ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA

DIPARTIMENTO di
INGEGNERIA DELL'ENERGIA ELETTRICA E DELL'INFORMAZIONE
"Guglielmo Marconi"
DEI

MASTER'S DEGREE
IN
TELECOMMUNICATIONS ENGINEERING

---

*[*9205*]*

Master Thesis
in
Laboratory of Networking M

Orchestration of a MEC-based multi-protocol IoT environment

Candidate
BASSI LORENZO

Supervisor
Prof. WALTER CERRONI

Co-supervisor
DAVIDE BORSATTI

Academic Year: 2020/2021

Session II

CONTENT

ACRONYMS

A

AMQP →	Advanced Message Queuing Protocol

B

BSS →	Business Support System

C

CA →	Certificate Authority
COAP →	Constrained Application Protocol
COTS →	Commercial Off-The-shelf
CNF →	Containerized Network Function
CNTT →	Common NFVI Telecommunications Taskforce
CPS →	Cyber-Physical System
CRL →	Certificate Revocation Lists

D

DDS →	Data distribution Service
DS →	Discovery Server

E

EC →	Electrotechnical Commission
EM →	Element Manager

F

FCAPS →	Fault, Configuration, Accounting, Performance and Security

G

GDS → Global Discovery Server

GUID → Globally Unique Identifier

I

IoT → Internet of Things

IIoTaaS → Industrial IoT as a Service

IPA → Ironic Python Agent

K

KNF → Kubernetes-based Network Function

KPIs → key performance indicators

KDU → Kubernetes Deployment Unit

L

LDS → Local Discovery Server

LSD-ME → LSD with Multicast Extension

M

MAQP → Advanced Message Queuing Protocol

MEAO → Multi-access Edge orchestrator

MEC → Multi-access Edge Computing

MI → Monitored Item

mDNS → multicast DNS

MQTT → Message Queuing Telemetry Transport

N

NBI → NorthBound Interface

NFV → Network Function Virtualization

NFVI → Network Function Virtualization Infrastructure

NFVI-PoP → NFVI Point of Presence

NFVO → NFV Orchestrator

NFV-MANO → NFV Management and Orchestration

NFPs →Network Forwarding Paths

NSD → Network Service Descriptor

NSI → Network Slice

O

OSM → Open Source MANO

OSS → Operation Support system

OPC UA → Open Platform Communication Unified Architecture

UASC → OPC UA Secure Conversation

P

PDU → Physical Deployment Unit

PNF → Physical Network Function

PNFD → Physical Network Function Descriptor

Q

QoS → Quality of Service

R

RO → Resource Orchestrator

S

SDN → Software Define Networking

SO → Service Orchestrator

SOA → Service Oriented Architecture

V

VCA → VNF Configuration and Abstraction

VDU → Virtual Deployment Unit

VI → Virtualization Infrastructure

VIM → Virtualised Infrastructure Manager

VL → Virtual Link

VLD → Virtual Link Descriptor

VNF → Virtualized Network Function

VNFC → VNF Components

VNFD → Virtualized Network Function Descriptor

VNFM → VNF Manager

VNFFGD → VNF Forwarding Graph Descriptor

VNF-FG → VNF Forwarding Graph

VRF → Virtual Routing and Forwarding

ABSTRACT

Nowadays we are witnessing to a continuous increasing of the number of IoT devices that must be configured and supported by modern networks. Considering an industrial environment, there is a huge number of these devices that need to coexist at the same time. Each one of them is using its own communication/transport protocol, and a huge effort needs to be done during the setup of the system. In addition, there are also different kind of architectures that can be used. That’s why the network setup is not so easy in this kind of heterogeneous environment.

The answer to all these problems can be found in the emerging cloud and edge computing architectures, allowing new opportunities and challenges. They are capable of enable on-demand deployment of all the IoT services.

In this thesis is proposed a Multi-access Edge Computing (MEC) approach to face all the possible multi-protocol scenarios. All the services are transformed into MEC-based services, even if they are running over multiple technological domains.

As result, was proved that this kind of solution is effective and can simplify the deployment of IoT services by using some APIs defined by the MEC standard.

As above mentioned, one of the most important tasks of these new generation’s networks is to be self-configurable in very low amount of time and this will be the scope of my research.

The aim of this thesis is to try to reduce as much as possible the time that a certain network requires to be self-configured in an automatic way considering an Industrial IoT as a Service (IIoTaaS) scenario.

INTRODUCTION

In the last years, we have seen an increase of the number of devices that are connected to the Internet. Nowadays, in all the application fields, there is the necessity of automation and control that is granted by IoT devices. Considering as example an industrial scenario, a huge number of IoT devices are used to manage and control production lines.

The number of businesses that use the IoT technologies has increased from 13% in 2014 to about 25% today. The worldwide number of IoT-connected devices is projected to increase more than 40 billion by 2023 [1].

Another important aspect to consider, is the vehicular communication industry. For sure in the next years, we will see an increasing number of vehicles will be connected to some kinds of networks.

It is obvious that current networks and infrastructures are not suitable to handle such number of devices that request a large variety of services and requirements.

It is evident that we need to develop new infrastructures capable of a self-autonomous configuration to guarantee all the requested services and suitable programmable platforms. This trend is also accompanied to the new 5G technologies which enables hyperconnected-machines using massive machine-type and ultra-reliable low latency communications.

This led to the concept of Industry 4.0, which can take advantage of the integration of IoT devices into modern network technologies like Network Function Virtualization (NFV) and Software Defined Networking (SDN).

One of the most important tasks of these new generation's networks is to be self-configurable in very low amount of time. This will be the scope of my research. There is a huge variety of components and different technologies from different manufacturers which need to communicate each other.

All these factors led Edge and Cloud Computing a part of the answer of all these necessities. That's why they are one of the most active areas of development. This kind of solution bring some compute, storage and network capabilities in the user premises allowing a local processing of the data without the need to reach the cloud.

There are lots of standardization bodies, such as ETSI, GSMA (and the previously mentioned 3GPP) which are trying to do design the architecture of edge systems which can handle and support all the characteristics previously mentioned.

NFV applies the Cloud Computing approach to the networking to make networks scalable and adaptable. In this way it is possible to create a network that is configurable in an automated way despite the big number of services and applications that are requested. All the resources and capabilities are virtualized and offered “as a service”.

After that this small introduction, it is clear that the transition to Industry 4.0 is strictly related to the adoption and the development of these new technologies. In this way IoT devices can benefit of high connectivity and processing in a full automated environment using appropriate GUIs and platforms. This led to the inception of an Industrial IoT as a Service (IIoTaaS) model.

The aim of this thesis is to try to reduce as much as possible the time that certain networks require to be self-configured in an automatic way considering an IIoTaaS scenario where a multitude of sensors using their own communication/transport protocol, need to coexist inside it.

During this research, I used Open Source MANO, OpenStack and Kubernetes to manage and orchestrate the setup of a certain network, trying to reduce as much as possible the time needed to complete the task. In this way was possible to unify under the same framework the orchestration fog and edge resources. In particular, the framework used is compliant with the Multi-access Edge Computing (MEC) which can bring all the functionalities offered by the MEC-based approach.

As result of this study shows that the automation setup and deployment of an IIoTaaS environment is feasible. The whole task can be accomplished in tens of seconds (always under 1 minute).

The document is structured as follows. Chapter 1 briefly illustrates the use of the virtualization in modern technologies.

Chapter 2 has the role to introduce the Network Function Virtualization paradigm and the architectural framework designed by ETSI.

Chapter 3 continues the illustration of the ETSI framework, introducing Open Source MANO from the software point of view.

Chapter 4 is focused on a detailed explanation of an internal component of the framework in question, which is called MEC Platform.

Chapter 5 is dedicated to the description of OpenStack, which is a software used for the management of virtual machines.

Chapter 6 introduces the two transport protocols used in this thesis. They are MQTT and OPC UA (with more emphasis on the latter). Chapter 7 then is dedicated to a detailed and deep explanation of the OPC UA protocol.

In the end, Chapter 8 presents the activity that was carried out for the deployment of the desired system, including all the testing and measurements carried. In the end, conclusions are carried.

