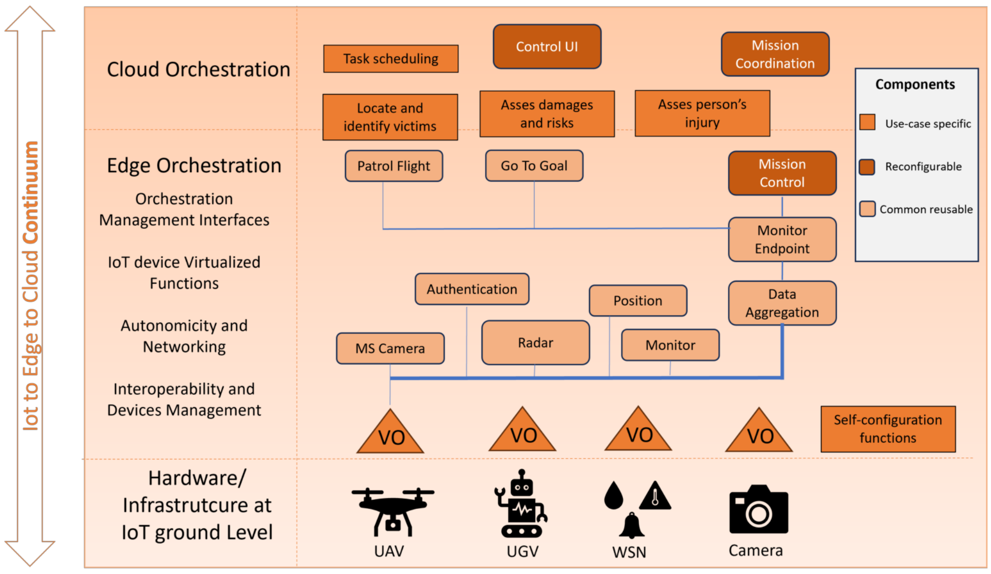


Fig. 3.3. VO stack architecture in the IoT to Edge to Cloud Continuum

Illustrated in Figure 3.4 is the layered architecture of the VOStack, highlighting the hierarchical arrangement of its three constituent layers. This structured design aims to provide a comprehensive and adaptable solution for IoT device virtualization, guaranteeing that the VOStack can effectively accommodate a diverse range of devices and applications while upholding flexibility and scalability.

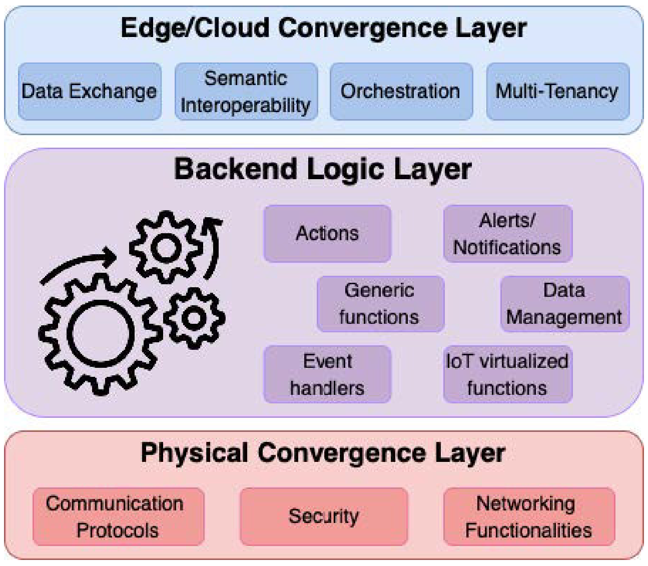


Fig. 3.4. VOStack layers [3]

3.3.1 Physical convergence layer

This layer serves as the cornerstone for connecting IoT devices to the computing continuum infrastructure, addressing the fundamental challenges of device integration. Firstly, it enables device registration tasks, such as registering new devices with VOs and establishing initial connections. In terms of connectivity, the VO supports a wide range of communication protocols commonly used in the IoT domain, spanning the application layer (MQTT, CoAP, HTTP), network layer (IPv4, IPv6), and transport layer (TCP, UDP). This extensive support ensures that the vast majority of IoT devices can seamlessly connect and communicate with their virtual counterparts. Considering the limited security capabilities of many devices, the layer incorporates authentication and authorization mechanisms (e.g., OAuth 2.0) to secure communication between devices and applications. Additionally, it simplifies the coordination of multiple IoT devices or clusters by providing autonomic and self-* functionalities [63], enabling devices to adapt and manage themselves effectively. To address the intermittent connectivity of devices, a suite of network-oriented functionalities is provided, including dynamic routing protocols, time-sensitive networking mechanisms, and mobility management schemes. These functionalities ensure that clients can maintain uninterrupted access to device information even if the device temporarily loses connection with its VO. In essence, this layer acts as a robust and versatile communication bridge between IoT devices and the broader computing continuum, facilitating seamless integration and secure data exchange.

3.3.2 Edge/Cloud convergence layer

This layer serves as the bridge between the VO and the application and orchestration layer, facilitating communication between the VO and entities such as data consumers, applications, or users through various interfaces. It supports a range of communication protocols, including HTTP, MQTT, and CoAP. This layer essentially acts as the gateway for exposing and consuming IoT devices. It provides functionalities for managing incoming requests, generating responses (e.g., data retrieval, action triggers, alert notifications), and handling multi-tenancy aspects (e.g., multiple requests for IoT device information). Additionally, this layer supports orchestration-related functions, such as monitoring the VO's status (e.g., container monitoring), managing VO deployment across the computing infrastructure (e.g., start, stop, restart, destroy), and handling elasticity and migration actions.

3.3.3 Backend Logic Layer

This layer is responsible for enhancing the functionalities and capabilities of IoT devices. It encompasses the logic related to the IoT device's operational behaviors, advanced functionalities, and services that the Object/Device can perform. Primarily, VOs can declare *alerts* based on IoT device state changes (e.g., a device suddenly restarting) or data-driven *notifications* (e.g., a sensor's temperature rapidly increasing). This functionality is closely linked to interaction with the storage entity, as observing past data values is often crucial in many scenarios.

When creating a virtual counterpart of an IoT device, it is essential to introduce a set of *actions* and *behaviors* that the VO can dictate to the IoT device. This enables the VO to reconfigure or remotely heal a device. Additionally, following an event-based logic, actions can be triggered by either monitored data (e.g., alerts and notifications from a sensor) or commands received from the application and/or orchestration side (e.g., an application provider may want to modify the behavior of a sensor, such as changing the polling period of measurement when a given threshold is exceeded). For each defined action, a mechanism is designed to support action-related policies that implement multi-tenancy characteristics. It is crucial that the set of actions, alerts, and notifications are reconfigurable and their definitions are not limited or heavily dependent on the specific use case.

Furthermore, the modularity of IoT functions in the edge and the cloud infrastructure is considered a key challenge to enabling modern applications and cloud-native IoT solutions. To address this challenge, two main categories of functionalities are defined to support the basic operations required for the interplay between IoT and

Applications: (i) *IoT Device Virtualized Functions* and (ii) *Generic/Supportive Functions.*

IoT Device Virtualized Functions (IDVF) encompass functions that handle a portion of an application's business logic. They are responsible for deploying and managing IoT-specific functionalities, such as video transcoding for cameras, image processing and analysis for remote healthcare devices, or face detection sensors. The primary goal of IDVFs is to offload computationally intensive tasks from IoT devices and transfer them to VOs running on nearby edge computing infrastructure. This virtualization approach enables the integration of IoT functions into edge computing applications and facilitates their dynamic management by edge and cloud computing orchestration platforms. IDVFs are primarily envisioned to be provided in the form of cVOs, representing an advanced capability of a VO.

Generic/Supportive Functions (GSFs) encompass a set of supportive functions that can be applied horizontally across all instantiated VOs for an application. These functions provide generic support for IoT-oriented functionalities, such as distributed data management, data aggregation, filtering, firewalling, authentication, and failure handling. Additionally, they support functionalities at the edge part of the infrastructure, such as service discovery and telemetry. Given their broad applicability, GSFs should be incorporated into the basic VO implementation and can be activated on demand based on the specific needs of the application.

VOStack adheres to a microservices-based approach, depicting cloud-native applications as an application graph consisting of independently deployable components. This approach ensures modularity, openness, and interoperability, especially with orchestration platforms. The deployable components, encompassing application functionalities such as supportive functions and IoT device virtualized functions, can be deployed either in the cloud or at the edge of the continuum. The VO stack offers these components as generic functionalities adaptable to the specific needs of an application. Each component in the application graph is accompanied by a sidecar, aligning with the service-mesh approach, which can be activated on demand.

Orchestration management interfaces 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 3.5, we depict a high level representation of the approach proposed within NEPHELE where application graphs may be interlinked with VOs and cVOs based on well defined HTTP-based interfaces. In Figure 3.1, 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. Therefore, the (c)VO is considered as 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.

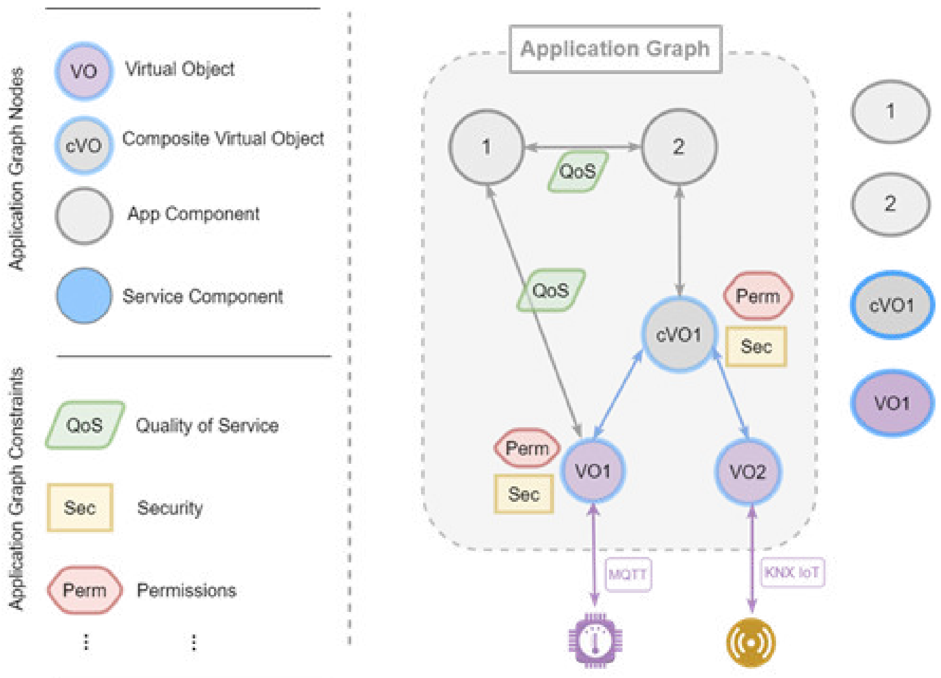


Fig. 3.5. NEPHELE Orchestration Interfaces

4

Softwarized and Virtualized environment for VO

This Chapter outlines the key attributes of the Cloud Computing paradigm and its evolution into the Multi-access Edge Computing (MEC) in the context of 5G softwarized and programmable networks. Paradigms like Software Defined Networking (SDN) and Network Function Virtualization (NFV) are also summarised.

4.1 The Cloud Computing

The advancement of cloud technologies and architectures has been pivotal in shaping the current 5G ecosystems [64]. 5G technologies not only offer the means to implement the "softwarization" of telecommunication infrastructures but also enable diverse expressions of the cloud, such as edge computing [65]. The key factor contributing to the success of these technological progressions lies in the architecture of the cloud, which allows for the slicing of roles and resources of all involved stakeholders. This separation facilitates the creation of multiple vertical markets, harnessing the flexibility of modern virtualization technologies and the ensuing "as-a-Service" paradigm.

In modern cloud contexts, every architectural aspect can be delivered to the end customer for direct utilization, with specific reference to the provisioning of infrastructure Infrastracture as a service (IaaS), platform Platform as a service (PaaS), and software Software as a service (SaaS) (Figure 4.1):

- *Software as a Service (SaaS)*: represents the most abstract level. The user can use the applications provided by the service provider running on remote infrastructure. The user accesses the applications through a graphical interface (e.g., via a web browser, dedicated app) across multiple devices (e.g., smartphones, PCs) but has no control over the applications, data, or platform characteristics (in terms of hardware, software, operating system configuration). Typical examples at this level include email applications through a web browser or social networks.

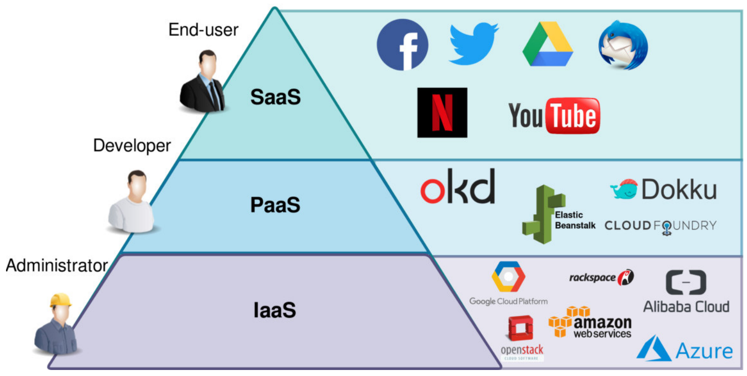


Fig. 4.1. Cloud service models [5]

- *Platform as a Service (PaaS)*: is the intermediate level in terms of abstraction. The user has a development environment where they can upload their applications, but they have control only over the applications they upload and their configuration, not over the infrastructure itself (operating system, memory, disk storage, network).
- *Infrastructure as a Service (IaaS)*: represents the level with the lowest abstraction. The user has the ability to partially customize the configuration with the service provider, in terms of provisioning certain resources (operating system, number of allocated CPUs, RAM, disk storage, network capacity), but not the physical infrastructure that delivers the cloud service.

The development of SaaS is particularly foundational for 5G-ready applications. To be suitable for use in the 5G domain, an application must be decomposed into a collection (chain) of *microservices* developed using a cloud-native methodology. In this approach, each microservice has its isolated execution environment, handles specific well-defined functionalities, and can be instantiated multiple times. While the term "cloud-native" lacks a specific definition, it is generally used to indicate that application components possess properties that enable instantiation in a cloud environment and fully exploit the benefits offered by such usage.

Collaboration among microservices constituting a 5G-ready application is based on a logical graph defining their interactions. Notably, infrastructure programmability aspects affecting the application's behavior should be considered during development. It is advisable to take into account the resources available (such as VCPUs, RAM, storage space, bandwidth, security constraints, etc.) during the development phase. Although the characteristics of a cloud-native component are necessarily heterogeneous, certain attributes are essential to facilitate interaction within the application graph and manage various stages of activity after instantiation (lifecycle). These include the presence of configuration parameters and quality of service levels, interfaces for graph composition, and programmability for lifecycle management. Moreover, while the absence of state would be ideal for managing components of a 5G-ready application, it is desirable to have at least the component's state separated from the image to facilitate migrations.

Within a cloud ecosystem, an orchestration layer is used to manage interactions among application components and the application's lifecycle. Typically, there is an orchestrator for each industry in the ecosystem, managing one or more applications. However, the roles and interactions between interested parties, and the resulting reference architecture, can vary significantly based on the context. IaaS providers offer their computing and/or network infrastructures using management interfaces called Virtual Infrastructure Manager (VIM). Through VIMs, customers (typically third-party platforms or developers) can monitor and manage the entire lifecycle of their services including physical, network, computing, and memory resources, with service activation and deactivation and modification of dedicated resources.

Currently, *OpenStack* [66] [67] represents de facto standard for implementing VIMs. To better manage the complexity and, more importantly, the specificity of the many services supportable in the cloud, many of these operations are delegated from VIM to one or more orchestrators. These orchestrators can be dedicated to individual entities, such as PaaS and SaaS providers, and operate in a cascading fashion, automating many of their functionalities.

To ensure the isolation of each entity within the infrastructure, a fundamental role is also played by the runtime environments used for services, both application and network. In general, an application can be designed to run through a container or in a hypervisor. A container is a software module in a filesystem containing all the necessary system tools and libraries for the proper functioning of the application. A hypervisor is a Virtual Machine (VM) manager that can operate above physical hardware or an operating system. The hypervisor virtualizes the workload of applications into various VMs to ensure the flexibility and stability provided by virtualization. The choice of one runtime environment over the other must be made by balancing the needs related to access to physical resources and ease of migration. As highlighted by ETSI [68], the most common requirements relate to the isolation of an application's runtime environment, efficiency in the use of physical resources, performance level compared to execution in the native environment, ease of runtime environment management, and portability. Furthermore, resource management is crucial for a proper allocation of capacity to ensure coverage of the requirements of each application. The optimal use of physical resources is achieved when an application runs in a container rather than a VM [69]. Regarding storage, the hypervisor consumes memory space for each VM, while containers utilize the same space for each application with small increments, ensuring less waste. Additionally, the waiting times for the instantiation of the runtime environment for a new application are much lower for containers than for VMs. Taking these considerations into account, the project is evaluating the use of Linux containers or Docker Compose [70] for creating and managing applications designed to run in multiple containers.

4.2 Network Function Virtualization (NFV) and Software Defined Networking (SDN)

In the historical context, telecommunication networks were heavily dependent on numerous dedicated and proprietary hardware components, each specifically designed for distinct network functions and routing. Introducing or modifying a service often required acquiring or designing new dedicated physical equipment. These devices not only occupied physical space and consumed substantial energy but also presented challenges in terms of integration and faced rapid obsolescence due to the swift evolution of technology. The paradigms of NFV and SDN offer a viable solution to these challenges through the application of virtualization techniques. Instead of relying on dedicated physical hardware, network functions are instantiated as software instances known as Virtual Network Functions (VNFs), and the network is defined using software, referred to as Software-Defined Network (SDN). For example, VNFs can be created, relocated, and managed across different locations within the network, providing unparalleled flexibility and adaptability.

This paradigm shift obviates the need for frequent physical hardware upgrades, as network functions can be updated or replaced in a software-driven manner, bringing about a transformative change in both the operational and evolutionary aspects of networks.

4.2.1 Network Functions Virtualization (NFV)

NFV [71] is an architectural network paradigm that leverages information technologies to virtualize entire classes of network functions/devices into independent software components that can be chained together to create various communication services, more or less complex. The virtualization of a network must ensure transparency towards users, a level of performance comparable to that of physical devices, and proper interaction with these devices. Through NFV, network operators can utilize generic reconfigurable hardware, thereby reducing infrastructure costs, and decouple the hardware lifecycle from that of the software. Other benefits of NFV include the rapid creation and activation of services, scalability, and enhanced security through easier separation and isolation. The standardization of NFV is carried out by the ETSI NFV Industrial Study Group [72], which is responsible for defining the architecture and interfaces for transforming physical functions into software applications, called Virtual Network Function (VNF), that run in VMs or containers. For this transformation and the subsequent provisioning and management of services, OpenStack [66] is a strong candidate, providing highly configurable interfaces for both public and private clouds.