CHAPTER 1: VIRTUALIZATION TECHNOLOGY IN MODERN NETWORKS

As previously mentioned in the introduction section, the number of services that are simultaneously running networks is increasing day by day. If we take into consideration 5G networks, it is necessary to provide a huge number of services and functionalities.

New networks implement a huge number of dedicated functionalities, which are generally called network function or "middle-boxes". A middlebox is defined as any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host [2].

In general, functions that are implemented inside middle boxes are:

- Security functions: Firewall, Intrusion Detection/Prevention Systems (IDS/IPS).
- Performance functions: Proxy/Caches, WAN optimizers, Protocol accelerators

This new kind of network architecture cannot be sustained by classical networks where we use specialized hardware for all the functions that are needed. Nowadays the continuous evolution on services, require a new model, where all resources and capabilities are virtualized and offered "as a service" allocated on-demand through the cloud.

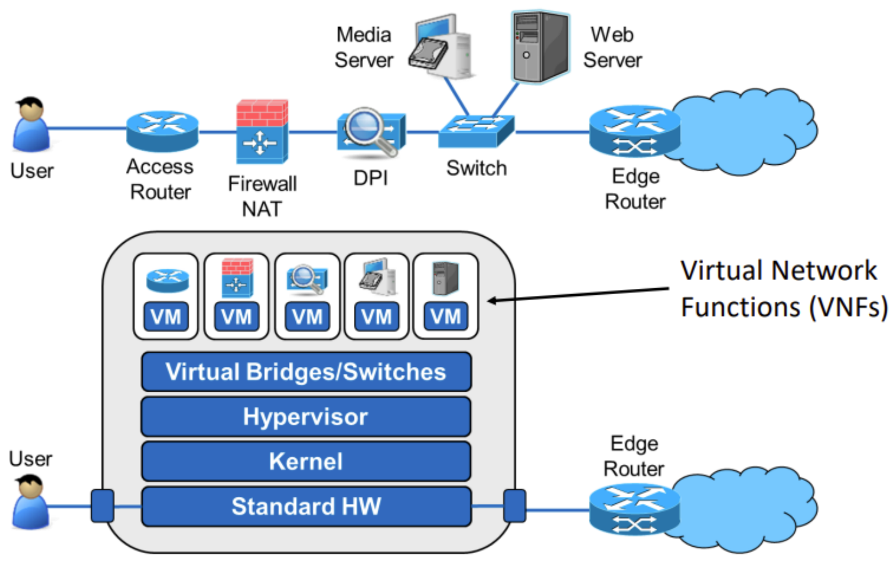


Figure 1.1 : Example architecture VNF

To do that, some solutions like fog and edge computing are adopted. It this way is possible to decentralize network resources in the premises of the final user. This is a very power solution in an IIoTaaS scenario because allow to implement all the requested functionalities.

In addition to this, 5G network slicing is an advanced approach that allow all the features of remote monitoring and dynamic service allocation that are requested.

And here is where the concepts of NFV and SDN comes to our aid.

Network Function Virtualization (NFV) and **Software Define Networking (SDN)** are two approaches that are based on the concept of network abstraction.

SDN → Separate network control functions from network forwarding functions. It abstracts physical networking resources and moves decision making to a virtual network control plane.

NFV → Abstract network forwarding and other networking functions from the hardware on which it runs. As result, it virtualizes all physical network resources beneath a hypervisor, which allows the network to grow without the addition of more devices.

Even if the two paradigms are quite different, have a common final purpose that is to make networks more flexible and dynamic allowing them to be adaptable to all types of architectures and infrastructures.

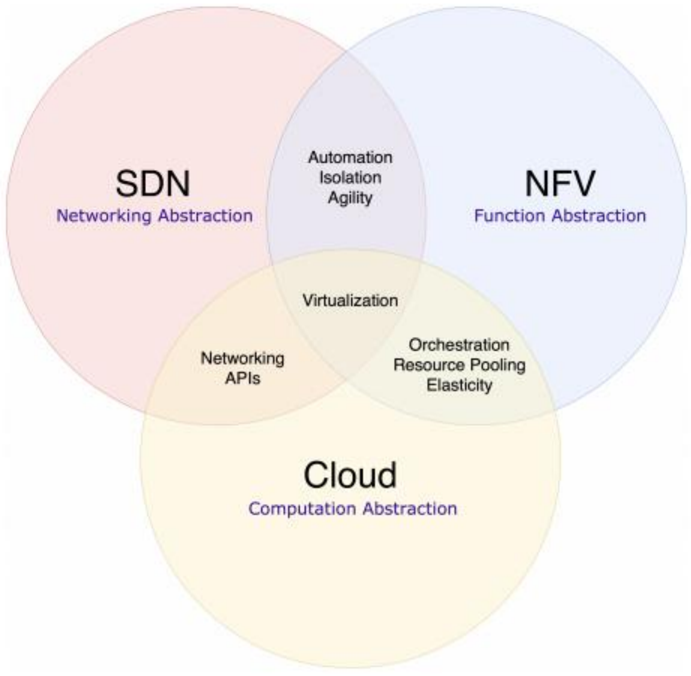


Figure 1.2 : SDN/NFV/Cloud paradigm in comparison

An important aspect that Network Virtualization can provide, is the possibility do deploy a huge number of services and functions without the need of specialized hardware components.

All the middle-boxes previously mentioned are no more running on a specific hardware but are running on a virtualized one. In this way there is no more the problem to be restricted to a single vendor.

To clarify what I mean, I will consider an example:

Suppose a case where we have a network managed by a Cisco router. Cisco provides specialized hardware that can perform in a very efficient way all the functionalities that are requested. Suppose now that a new protocol is realised. Unfortunately, we would be not able to use that protocol, because we are restricted to the functionalities provided by the vendor Cisco. If we want to use that new software, we need to wait that a new update form Cisco will add that new functionality.

In this example is clear that if we choose a classic network infrastructure, we are bounded to commands and functionalities that are defined by the vendor we choose.

If instead of using a specialized hardware, we would have chosen a general-purpose hardware, we would have the possibility to virtualize the specific function that is needed. In this way would be possible to use that new functionality as soon as is realised.

From my point of view, this is the most important advantage that NFV can provide. In old networks, VNF approach was not used. As consequence was used a specialized hardware that execute all the services that are requested.

This approach cannot be used in modern networks. As previously mentioned, the number of network services that are requested is increasing day by day. For this reason, it is impossible to have a network infrastructure that has a specialized hardware. In this case would be very difficult, if not impossible, to deploy a new service that requires a certain hardware that is not present in that infrastructure.

NFV/SDN paradigm can solve all these problems because are able to provide:

- Possibility to virtualize every kind of hardware: Therefore, it is possible to deploy every type of service resolving the need of a specialized hardware.

- Possibility of scalability: Allow to reconfigure and scale the network depending by the traffic and services that are requested.

- Portability: Possibility to deploy network functions over heterogeneous hardware architecture.

- Control and data plane are decoupled. → Applications and networks services see the underlying physical infrastructure as an abstracted virtual entity.

- Network intelligence logically centralized.

- Management and Orchestration: Node control plane functions are directly programmable.

1.1 **Network Function Virtualization (NFV) Architecture**

The concept of NFV is standardized by the European Telecommunications Standards Institute (ETSI) [3].

The main goal of this concept is the possibility to make possible the dynamic configuration of cloud environments. In this way networks can be adaptable and reconfigured based on the current demand of services and user traffic.