The incorporation of Network Function Virtualization (NFV) offers a range of compelling advantages [6]:

- Cost Efficiency: The transition to standardized hardware components and the reduction of reliance on specialized devices result in substantial reductions in both capital expenses (CAPEX) and operating expenses (OPEX).
- Operational Flexibility: NFV facilitates the rapid deployment and scaling of network services, providing the capacity to adapt to evolving demands without being constrained by the limitations of physical hardware.
- Service Agility: NFV enables the swift introduction or modification of services, leading to a more expedited time-to-market and heightened responsiveness to the dynamic shifts in the market.

In summary, NFV emerges as a transformative force in the realm of telecommunication networks, offering a more agile, cost-effective, and scalable approach to the provisioning of services and the management of networks.

The NFV architecture

Understanding the functionality and application of NFV hinges significantly on grasping its architecture, which comprises three fundamental elements: Network Function Virtualization Infrastructure (NFVI), VNFs, and NFV Management and Orchestration (MANO). Figure 4.2 shows the NFV architecture composition.

The NFV Infrastructure (NFVI). NFVI stands as a pivotal platform, seamlessly integrating hardware and software resources to serve as the bedrock for deploying VNFs. Physical resources, comprising commercially available computing hardware, storage infrastructure, and network components, provide the underlying muscle, ensuring adequate processing power, storage capacity, and connectivity for VNFs to

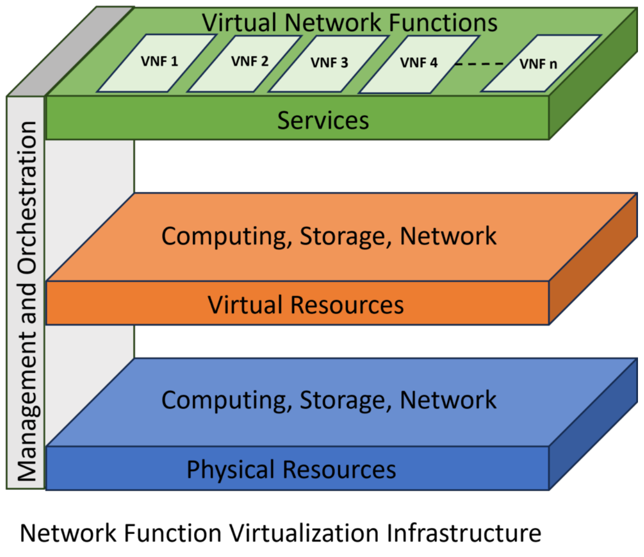

Virtual Network Functions
VNF 1
VNF 2
VNF 3
VNF 4
VNF n
Services
Computing, Storage, Network
Virtual Resources
Computing, Storage, Network
Physical Resources
Management and Orchestration
Network Function Virtualization Infrastructure

Fig. 4.2. NFV architecture [6]

operate effectively. In contrast, virtual resources represent distilled versions of these physical components, achieving abstraction through a virtualization layer, typically anchored by a hypervisor. This layer effectively isolates virtual resources from their physical counterparts, enabling their seamless management and orchestration. In a data center setting, computing and storage resources are typically manifested as VMs, while virtual networks materialize as interconnected virtual links and nodes.

Virtual Network Function. A Network Function (NF) is a fundamental building block within a network infrastructure, characterized by well-defined external interfaces and a specific functional behavior. Examples include devices in a home network, such as a Residential Gateway (RGW), and traditional network functions like Dynamic Host Configuration Protocol (DHCP) servers and firewalls. In contrast, a VNF represents the virtualization of an NF, implemented on virtual resources like Virtual Machines (VMs). A single VNF may comprise multiple internal components and can be deployed across multiple VMs, with each VM hosting a distinct component of the VNF.

The NFV Management and Orchestration (MANO). The MANO framework stands as a critical component of the NFV architecture, orchestrating and managing the lifecycle of physical and/or software resources underpinning VNFs. It ensures the efficient allocation, coordination, and optimization of these resources to deliver seamless end-to-end network services. The role of orchestration in NFV encompasses the automated arrangement, coordination, and management of VNFs and network services. This involves dynamically allocating resources, instantiating VNFs, and chaining these functions to create cohesive services. Orchestration aims to optimize resource utilization, enhance service reliability, and facilitate rapid service deployment and scaling. The MANO framework comprises three primary components:

- NFV Orchestrator (NFVO): The NFVO serves as the orchestrator of NFV infrastructure resources and services. It governs the lifecycle of network services, from inception to termination, and collaborates with VNF managers to manage the lifecycle of VNF instances.
- VNF Manager (VNFM): The VNFM oversees the lifecycle of VNF instances, encompassing instantiation, scaling, updating, and termination of VNFs. It also coordinates with the NFVO for resource allocation and with Element Management Systems (EMS) for configuration and fault management of VNFs.
- The VIM acts as the steward of virtualized resources, ensuring their availability for deploying VNFs. It seamlessly interfaces with the NFVO and VNFM to provision the necessary resources and report on their operational status, ensuring the seamless deployment and operation of VNFs.

The ever-changing and adaptable characteristics of virtualized resources and services call for the creation of advanced algorithms and strategies to uphold optimal performance and ensure service availability. Crucial challenges involve guaranteeing uninterrupted service continuity VNF migrations, proficiently overseeing VNFs from multiple vendors, and establishing robust security measures to protect the virtualized environment. Despite these challenges, the intrinsic flexibility and agility of NFV orchestration present substantial prospects for swift service deployment, effective resource utilization, and the capacity to adjust to evolving network conditions and requirements.

NFV Tools

A variety of tools and platforms have emerged to facilitate NFV orchestration, with some of the most notable ones being:

- Docker: This platform enables developers to package applications into containers, which are standardized executable components combining application source code with the operating system. In the context of NFV, Docker is employed to containerize Virtual Network Functions (VNFs), ensuring consistent deployment across diverse environments.
- Kubernetes: An open-source container orchestration platform designed to automate the deployment, scaling, and operation of application containers. In NFV

applications, Kubernetes is capable of managing and orchestrating VNFs packaged as containers, offering features such as auto-scaling, load balancing, and self-healing.

- OpenStack Tacker: As an official OpenStack project, Tacker provides NFV orchestration functionalities, encompassing VNF lifecycle management and network service orchestration.
- ONAP (Open Network Automation Platform): A comprehensive platform dedicated to real-time, policy-driven orchestration and automation of both physical and virtual network functions. ONAP facilitates the design, creation, and management of VNF services.

4.2.2 Software Defined Networking (SDN)

The introduction of SDN [73] [74] adds further capabilities to the NFV paradigm. With SDN, traffic can be injected not only based on IP addressing but also on a flow basis, achieving greater granularity in network traffic management. SDN and NFV offer mutual advantages but can also be used individually.

SDN is a technological paradigm based on the separation of hardware from the software of dedicated network devices, allowing the execution of this software not only in the network infrastructure but also in the cloud and on server architectures. This decomposition, in particular, enables the separation of the typical *control plane* and *data plane* functions of a traditional router, providing extensive abstraction possibilities, such as the creation of distributed virtual networks under the same control. Thanks to a set of APIs provided by SDN, it is possible to define a global view of the network within the control plane through an abstract graph and its corresponding element control. The global network view is maintained in an operating system, which can also manage resources dedicated to VNFs by scaling vertically, allocating more/less resources to a single VNF, or horizontally, adding/removing new instances of a VNF.

The Northbound interfaces of the SDN controller, allowing application interaction with the network, are specific to the controller. As for the Southbound interfaces, the most common is probably the *OpenFlow (OF)* protocol [75]. In general, Southbound interfaces are not service-oriented and as such, need to be extended to handle more specific contexts. Open Source Mano (OSM) [76] for NFV service orchestration and OpenDaylight (ODL) [14] as the SDN controller.

4.3 5G and Edge computing

4.3.1 5G

The fifth generation of mobile networks (5G) brings about the definitive integration of telecommunications networks and computing/storage resources, creating a new infrastructural paradigm to ensure convergence between fixed and mobile access. This leads to complete service ubiquity, opening doors to economic and social innovations previously unrealizable [77]. 5G primarily leverages programmability and softwarization paradigms to enhance radio technology performance in terms of capacity, transmission speed, and connectivity.

Unlike previous generations of mobile communications, the 5G architecture is service-centric, orchestrating all real and virtual entities (devices and applications) collaborating to deliver services. Hence, the term Service-Based Architecture (SBA) is often used to describe this paradigm. This implies that most architectural elements consist of network functions that can expose their services to any platform authorized to interact with them through common interfaces (APIs). While these functions are primarily VNFs, the presence of physical functions is not excluded a priori. Therefore, the key components of a 5G network remain user devices, the access network, and the core network, with the architecture of both networks significantly modified to adhere to the SBA.

The major innovation introduced in the access network (New Generation (NG)-Radio Access Network (RAN) is the division of functionalities traditionally provided by base stations into centralized and distributed components. This division aims to improve scalability, implementation and management costs, and make performance proportional to actual loads. In practice, the division, commonly recognized by standardization bodies, involves protocol mechanisms up to level 2.5 at the site and higher-level functionalities in centralized servers. This division is entirely transparent, and neither the core network nor other antennas perceive it.

Modularity principles are also observed in the realization of the core network [78], where various network functions are separated between the user plane and the control plane, following the practices of SDN and NFV-based techniques. The interaction between network functions can be direct or mediated by dedicated functions in the control plane. Both communications between the network functions composing the core and those towards external entities are conducted through REST interfaces, aligning with current trends in vertical application development, especially in industrial automation.

Figure 4.3 illustrates the 5G network architecture and highlights the main functions of the core network. At the user plane level, the most crucial function is played by the User Plane Function (UPF), which takes on the role of the S/P-GW in 4G but additionally can inject traffic directly towards applications using forwarding rules determined by the applications themselves. These rules are mediated by a control plane function, the Session Management Function (SMF). At the control plane level, a fundamental role is played by the Network Exposure Function (NEF), allowing the exposure of core network functionalities to third parties (such as vertical industries and external service providers) beyond the operator's domain.

To provide maximum flexibility, methodologies for allocating these functions are being explored to meet the specific characteristics of various vertical industries, such as mobility requirements or direct communication between instances or components of their applications [79]. For this reason, 5G is the first generation of mobile technologies to provide intrinsic virtualization capabilities in the form of the network slicing concept [80]. A network slice is defined as a set of network functions, along with their resources, configured to form a complete logical network capable of meeting all communication requirements of a specific business. To adhere to this definition, a slice must provide all network levels (from access to core) and interfaces for access.

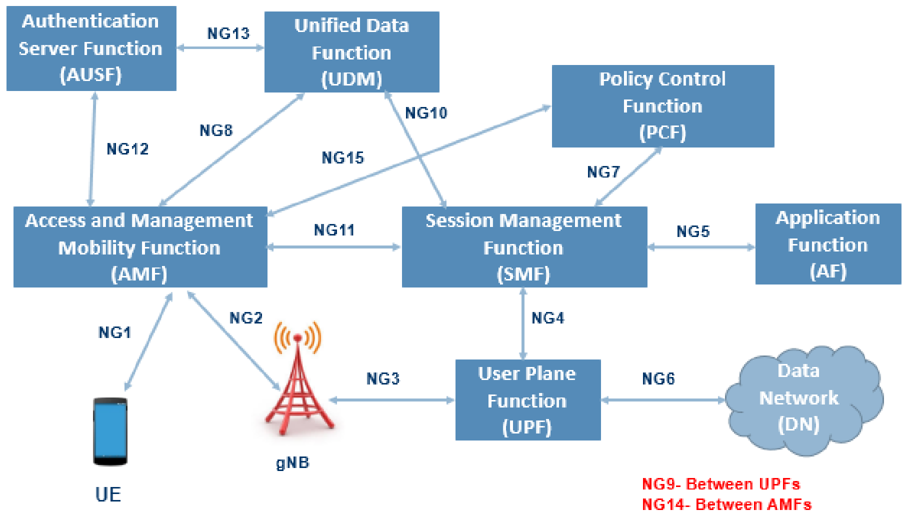

Authentication Server Function (AUSF)
NG13
Unified Data Function (UDM)
Policy Control Function (PCF)
NG8
NG10
NG12
NG15
NG7
Access and Management Mobility Function (AMF)
NG11
Session Management Function (SMF)
NG5
Application Function (AF)
NG1
NG2
NG4
NG3
User Plane Function (UPF)
NG6
Data Network (DN)
gNB
UE
NG9- Between UPFs
NG14- Between AMFs

Fig. 4.3. 5G reference architecture (*ref: https://www.techplayon.com/5g-reference-network-architecture/*)

4.3.2 Multi-access Edge computing (MEC)

The proliferation of various new use cases, including the IoT, has significantly contributed to the surge in network traffic. Activities such as the consumption of high-definition video (UHD, 4K) have witnessed a substantial increase, driven in part by the expanding role of Content Delivery Networks (CDNs). This shift in network dynamics has propelled the evolution of cloud computing towards edge computing. Concurrently, emerging applications like Augmented Reality (AR), characterized by their computational intensity and sensitivity to delays, are experiencing substantial growth. Both video traffic and AR applications necessitate low latency and high throughput, making them prime candidates for leveraging storage and processing resources in close proximity to users. This approach not only enhances the QoE for users requiring high-speed connections but also alleviates congestion in the core network. Fulfilling content requests locally through the access network eliminates the need for establishing peer-to-peer connections with remote servers.

The ETSI Industry Specification Group (ISG) first coined the term Mobile Edge Computing (MEC) to describe the trend of siting cloud capabilities in close proximity to mobile end devices at the RAN premises. In September 2016, the ETSI ISG renamed Mobile Edge Computing to "Multi-access Edge Computing" to broaden its applicability to heterogeneous network technologies, including WiFi and fixed access, in addition to cellular networks.

In the IoT sector, MEC emerges as a pivotal player, enabling objects to gain intelligence by communicating data about themselves and their measurements. Typically, IoT objects operate with limited computational capacity and memory. MEC facilities, situated at the Edge (i.e., Virtualized Operator), assume the role of storing and aggregating data from IoT sensors and actuators. This type of traffic demands very low latency and a substantial amount of memory to deliver efficient real-time services. Moreover, the sheer volume of traffic generated by billions of new IoT devices has the potential to overwhelm the network if directed to the remote cloud, increasing the load on the core network.

The MEC technology [81] is recognized as crucial for bringing applications inside the network operator's infrastructure [82], allowing proximity to users and information about them to deliver increasingly better service quality, high levels of customization, and even entirely new applications. This is facilitated by collaboration with the aforementioned NFV and SDN paradigms. In particular, MEC and NFV can be seen as complementary technologies: while NFV provides network services and functions, MEC handles applications at the 7th layer of the OSI stack. Considering the similarity of infrastructural dependencies between the two, ETSI [83] is attempting to identify guidelines to enable their cooperation. The architecture defined for this purpose includes a host physically containing the MEC platform and its applications, a manager for controlling the platform's functionalities, and an orchestrator tasked with managing applications hosted on multiple hosts. In this perspective, the ability to interact with VIMs belonging to different owners plays a fundamental role in extending the plethora of virtual resources to build applications and network slices.

Given the above, it is evident that ensuring resource isolation in a multi-ownership context is even more critical. To this end, sophisticated multi-site resource management mechanisms are necessary and must be capable of supporting heterogeneous VIMs. This involves implementing specific APIs for each VIM, appropriate levels of abstraction, aggregating available resources across various sites, interconnecting resources assigned to a specific entity, and providing these mechanisms in the most automated manner possible. At present, the two most advanced solutions, both part of the OpenStack project, are KingBird [84] and Tricircle [85].

Beyond MEC, other proposals in the literature, such as Fog Computing and Cloudlet, offer alternative approaches for processing requests at the edge of the network. A comprehensive comparison of these three Edge Computing technologies is presented in [86].

ETSI MEC Architecture

ETSI MEC places its primary emphasis on the system and host levels, as illustrated in Figure 4.4. In essence, it involves a *MEC host*, representing the platform linked to an individual Edge Server, engaged in interactions with other MEC hosts. Additionally, there exists a host-specific management mechanism referred to as *MEC host level management*, along with a system-wide management level known as *MEC system level management*.

Principal components of the MEC architectures are:

- *MEC Host*: It is a logical entity that includes the MEC platform and the virtualization infrastructure, providing computing, storage, and network capabilities to MEC applications. The virtualization infrastructure consists of a Data Plane that enforces traffic forwarding rules received from the MEC platform and manages routing between applications, services, and the network.
- *MEC Platform*: It offers the necessary functionalities for running virtualized *MEC applications*, incorporating a service registry and, if needed, an advertising service

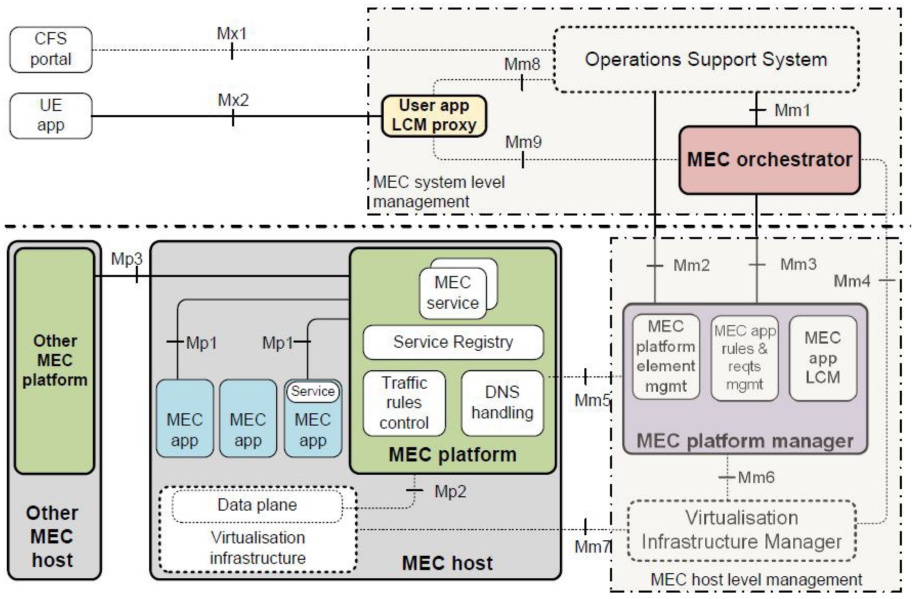


Fig. 4.4. ETSI MEC architecture

that promotes the existence of particular services in the MEC network, facilitating the discovery of applications. Additionally, the MEC platform facilitates the configuration of the local Domain Name System (DNS) to guide user traffic toward MEC applications. Moreover, the MEC platform is capable of deploying virtualized applications as services to assist other applications, known as *MEC services*.

- *MEC Apps*: Executed within virtual machines on the Virtualization Infrastructure, these applications have the capability to utilize MEC services and offer services not only to the broader MEC platform but also to users who have initiated service requests.
- *Virtualization Infrastructure*: This infrastructure serves as the virtualization foundation, providing resources to MEC applications. It encompasses a Data plane enabling communication between applications and the broader infrastructure.
- *MEC Orchestrator*: Functioning as the application orchestrator, the MEC Orchestrator maintains a comprehensive overview of managed MEC hosts, considering available resources, provided services, and the overall topology, which outlines how various MEC hosts are interconnected. The MEC Orchestrator collaborates with MEC host level management to prepare the infrastructure for supporting applications. It plays a crucial role in selecting the most suitable MEC host for each application and manages processes related to stopping and migrating MEC apps when necessary.

- *MEC System Level Management*: During the instantiation phase of a MEC application, the MEC system level management ensures the validation of the service and specific application requirements, such as the maximum allowable latency.
- *Operation Support System (OSS)*: This component intercepts requests to instantiate a MEC app from users or third parties. It plays a vital role in determining, based on agreed-upon Service Level Agreement (SLA)s with the operator, whether a request can be fulfilled. Subsequently, it forwards the request to the MEC orchestrator.
- *MEC Platform Manager*: This block is responsible for overseeing the lifecycle of virtualized MEC applications, including instantiation, migration, scaling, and termination. The MEC platform manager interacts with the MEC orchestrator, providing information on events related to MEC apps, enabling the orchestrator to make informed decisions.
- Virtualization Infrastructure Manager (VIM): Responsible for managing and configuring the virtualization infrastructure for MEC app execution. Its tasks include allocating, managing, and releasing resources associated with various MEC apps.

4.4 Benefits of edge computing and virtualization

4.4.1 From Cloud to Edge computing

Cloud computing has garnered significant attention in the realm of enterprise IT infrastructure, providing more favorable solutions to address the growing demands for processing and storage (i.e. Big Data and Blockchain). However, the substantial impact of IoT, driving the diffusion of wearable devices, smart environments, and a general rise in M2M communications, has altered the landscape of devices utilized at the network edge. In response to these developments, ETSI formulated the MEC architecture, as detailed in the previous section, to cater to these evolving requirements. The primary advantages that a MEC solution can provide include:

- *QoS with real-time requirements and lower latencies.* The latest generation of mobile devices imposes increasingly high QoS demands due to mobility and the stringent requirements of real-time and interactive applications. Cloud computing alone may not be the optimal solution as packets must traverse numerous nodes before reaching remote servers located in distant networks. The MEC approach reduces latency in accessing cloud services.
- *Increase of Battery life.* Concerns about battery life are crucial for mobile devices. MEC offers the advantage of running tasks on the edge infrastructure instead of

the device itself, contributing to enhanced battery life. While cloud computing provides a similar benefit, it incurs higher battery consumption due to data transmission costs. Longer routes increase the probability of packet loss, leading to more retransmissions and elevated battery consumption [87].

- *Lower congestion in the core network.* Providing services closer to the user reduces overall network congestion, as input data no longer need to traverse the entire network. Storing data items, such as high-definition multimedia content, in MEC servers is particularly beneficial for popular content, avoiding the transmission of large identical packets that would occupy significant bandwidth in the core network.
- *Scalability.* Deploying services and applications by replicating them as virtual machines presents an opportunity to enhance the scalability of network management, even in the face of the substantial traffic generated by IoT devices.
- *Resilience.* Running applications and services at the edge ensures that, in the event of anomalies or errors, the problem does not impact the entire network. This localized approach facilitates easier problem resolution.

4.4.2 Network virtualizaion and VO

Delving into the interaction between VOs and the virtualized environment, we find that in the dynamic landscape of telecommunication networks for the IoT, both NFV and VOs assume pivotal roles. Their coexistence and collaboration within the same ecosystem are not mere happenstance but are driven by the overarching objective of establishing a more flexible, scalable, and efficient network infrastructure.

Leveraging tools such as Docker and Kubernetes, VNFs seamlessly integrate with VOs, allowing for dynamic and context-aware network services. For example, a VO representing a sensor can interact with another service, both functioning as VNFs orchestrated by Kubernetes, ensuring real-time data processing and analytics. Moreover, they can enable advanced services like real-time analytics, context-aware networking, and adaptive resource allocation.

The synergy between NFV and VOs aims to abstract the physical layer, fostering more dynamic and adaptable systems. NFV concentrates on decoupling network functions from dedicated hardware, while VOs represent the abstraction of physical objects in the IoT domain. The convergence of these concepts in modern networks underscores the industry's shift towards a more software-centric approach, fostering rapid innovation and adaptability. For instance, a VO representing a air quality sensor in an industrial IoT setup can collaborate with a VNF to prioritize its data during a critical spike.

A unified management and orchestration, with both network functions and IoT objects being virtualized, not only simplifies network management but also ensures optimal resource allocation based on both network and IoT demands. An example is reported in the next subsection. This vision transports the Network towards a future of telecommunication and IoT networks—virtualized, integrated, and highly adaptive.

4.4.3 The NEPHELE synergetic meta orchestration framework

The synergetic meta-orchestration framework provided in the NEPHELE project[1] aims to address the challenges posed by hyper-distributed applications in the next generation compute continuum. Existing orchestration platforms focus on specific areas such as the cloud, edge, or IoT within the compute continuum. However, they lack the inherent suitability to handle the orchestration complexities of hyper-distributed applications. These platforms typically hold the responsibility for resource allocation across the continuum but lack control, knowledge, and authorization for effective horizontal scheduling of different application components.

Notably, varying levels of access and control exist for IoT devices, edge, and cloud computing resources, managed by different platforms or providers. As applications become more distributed, the coherence of failures diminishes, and the distance between cause and effect increases. To address these challenges, a meta-orchestration level is essential to facilitate synergy among different orchestration systems/platforms by generalizing and modeling their orchestration modules.

In the NEPHELE project, the meta-orchestration framework adopts a "system of systems" approach (Figure 4.5, where complex systems (orchestration modules in different parts of the continuum) are managed by a large-scale concurrent and distributed system. Technological advancements in 5G and beyond networks, Artificial Intelligence (AI), and cybersecurity are integral components and will be seamlessly integrated as pluggable "systems" in the synergetic meta-orchestration framework. In summary, NEPHELE envisions enabling efficient, reliable, and secure end-to-end orchestration of hyper-distributed applications across programmable infrastructure spanning from Cloud-to-Edge-to-IoT. It aims to eliminate existing openness and interoperability barriers in the convergence of IoT technologies with cloud and edge computing orchestration platforms. The project also introduces automation and decentralized intelligence mechanisms powered by 5G and distributed AI technologies.

[1] https://nephele-project.eu/

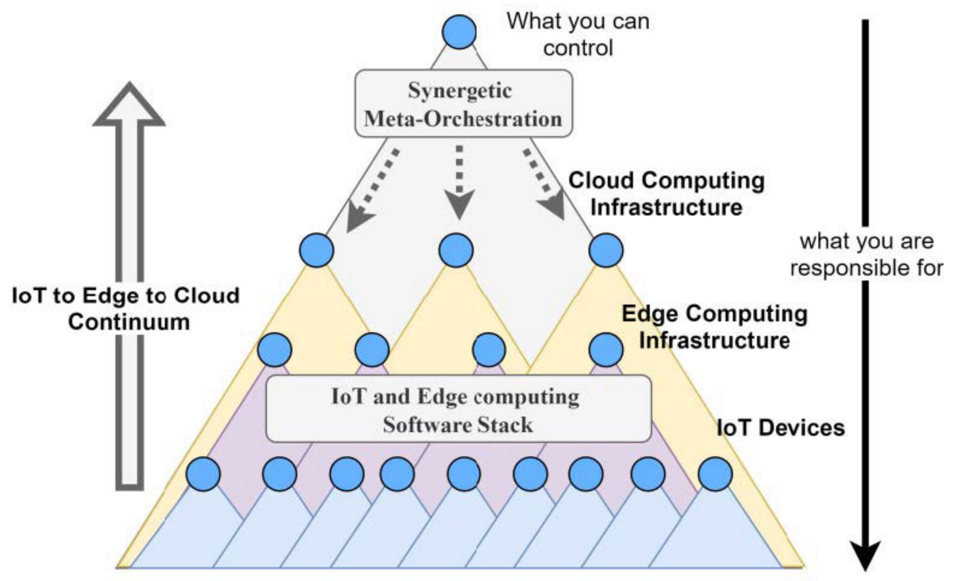

What you can control
Synergetic Meta-Orchestration
Cloud Computing Infrastructure
what you are responsible for
IoT to Edge to Cloud Continuum
Edge Computing Infrastructure
IoT and Edge computing Software Stack
IoT Devices

Fig. 4.5. NEPHELE synergetic orchestration in the computing continuum (*ref: Nephele EU project https://nephele-project.eu/)*

5

The VO and cVO implemented

In this Chapter, the developed VO is described as a software component that represents the enhanced virtual counterpart/extension of a physical device, often constrained. A VO can also be seen as a microservice, which is a service offered by the network to reduce direct communications to the physical device and, at the same time, extend/enrich the sensor data through the processing of metadata useful to the application context without further burdening the device.

5.1 The VO model for semantic interoperability

The most suitable approach for representing IoT devices involves leveraging semantic technologies [47]. Consequently, the VO enriches the semantic aspects of data and functionalities provided by IoT devices. The outcome of this semantic description is the VO model, encompassing characteristics such as object features, object location, resources, services, and quality parameters provided by objects. The VO model, designed as software for a specific service, is independent of a particular device; it initializes at startup based on the properties of the corresponding physical counterpart it represents. This initialization relies on a purpose-built configuration file, to which a section of this chapter called *Descriptor file* is dedicated.

In OMA-LwM2M, a Device Description File (DDF) of Objects is provided through an XML configuration file, defining the object structure and its resource data. The data producer, hosting objects and resources, is termed the OMA-LwM2M client, while the data consumer is the OMA-LwM2M server. Both entities only need to possess the same configuration file for information serialization and de-serialization. The public registry[1] supplies objects defined by OMA, Internet Protocol Security Option (IPSO), and standard objects from third-party organizations. Additionally, developers can define customized objects following technical specifications.

[1] https://technical.openmobilealliance.org/OMNA/LwM2M/LwM2MRegistry.html

The VO, based on OMA-LwM2M, aims to achieve pivotal objectives, including overcoming platform heterogeneity, ensuring interoperability, enhancing search and discovery, and alleviating the burden on constrained devices. It provides a semantic description of the physical counterpart to ensure a shared understanding of its features and capabilities among potential consumer applications. Specifically, it abstracts the embedded components, thereby decoupling the specific hardware and software platform implementations. Consequently, the VO exposes the capabilities of the relevant physical device to interested applications, facilitating transparent access to intelligent heterogeneous resources. This feature is particularly advantageous for sophisticated applications relying on, for example, AI inference capabilities. The semantic description, both in a general sense and specifically for AI-empowered IoT devices, streamlines search and discovery procedures, identifying the most suitable components for a given task based on the demands of the requesting application. Furthermore, this abstraction of IoT device capabilities ensures interoperability, overcoming fragmentation, and acts as a proxy between the physical device and consumer applications. It effectively responds to requesting applications, making IoT devices available in an interoperable manner to all interested applications.

The semantic description effectively addresses heterogeneity and fosters interoperability in the IoT domain, mitigating the existence of vertical silos. Additionally, it proves instrumental in supporting search and discovery operations. These mechanisms facilitate the identification of the most suitable device for a given application's task.

5.2 Tools and platforms

The VO and cVO code has been developed during the research activity to encompass the design requirements illustrated in the first chapters of the thesis work. The following tools were used to develop the microservice to fulfill all the requirements and to facilitate further future developments. Some of them were used directly for the development of the VO interfaces and functions; others, however, were used to integrate the service within the virtualized Edge computing environment (i.e. Docker).

5.2.1 The Spring Boot service framework

Spring Boot is a powerful and widely adopted framework in the Java ecosystem. It is designed to simplify the process of building robust and scalable Java-based applications. The evolution of enterprise software development has led to the emergence of frameworks that streamline and simplify the creation of applications. One such framework is Spring Boot, an open-source extension of the Spring framework that addresses common challenges faced by developers during the development lifecycle. It is known for its convention-over-configuration approach, which minimizes boilerplate code and enables developers to focus on business logic.

At its core, Spring Boot leverages the Spring framework's capabilities, building upon the Inversion of Control (IoC) and Dependency Injection (DI) principles. It follows the convention-over-configuration paradigm, reducing the need for explicit configuration. By adopting sensible defaults, developers can create applications with minimal setup, enhancing productivity and reducing the cognitive load associated with configuration. It embraces a modular and layered architecture, promoting the use of reusable components and promoting code organization.

One of the standout features of Spring Boot is its support for embedded servers, such as Tomcat, Jetty, and Undertow. This eliminates the need for external server deployment, simplifying the deployment process and making applications more self-contained. Spring Boot is well-suited for building microservices-based architectures, allowing developers to create modular, independently deployable services. This aligns with contemporary software development trends and supports the scalability and maintainability of applications.

Spring Boot stands as a pivotal framework in the Java ecosystem, offering a pragmatic and efficient solution for building robust and scalable applications. As technology continues to evolve, Spring Boot remains a cornerstone in the landscape of Java-based application development.

The implemented VO leverages on this framework to facilitate development by using components provided by the framework, such as Structured Query Language (SQL) drivers, libraries to manage HTTP calls and reception of requests via API endpoints, and interconnection with further third-party services. In addition, the framework enables dynamic configuration at service startup through the use of a single YetAnother Mark-up Language (YAML) file. The VO uses this functionality to manage the Descriptor file, which describes the physical device and the functionalities that the VO will acquire, and which will be better described in the following sections.

5.2.2 Apache Maven

Apache Maven [88] [89] stands as a pivotal tool in the realm of software development, revered for its dual role in build automation and project management. At its core is the Project Object Model (POM), an XML file that encapsulates crucial project details, including dependencies, plugins, and configurations. This POM serves as the lynchpin for Maven, offering a centralized hub of information from which it orchestrates the intricacies of a project. A notable feature is Maven's adeptness in managing project dependencies. By leveraging a central repository, Maven automates the download of required libraries and frameworks, alleviating the manual burden of dependency management. Embracing the philosophy of *convention over configuration*, Maven defaults to sensible configurations, reducing the need for explicit specifications, provided the project aligns with established conventions. Plugins, the workhorses of Maven, extend its functionality and allow developers to tailor the build process to specific project requirements. Maven's lifecycle and phases structure tasks logically, encompassing activities like compilation, testing, packaging, installation, and deployment. The concept of a consistent project structure is intrinsic to Maven, fostering clarity and uniformity across different projects. Moreover, its integration with various Integrated Development Environment (IDE)s enhances the development experience, seamlessly aligning with tools like Eclipse [90], IntelliJ IDEA [91], and Visual Studio Code [92].

In essence, Apache Maven's significance lies in its ability to streamline and automate the complexities of project management and build processes, offering a standardized yet flexible framework for developers. For the most current and detailed information, referring to the official Apache Maven documentation [89] is recommended, as the software landscape undergoes evolution and refinement over time.

5.2.3 Eclipse Leshan

Eclipse Leshan[2] is an open-source project under the Eclipse IoT working group. It provides a lightweight and scalable implementation of the OMA LwM2M protocol, which has been described in previous sections. The project aims to implement a solution for a communication protocol for device management and service enablement in the context of the IoT.

Leshan is not a monolithic standalone project but it offers libraries that assist developers in creating their Lightweight M2M server and client. The project includes a LwM2M client, server, and bootstrap server demonstration, serving as an illustration of the Leshan API and for testing purposes. Leshan depends on the Eclipse IoT Californium project for implementing CoAP and DTLS.

The Leshan project is developed using Maven and it is composed by following server and client packages:

- bsserver demo: it is the LwM2M Bootstrap server demo for client configuration at start-up.

[2] https://projects.eclipse.org/projects/iot.leshan

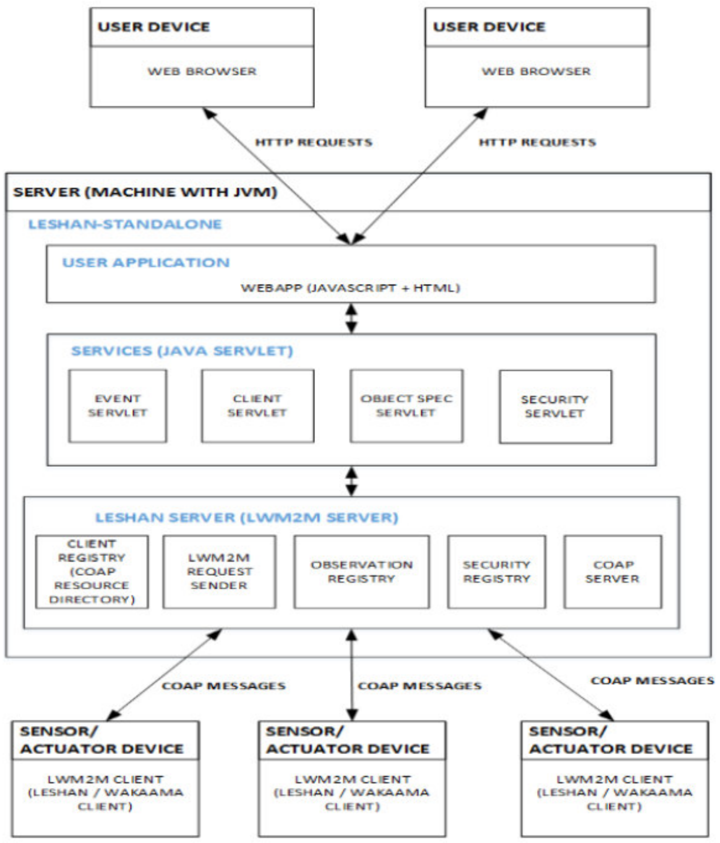


Fig. 5.1. Leshan architecture [7]

- Client cf: *client cf* is dependent of *Californium* [93] and contains specific code about Californium generally. This is mainly binding from CoAP to LWM2M concept for the physical device. Basically, it is the LwM2M client Californium interface for using CoAP protocol.
- Client core: it is the client (physical device) core package for bootstrap, registration, mandatory object Enabler (Security 0, Server 1, Device 3), observer enabler on device registration, Request sender (Sync and Async), and etc. Client core is not dependent of Californium and contains only LWM2M logic (no Coap logic) and ideally most of the LWM2M logic should be implemented here.
- Client demo: it is the module to run a client demo which emulate a physical device and will connect to the LwM2M server. It has two sensor LwM2M objects by default: Temperature (Object id: 3303), and Location (Object id:6). The jar (JAVA executable) file is located inside the path: *leshan-client-demo/"ẗarget/"l̈eshan-client-demo.jar*.
- Core: it is the LwM2M Enabler according to the OMA-LwM2M standard.