There are several advantages that NFV can offers with respect to the classical networks:

- Portability and Interoperability
- Performance
- Management and Orchestration
- Security
- Network stability
- Energy efficiency
- Adaptability, co-existence and integration with existing platforms

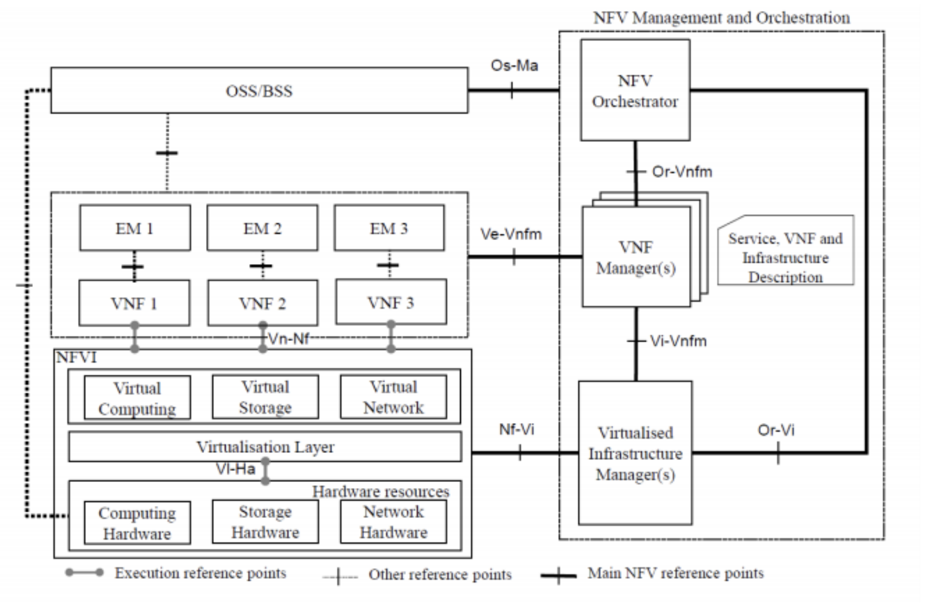


Figure 1.3: ETSI Architecture

As shown in Figure 1.3, there are some functional blocks in the NFV architecture.

1.2 **Methods of Virtualization: VM, LXC, Docker, Kubernetes**

In this section is presented a briefly introduction about all the different virtualization technologies that are used for the purpose of this thesis.

One of the goals of modern software development is to keep several applications running on the same host, but at the same time, each one isolated from the others.

This can be done by using Virtual machines or Linux containers.

1) **Virtual machine (VM)**:

   Require a lot of resources (memory and CPU). It is needed to virtualize both kernel and hardware.

   Is based on the Hypervisor-based virtualization.

   Hypervisor is a process that is running in the host machine. Its job is to separate OS and application from the hardware of the host.

   This solution is widely used in supercomputer, where is necessary to run multiple VMs simultaneously. All the VMs can utilize the CPU and memory resources that are allocated for them, but at the same time are all independent one from each other.

2) **Container Based**:

   Are a very used solution nowadays.

   Containers are piece of software that can be executed directly on the kernel of the host. This means that is not required the virtualization of the hardware. Only the virtualization of the OS is needed.

The two types of virtualizations that I have mentioned have the advantage to allocate dynamically resources that are needed.

There are also some problems too:

Virtual machines → Require resources of virtualize Hardware and OS.

Containers → Require resources for the virtualization of the OS.

Here is where the Docker software can come to our aid. When the software is launched, the Docker daemon can interact directly with the kernel host and allow the interaction between the host machine and all the Docker containers that are running.

Docker → No need of virtualization of Hardware or OS.

The container is directly executed on the Kernel Host.

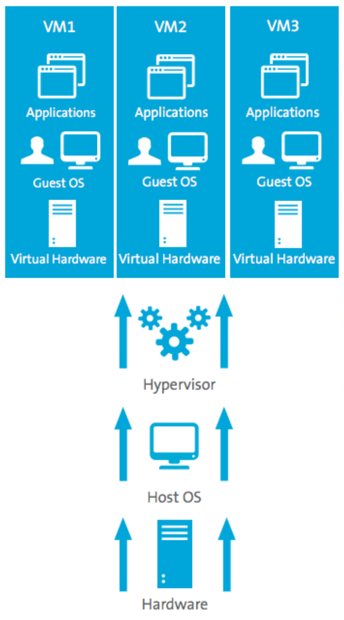
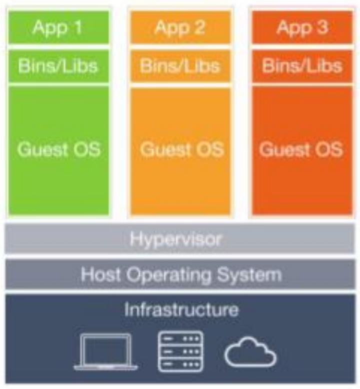
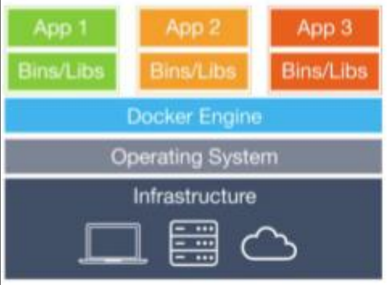
Hypervisor	Container	Docker
 VM1 VM2 VM3 Applications Guest OS Virtual Hardware Hypervisor Host OS Hardware	 App 1 App 2 App 3 Bins/Libs Guest OS Hypervisor Host Operating System Infrastructure	 App 1 App 2 App 3 Bins/Libs Docker Engine Operating System Infrastructure

Table 1: Virtualization Technologies

1.3 **Kubernetes**

Kubernetes (K8s) is an open-source software designed for automating deployment, scaling, and management of containerized applications [4].

To achieve this result, all the containers are organized into high-level logical units which allow the management and control of them.

The main features which are guaranteed by the software are:

- **Automated rollouts and rollbacks** and **Automatic bin packing:**
  K8s can monitor the current status of the application and perform some configuration changes in real-time. Based on the resource requirement can guarantee best-effort services. This feature is also denoted as *horizontal scaling*.

- **Storage orchestration**

- **Secret** and **configuration management (extensibility)**
  The application can scale, reconfigure or update the current configuration without the need of building from scratch the entire application.

- **IPv4** or **IPv6 dual stack**

- **Self-healing**
  K8s can restarts containers that fail or in case there are other problems in an automated way, without the need of advertise of the problem to the client.

All these features are obtained by means of APIs which are accessible through the bash with the command *kubectl*.

1.3.1 *Kubernetes Architecture*

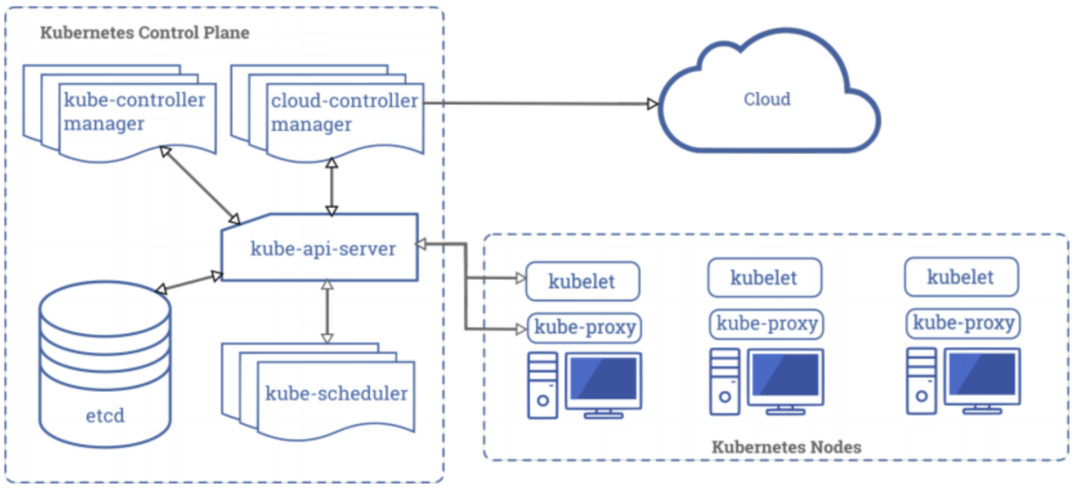


Figure 1.3: Kubernetes architecture: Control Plane + Nodes

As shown in Figure 1.3, K8s architecture can be splitted into two different parts:

1) **Control Plane (master node)**

   The control plane represents the core of the application. Inside it we can find the *master node* which is responsible to control the entire status of the system through different functional blocks:

   - *Kube-controller-manager*: Is a daemon process which act as a controller. In this way make possible to deploy the wanted application.

   - *Cloud-controller-manager*: Is responsible of interconnection the cluster with the cloud provider's API.

     Kube-api-server: Is the functional block responsible to manage the REST API interface.

   - *Kube-scheduler*: Depending by the nodes characteristic assign to them all the tasks to be accomplished.

- *Etcd*: Is a sort of database which contains all the information about the current state of the system.