- Core cf: it is the core Californium interfaces module implementation. CoAP to LwM2M binding.
- Integration tests: it is the package to run tests over the implemented code.
- Server cf: it is the LwM2M server enabler, a Lightweight M2M server. This implementation starts a Californium CoAP-client with a unsecured (for coap://) and secured endpoint (for coaps://). This CoAP client defines a rd resource as described in the LWM2M specification.
- Server core: it is the core server package for operations: bootstrap, Model, request, registration, security, etc.
- Server demo: it is the module to run a server demo which will accept client connections. It has different HTTP servlets defined for APIs. One can find the .jar file (JAVA executable) inside the module at *leshan-server-demo/"ẗarget/"l̈eshan-server-demo.jar*.

The Leshan project has been particularly important in the realization of this work as it was used as a starting point and inspiration for the development of the virtualization of physical devices through the use of the OMA-LwM2M standard. Currently, in particular, the developed VO uses the Leshan server package to implement the CoAP-LwM2M interface. The Leshan architecture is depicted in Figure 5.1.

5.2.4 Eclipse Paho

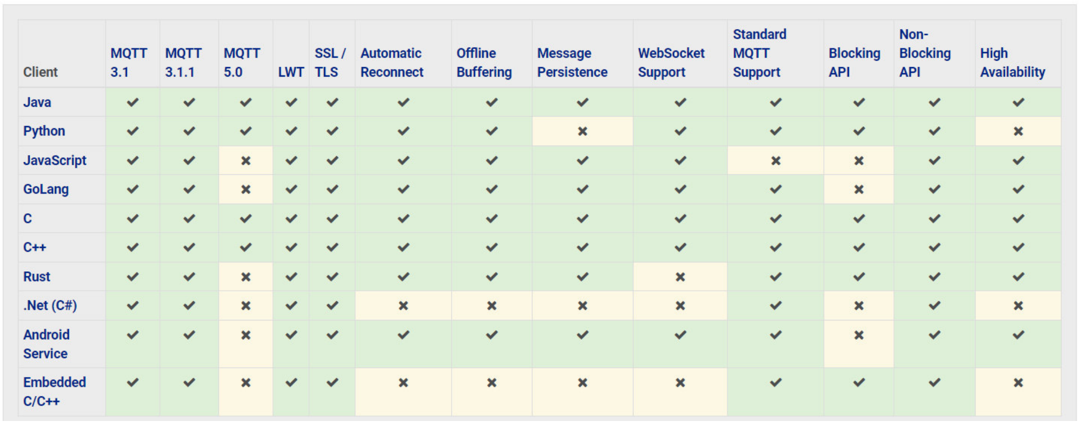
The Eclipse Paho project [8] is an open-source project which encompasses a suite of open-source libraries that provide reliable implementations of MQTT and MQTT for Sensor Networks (MQTT-SN). The Eclipse Paho libraries are meticulously crafted to ensure reliable and high-performance data exchange between devices and applications. They handle the intricacies of MQTT and MQTT-SN protocols, ensuring efficient and reliable message delivery. Moreover, project extends its reach by providing implementations for a wide range of programming languages, including C, Java, Python, and C #. This cross-platform compatibility eliminates language barriers and facilitates the integration of MQTT and MQTT-SN into applications written in different languages. Additional features are listed below:

- Modular Architecture and Backward Compatibility: The Eclipse Paho project adopts a modular architecture, allowing developers to selectively incorporate desired features while maintaining compatibility with older versions. This modularity enhances flexibility and simplifies integration.
- Thread-Safe and Memory-Efficient Design: The Eclipse Paho libraries are meticulously designed to be thread-safe and memory-efficient, ensuring their suitability

for resource-constrained IoT devices. This optimization ensures optimal performance in resource-limited environments.

- Support for Advanced Features: The Eclipse Paho project extends beyond basic messaging by supporting advanced features, such as authentication, encryption, and QoS mechanisms. These features enhance security and reliability in IoT applications.
- Extensive Documentation and Community Support: The Eclipse Paho project provides comprehensive documentation and a thriving community of developers who are eager to assist and answer questions. This support ensures that developers can effectively leverage the project's capabilities.

Figure 5.2 resumes all the Paho implementation and their functionalities with respect the used programming language.



Client	MQTT 3.1	MQTT 3.1.1	MQTT 5.0	LWT	SSL / TLS	Automatic Reconnect	Offline Buffering	Message Persistence	WebSocket Support	Standard MQTT Support	Blocking API	Non-Blocking API	High Availability
Java	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Python	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗
JavaScript	✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓
GoLang	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
C	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C++	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Rust	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
.Net (C#)	✓	✓	✗	✓	✓	✗	✗	✗	✗	✓	✗	✓	✗
Android Service	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Embedded C/C++	✓	✓	✗	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗

Fig. 5.2. Paho MQTT client comparison [8]

The Eclipse Paho project offers a comprehensive and technically robust solution for IoT developers seeking reliable and efficient communication between devices and applications. Its cross-platform compatibility, open-source nature, and advanced features make it an indispensable tool for building cutting-edge IoT solutions. The project's commitment to quality, security, and community support ensures its continued relevance and value in the ever-evolving IoT landscape. The code is available at Eclipse repository on GitHub [94].

The VO developed in this thesis implements a Paho MQTTAsyncClient [95]. In contrast to the conventional synchronous client, this particular client employs a multi-threaded mode, allowing for concurrent execution of multiple operations. This design does not only markedly reduce execution times, but it also mitigates potential service queue congestion associated with heightened traffic on the interface.

5.2.5 Docker

Docker is a containerization platform that enables developers to package applications and their dependencies into lightweight, portable containers. These containers can run consistently across different environments, from a developer's laptop to a testing environment or a production server. Docker containers encapsulate the application, libraries, and other dependencies, ensuring that the application runs reliably and consistently regardless of the environment. Docker is often used in conjunction with microservices to containerize individual services. Each microservice can be packaged as a Docker container, providing consistency in deployment and execution. This combination allows for easier management of dependencies, simplified scaling, and improved portability across various environments. Many organizations adopt Docker and microservices to achieve greater agility, scalability, and maintainability in their software development and deployment processes.

In the orchestration of numerous instances of the VO, Docker played a pivotal role, facilitating the simultaneous activation of multiple VOs and thereby ensuring enhanced scalability and flexibility. Notably, Docker's utility extended beyond the VO, encompassing the initiation and management of instances of InfluxDB crucial for real-time data handling and storage as more comprehensively explained in the dedicated paragraph. This approach contributed to establishing a coherent environment wherein all components, spanning from the VO to the databases, were uniformly managed. Such uniformity aimed at reducing complexities and mitigating potential points of failure in our research endeavors.

An example of VO container instantiation steps for a virtualized environment are described below and in Figure 5.3:

1. The repository requests the OSS to instantiate elementary VOs.
2. The OSS asks the VIM to install a VM (with pre-installed Docker) for each physical device for which a virtual counterpart is desired.
3. The NFVO requests the VIM to perform the specialization of generic VMs into elementary VOs, including the addition of Docker compose files related to the machinery (interfaces, etc.).
4. Upon completion of the specialization, the OSS communicates the address of each VO to the VO registry.

Listing 5.1. Docker compose example

```
ADD file:e36038a1f6ee02f3f9a7db183e116f58d277e97cb7e71032634097d8
    02654d02 in /
```

```
CMD ["bash"]
 EXPOSE 22
 EXPOSE 8080
 ENV TZ=Europe DEBIAN_FRONTEND=noninteractive
/bin/sh -c sed -i 's/# \(.*multiverse$\)/\1/g' /etc/apt/sources.
    list &&   apt-get update &&   apt-get -y upgrade &&   apt-get
     install -y build-essential &&   apt-get install -y software-
    properties-common &&   apt-get install -y byobu curl git
    htop man unzip vim wget &&   apt-get install -y default-jre
    &&   apt-get install -y tzdata &&   mkdir /root/vo &&   mkdir
     /root/vo/config
ADD file:65ae4c05cec1c7d15ec0e082504be9bfad7fd4ac57efc1ba179cb675
    6cab3e68 in /root/.bashrc
ADD file:ada32699a865dcddf0e1c5b8f3dc780b9dbef40411e9d6903388caf8
    446d4cef in /root/.gitconfig
ADD dir:171c000765d21f266d9f846756413539b23aff1a669c39ab56464744c
    6499123 in /root/.scripts
ADD file:65e25b47f5774cb9c311adfe1e91b0c4ae86d0c857040f7018415aae
    8983cc1a in /root/vo/vo-1.0-SNAPSHOT.jar
 ENV HOME=/root/vo
WORKDIR /root/vo
 ENTRYPOINT ["java" "-jar" "vo-1.0-SNAPSHOT.jar"]
```

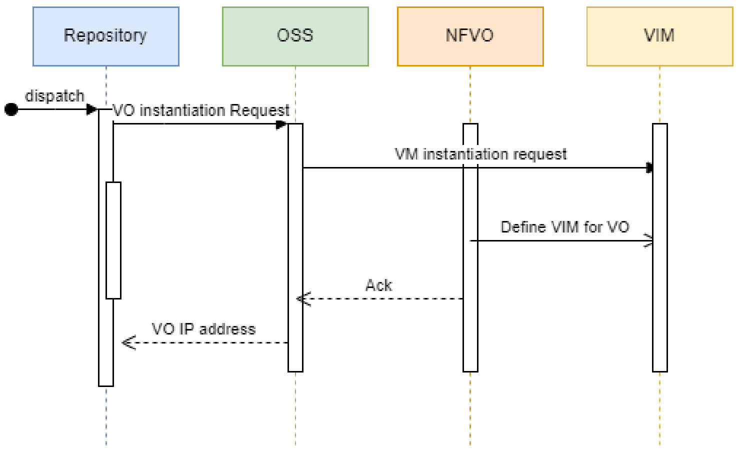


Fig. 5.3. VO instantiation procedure

5.3 VO Architecture implementation

The milieu enveloping a VO is intricate, comprising diverse components crucial for its seamless operation and integration within the broader computing continuum. In this section, we delve into these key components, elucidating their individual roles and functionalities.

The VO acts as a *man-in-the-middle* between the physical device and all consumers (applications or other VOs/cVOs) that want to exchange data with the object (i.e device, car, machine, etc...). The VO is located between the services and the resources to act as an advanced physical device from the perspective of the resource consumers, that is, the services. From the point of view of the physical device, on the other hand, the VO is the server enabled to receive messages, updates, manage the resources exposed by the sensors in the device and the device itself. The advanced features of the VO allow it, for example, to use multiple communication protocols (MQTT, HTTP, etc.).

The VO, in essence, is conceived as a microservice constituted by:

- Southbound Interfaces, to physical devices;
- Northbound interfaces, towards consumer applications and cVOs;
- 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 VO Container, and interfaces to a non-relational datastore (InfluxDB) deployed external, but close, to the VO container for high data rate resource values.
- Backend logic core for processing and enhancing functionality.

Figure 5.4 depicts the distinct layers that constitute the implemented VO. The levels of the implemented VO reflect the levels of the VOStack architecture. The fact that the implemented VO is fully compatible with the VOStack architecture means that it can be easily integrated with other VOs and services that are also based on the VOStack architecture.

The architecture of the VO has been designed in a modular fashion, with functions of the VO exposed through interfaces, APIs. This design aims to achieve greater scalability and extensibility of the code for future VO enhancements, as detailed in the following sections.

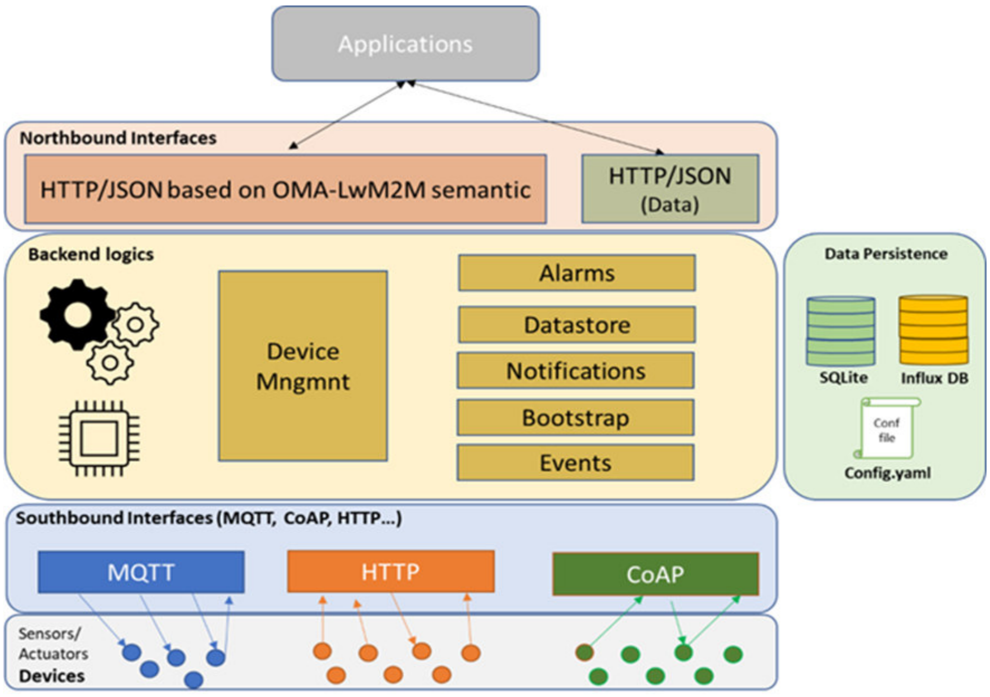


Fig. 5.4. VO architecture

5.4 Datastore

IoT devices by definition generate a large amount of data, creating *Big data*. Big data are generally understood as large collections of data, characterized by a volume, speed, and variety that require the use of specific technologies and analytical methods to extract value or knowledge [96].

Over time, IoT devices have contributed to the exponential growth of data stored in the cloud, posing challenges to the network, particularly due to the high volume of small-sized data packets transmitted across the entire infrastructure. The virtualization of IoT devices addresses these issues by attempting to shift the burden of storage and transfer to the edge. This implementation aims to prevent packets from continuously traversing the entire network and allows for the edge-based storage of time series data with broader temporal windows than those feasible on physical devices. This data can then be consumed by applications, including cloud applications, at different times and intervals, providing a more efficient and flexible approach.

The implemented VO is proficient in the retention of diverse data types associated with its operational state, encompassing information pertaining to its core identity as a device, supplementary extended objects, and other enduring data with protracted update intervals, spanning duration such as months or years. Simultaneously, the

VO preserves data related to monitored parameters characterized by notably shorter lifespans, typically on the order of seconds or milliseconds.

Considering the heterogenity of data to be stored, the VO implements two different data-storage with different purpose. A *SQLlite* [97] database is implemented as VO container data volume to efficiently manages long-term data, primarily written during the start-up phase while VO is modelling itself with respect the physical device, without concurrent operations. An *InfluxDB* [98] instance is implemented for high-frequency (up to 20Hz) data management. The interaction between the two datastore is described in the schema of Figure 5.5.

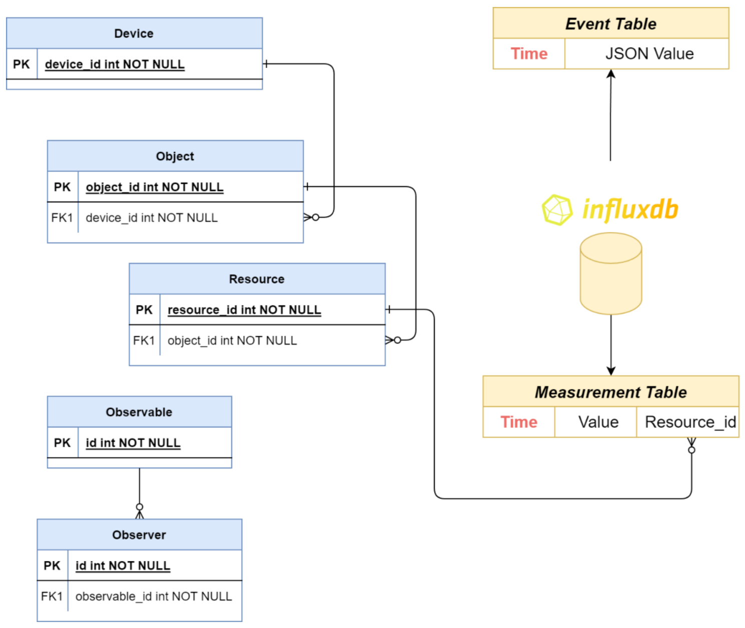

Device
PK device_id int NOT NULL
Object
PK object_id int NOT NULL
FK1 device_id int NOT NULL
Resource
PK resource_id int NOT NULL
FK1 object_id int NOT NULL
Observable
PK id int NOT NULL
Observer
PK id int NOT NULL
FK1 observable_id int NOT NULL
Event Table
Time
JSON Value
influxdb
Measurement Table
Time
Value
Resource_id

Fig. 5.5. Datastore Schema

5.4.1 SQLite

SQLite has been the first datastore implemented in the VO, and it is a self-contained, serverless, and zero-configuration relational database management system, which has garnered significant attention in academic circles for its lightweight yet robust characteristics. Designed for embedded systems and scenarios with low to moderate database demands, SQLite stands out for its simplicity, efficiency, and ease of integration.

The tables implemented in this database are related to device description according to the LwM2M models. Those tables are:

1. Device: this table contains the descriptive attributes of the device such as VO name (endpointname), lwm2m version, registrationId, etc. These data are ac-

quired from a descriptor file injected in the VO at start-up. Specifically, the columns are:

a) id: specifies the primary key of the entity and it is automatically generated;
b) additional registration attributes: (optional) for future development;
c) address: IP address and port exposed for services;
d) binding mode: indicates the interface type used for the device communication;
e) created at: timestamp when the VO is created;
f) endpoint: the unique name of the device;
g) lifetime: lifetime- keep-alive (not used at the moment);
h) lwm2m version: version of the LwM2M protocol used;
i) registration endpoint address: not used at the moment;
j) registration id: unique identifier of the registration;
k) resource type: type of semantics used (i.e. OMA-LwM2M);
l) root path: path for the exposed resources;
m) secure: [TRUE/FALSE] enable/disable the security protocol;
n) sms number: SIM number of the device;
o) updated at: timestamp of the last update.