2) **Nodes/pods**

   K8s generic nodes are all the machines which are responsible to execute all the tasks assigned by the master node. A single node is called pod. A pod can be seen as instance that has some tasks to execute. To execute them, each node has a runtime engine container what can be handled by some platform like Docker or Open Container.

   To make everything working, some processes are running inside each node:

   Kubelet: Is a sort of daemon responsible of the interaction with the master node. We can see it as a sort of agent which is responsible to control that all the containers are running in the pod.

   - *Kube-proxy*: Is a proxy responsible of the management of the traffic between the internal network of cluster and all the external ones.

In this chapter is provided only a brief introduction to K8s. A further development of this can be found in Section 3.10, when this software will be used in combination with OSM.

1.4 **Virtualized Network Functions (VNFs)**

Are all the network functions that are virtualized through the utilization of software components like Virtual Machines (VMs) or Containers. We have multiple ways to do that. It is in fact possible that a VNF is deployed in a single VM or can be composed by several software components called VNF Components (VNFC). Through the interaction of these components, it is possible to obtain the wanted function.

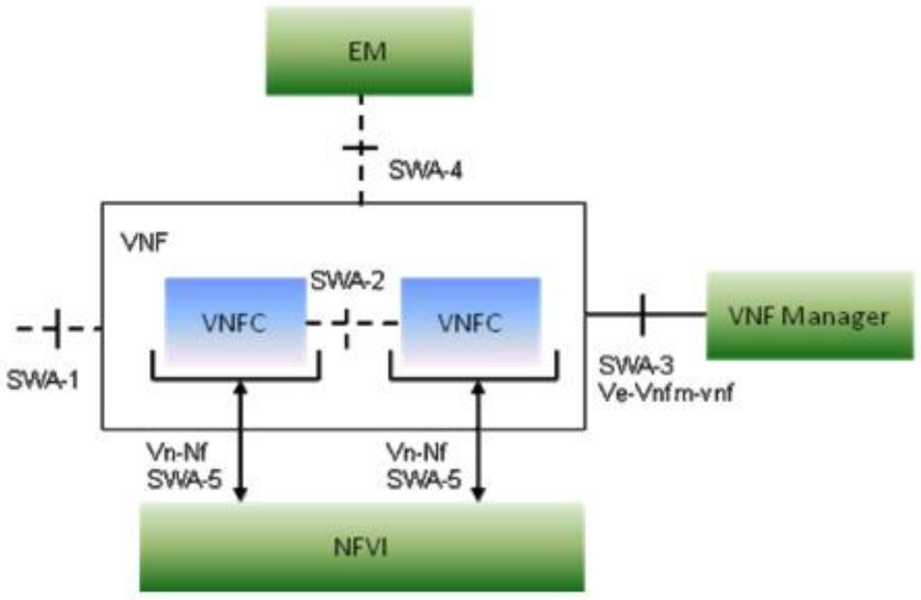


Figure 1.4: VNF Architecture [5]

We can also see that there are some interfaces that link all the VNFC that are inside the VNF block with other functional blocks: EM, VNF Manager and NFVI that will be descripted soon.

1.4.1 *Element Manager (EM)*

Is the entity that is responsible of the management of FCAPS (Fault, Configuration, Accounting, Performance and Security) of the VNF:

-*Fault*: management of fault in the network function

-*Configuration*: Configuration of the network function by means of several parameters

-*Accounting*: Tracking network utilization information, such that individual users, departments, or business units can be appropriately billed or charged for accounting purposes [6].

-*Performance*: collection of data regarding the performance of the network function.

-*Security*: security management for the access to the configuration of the network function.

1.4.2 *Network Function Virtualization Infrastructure (NFVI)*

This block represents the core of the entire infrastructure because it includes the totality of all general-purpose hardware components that is used by the infrastructure.

All the virtual components that are created run over this general-purpose hardware.

From Figure 1.5 it is possible to see that there are three types of resources:

- *Computer domain resource*: It is subdivided into 3 other categories:
    - Computing Hardware: In general, we have a COTS (Commercial Off-The-shelf) hardware because as previously mentioned is necessary to have a general-purpose hardware where is possible to implement every kind of function.
    - Storage Hardware: volumes of storage at either block or file-system level

- Network Hardware: includes switching functions and links.

- *Hypervisor Domain resources*: Is the part related to the virtualization process. Virtual compute and Virtual storage are classified inside this category.

- *Network domain resource*: This includes Network hardware (physical resources) and Virtual networking.

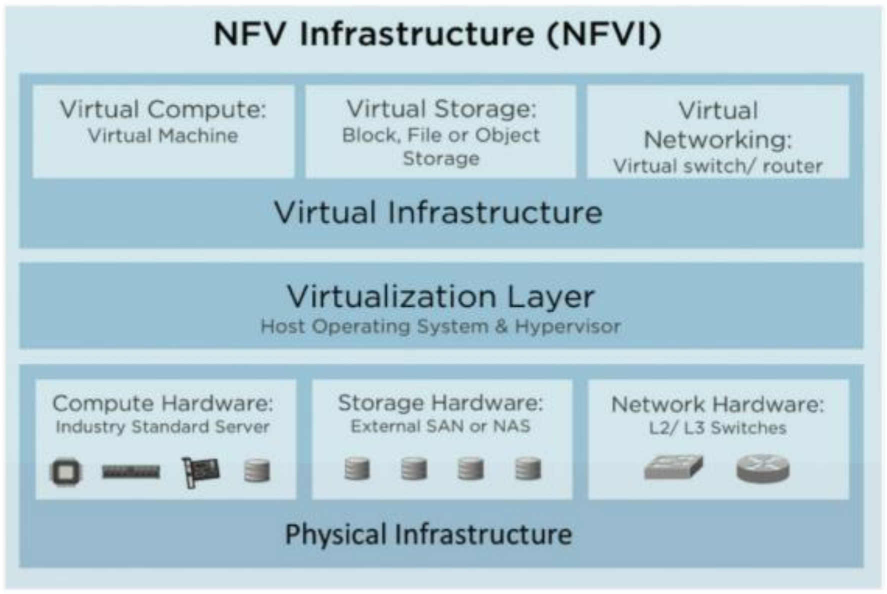

NFV Infrastructure (NFVI)
Virtual Compute: Virtual Machine
Virtual Storage: Block, File or Object Storage
Virtual Networking: Virtual switch/ router
Virtual Infrastructure
Virtualization Layer
Host Operating System & Hypervisor
Compute Hardware: Industry Standard Server
Storage Hardware: External SAN or NAS
Network Hardware: L2/ L3 Switches
Physical Infrastructure

Figure 1.5: NFVI Infrastructure

It is clear that NFVI resources can be both virtualized and non-virtualized resources.

Is defined NFVI Point of Presence (NFVI-PoP) a node that allocates some resources that are needed for a certain VNF.

The allocation of those resources can be a complex task. To manage and orchestrate all these operations there is a specific entity called NFV-MANO that will be explained in the next section.

As already said in the introduction part, the NFVI has a very important role in the process of virtualization. It is in fact able to provide to all those software components of the VNFs to be decoupled from the hardware resources. As result, is possible to ensure portability on different hardware platform. We can see it as a sort of IaaS because has the possibility of delivering VNFs that support the actual services that are provided by a Network Operator.

1.4.3 *Operation Support system (OSS) and Business Support System (BSS)*

Is the functional block that is in general used by vendors and telco operators for the management of the entire system.

In particular, the OSS has the role of monitoring the Quality of Service (QoS) while the BSS has a role related to the management of the commercial transaction's telco operator with its customers.

1.5 **Physical Network Functions (PNFs)**

In general, Physical Network Functions are implemented using a set of software modules deployed on a dedicated hardware.
Nowadays the approach is moving in the direction of a cloud environment where everything is based on cloud resources.
In this way is possible to offer scalability, security, and the possibility to upgrade the PNF as soon as is needed.

1.6 **Containerized Network Functions (CNFs)**

In the previous section I have explained and analized what are the characteristics of virtualization offered by containers.
As general case is possible to use containers as boxes inside which is possible to deploy the wanted service.
We can use LXC or a software container manager like Docker or Kubernetes for the deployment of the container.

In case is used Kubernetes, the NF that is created is called Kubernetes-based Network Function (KNF).

CHAPTER 2:
NFV MANAGEMENT AND ORCHESTRATION (MANO)

We have previously mentioned the NFVI. It is the entity that can guarantee the decoupling of Network Function from the hardware infrastructure which they are usually tight to. This process of decoupling imposed by NFV exposes a new set of entities (VNFs) and a new set of relationships with the NFVI.

This process is quite complex because requires the collaboration of some functional blocks described before. To handle the management and orchestration of all these blocks, comes to our aid the NFV-MANO.

The NFV Management and Orchestration (NFV-MANO) is in charge of manage the NFVI and orchestrate the allocation of resources needed by the Network Services and VNFs. This functional-management block has all the components to coordinate all the operations needed for the deployment of them.

NFVI → Manage and orchestrate virtualized and non-virtualized resources. Virtualized resources are catalogued to be offered as abstracted services. They can be distributed over multiple NFVI-PoPs and the management and orchestration. The allocation and release of them is a dynamic process that is very complex and is done by the VNF lifetime (explained in the next section).

VNFs → FCAPS and lifecycle management. This are essential tasks because provide some monitor KPIs that are stored inside a monitor template and will be used for scaling operations.

NS → Responsible for the Network Service lifecycle management (will be treated soon)

2.1 **NFV MANO architectural framework**

The NVF Management and Orchestration (NFV-MANO) architectural framework has the role to manage the NFVI and orchestrate the allocation of resources and Network Functions requested for the deployment of Network Services and VNFs. The decoupling imposed by NFV requires a new and different set of management and orchestration functions.

The NFV MANO architectural framework identifies the following functional blocks:

2.1.1 – *Virtualised Infrastructure Manager (VIM)*

The Virtualised Infrastructure Manager (VIM) is responsible for controlling and managing the NFVI compute, storage and network resources, that are provided by an operator.

All these resources are kept under the control of the VIM that manages their virtualization using hypervisors and other network controllers.

Some important aspects and functions are performed by the VIM:

- Manage and orchestrate the allocation of upgrade/release/reclamation of NFVI resources.

- Control the assignment of the virtualised resources to the physical compute, storage, networking resources.

- Supporting the management of VNF Forwarding Graphs.

- Managing the information related to hardware and software resources. In this way it is possible to know in advance what will be the performances provided.

- Management of the virtualised resource.

- Management of software images (add, delete, update, query, copy) as requested by other NFV-MANO functional blocks (e.g. NFVO)

- Collection of performance and fault information (e.g. via notifications) of hardware resources (compute, storage, and networking) software resources (e.g. hypervisors), and virtualised resources (e.g. VMs);

2.1.2 – *NFV Orchestrator (NFVO)*

The NFVO is the entity that has the role of taking orchestration decisions over the virtualization of all those resources that are under his authority. In general, has the role of managing physical, computing, storage, and network resources.

The two main important functionalities that it performs are:

1) Lifecycle management of NSs, fulfilling the NS Orchestration functions. This is done by the Network Service Orchestrator. The Resource Orchestrator keeps under control of all the instances and resource that are allocated for all the VNFs. In this way is possible to keep track of the utilization of them but also manage their utilization.

2) Orchestration of NFVI resources across multiple VIMs. This is done by the Resource Orchestrator (RO). An important functionality performed by the Orchestrator is the deployment of the so-called Network Service Descriptors (NSDs). A descriptor can be seen as a sort of catalogue where are stored some information and characteristics regarding a specific NS. (NSDs will be described in the next section).

These two functionalities that I reported above can summarize in an exhaustive way the role of the NFVO.

All the functionalities and capabilities offered by the NFVO through the Network Service Orchestrator are [7]:

- Management of NS deployment templates and VNF Packages (e.g., on-boarding, validation).

- NS instantiation and NS instance lifecycle management, (e.g., update, query, scaling, performance measurement, event collection and correlation, termination).

- Management of the instantiation of VNF Managers (where applicable)

- Management of the instantiation of VNFs, in coordination with VNF Managers.

- Validation and authorization of NFVI resource requests from VNF Managers.

- Management of the integrity and visibility of the NS instances through their lifecycle, and the relationship between the NS instances and the VNF instances, using the NFV Instances repository.

- Management of the NS instances topology, (e.g., create, update, query, delete VNF FGs).

- NS instances automation management.

- Policy management and evaluation for the NS instances and VNF instances.

- Validation and authorization of NFVI resource requests from VNF Managers.

- NFVI resource management across operator's Infrastructure Domains including the distribution, reservation and allocation of NFVI resources to NS instances and VNF instances by using an NFVI resources repository.

- Supporting the management of the relationship between the VNF instances and the NFVI resources allocated to those VNF instances by using NFVI Resources repository and information received from the VIMs.

- Policy management and enforcement for the NS instances and VNF instances (e.g., NFVI resources access control, reservation and/or allocation policies, placement optimization based on affinity and/or anti-affinity rules as well as geography and/or regulatory rules, resource usage, etc).

- Collect usage information of NFVI resources by VNF instances or groups of VNF instances, for example, by collecting information about the quantity of NFVI resources consumed via NFVI interfaces and then correlating NFVI usage records to VNF instances.

All these functionalities provided by the Resource Orchestrator allow to the NFVO to perform correctly all the tasks required. In particular the NFVO services are used to support the access to the NFVI resources in an abstracted manner independently of any VIMs, as well as governance of VNF instances sharing resources of the NFVI infrastructure.

2.1.3 – *VNF Manager (VNFM)*

The VNFM is the functional entity that has the role to manages lifecycle management of VNF instances. To each instance is associated a VNF Manager that perform some important functions:

- VNF instantiation
- VNF instantiation feasibility checking.
- VNF instance software update/upgrade.
- VNF instance modification.
- VNF instance scaling out/in and up/down.
- VNF instance-related collection of NFVI performance measurement results and faults/events information.
- VNF instance assisted or automated healing.
- VNF instance termination.
- VNF lifecycle management change notification.
- Management of the integrity of the VNF instance through its lifecycle.

All these instances for the management of the lifecycle are specified and contained inside a template called Virtualized Network Function Descriptor (VNFD).

All the VNFDs are stored inside the NFV catalogue. NFV-MANO uses a VNFD to create instances of the VNF it represents and also to manage the lifecycle of those instances.

The NFVO is a very important function block because through the information that are stored inside the templates VNFDs can guarantee the flexible deployment and portability of VNF instances on multi-vendor and diverse NFVI environments.

To do that, hardware resources need to be properly abstracted, and this is done by creating the proper VNFDs.

2.2 **NFV-MANO data repositories**

From Figure 2.1 is shown that there are four types of data repositories:

1) NS Catalogue:

   It is a sort of repository that store all the on-boarded network services and is used during the creation and management of the deployment templates (will be described on section 2.3).

2) VNF Catalogue:

   It is a sort of repository that store all the on-boarded VNF Packages.

3) NFV instances Repository:

   Holds information of all VNF instances and Network Service instances. For each Network service we have a different record that is updated during its lifecycle to have a full description of the actual status of all the NS that are currently offered.

4) NFVI Resources Repository:

   Contains all the information about all the NFVI resources (available or reserved or allocated). NFVI Resources Repository has a fundamental role because through the cooperation with the NFVO can track the reservation/allocation of NFVI resources.

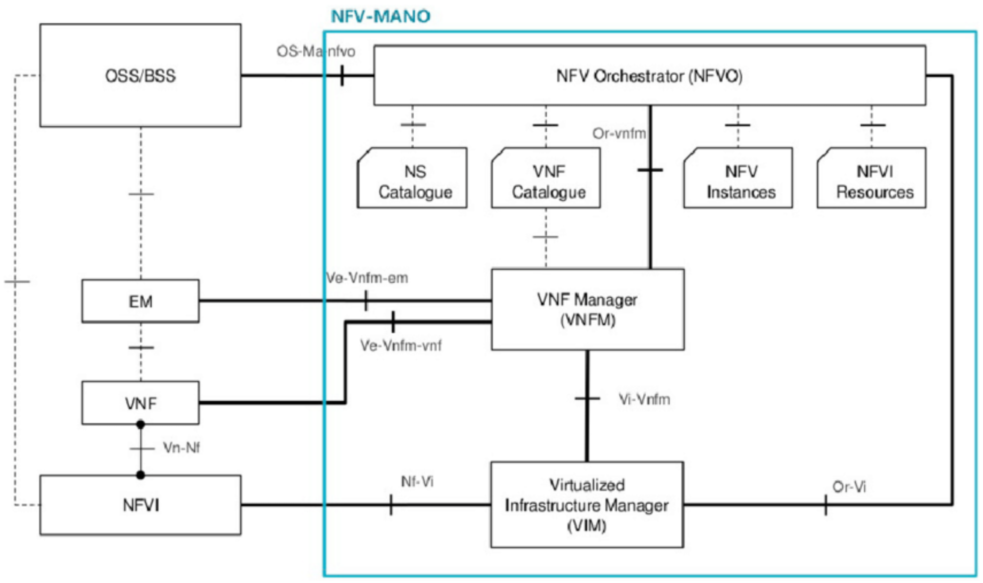


Figure 2.1: NFV-MANO Architecture

We can also identify some **reference points** that can be seen as interfaces that interconnect two different blocks of the structure.