2. Object: contains the LwM2M objects contained by the VO and their attributes as defined by OMA-LwM2M. The table is linked with *Observable* table and contains the following columns:
   a) id: check observable;
   b) name: check observable;
   c) created at: timestamp when the object is added to the VO system;
   d) description: small description of the object;
   e) instance id: different object can be instantiate multiple times and all of those instances are stored in the object table. This varchar indicates the object instance;
   f) mandatory: Mandatory (must be supported by all LwM2M Client implementations) or Optional (may not be supported);
   g) object id: the object id following the OMA Lwm2m registry;
   h) updated at: last timestamp in which the VO received an update from the device;
   i) device id: the device id where the object is built in;
3. Observable: they are the entities (instance or resource) placed under observation by one or more observers. The Northbound interface of the VO gives the possibility through an API to get notified for each update that an object or a resource has.

This capability to be observed gives the object the attributes *observable*. So when an application sends an observe request for a given object or resource, the object or the resource is stored in the table observable. This table stores the object created by the class that extends *observable*:

   a) id: specifies the primary key of the entity and it's automatically generated;
   b) name: specifies the name of the observable entity.

4. Observer: The observers are the objects (application, other VOs, cVOs) that observe. In other words, the observer table is the list of the consumers that have requested the observation on a resource or an object instance. The observer will be notified for each resource or object instance update.
   a) id: uniqueID of the observer;
   b) address: address where the observer response has to be sent. This address is sent in the payload of the observe request. In the case of null payload, this address is equals to the request sender;
   c) created at: timestamp when the observer is instantiated;
   d) name: not used at the moment;
   e) one shot: this is used when southbound interface does not implement ReST protocols. For instance, MQTT is not a Request/response protocol. So that, this flag is used to enable an Async Request/Response behaviour into northbound interface, HTTP, and the external application (Request) making one shot observable the required resource until the southbound, MQTT, get the new data;
   f) updated at: timestamp of the last update;
   g) observable id: uniqueID of the object for which the observer want to receive notifications.
5. Resource: stores the list of resources made available by enabled OMA-LwM2M objects. It contains:
   a) id: uniqueID of the resource;
   b) name: name of the represented resource (ie. Sensor Value, Min Measured Value);
   c) created at: timestamp when the resource is created;
   d) description: brief description of the resource;
   e) instance type: check standard Oma-lwm2m;
   f) mandatory: check standard OMA-LwM2M. It defines if the resource is mandatory to be present in the LwM2M object model

g) operations: shows the type of the operations that can be applied to the resource: NONE, R(read), W(write), RW, E(executable), RE, WE, RWE;
h) range enumeration: the range reachable of the resource value;
i) resource id: OMA registry uniqueID of the resource;
j) type: indicates the type of resource (STRING, INTEGER, FLOAT, BOOLEAN, OPAQUE, TIME, OBJLNK);
k) units: unit of measurement of resource;
l) updated at: timestamp of the last update;
m) object id: Object to which the resource refers.

5.4.2 InfluxDB

InfluxDB is an open-source Time Series Database (TSDB) designed to efficiently store and analyze high-volume, high-velocity time-stamped data. It is widely used for monitoring and analyzing IoT data, as well as for building real-time applications in various domains, including finance, manufacturing, and healthcare.

Key Features of InfluxDB are:

- Scalability: InfluxDB is horizontally scalable, meaning it can be deployed across multiple servers to handle large volumes of data. This makes it suitable for storing and analyzing data from large-scale IoT deployments.
- Real-time Performance: InfluxDB is optimized for real-time data ingestion and analysis. It can ingest millions of data points per second and provide near-real-time data access. This makes it ideal for building applications that need to react to changes in data in real time.
- Query Language: InfluxDB supports a powerful query language called InfluxQL, which enables users to easily query, filter, and analyze time series data. InfluxQL is based on SQL, making it familiar to many users.

Moreover, InfluxDB integrates seamlessly with Grafana [99] [100], a popular data visualization tool. This allows users to create interactive dashboards and charts to visualize their data. The Event table stores all the event received by the VO and it is implemented like an InfluxDB table and has two columns:

1. timestamp: timestamp when the VO is notified by the device;
2. JSON: given the fact the VO-LwM2M communicates in the Southbound using JSON, each event arrives in JSON format and it is saved in row in this table.

The Measurement table is used to store the values notified by the device, for each resource. It is a InfluxDB table and it has 4 columns:

1. timestamp: denotes the timestamp of the data entry;
2. value: holds the actual measurement value. For convenience, this attribute is further divided into various variables based on the value type, such as ”valueString”, ”valueInteger”, and so on. However, only one of these variables will be instantiated and populated with the measurement’s value;
3. resource id: represents the LwM2M resource receiving updates;
4. type: specifies the kind of value being received.

5.5 Interfaces

The VO represents a service designed to communicate with a diverse range of devices foremost and services secondarily. To achieve this objective, the implemented VO is capable of utilizing prevalent IoT protocols such as CoAP and MQTT, as well as the foundational web protocol, HTTP. In particular, two sets of interfaces have been implemented, dedicated respectively to communications with physical devices (*Southbound interfaces*), and communication with other VOs, cVOs, and VOStack prosumers (*Northbound interfaces*). Additionally, a distinct group of interfaces is designated for communication with the respective database(s) to perform specific data extraction functions.

5.5.1 Southbound Interfaces

This section of the service is specifically allocated to the interfaces responsible for engaging in communication with physical devices for both data acquisition and device management purposes. From a spectrum of IoT protocols scrutinized for the instantiation of these interfaces, the decision was made to implement the CoAP, MQTT/TCP, and HTTP protocols, as elucidated earlier in the document. Irrespective of the selected protocol, adherence to the LwM2M standard is maintained, necessitating that the payload consistently conforms to the OMA-LwM2M model, utilizing the JSON format.

Communication between physical devices and VOs is facilitated through interfaces utilizing the URI path format ObjectID/ObjectInstanceID/ResourceID, corresponding to the owned objects declared by the physical device. The interfaces adhering to the standard encompass the following operations:

- READ
- WRITE
- EXECUTE

- OBSERVE
- DELETE OBSERVE

Both CoAP and HTTP, being ReST-based protocols, easily map resources addressing using the URI path. However, this mapping is not as straightforward for the Publish/Subscribe MQTT protocol, where resources are addressed using topics. Over time, LwM2M has evolved to also allow use with the Publisher/Subscriber MQTT type protocol. This evolution, to simplify, provides that the MQTT message contains all the information that allows the receiver to reconstruct the type of REST packet that would have been transmitted using the CoAP protocol. The research work carried out made it possible to create and improve the integration of semantic models and the use of the primitives envisaged by the standard for the MQTT protocol, the result of which is summarized in the Tables 5.1, 5.2, and 5.3.

OMA-LWM2M	CoAP		
Primitives	Method	API path (Requests)	payload
READ	GET	/objID/inst.ID/resID	No
WRITE	PUT	/objID/inst.ID/resID	Yes
EXECUTE	POST	/objID/inst.ID/resID	No
OBSERVE	GET	/objID/inst.ID/resID/?observe=0	No
DELETE Observ	GET	/objID/inst.ID/resID/?observe=1	No

Table 5.1. CoAP-OMA LwM2M VO binding

OMA-LWM2M	MQTT		
Primitives	Method	Topic path (Requests like)	payload
READ	cmnd	deviceID/objId/instId/resId	No
WRITE	cmnd	deviceID/objId/instId/resId	Yes
EXECUTE	cmnd	deviceID/objId/instId/resId	Yes
OBSERVE	cmnd	deviceID/objId/instId/resId/observe	Yes (on)
DELETE Observ	cmnd	deviceID/objId/instId/resId	Yes (off)

Table 5.2. MQTT-OMA LwM2M VO Request binding

In this implementation, a distinction will be made between:

- Topic: Resource identifier (e.g., /deviceID/objId/instId/resId/);
- FullTopic: Comprising a prefix (e.g., cmnd).

OMA-LWM2M	MQTT		
Primitives	Method	Topic path (Response like)	payload
READ	stat	deviceID/objId/instId/resId	Yes
WRITE	stat	deviceID/objId/instId/resId	Yes
EXECUTE	stat	deviceID/objId/instId/resId	Yes
OBSERVE	stat	deviceID/objId/instId/resId/observe	Yes
OBSERVE	tele	deviceID/objId/instId/resId/	Yes
DELETE Observ	stat	deviceID/objId/instId/resId	Yes

Table 5.3. MQTT-OMA LwM2M VO response binding

Topics adhere to the standard modeling of the LwM2M protocol, which assigns a unique URI to each resource. For example, the topic for the value resource 5700 of instance 0 of the temperature object 3303 of the hot water device B01 will be constructed as follows: B01/3303/0/5700.

Full Topics, in addition to the topic, incorporate a prefix (e.g., cmnd). These prefixes are implemented to prevent the creation of potential loops between MQTT topics. Within this implementation, three distinct prefixes will be utilized:

- *cmnd*: Prefix for issuing commands or requesting status updates
- *stat*: Reports the status or configuration messages
- *tele*: Reports telemetry information at specified intervals.

The solution requires that the MQTT message continues to contain the same JSON format present in the payload provided by the OMA-LWM2M standard for the CoAP protocol. This solution has proven to be particularly advantageous in achieving levels of interoperability. In particular, the VO leverages on using the same JSON, it allows a simpler integration of the two IoT protocols, CoAP and MQTT, avoiding duplicating the backend functions that manage the data.

Comparison between CoAP and MQTT

MQTT and CoAP, integral communication protocols in the IoT, exhibit notable distinctions in their fundamental characteristics. MQTT functions as a many-to-many communication protocol, facilitating the exchange of messages among multiple clients through a centralized broker. Noteworthy for decoupling publishers and subscribers, MQTT relies on the broker to forward messages. While MQTT incorporates support for persistence, its optimal performance is observed when serving as a real-time data communication channel. Conversely, CoAP primarily operates as a one-to-one protocol, governing the transfer of information between clients and servers. Despite incorporating functionality for resource observation, CoAP finds greater applicability in a state transfer model, diverging from a paradigm exclusively reliant on events. A critical differentiator lies in the metadata support for message comprehension. MQTT lacks inherent mechanisms for labeling messages with types or other metadata, necessitating a prior understanding of message formats for effective communication among clients. In contrast, CoAP provides built-in support for content negotiation and discovery, enabling devices to mutually explore and establish efficient data exchange methodologies. The selection between MQTT and CoAP is contingent upon the specific requirements and nuances of the application at hand, each protocol presenting its unique advantages and considerations in the context of IoT communication.

5.5.2 Northbound Interfaces

The VO exposes these interfaces to interact with other VOs, cVOs, applications, and/or services. The implementation leverages the RESTful protocol, specifically the HTTP. This interface level draws inspiration from the methods employed in the OMA LwM2M protocol to facilitate device management and information notification operations. The specifications for each interface (endpoint) are succinctly outlined later in the document, encompassing operations such as READ, READ Realtime, WRITE, EXECUTE, and OBSERVE.

READ. The READ operation is initiated when an application requests information from the VO. This information can pertain to:

- General details about the VO;
- Information originating from an object within the VO;
- Information from an instance of an object;
- Information from a specific resource.

In the case of information from an LwM2M object, the data provided encompasses all instances of the requested object. The VO can handle two distinct types of READ requests. The first is a standard READ request (READ), which returns the last stored value by the VO. The second is the real-time request (READ Realtime), elaborated upon below, which is forwarded to the physical device to obtain an updated value. The choice between these READ requests depends on the specific requirements of the application.

API Details:

- 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 desired level of information, it is imperative to specify the corresponding URI path. For instance, if the requested information is the value of the resource 5700 of the instance 0 of the temperature object 3303, the URI would be the following: *http://(VOnameOrIPAddr)/api/clients/deviceId/3303/0/5700.*

WRITE. The WRITE operation is initiated when an application submits a WRITE request to the VO with the intention of writing a single value or multiple values to a specified resource or instance. The VO subsequently forwards the request to the device and retains the provided values. This functionality is applicable to both resources and instances.

Write Interface Details:

- 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"]
  - Resource: "id":"resID","value":"+01"

Example for LwM2M *ObjecId 3* and *InstanceID 0* (uripath: /3/0/):

Listing 5.2. Write LwM2M Instance example

```
{
  "id": "0",
  "resources": [
    {"id": 14, "value": "+01"},
    {"id": 15, "value": "Europe/Reggio Calabria"}
  ]
}
```

Example for LwM2M Resource 14 (uripath: /3/0/14):

Listing 5.3. Write LwM2M Instance example

```
{"id": 14, "value": "+01"}
```

This specification provides a comprehensive definition of the WRITE interface, encompassing the relevant HTTP method, interface path, resource path, parameters, and payload structure for both instances and resources.

EXECUTE VO processes an execution request. The Execute operation serves to initiate or trigger specific actions and is exclusively applicable to a single resource. The interface specifications are outlined below:

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

OBSERVE

The VO receives an OBSERVE request from an application, intending to monitor a resource, an object instance, or an entire object. Upon initiation of observation, the observer is enlisted in a dedicated list. This list functions as a reference for the VO to notify the application of any new incoming values associated with the observed entity. Consequently, the addition of a new observer for the same entity occurs seamlessly, bypassing the necessity of forwarding new OBSERVE requests through the southbound interface and subsequently to the device.

Interface Details:

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

The response to the OBSERVE request takes the following format:

Resource

Listing 5.4. Observe LwM2M Resource example

```
{
  "event": "NOTIFICATION",
  "data": {
    "ep": "VOid",
    "res": "/3303/0/5700",
    "val": {"id": 5700, "value": 55}
  }
}
```

Instance

Listing 5.5. Observe LwM2M Resource example

```
{
  "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"}
      ]
    }
  }
}
```

This format encapsulates the pertinent information associated with the observed entity, facilitating efficient communication between the VO and the application.

To facilitate asynchronous data exchange between the VO and cVO services, a specific observe operation interface has been implemented. In particular, the cVO has the ability to subscribe to all resources within a specific VO through a single HTTP POST request, as detailed below:

- HTTP Method: POST
- Interface: api/cvo/register
- Port: 8080
- Parameters (optional):
- Content-Type: application/json
- Payload (optional): Observer's IP address

Note that even using VO inside orchestrable environment, the VO APIs are unable to read the real IP of the querying services because the VO is shielded, for instance, inside container cluster. So,Observer has to declare its APIs IP inside request payload explicitly. Moreover, this option can be useful if consumer, the observer, would use separated IPs for async receiving interfaces.

Anyhow, the response to both types of Observe requests is sent to the same IP address that made the Observe request or to the one declared in the request's payload, through an HTTP connection with the following parameters:

- HTTP Method: POST
- Address: Requester's IP
- Port: 8080
- Interface: api/notify
- Payload (example in the case of object 3304, Humidity):

Listing 5.6. Observe LwM2M Resource response example

```
{"event":"NOTIFICATION","data":{"ep":"D003","res":"/3304/0/5700","val":{"id":"5700","value":"44","timestamp":"2023-10-12 14:34:55.103"}}}
```

Listing 5.7. Observe LwM2M Resource response example

```
{"event":"NOTIFICATION","data":{"ep":"D003","res":"/3304/0","val":{"id":0,"resources":[{"id":5601,"value":30},{"id":5602,"value":60},{"id":5700,"value":33},{"id":5701,"value":"%"}]}}
```

DELETE

This functionality is employed to cancel a previously initiated OBSERVE. The interface specifications are outlined below:

- HTTP Method: DELETE
- Interface: api/clients
- Resource Path: /deviceId/objectID/InstanceID/ResourceID(opt.)/observe
- Parameters: null
- Payload: (optional) Observer IP address

5.5.3 Datastore Interfaces

The VO has been implemented to provide enriched functionalities with respect the physical device. In this context, particularly concerning data management, the VO exposes dedicated APIs on the endpoint *api/data*. These APIs enable data aggregation and data extraction operations, as more specifically described below.

Data extraction by value

This interface enables the search for a specific LwM2M resource value, only resource, within the logged data. For instance, it enables the search for the value (resId=5700) of the humidity sensor (objectId=3304) attained a specific value (val=40). Additionally, this request allows the selection of the type of operator to be employed in the search for the matching criteria: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

Listing 5.8. Datastore API Request data example using cURL libraries

```
curl --location 'http://192.168.35.125:8080/api/data/D004/3304/0/
    5700/value?value=40&operator=1'
```

Extraction of the last n recorded values The interface enables the extraction of a finite set of values (n) from a singular 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 15 values, in chronological order, recorded on that resource:

Listing 5.9. Datastore API Request of number of samples example using cURL libraries

```
curl --location 'http://192.168.35.125:8080/api/data/D003/3304/0/
    5700/limit?limit=15'
```

Data extraction over a period of time This interface permits consumer applications to retrieve a historical record of data for various purposes, such as constructing specific diagrams embedded in dedicated dashboards. The dates should be provided in the *SimpleData* format[3], as elaborated upon later in the document.

The API endpoint is defined as:

- HTTP method: GET
- Interface: api/data

[3] https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html

- 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 60 second time interval, form 2023-04-13 11:34:43 to 2023-04-13 11:35:43:

Listing 5.10. Datastore API Request of period data example using cURL libraries

```
curl --location 'http://192.168.33.42:8080/api/data/D004/3304/0/5
    700/date?startDate=2023-04-13%2011%3A34%3A43&endDate=2023-04-
    13%2011%3A35%3A43'
```

5.6 Backend logics

The backend, on the other hand, is where the application's business logic, data processing, and storage take place. Backend logics involve the algorithms, processes, and rules that govern how data is processed, how business rules are enforced, and how the application functions behind the scenes.

In the VO logics are implemented in order to provide a set of functions supported by both the "IoT Device Virtualized Functions" and the "Generic/Supportive Functions" layers within the VOStack. The layer dedicated to IoT Device Virtualized Functions focuses on handling specific aspects of an application's business logic. In contrast, the Generic/Supportive Functions Layer encompasses a suite of functions that can be broadly applied across all instantiated Virtual Objects (VOs) for an application.

At the Generic/Supportive Functions Layer, we adopt a service mesh approach, where the development of generic supportive functions, including but not limited to elasticity management, and telemetry, is the focal point. Telemetry functions, alongside elasticity management actions for IoT application components, are provided. When representing an IoT device, a VO offers the supported IoT Virtualized functions to the application, while other supportive functions can be activated as part of the application graph.