- Os-Ma-nfvo, interface between OSS/BSS and NFVO
- Ve-Vnfm-em, interface between EM and VNFM
- Ve-Vnfm-vnf, interface between VNF and VNFM
- Nf-Vi, interface between NFVI and VIM
- Or-Vnfm, interface between NFVO and VNFM
- Or-Vi, interface between NFVO and VIM
- Vi-Vnfm, interface between VIM and VNFM

2.3 **MANO Descriptors**

All the services that need to be deployed by the vendor operator need to be described in some way.

The description of a NS is done though some specific elements:

- *Virtualised Network Function Descriptor (VNFD)*:

It is a sort of index-template that contains all the references to all other descriptors which describe components that are part of that Network Service. It is stored inside the VNF catalogue.

- *Physical Network Function Descriptor (PNFD)*:
  Describes all the resources and connectivity information that are necessary for the interconnection between a Virtual Link (VL) and a Physical Network Function (PNF)

- *Virtual Link Descriptor (VLD)*:
  It is a template that describes resource requirements of a virtual link. All these information is used by the NFVO to correctly orchestrate all the other functional blocks in order to satisfy them.

- *VNF Forwarding Graph Descriptor (VNFFGD)*:
  It is a description of the Network Service that we want to implement. It contains all the references to the VNFs and PNFs and Virtual Links that connect them.

- *Network Service Descriptor (NSD)*:
  The Network Service Descriptor is a template file, that contains all the information and parameters used by the Orchestrator (NFVO) for deploying network services [8].
  It can be seen as a sort of container or index that keeps all the data of the other descriptors. Also provides a description of some components that are part of the Network Service. Through the NSDs the NFVO can handle the life cycle management of a Network Service.

2.4 **Forwarding Graph**

Each network service can be represented in a schematic way using a VNF *Forwarding Graph (VNF-FG).*

This means that a NF can be fully described through this abstract representation that defines the sequence of VNFs, PNFs and the set of logical and virtual links (VLs) that a certain packet has to traverse in order to make possible the wanted service.

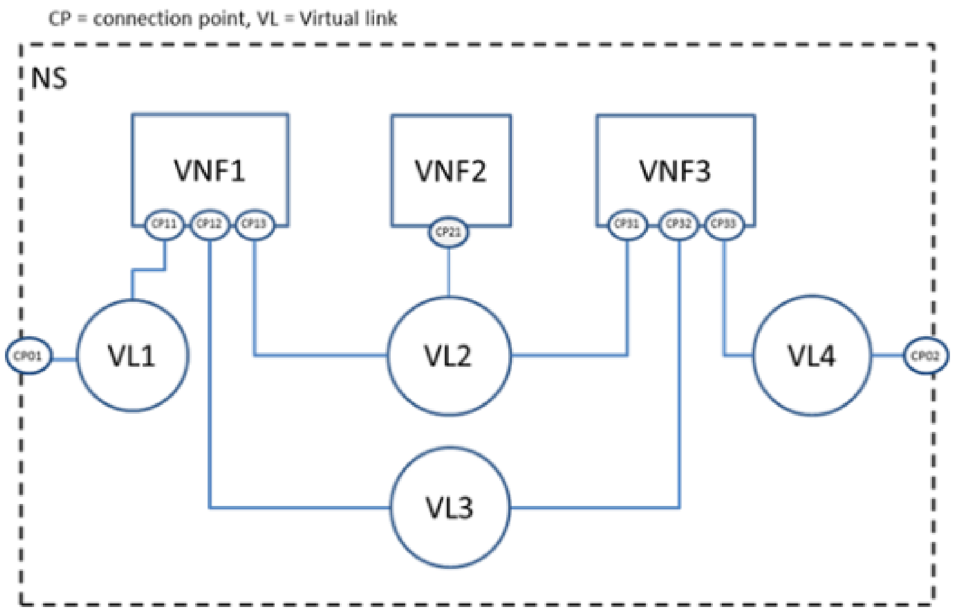


Figure 2.2: Example of a Forwarding Graph [9]

To summarize, a Forwarding Graph can give some information to the service provider that is working to develop the NS and understand how the traffic is flowing inside the network.

Some Network Forwarding Paths (NFPs) can be identified and be used to forward the packet to the correct interface depending by the service that we are considering.

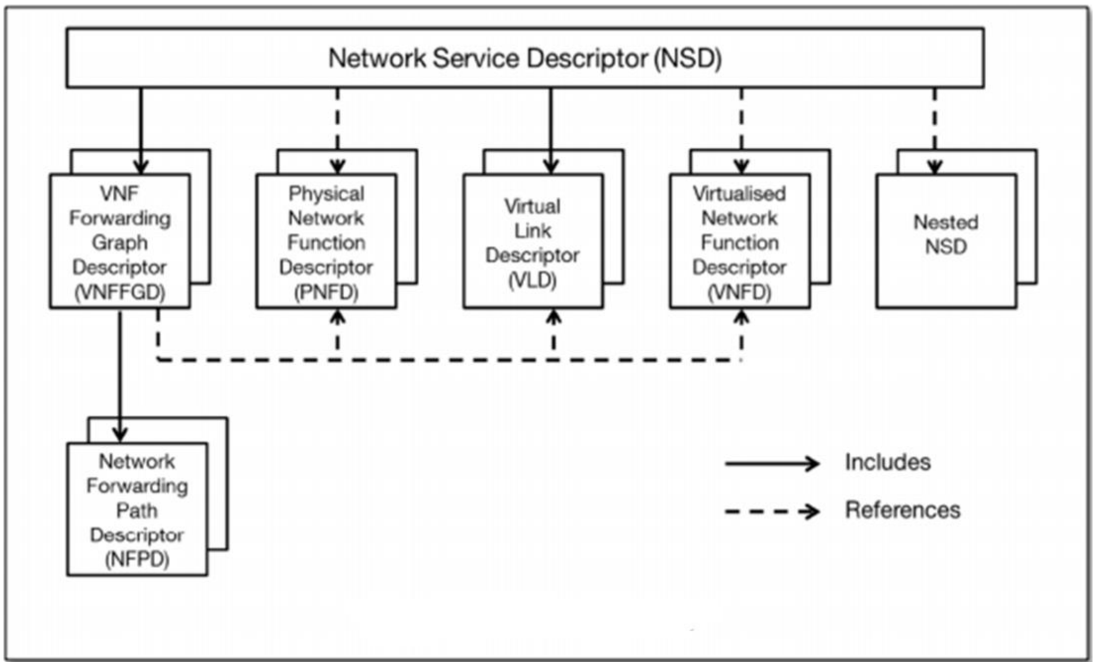

Network Service Descriptor (NSD)
VNF Forwarding Graph Descriptor (VNFFGD)
Physical Network Function Descriptor (PNFD)
Virtual Link Descriptor (VLD)
Virtualised Network Function Descriptor (VNFD)
Nested NSD
Network Forwarding Path Descriptor (NFPD)
Includes
References

Figure 2.3: Network Service Descriptor elements

At the end of the day, we can state that a NS can be seen as an interconnection of multiple Network Functions arranged as a set of functions with unspecified connectivity between them or according to one or more forwarding graphs.

CHAPTER 3: OPEN SOURCE MANO (OSM)

As previous mentioned, management and orchestration are essential elements in a network deployment. There is huge number of different services that Industries want to deploy and a big number of vendors providing different VNFs.

For this reason, is very important to find a way to combine all these different VNFs and "hope that everything will work fine". Even if we have all the VNF descriptors, is difficult to predict what would be the integration of different VNFs under the same network.

The reason of that is because VNF descriptors are only a theoretical description of a certain service and does not represent the complexity of the real environment.

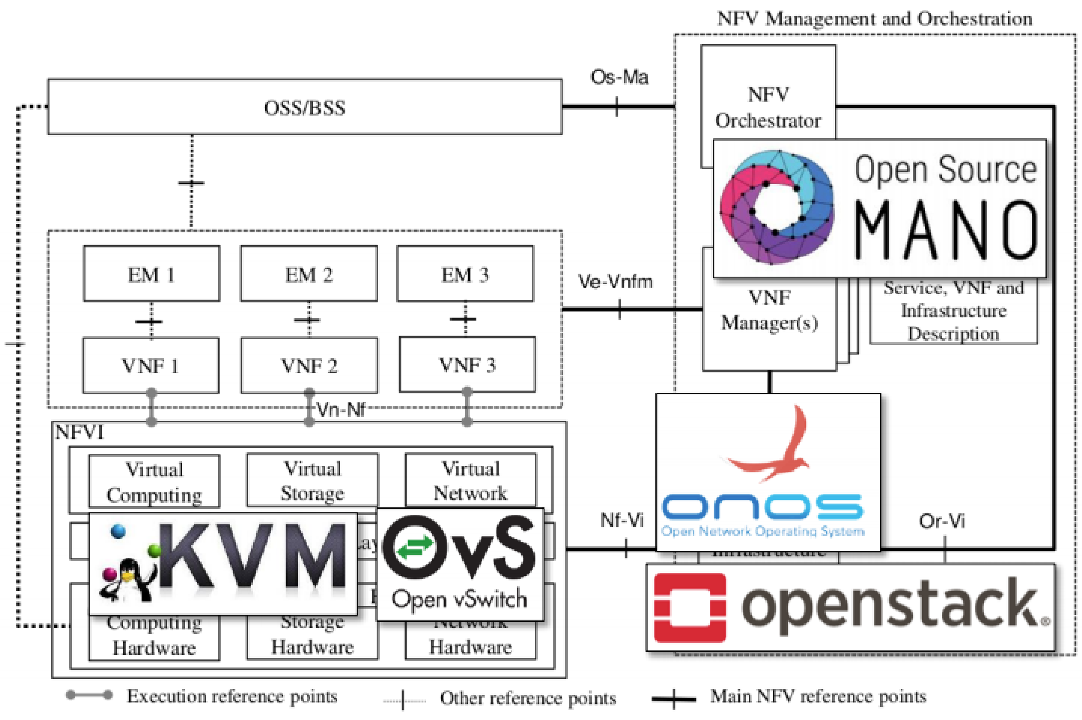


Figure 3.1: ETSI Architecture Services and Vendors

The problem of the approach “hoping that everything will work fine” cannot be applied in all the application domains.

If we consider the case of a vehicle communication domain, we must be sure that the network will be stable and working as expected. In that domain, network operators and engineers have people’s life in their hand.

Here is where OMS software shows it’s potential. In fact, is capable to enable an eco-system where different VNF vendors can offer their service and apply it to different network infrastructures.

Open-Source MANO (OSM) is an open-source management and orchestration (MANO) platform aligned with the ETSI NFV specifications [10].

As shown in Figure 3.1, the OSM architecture is responsible of the orchestration of the NFVO and VNFM.

The VIM instead is out of the focus of this functional block. This because it already exists a certain number of existing management infrastructures like ONOS or OpenStack that are specialized over the VIM functionalities.

The OSM software can be described by means three different software components:

1) *Resource Orchestrator (RO)*:

   Acts as NFVO as described for the MANO architecture. It is responsible of the orchestration of resources and is composed by some software python scripts.

   It is possible to interact with this software component by means of the NorthBound Interface (NBI), by using some specific APIs [11].

2) *VNF Configuration and Abstraction (VCA)*:

   Is responsible of the lifecycle management of the VNF and manage them in real time, like the VNF Manager in the MANO framework. To do that is required the Juju software that has the role to concentrate all the resources needed for the deployment of the VNF in a single application.

   Juju Software will be treated in a next section.

3) *Service Orchestrator (SO)*:

Is responsible of the correct functioning of the VCA and RO, but also other important functions. In fact, can control the interaction between JUJU and RO.

In addition, provides a GUI for the entire management of the whole OSM framework.

OSM is an open software that offers a well-known Information Model (IM) aligned with standard ETSI NFV SOL006 that can be used to model all the Network Services required by operators without worrying of the virtualization of resources and the underlying structure. SOL006 is described through YANG models.
YANG is a data model language that is used to define entities and their structure, but also how all those entities relate each other.
The YANG model is defined by standard, but it is possible to add new features that are not yet present in the standard (*augment* as shown in Figure 3.3).

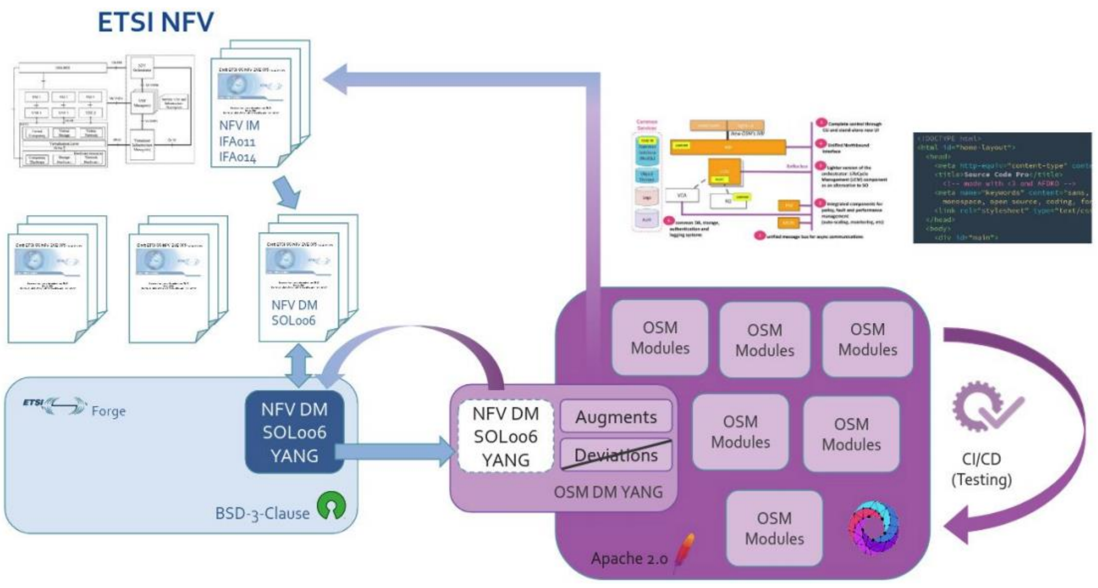


Figure 3.2: YANG model

At the end of the day, through this software is possible to define which entities exist within the NFV domain and how to relate them to create descriptors for the deployment of the Network Service we want to implement.

Through these entities is possible the management and the automation of the entire lifecycle of NF or NS or Network Slices.
A Network Slice (NSI) can be seen as an aggregation of Network Service. This is an independent entity that can be deployed repeatedly. This means that the same slice can deployed multiple times choosing the NSs contained inside every slice.

3.1 **Network Service Lifecycle**

During the deployment and cycle of Network Service, we can identify a series of common steps that need to be followed to properly accomplish the task.
We can call lifecycles of a VNF all the steps that allow to the VNF to be instantiated, managed, scaled up or down, and terminated when is no more needed.
As previously mentioned inside the MANO framework there are some functional blocks that work together for the correct deployment of the VNF.

- NFVO → Responsible of the VNF Orchestration but also controls the management of the VNF Manager.
- VIM → Is responsible of the management of the virtualized network resources.

The VNF lifecycle is not controlled by a single functional block but involve many of them. To enter more in details, the validation of the VNF Lifecyle is a complex work made mainly through the collaboration of the NFVI, NFVM and the NFV Function.
This process is essential in the efficiency of the Network Service. If all the steps are done correctly, we can reduce a lot the time for the deployment of certain services.
To do that, some validation steps are performed to achieve the best performance for the setup of the service.
The MANO framework performs some evaluation tests that can be seen as a sort of benchmark tests. In this way is possible to evaluate what are the actual performance offered in terms of features and scalability that are provided by the NFV.