The choice and specification of IoT virtualized functions and supportive features are primarily guided by use cases, covering a wide range of IoT devices, applications, and services. Different functions must be accommodated to cater to the unique aspects of each use case. While certain IoT devices and functionalities are specialized for a specific use case, others share commonalities across various scenarios. For instance, ground robots and ultrasound probes are specific to post-disaster and e-Health use cases, while environmental sensors are applicable in several use cases. In this section generic function implemented in this VO will be described, taking into account that more functionalities can be provided by external services, named *sidecar* when strictly interconnected to VOs, using VO interfaces and specific use-case OMA-LwM2M models.

The set of generic functions applicable across all instantiated VOs within an application encompasses several critical aspects. These functions contribute to the overall robustness, efficiency, and reliability of the application. Here is an overview of these functions:

- Telemetry and Monitoring Components: These components are designed to continually monitor and assess various aspects, including network connectivity, sensor status, and the operational status of robots. They play a crucial role in maintaining situational awareness and can trigger predefined actions in response to events such as low energy levels or disconnections.
- Storage and Replay Software Component: This function involves the development of a software component dedicated to storing and replaying historical data related to robot actions and tasks. It facilitates in-depth analysis of past events, providing valuable insights into the performance and behavior of the application over time.
- Data Aggregation: Data aggregation involves the consolidation and summarization of data from multiple sources. In the context of the application, this function ensures that relevant data is collected and processed efficiently, contributing to informed decision-making and actionable insights.
- Elasticity Management for VO Deployment: Elasticity management is crucial for adapting the infrastructure resources dynamically to changing workloads. This function enables the deployment of VOs in a Kubernetes Cluster, utilizing Custom Resource Definitions (CRDs) and predefined rules for horizontal and vertical scaling. It ensures optimal resource utilization and responsiveness to varying demands.
- Alarms: The alarms function involves the exposure of thresholds and associated alarm resources. This allows consumers to monitor trends in resource behavior, and when predefined thresholds are reached, notifications about anomalous trends are triggered. Alarms contribute to proactive monitoring and timely responses to potential issues.

In summary, these generic/supportive functions form a comprehensive toolkit that enhances the overall performance, and adaptability of the application, making it well-equipped to handle diverse scenarios and challenges.

Following subsection describes functionalities more in detail.

5.6.1 Telemetry

By harnessing the Observe primitive outlined in the OMA-LwM2M standard, both the VO and cVO implement a efficient telemetry mechanism. This mechanism is designed to autonomously communicate changes in the status of a resource to the data consumer, often referred to as the Observer. In the initiation phase, the VO, stationed at the northbound interfaces, eagerly awaits an OBSERVE request from an application. This request serves as an explicit expression of interest in observing a specific LwM2M Resource or Object instance. Upon receipt of such a request, the VO promptly responds by registering the observer in a dedicated list. This list is meticulously curated for entity observation, accommodating entries for both Resources and Instances. Subsequently, when an entity, be it a Resource or an Instance, is under observation and undergoes a change in its value, the VO consults the registered observe list. This list becomes the conduit through which the VO notifies the observer application about the updated value from the observed entity. This telemetry notification process ensures that the Observer stays abreast of any alterations in the status of the observed entity.

A noteworthy facet of this mechanism lies in its adaptability. If a new observer expresses a desire to monitor the same entity, the VO seamlessly integrates the new observer into the existing list. This dynamic addition of observers occurs without the need for intricate procedural interventions.

In essence, this telemetry mechanism operates with a keen focus on efficiency and responsiveness. Its dynamic nature facilitates the continual addition of new observers, enriching the real-time monitoring capabilities of the VO. This approach aligns with the broader goal of ensuring that Observers receive timely and comprehensive updates on the evolving status of the observed entities within the IoT ecosystem.

5.6.2 Data storage

As described in previous sections, the VO is endowed with the capability to accommodate diverse data storage strategies, a versatility dictated by the inherent nature of the data it handles. This adaptive approach to data storage manifests in the utilization of both internal and external datastores, each serving distinct purposes.

Internally, the VO leverages lightweight solutions, exemplified by the implementation of SQLite. This choice is particularly well-suited for historical data, where frequent temporal updates are less prevalent. SQLite provides an efficient and streamlined mechanism for historical data storage, ensuring that the VO's operations related to less dynamically changing information remain optimized. Conversely, for scenarios demanding a more robust and high-performance data processing infrastructure, the VO turns to external data storage solutions. Here, the focus shifts to handling substantial volumes of time-related data, especially those characterized by high update frequencies. In this context, InfluxDB emerges as a prime example of an external datastore that aligns with the demands of managing large quantities of time-series data.

This dualistic approach to data storage, with a judicious selection between internal and external solutions, underscores the VO's adaptability to the diverse requirements of data management. The choice of the data storage strategy is intricately linked to the nature of the data at hand and deployment environment, ensuring optimal performance and resource utilization. The scalable and stateless configuration of these data storage mechanisms is strategically aligned with the VO's status. This configuration is pivotal for maintaining seamlessness in orchestration and facilitating elasticity management within a VOStack. The ability to dynamically scale and adapt to evolving workloads is integral to the VO's role in the larger ecosystem, contributing to its resilience and efficiency in handling diverse data processing scenarios.

5.6.3 Data aggregation

Within the NorthBound interfaces, the cVO extends accessibility to historical data through dedicated interfaces, seamlessly facilitating direct retrieval of recorded data within a specified timeframe. These interfaces offer a versatile set of functionalities designed to cater to diverse data extraction requirements.

In essence, these interfaces establish a robust foundation for interacting with historical data, offering a rich set of extraction options. Whether the need is to retrieve data by type, value, or within a specified timeframe, the VO's and cVO's NorthBound interfaces provide a comprehensive suite of tools for users to navigate and extract meaningful insights from the stored historical data.

5.6.4 Elasticity Management

The VO, instantiated into a virtualized VOStack, can be elasticity managed by orchestrator in order to jointly provide *Elasticity Management* functionalities like:

- Constant connectivity monitoring to guarantee a reliable and resilient connection, enhancing the overall robustness of the IoT ecosystem;
- VO resilience to discontinuous connected physical device: The VO autonomously manages such situations to uphold data provisioning and integrity;
- Resource scalability: Thanks to VO containerization, the orchestrator can continuously monitor the resources allocated to the VO microservice and optimize their allocation according to needs and, possibly, also carry out migration actions thereof.

It allows VOs to operate effectively, even in dynamic and challenging scenarios, contributing to the reliability and continuity of IoT applications.

5.6.5 Alarms

By leveraging OMA-LwM2M semantic model OMA-LwM2M, the VO proficiently exposes thresholds and associated alarm resources. This strategic utilization allows consumers to effectively monitor resource trends and receive timely notifications upon detection of anomalous conditions. To facilitate manageable alarm systems, OMA-LwM2M objects are structured to incorporate three specific resources within their models:

- Alarm State (ID 6013): This resource serves as a binary indicator, representing the True/False status of the alarm. It provides a clear signal of whether the alarm is currently active (True) or inactive (False).
- Alarm Set Threshold (ID 6014): Operating as a dynamic parameter, this resource is instrumental in establishing the threshold level for activating the alarm. Notably, it supports both reading (READ) and writing (WRITE) operations, providing flexibility for users to configure and adjust the threshold as needed.
- Alarm Set Operator (ID 6015): This resource, both readable and writable, plays a crucial role in conjunction with the Set Threshold. It determines the triggering conditions for the alarm and must be set to one of the following values:
  - Greater than or equal to: The alarm triggers when the sensor value is greater than or equal to the specified threshold.
  - Less than or equal to: The alarm triggers when the sensor value is less than or equal to the specified threshold.

This structured approach to alarm management within OMA-LwM2M objects ensures a robust and customizable system. By incorporating these key resources in LwM2M models, the VO enhances its ability to facilitate alarm monitoring and response, contributing to a comprehensive and adaptive IoT environment.

5.7 Descriptor file

Upon instantiation, a VO is accompanied by an indispensable configuration file, in YAML format, which is parsed at start-up of Spring Boot service. It is injected dynamically into docker container by orchestrator with the name of *application-registration.yaml*, depending on physical counterpart device, and is read at start-up by Spring. This configuration file serves as a repository for setup information essential for the proper functioning of the VO. It includes details about the corresponding physical device, such as its address, communication protocol, version, sensors, and IP address. The assignment of the IP address depends on the context and it can, for instance, be directly allocated by the infrastructure manager and dynamically inserted into the configuration file during the creation of the VO. In addition to information about the physical device, the configuration file holds various other crucial data points. For instance, in OMA-based VOs, it includes OMA object IDs, and in cases where the VO utilizes MQTT, the file contains the MQTT broker's address. These details are instrumental in managing the components that constitute the VO stack.

Moreover, the *Descriptor* file plays a pivotal role in orchestrating the seamless integration and operation of the VO within the broader system. It ensures that the VO can effectively communicate and interact based on its predefined settings. By encapsulating essential parameters, this file acts as a guiding document for the VO, facilitating its coherent participation in the overall system architecture.

The descriptor file is the key that allows the VO microservice to adapt to any IoT object, which, described through the OMA-LwM2M semantics, can be composed of numerous LwM2M Objects representing both the hardware and software components of the device. An example of a descriptor file is shown in the following listing:

Listing 5.11. Descriptor file example for a Brewery filter

```yaml
vo:
  device:
    endpoint: D002 #unique name of physical device
    registrationId: ul9mXXFFF #registration ID
    address: 127.0.0.1:8080 #IP address:port of VO container
    Version: 1.1 #VO version
    lifetime: 30 #lifetime/keep-alive
    bindingMode: M #southound interface (H=HTTP; M=MQTT; U=COAP/
       UDP;)
    rootPath: / #URI path root
    resourceType: oma.lwm2m #semantic type
    secure: false # True/false
    additionalRegistrationAttributes:
  objectLinks: #lists of LwM2M object instances
    - 26243/0
    - 26243/1
    - 3330/0
    - 3306/0
    - 3306/1
    - 3306/2
    - 3306/3
  url: tcp://192.168.2.11:1883 #server IP (i.e.  MQTT broker)
  cleanSession: true # MQTT option
  username: #MQTT usr
  password: #MQTT pssw
  mqttQos: 0  #MQTT QoS
```

5.8 Coding

As described in previous section, the implementation of VO codebase relies on the Spring Boot framework, a key component within the broader Spring ecosystem. A notable advantage of Spring Boot is its automatic configuration based on the project's libraries, eliminating the need for explicit bean specifications in configuration files. The architecture within the Spring Boot framework is structured into layers, with the *Service* and *Controller* layers playing crucial roles (Figure 5.6).

The Service Layer handles the business logic of the application, interacting with the data access layer and executing CRUD operations: Create, Read, Update, and Delete. Classes designated as services, marked with the *@Service* annotation, are injected into controllers or other services, following the principle of inversion of control and maintaining a clear separation of concerns.

On the other hand, the Controller Layer, marked with *@Controller* or *@RestController*, acts as an intermediary between the model and the view. It processes client requests, leverages services, and returns the appropriate view or data. In the context of ReSTful web services, the controller manages and responds to incoming HTTP requests, often delivering JSON or XML data.

To ensure a robust and maintainable implementation aligned with Object-Oriented Programming (OOP) principles, a layered approach was adopted. This involved, for instance, defining a general interface for the *ClientService*, creating an abstract class encapsulating shared code between MQTT and CoAP services, and developing concrete classes, *ClientServiceM* and *ClientServiceU*, which extend the abstract class. This approach ensures a modular, maintainable, and reusable codebase.

The organization outlined above is maintained through the Spring Boot *@ConditionalOnProperty* annotation, allowing the conditional registration of beans based on configuration property presence. The configuration file, named *Descriptor* determines the communication protocol utilized by the VO during instantiation using bindingMode parameter: M (MQTT), or U (UDP, or H (HTTP).

For MQTT implementation, the *MqClient* class is employed, managed by ClientServiceM, a Spring Boot service created when the VO is instantiated in MQTT mode. This service handles Controller calls using the MQTT *Paho Async Client* library, enabling concurrent operations during connection or subscription. Additionally, the Eclipse Leshan project contributes ready-to-use classes for implementing an LwM2M CoAP server. Specific classes from the Leshan project are imported and modified to align with the VO's requirements for seamless integration.

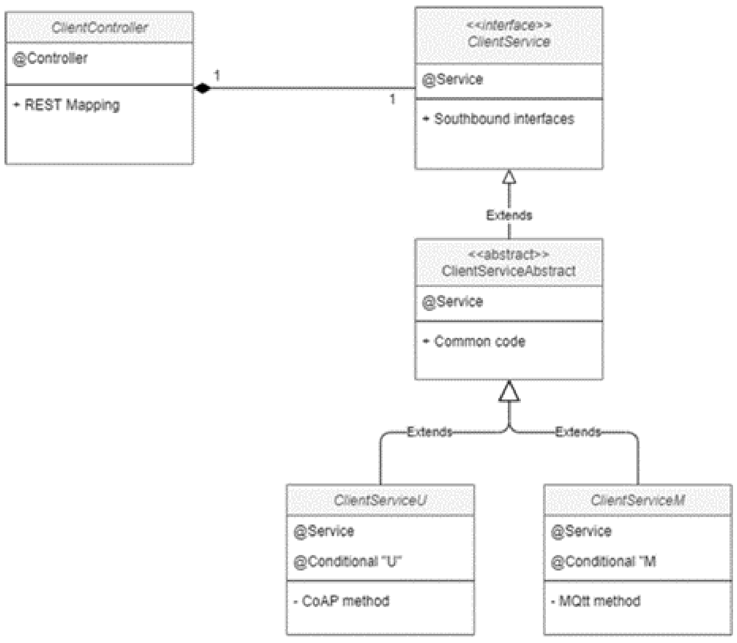


Fig. 5.6. Class diagram describing ClientController and ClientService relationship

The VO code, as well as cVO code, is open-source, constantly evolving, and accessible into Eclipse Git repository[4].

5.9 The composite VO

A cVO represents a fusion of semantically interoperable VOs and delivers services aligned with user perspectives and application requirements. Utilizing discovery mechanisms, the cVO facilitates the reuse of both existing VOs and cVOs across diverse applications, potentially extending their utility beyond their original development context [101].

The implemented cVO inherits the majority of functionalities from the VO. Consequently, during the development process, the decision was made to keep both entities integrated within the same project. In practice, the VO and cVO share the same codebase, enabling real-time selection between them through the use of a dedicated configuration file, the Descriptor.

5.9.1 Architecture

The cVO architecture depicted in Figure 5.7, shows the differences with respect VO. In particular, it is clear that IoT application protocol interfaces are not implemented at *Southbound interface* because cVO is not directly involved in communication with the physical device.

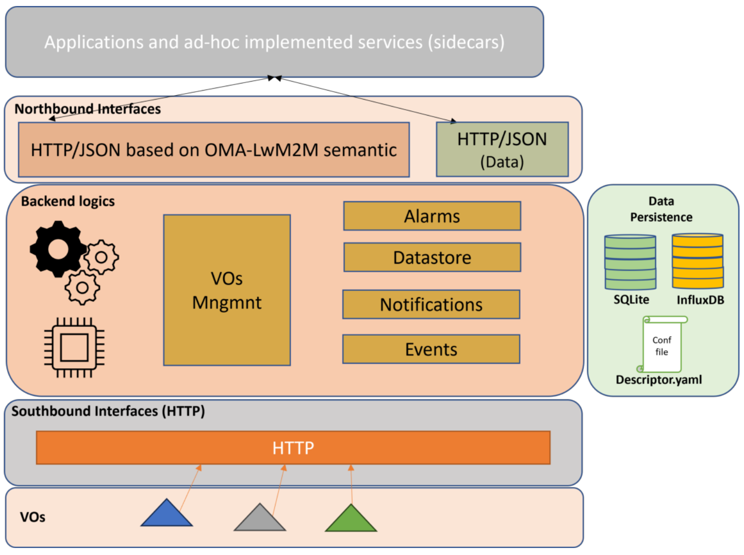


Fig. 5.7. cVO architecture

[4] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-lwm2m

Moreover, northbound interfaces are oriented to communication with enhanced services and applications.

5.9.2 Container instantiation

Unlike VO, which is primarily conceived as a native VNF of the infrastructure to integrate IoT devices, the cVO is usually instantiated to serve a specific service/application restricted to a particular use case. For this reason, the process of creating the cVO within the virtualized environment differs slightly from that of the VO and is initiated by specific inputs from the "consumer" application. Usually, an application can request the creation of a cVO, which is the aggregation of multiple elementary VOs and outputs from other applications. The request process unfolds as follows (Figure 5.8):

1. The application requests the creation of a cVO and communicates its specifications.
2. The VO registry forwards the cVO creation request to the OSS.
3. The OSS requests the VIM to install a VM with pre-installed Docker.
4. The NFVO requests the VIM to specialize the generic VM by transmitting the Docker compose file related to the cVO.
5. The OSS informs the VO registry of the successful creation and enables access.
6. The VO registry provides the application with the address to interact with the cVO.

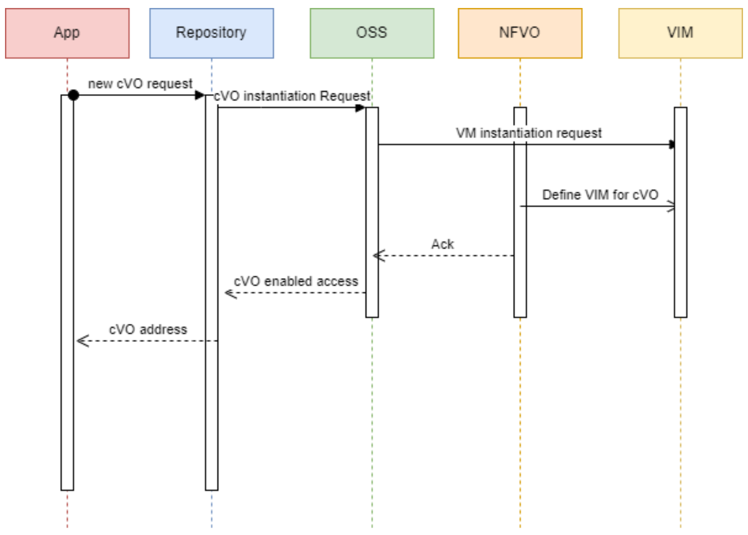


Fig. 5.8. Example of cVo instantiation in virtualized environment

5.9.3 Interfaces

The communication protocol employed among the VOs and cVO is intentionally crafted to be user-transparent, leveraging on *Notify* mechanism. Essentially, when there is a fresh resource update originating from the southbound side, the corresponding VO promptly updates cVO. Subsequently, this composite object assumes the crucial task of updating the northbound application, thereby ensuring real-time data synchronization and minimizing latency. The articulated architecture, characterized by a distinct allocation of roles and responsibilities, guarantees scalability, adaptability, and efficient management of real-time data emanating from diverse sources.

The cVO subscribes as an Observer to all the resources made available by the VO through the use of its dedicated interface, called *CvoService*. The interface functions similarly to the *Observer interface* normally provided by the VO. Therefore, the IP address of the cVO is added as an observer for all observable resources within the VO.

5.9.4 Descriptor file

The configuration file of the cVO is designed to indicate which VOs will be part of the service chain for which the cVO has been instantiated. Unlike the VO, the cVO currently does not have the ability to expose its services outside the HTTP protocol.

Listing 5.12. Descriptor file example for a cVO

```
composite-vo:
  endpoint: deviceID_CVO #unique name of physical device
  registrationId: u67sffvkkkk #iregistration ID
  address: 127.0.0.1:8080 #ip:port of cVO container
  Version: 1.0 #code version
  lifetime: 30 #lifetime/keep-alive
  bindingMode: H #HTTP is the only available interface for cVO
  rootPath: / # rott path of exposed resources
  resourceType: oma.lwm2m #semantic type
  secure: false #
  additionalRegistrationAttributes:
  devices: #list of VOs to be connected to
    - id: D001 #VO unique name
      ip: 192.168.5.12:8080 #VO ip:port address
    - id: D002
      ip: 192.168.5.13:8080
    - id: D003
```

```
      ip: 192.168.5.14:8080
```

It is likely that in the near future, the cVO will also be able to expose interfaces based on different protocols, such as NGSI-LD for semantic-level communication or Kafka[5] and others at the application level, to implement integrated edge services.

[5] "https://kafka.apache.org/documentation/#design"

6

Proof of concept

This chapter presents some VOs and cVOs implementation placed in different contexts of national research and innovation projects. The first one has been developed in a Smart factory environment for beer production; the second deals with a Mobility as a Service (MaaS) scenario promoting a platform for urban integrated mobility services.

6.1 A virtualized ICT Platform for Industry 4.0: eBrewery

The project *eBrewery* (PON 2014-2020, code ARS 0100582) aimed to conceive a platform for the Smart Factory based on virtualization and slicing concepts, which were subjects of investigation in different domains (e.g., cellular systems) and not yet, at that time, sufficiently explored in an industrial context. This made the project highly innovative, as it envisioned an ICT platform with fully virtualized functions and devices, creating virtual slices based on the same HW platform, each specialized to support specific applications related to industrial processes. In particular, the project focused on creating a virtualized network infrastructure with two levels of orchestration. The first level is associated with the application domain, and the second is linked to the IoT domain (Figure 6.1).

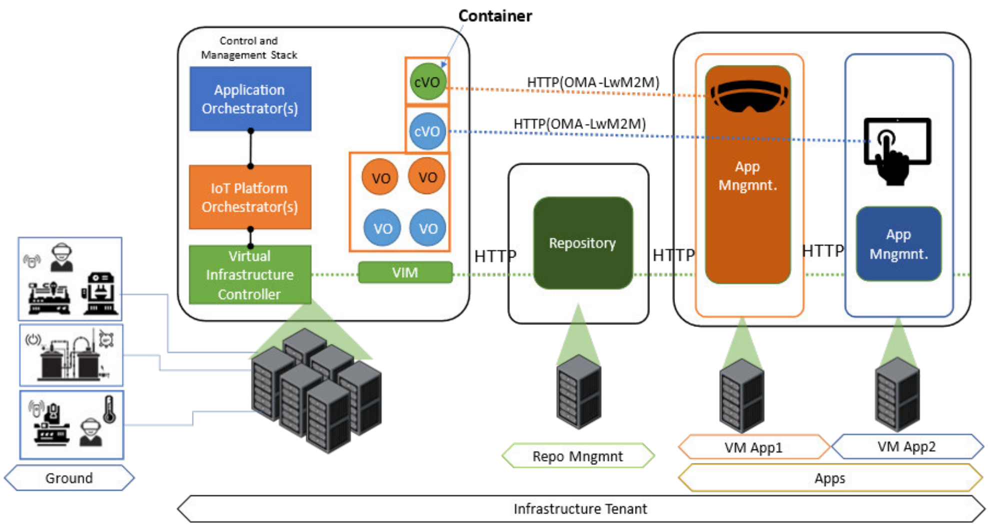


Fig. 6.1. eBrewery Infrastructure architecture

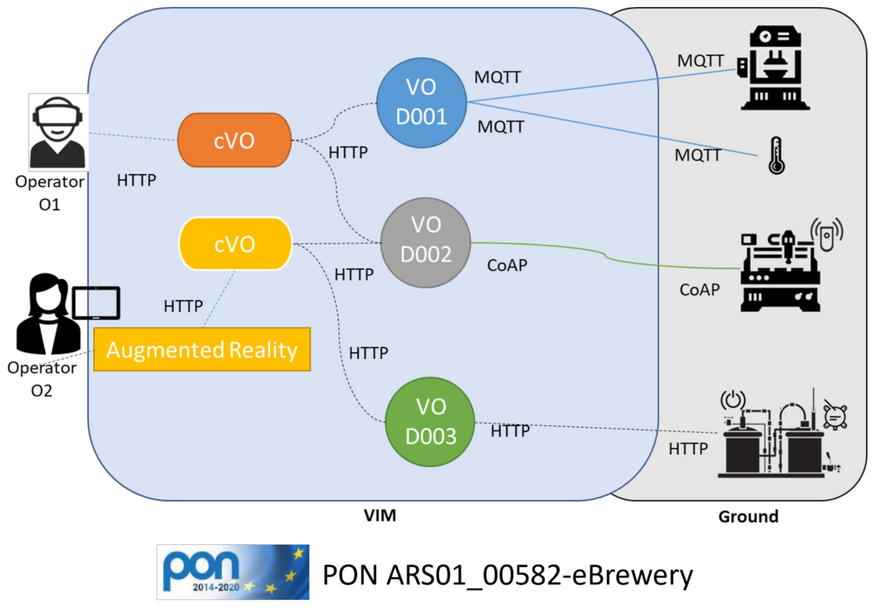


Fig. 6.2. Ebrewery Architecture of IoT domain

Devices have been virtualized through the implementation of VOs and cVOs specifically designed by developing custom semantic models following the OMA-LwM2M standards. Three distinct VOs and two different composite VOs were developed, one for each application used in the specific use case (Figure 6.2).

As shown in Figure 6.2, a cVO is an aggregator of VOs which provides to applications a single point of access to the information the app need. It can be used, for instance, when an app like Augmented reality(AR) needs to interact with several VOs with several IP addresses.

Listing 6.1. Masher VO

```
vo:
  device:
    endpoint: D001 #device ID
    registrationId: ul9mXXFQQ #registration ID
    address: 127.0.0.1:8080 #ip:port container
    Version: 1.1 #version
    lifetime: 30 #lifetime keep-alive
    bindingMode: M #binding(H=HTTP; M=MQTT; U=COAP/UDP;)
    rootPath: / #path
    resourceType: oma.lwm2m #semantic
    secure: false #sicurezza True/false
    additionalRegistrationAttributes:
  objectLinks: # objectinstance
    - 3303/0 # Temperature object 1
    - 3303/1  # Temperature object 1
    - 3303/2  # Temperature object 1
    - 3326/0    # PH
    - 26242/0   # FAN
    - 26243/0   #Grado plato
    - 3306/0    #Actuator heating band 1
    - 3306/1    #Actuator heating band 2
    - 3306/2    #Actuator heating band 3
    - 3306/3    #Actuator masher shaker
    - 3306/4    #Actuator recycler
    - 3306/5    #Actuator valve ejection
  url: tcp://172.143.5.3:1883 # broker MQTT
  cleanSession: true #MQTT
  username: #MQTT usr
  password: #MQTT pssw
  mqttQos: 0  #MQTT QoS
```

The three different VOs were a masher (Listing 6.1), a filter (Listing6.2), and a fermenter (Listing 6.3). Each one has been defined by a differente decriptor file.

Data collected by VOs and cVO were then displayed by two different applications of Augmented Reality (AR) designed to (i) support the monitoring and management of the production process in the field in a semi-artisanal manner (Figure 6.3) and (ii) develop a predictive maintenance platform for the plant.

Listing 6.2. Filter VO

```
vo:
  device:
    endpoint: D002 #device ID
    registrationId: ul9mVV34RTz #registration ID
    address: 127.0.0.1:8080 #ip:port container
    Version: 1.1 #version
    lifetime: 30 #lifetime keep-alive
    bindingMode: M #binding(H=HTTP; M=MQTT; U=COAP/UDP;)
    rootPath: / #path
    resourceType: oma.lwm2m #semantic
    secure: false #sicurezza True/false
    additionalRegistrationAttributes:
  objectLinks: # objectinstance
    - 26243/0   #Grado Plato
    - 26243/1   #Grado Plato Boiler
    - 3330/0    #Turbidity
    - 3306/0    #Actuator Transfert form filter to Boiler
    - 3306/1    #Filter shaker
    - 3306/2    #Boiler shaker
    - 3306/3    #Pump
  url: tcp://172.143.5.3:1883 # broker MQTT
  cleanSession: true #MQTT
  username: #MQTT usr
  password: #MQTT pssw
  mqttQos: 0  #MQTT QoS
```

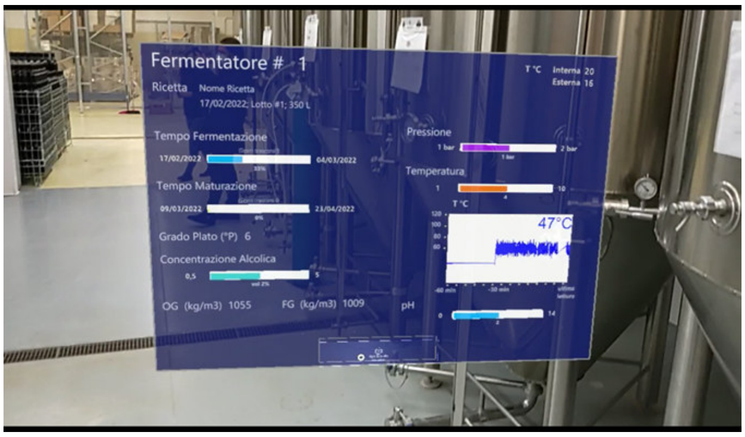


Fig. 6.3. eBrewery Mesher Panel

Listing 6.3. Fermenter VO

```
vo:
  device:
    endpoint: D003 #device ID
    registrationId: ul9wwwbolK #registration ID
    address: 127.0.0.1:8080 #ip:port container
    Version: 1.1 #version
    lifetime: 30 #lifetime keep-alive
    bindingMode: M #binding(H=HTTP; M=MQTT; U=COAP/UDP;)
    rootPath: / #path
    resourceType: oma.lwm2m #semantic
    secure: false #sicurezza True/false
    additionalRegistrationAttributes:
  objectLinks: # objectinstance
    - 26243/0 #grado plato
    - 3320/0    #alcohol concentration (%) with min threshold
    - 3320/1    #alcohol concentration (%) with max threshold
    - 3325/0    #OG
    - 3325/1    #FG
    - 3/0       #device (serial number)
    - 3323/0    #Pressure
    - 3303/0    #Temperature (external)
    - 3303/1    #Temperature (internal)
    - 3326/0    #PH
    - 3308/0    #Set point Temperature
    - 30000/0   # Production batch
  url: tcp://172.143.5.3:1883 # broker MQTT
  cleanSession: true #MQTT
  username: #MQTT usr
  password: #MQTT pssw
  mqttQos: 0  #MQTT QoS
```

6.2 Device Virtualization in MaaS environment

The scenario of mobility has certainly not been exempt from technological developments involving the IoT and telecommunication systems. On the contrary, taking advantage of these developments makes it possible to create new scenarios for sustainable mobility that can be provided as a scalable, customizable, and shared service.

Modern Mobility as a Service (MaaS) systems, in fact, extensively utilize solutions based on MEC technologies and IoT to improve the quality of life for citizens within increasingly smart areas, known as smart cities. The work proposed in [9], an evolution of the previous work [102], outlines the design of a framework for collecting and processing data produced by commuters and public transportation. In particular, the paper suggests the virtualization of physical devices for both public transport users, commuters, through the creation of VOs of their smartphones, and for public transportation through the creation of VOs for their respective On Board Units (OBUs). The VOs, or DTs, are hosted within an edge infrastructure according to ETSI-MEC standards (Figure 6.4).

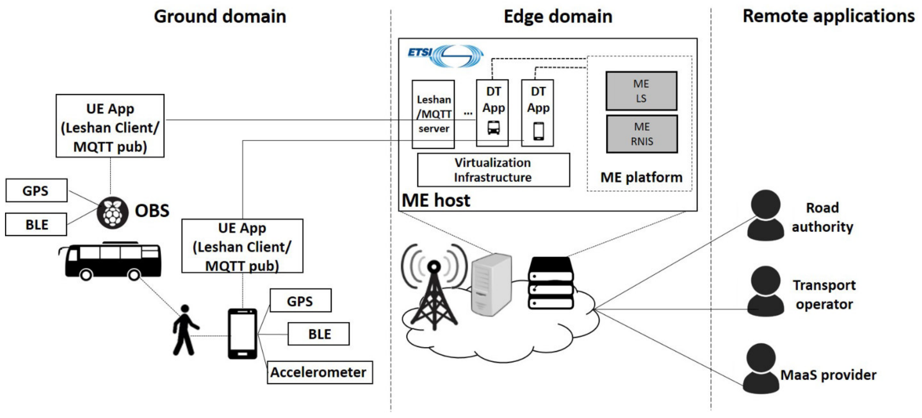


Fig. 6.4. MaaS scenario [9]

Furthermore, the work has considered the use of two of the most widely adopted application protocols for IoT, MQTT and CoAP, both associated with OMA-LwM2M. The presented Proof of Concept (PoC) confirmed the viability of the proposal and provided valuable insights into the effectiveness and efficiency of the messaging protocols for interactions between physical devices and their corresponding DTs, the VOs (Figure 6.5).

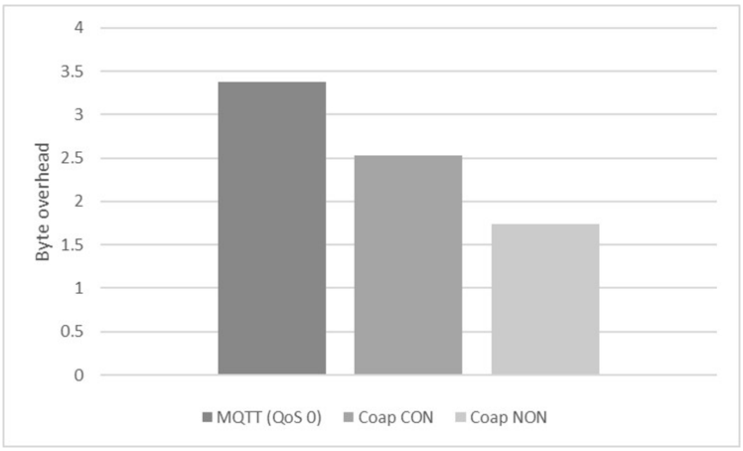


Fig. 6.5. MaaS: Byte overhead for the two compared messaging protocols. [9]

7

Open issues and Conclusions

In contemporary scenarios, Virtual Objects (VOs) have become integral components, demonstrating a capacity to acquire, analyze, and interpret contextual information. Their significance lies in bolstering the security of IoT devices and mitigating challenges related to solution interoperability. Notably, VOs facilitate the management of diverse standards and models, fostering collaboration between platforms and promoting the sharing of resources. The implemented solution of VOs and cVOs, as detailed in Chapter 5, seamlessly aligns with this context, offering various supplementary functionalities independently of the physical device. These functionalities encompass:

- Semantics: The VO employs a semantic standard, OMA-LwM2M, enabling the abstraction and control of any IoT device and the transmission of data in well-defined formats.
- Communication Protocols: The VO can utilize multiple protocols such as MQTT, CoAP, and HTTP for both communications with the physical device, maintaining semantic standards, and with applications, the "consumers."
- Energy Consumption of the Physical Device: It avoids unnecessary queries to the physical device by aggregating requests or responding on its behalf. Additionally, it allows the virtualization of functionalities not strictly related to the hardware.
- Enhancement of Functionalities: Leveraging resources available in virtualized environments, the VO implements advanced features without imposing a burden on the physical device, including Telemetry, Data Aggregation, Alarming, and Data Historization, even in the form of time series to facilitate the implementation of business intelligence logic.
- Scalability: The VO is implemented in the JAVA language and utilizes modern programming tools with the idea of providing expandable modular components accessible through interfaces.

- Orchestration: The VO is instantiable, hence orchestratable, within virtualized network infrastructures implementing SDN and NFV functionalities through the use of Docker and Docker Compose.

The research activities outlined in this thesis aimed to contribute towards the creation of a VOStack. The goal is to enable seamless interaction between the IoT ecosystem, with its distinctive technologies, and the computing continuum of the network infrastructure, spanning from edge to cloud. The implemented VO, contributing to the creation of a VOStack [3], extends the IoT device resources into the virtualized and hyper-connected and hyper-distributed computing continuum where the expansion potential is unlimited.

However, several issues still remain open including three main challenges:

- *Security*: In a hyper-connected infrastructure, all entities are particularly interdependent from a security perspective. The compromise of a single entity could disrupt the provision of all services and chains of services interconnected with it. Some proposals, such as Verificable Credentials (VCs) [103] and Decentralized Identifiers (DIDs) [104], have been suggested for distributed systems, while more traditional solutions can be employed for communications between physical devices and VOs, including Transport Layer Security (TLS) [105] and Object Security for Constrained RESTful Environments (OSCORE) [106].
- *Semantic interoperability*: this thesis work adopted a standard semantic description of IoT devices. However, the IoT landscape is extensive and to date significant efforts and various levels of interoperability are required to enable most, if not all, IoT objects and services operating within the continuum to communicate with each other seamlessly.
- *Continuum Orchestration*: the integration of IoT technologies with edge and cloud computing technologies, offering transparent deployment and orchestration solutions for IoT applications across the computing continuum ensuring lifecycle orchestration paradigms to efficiently leverage network resources and services.

Future work will be oriented towards addressing the remaining gaps in semantic interoperability and cooperation between services in edge computing, striving to propose increasingly integrated solutions in IoT environments and enhancements of VO generic and supportive functions. A brief list of functionalities to be addressed in future developments of the VO is:

- Enhance semantic interoperability by the introduction of further standard like WoT and NGSI-LD.

- Cognitive management: Cognitive mechanisms at the VO level to enable self-management and self-configuration of real objects. 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.
- Introducing new OMA-LwM2M standard functionalities like multi-objects request.
- Create Interface to integrate AI functionalities and services.

References

1. A. P. Castellani, S. Loreto, A. Rahman, T. Fossati, and E. Dijk, “Guidelines for Mapping Implementations: HTTP to the Constrained Application Protocol (CoAP).” RFC 8075, Feb. 2017.
2. OMA, “Lightweight machine to machine technical specification: Core,” *https://www.openmobilealliance.org/release/LightweightM2M/V1_2_1-20221209-A/OMA-TS-LightweightM2M_Core-V1_2_1-20221209-A.pdf*, 2022-12.
3. D. Spatharakis, I. Dimolitsas, G. Genovese, I. Tzanettis, N. Filinis, E. Fotopoulou, C. Vassilakis, A. Zafeiropoulos, A. Iera, A. Molinaro, *et al.*, “A lightweight software stack for iot interoperability within the computing continuum,” in *2023 19th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, pp. 715–722, IEEE, 2023.
4. Nephele EU Project, “High Level View of VO Usage,” *NepheleEUProject,https://nephele-project.eu/*, (2023).
5. Cetic, “The three levels of cloud,” *https://www.cetic.be/Optimize-your-costs-and-productivity-by-migrating-to-the-PaaS-use-case*, (2023).
6. R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, “Network function virtualization: State-of-the-art and research challenges,” *IEEE Communications surveys & tutorials*, vol. 18, no. 1, pp. 236–262, 2015.
7. L. "Rahaman, “"tutorial on leshan framework”,” *https://www.win.tue.nl/~lrahman/iot/files/LeshanTutorial*, (Accessed 2023.
8. Eclpse Foundation, “Paho project,” *https://wiki.eclipse.org/Paho*, (Accessed 2023.
9. C. Campolo, G. Genovese, A. Molinaro, and B. Pizzimenti, “Digital twins at the edge to track mobility for maas applications,” in *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pp. 1–6, IEEE, 2020.
10. V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate, “A survey on application layer protocols for the internet of things,” *Transaction on IoT and Cloud computing*, vol. 3, no. 1, pp. 11–17, 2015.
11. 5G Americas, “5G-The future of IoT, 5G Americas White Paper (2019),” *https://www.5gamericas.org/5g-the-future-of-iot/*, Accessed 2023.

12. GSMA, "Network slicing - use case requirements, gsma (2018)," *https://www.gsma.com/futurenetworks/wp-content/uploads/2018/04/NS-Final.pdf*.
13. J.-M. Fernandez, I. Vidal, and F. Valera, "Enabling the orchestration of iot slices through edge and cloud microservice platforms," *Sensors*, vol. 19, no. 13, p. 2980, 2019.
14. High Performance, Edge And Cloud computing (HiPEAC), "From hpc/cloud to edge/iot: A major paradigm shift for europe," 2023.
15. C Campolo, G Genovese, A Molinaro, B Pizzimenti, G Ruggeri, D Zappala, "An edge-based digital twin framework for connected and autonomous vehicles: Design and evaluation, computer networks," *Computer Networks, Elsevier, COMNET-D-23-01617 (Submitted)*, 2023.
16. C. Campolo, G. Genovese, A. Iera, and A. Molinaro, "Virtualizing ai at the distributed edge towards intelligent iot applications," *Journal of sensor and actuator networks*, vol. 10, no. 1, p. 13, 2021.
17. L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.
18. A. J. Jara, A. C. Olivieri, Y. Bocchi, M. Jung, W. Kastner, and A. F. Skarmeta, "Semantic web of things: an analysis of the application semantics for the iot moving towards the iot convergence," *International Journal of Web and Grid Services*, vol. 10, no. 2-3, pp. 244–272, 2014.
19. D. Arellanes and K.-K. Lau, "Evaluating iot service composition mechanisms for the scalability of iot systems," *Future Generation Computer Systems*, vol. 108, pp. 827–848, 2020.
20. M. Condoluci, M. A. Lema, T. Mahmoodi, and M. Dohler, "5g iot industry verticals and network requirements," in *Powering the internet of things with 5G networks*, pp. 148–175, IGI Global, 2018.
21. M. Simsek, A. Aijaz, M. Dohler, J. Sachs, and G. Fettweis, "5g-enabled tactile internet," *IEEE Journal on selected areas in communications*, vol. 34, no. 3, pp. 460–473, 2016.
22. C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Ca4iot: Context awareness for internet of things," in *2012 IEEE International Conference on Green Computing and Communications*, pp. 775–782, IEEE, 2012.
23. A. Botta, W. De Donato, V. Persico, and A. Pescapé, "Integration of cloud computing and internet of things: a survey," *Future generation computer systems*, vol. 56, pp. 684–700, 2016.
24. Cisco, "Cisco annual internet report," *https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf*, (2020).
25. J. P. Espada, O. S. Martínez, B. C. P. G. Bustelo, and J. M. C. Lovelle, "Virtual objects on the internet of things," *IJIMAI*, vol. 1, no. 4, pp. 23–29, 2011.
26. P. K. Verma, R. Verma, A. Prakash, A. Agrawal, K. Naik, R. Tripathi, M. Alsabaan, T. Khalifa, T. Abdelkader, and A. Abogharaf, "Machine-to-machine (m2m) communica-

tions: A survey," *Journal of Network and Computer Applications*, vol. 66, pp. 83–105, 2016.

27. 3GPP, "3gpp-service requirements for machine-type communications (mtc)," *https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=645*.
28. ETSI, "ETSI (2013) TS 102 690," *https://www.etsi.org/deliver/etsi_ts/102600_102699/102690/01.02.01_60/ts_102690v010201p.pdf*.
29. OneM2M, "oneM2M TS-0001-Functional Architecture," *https://member.onem2m.org/Application/documentApp/documentinfo/default.aspx?documentId=35400*, 2023.
30. IETF, "Internet standards," *https://www.ietf.org/standards/*, 2023.
31. OASIS, "Standards," *https://www.oasis-open.org/standards/*, 2023.
32. Open Mobile Alliance, "Oma technical specifications," *https://technical.openmobilealliance.org/index.html*, 2023.
33. OMA Spec Works, "Oma specwork homepage," *https://omaspecworks.org/about/*, 2023.
34. W3C Concortium , "W3c home page," *https://www.w3.org/mission/*, (2023).
35. IETF, "Concise binary object representation (cbor)," *https://datatracker.ietf.org/doc/html/rfc8949*, (Accessed 2023.
36. Internet Engineering Task Force (IETF), "The JavaScript Object Notation (JSON) Data Interchange Format," *https://www.rfc-editor.org/rfc/rfc8259*.
37. OASIS, "Mqtt: The standard for iot messaging,"
38. A. Banks and R. Gupta, "MQTT Version 3.1. 1," *OASIS standard*, vol. 29, p. 89, 2014.
39. C. M. Ramya, M. Shanmugaraj, and R. Prabakaran, "Study on zigbee technology," in *2011 3rd international conference on electronics computer technology*, vol. 6, pp. 297–301, IEEE, 2011.
40. C. Bisdikian, "An overview of the bluetooth wireless technology," *IEEE Communications magazine*, vol. 39, no. 12, pp. 86–94, 2001.
41. OMA-LwM2M, "Lightweight machine to machine registry," *http://www.openmobilealliance.org/wp/OMNA/LwM2M/LwM2MRegistry.html*, 2022-12.
42. ETSIGS,GS CIM 009 V1.7.1 , "Ngsi-ld api. context information management (cim)," *âĂœhttps://www.etsi.org/deliver/etsi_gs/CIM/001_099/009/01.07.01_60/gs_CIM009v010701p.pdf*, (2023-06).
43. Fiware, "Step-by-Step for NGSI-LD," *https://ngsi-ld-tutorials.readthedocs.io/en/latest/index.html*, (2023-05).
44. Fiware, "Orion-LD: Context Broker and CEF building block for context data management which supports both the NGSI-LD and the NGSI-v2 APIs.," *https://github.com/FIWARE/context.Orion-LD*, (2023-05).
45. Fiware, "Smart Data Models," *https://smartdatamodels.org/*, (2023-05).

46. I. Alam, K. Sharif, F. Li, Z. Latif, M. M. Karim, B. Nour, S. Biswas, and Y. Wang, “Iot virtualization: a survey of software definition & function virtualization techniques for internet of things,” *arXiv preprint arXiv:1902.10910*, 2019.
47. 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.
48. R. Giaffreda, “iCore: A cognitive management framework for the Internet of Things,” in *The Future Internet Assembly*, pp. 350–352, Springer, 2013.
49. M. Weyrich and C. Ebert, “Reference architectures for the internet of things,” *IEEE Software*, vol. 33, no. 1, pp. 112–116, 2015.
50. Q. Fan and N. Ansari, “On cost aware cloudlet placement for mobile edge computing,” *IEEE/CAA Journal of Automatica Sinica*, vol. 6, no. 4, pp. 926–937, 2019.
51. X. Sun and N. Ansari, “Edgeiot: Mobile edge computing for the internet of things,” *IEEE Communications Magazine*, vol. 54, no. 12, pp. 22–29, 2016.
52. O. Chukhno, N. Chukhno, G. Araniti, C. Campolo, A. Iera, and A. Molinaro, “Optimal placement of social digital twins in edge iot networks,” *Sensors*, vol. 20, no. 21, p. 6181, 2020.
53. S. Y. Jang, Y. Lee, B. Shin, and D. Lee, “Application-aware iot camera virtualization for video analytics edge computing,” in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 132–144, IEEE, 2018.
54. H. Kokkonen, L. Lovén, N. H. Motlagh, A. Kumar, J. Partala, T. Nguyen, V. C. Pujol, P. Kostakos, T. Leppänen, A. González-Gil, *et al.*, “Autonomy and intelligence in the computing continuum: Challenges, enablers, and future directions for orchestration,” *arXiv preprint arXiv:2205.01423*, 2022.
55. Savaglio C, Ganzha M, Paprzycki M, Bădică C, Ivanović M, Fortino G, “Agent-based internet of things: State-of-the-art and research challenges,” *Future Generation Computer Systems*, vol. 102, pp. 1038–1053, 2020.
56. M. Segovia and J. Garcia-Alfaro, “Design, modeling and implementation of digital twins,” *Sensors*, vol. 22, no. 14, p. 5396, 2022.
57. L. Atzori, J. L. Bellido, R. Bolla, G. Genovese, A. Iera, A. Jara, C. Lombardo, and G. Morabito, “Sdn&nfv contribution to iot objects virtualization,” *Computer Networks*, vol. 149, pp. 200–212, 2019.
58. M. A. Jarwar, S. Ali, and I. Chong, “Microservices model to enhance the availability of data for buildings energy efficiency management services,” *Energies*, vol. 12, no. 3, p. 360, 2019.
59. ETSI, “Network function virtualization: Architectural framework,” *http://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.01.01_60/gs_NFV002v010101p.pdf, 2013.* , (2013).
60. D. Raggett, “Compose: An open source cloud-based scalable iot services platform,” *ERCIM News*, vol. 101, pp. 30–31, 2015.

61. S. Alam, M. M. Chowdhury, and J. Noll, "Senaas: An event-driven sensor virtualization approach for internet of things cloud," in *2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications*, pp. 1–6, IEEE, 2010.

62. M. Presser, P. M. Barnaghi, M. Eurich, and C. Villalonga, "The sensei project: Integrating the physical world with the digital world of the network of the future," *IEEE Communications Magazine*, vol. 47, no. 4, pp. 1–4, 2009.

63. M. Sanchez, E. Exposito, and J. Aguilar, "Implementing self-* autonomic properties in self-coordinated manufacturing processes for the industry 4.0 context," *Computers in industry*, vol. 121, p. 103247, 2020.

64. D. Szabó, F. Németh, B. Sonkoly, A. Gulyás, and F. H. Fitzek, "Towards the 5g revolution: A software defined network architecture exploiting network coding as a service," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 105–106, 2015.

65. D. Soldani and A. Manzalini, "Horizon 2020 and beyond: On the 5g operating system for a true digital society," *IEEE Vehicular Technology Magazine*, vol. 10, no. 1, pp. 32–42, 2015.

66. Openstack, "The openstack project," *https://www.openstack.org/*, (2023).

67. A. Vogel, D. Griebler, C. A. Maron, C. Schepke, and L. G. Fernandes, "Private iaas clouds: a comparative analysis of opennebula, cloudstack and openstack," in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 672–679, IEEE, 2016.

68. ETSI, "ETSI Network Function Virtualization," *www.https://www.etsi.org/technologies/nfv*, (2023).

69. W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pp. 171–172, IEEE, 2015.

70. Docker, "Docker compose," *https://docs.docker.com/compose/startup-order/*, (2023).

71. Chiosi, M., et al., "Network functions virtualization: An introduction, benefits, enablers, challenges & call for action," *ETSI White Paper, Oct. 2012.*

72. ETSI ISG, "ETSI ISG web portal," *https://portal.etsi.org/tb.aspx?tbid=789&SubTB=789,795,796,801,800,798,799,797,802.*

73. W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A survey on software-defined networking," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 27–51, 2014.

74. D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014.

75. Open Networking Foundation, "ONF openflow," *https://opennetworking.org/sdn-resources/customer-case-studies/openflow/.*

76. ETSI OSM, "Open source mano," *https://osm.etsi.org/wikipub/index.php/Welcome_to_OSM.*

77. I. Markit, “The 5G Economy: How 5G technology will contribute to the global economy,” *https://www.ihs.com/Info/0117/5g-technology-global-economy.html*, 2023.
78. Next Generation Mobile Networks (NGMN) Alliance, “5g white paper,” *https://www.ngmn.org/publications/5g-white-paper-2.html*, 2023.
79. 3GPP, “Study on management and orchestration of network slicing for next generation network,” *TR 28.801*, 2018.
80. S. Thalanany and P. Hedman, “Description of network slicing concept,” *NGMN Alliance*, 2016.
81. ETSI GS MEC 002 2016, “Mobile Edge Computing (MEC)-Technical Requirements,” *http://www.etsi.org/deliver/etsi_gs/MEC/001_099/002/01.01.01_60/gs_MEC002v010101p.pdf*, 2023.
82. 5G PPP Architecture Working Group, “View on 5G Architecture,” *https://5g-ppp.eu/wp-content/uploads/2017/07/5G-PPP-5G-Architecture-White-Paper-2-Summer-2017_For-Public-Consultation.pdf*, 2023.
83. ETSI Multi-access Edge Computing Working Group, “ETSI Multi-access Edge Computing,” *http://www.etsi.org/technologies-clusters/technologies/multi-access-edge-computing*, 2023.
84. The OpenStack KingBird project, “The openstack kingbird project,” *https://wiki.openstack.org/wiki/Kingbird*, 2023.
85. Tricircle, “The OpenStack Wiki,” *https://wiki.openstack.org/wiki/Tricircle*, 2023.
86. S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang, “A survey on mobile edge networks: Convergence of computing, caching and communications,” *IEEE Access*, vol. 5, pp. 6757–6779, 2017.
87. M. Maray and J. Shuja, “Computation offloading in mobile cloud computing and mobile edge computing: survey, taxonomy, and open issues,” *Mobile Information Systems*, vol. 2022, 2022.
88. B. Mccullough, “Apache Maven,” *Syntax*, pp. 1–6, 2009.
89. Apache Software foundation, “Maven project Home page,” *https://maven.apache.org/*, (Accessed 2023.
90. Eclipse Foundation, “Eclipse IDE Home page,” *https://www.eclipse.org/ide/*, (Accessed 2023.
91. IntelliJ IDE, “Intellij ide home page,” *https://www.jetbrains.com/idea/*, (Accessed 2023.
92. Microsoft, “Visual studio code home page,” *https://code.visualstudio.com/*, (Accessed 2023.
93. Eclpse Foundation, “Californium,” *https://www.eclipse.org/californium/*, (Accessed 2023.
94. Eclpse Foundation, “Paho git project,” *https://github.com/eclipse/paho.mqtt.java*, (Accessed 2023.

95. Eclpse Foundation, “Paho mqtt async client,” *https://github.com/eclipse/paho.mqtt.java/blob/master/org.eclipse.paho.client.mqttv3/src/main/java/org/eclipse/paho/client/mqttv3/MqttAsyncClient.java*, (Accessed 2023.

96. A. De Mauro, M. Greco, and M. Grimaldi, “A formal definition of big data based on its essential features,” *Library review*, vol. 65, no. 3, pp. 122–135, 2016.

97. K. P. Gaffney, M. Prammer, L. Brasfield, D. R. Hipp, D. Kennedy, and J. M. Patel, “Sqlite: past, present, and future,” *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3535–3547, 2022.

98. J. Shahid, “Influxdb documentation,” 2019.

99. "Grafana", “"grafana web site ”,” *https://grafana.com/*, (Accessed 2023.

100. "Grafana", “"grafana, influx db datasource ”,” *https://grafana.com/docs/grafana/latest/datasources/influxdb/?src=ggl-s&mdm=cpc&cnt=145265838431&camp=nb-influxdb-bmm*, (Accessed 2023.

101. V. Foteinos, D. Kelaidonis, G. Poulios, V. Stavroulaki, P. Vlacheas, P. Demestichas, R. Giaffreda, A. R. Biswas, S. Menoret, G. Nguengang, *et al.*, *A cognitive management framework for empowering the internet of things.* Springer, 2013.

102. C. Campolo, D. Cuzzocrea, G. Genovese, A. Iera, and A. Molinaro, “An oma lightweight m2m-compliant mec framework to track multi-modal commuters for maas applications,” in *2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pp. 1–8, IEEE, 2019.

103. J. Glöckler, J. Sedlmeir, M. Frank, and G. Fridgen, “A systematic review of identity and access management requirements in enterprises and potential contributions of self-sovereign identity,” *Business & Information Systems Engineering*, pp. 1–20, 2023.

104. D. Reed, M. Sporny, D. Longley, C. Allen, R. Grant, M. Sabadello, and J. Holt, “Decentralized identifiers (dids) v1. 0,” *Draft Community Group Report*, 2020.

105. H. Mrabet, S. Belguith, A. Alhomoud, and A. Jemai, “A survey of iot security based on a layered architecture of sensing and data analysis,” *Sensors*, vol. 20, no. 13, p. 3625, 2020.

106. G. Selander, J. Mattsson, F. Palombini, and L. Seitz, “Object security for constrained restful environments (oscore),” tech. rep., 2019.