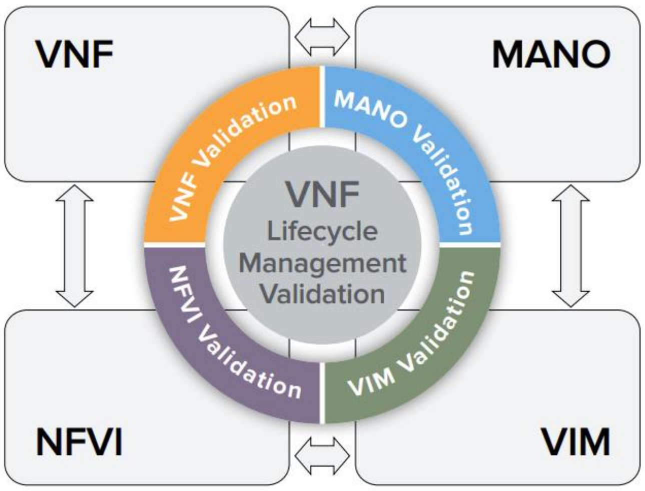


Figure 3.3: VNF Lifecycle

3.1.1 *Setup and configuration of a VNF*

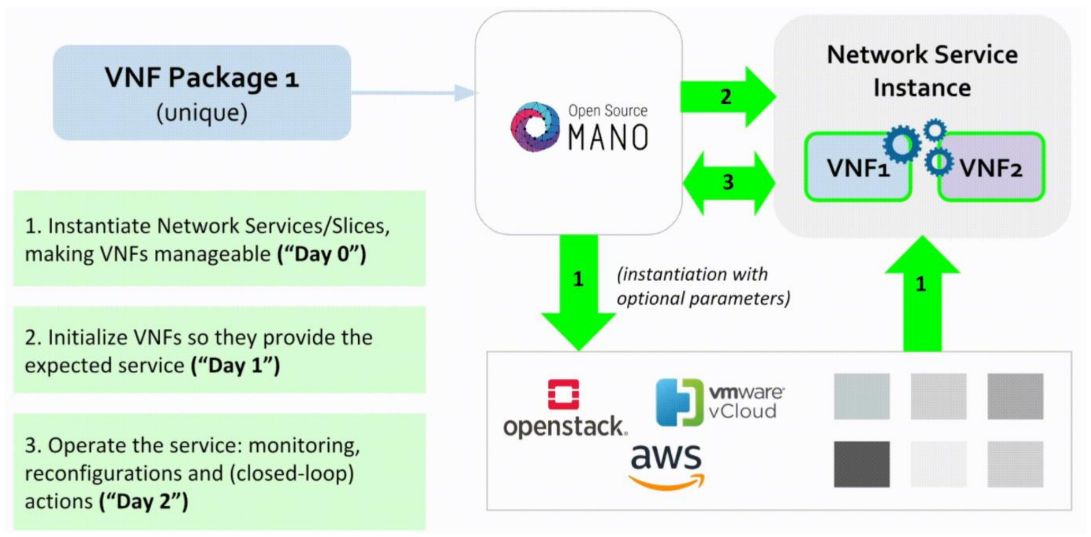

VNF Package 1
(unique)
Open Source MANO
Network Service Instance
VNF1
VNF2
1
2
3
1
(instantiation with optional parameters)
1. Instantiate Network Services/Slices, making VNFs manageable ("Day 0")
2. Initialize VNFs so they provide the expected service ("Day 1")
3. Operate the service: monitoring, reconfigurations and (closed-loop) actions ("Day 2")
openstack
vmware vCloud
aws

Figure 3.4: VNF Configuration [13]

The aim of the OSM is to take a Network vendor software and break it down into a series of descriptors written in SOL006 code. Through them is possible to model the wanted service and make it manageable (***Day 0***).

The service so far created can be located somewhere in the network. In the moment the service is launched, provides some base functionalities (***Day 1***).

The service so far create can be modelled and scaled in a horizontal way by adding new instances to that service (***Day 2***).

The setup of a VNF is based on some packages called *Charms*.

A *Charms* can be seen as book containing some information and pieces of software. Some useful and well-known scripts can be stored inside them [13].
This means that they are quite useful when we want to implement a certain application. By looking at them we could find some useful information for the implementation of the NS. The aggregation of multiple charms is called *Bundle*.
The concepts of *Charms* and *Bundle* will be treated in section 3.3.
The OSM platform is based on this concept for the deployment of network services. To do that, are used three types of software. They are called *Juju* and *Cloud-init* and *Charms.*

3.2 **Cloud Init**

Cloud-init is developed and released as free software under the GPLv3 open-source license. It was originally designed for the Ubuntu distribution of Linux in Amazon EC2 but is now supported on many Linux and UNIX distributions [15].

This software is used for the execution of scripts that can be uploaded un the cloud before the deployment of a service or virtual machine.

In particular, it is possible to identify three different types of scripts:

- Cloud config (used in OSM)
- Script shell
- Text only

Those scripts are executed in the beginning of the deployment of the VNF.

The cloud image that needs to be used by the VNF and initialize the system based on that.

The cloud-init scripts can execute some actions:

- Configure the users and hosts.
- Configure ssh keys.
- Create passwords.
- Add repository.

3.3 **Charms**

A charm is a collection of scripts and metadata that contains all the data and knowledge about a particular software product. Charms make it easy to reliably and repeatedly deploy applications, then scale them as required with minimal effort [16].

In the OSM system, charms are used for the deployment of cloud resources.

There are some important concepts related to the software that are: Cloud, Container, Controller, Model, Bundle and Machine [17].

3.3.1 *Cloud*

Is the term that identify all those resources (machines, instances, and storage) that are used for the deployment of the wanted NVF. As mentioned before, it is the component that classify Juju as IaaS like other cloud services such as AWS, Microsoft Azure, OpenStack-based cloud.

3.3.2 *Container*

Is not a keyword inside the Juju concept. It is used to refer an LXD-based machine.

3.3.3 *Controller*

The Juju controller is responsible of the initial creation of the cloud instance. Through the use of APIs can manage and control all the cloud resources that are needed. Juju controller will be explained more in detail in a next section.

3.3.4 *Model*

Is a sort of virtual space where all the cloud resources requested by the controller are allocated. It can be seen as a sort of VM. All the resources that are allocated for a specific controller are isolated form the others.

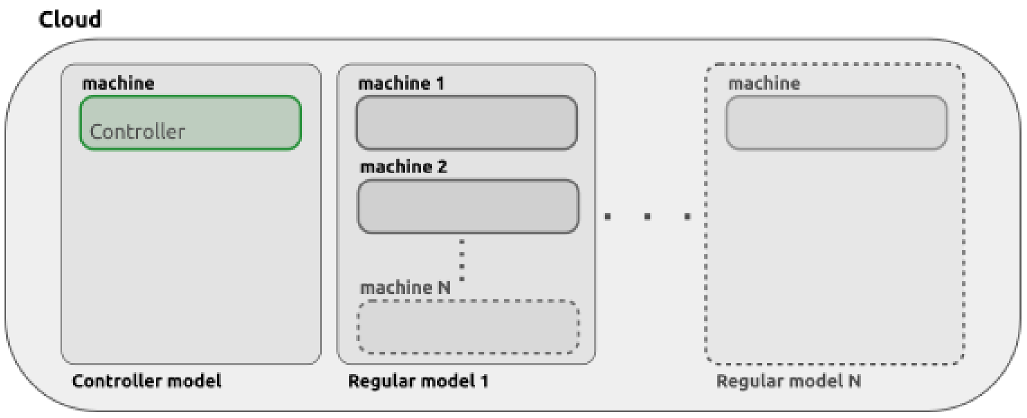


Figure 3.5: Collection of Charms

3.3.5 *Bundle*

A bundle is a collection of different charms that offer different services that are complementary each other and can be linked together to deploy the NS wanted with all the functionalities required.

3.3.6 *Machine*

Represents the cloud instance that is requested by Juju.

3.4 **Types of charms**

We can identify two types of charms depending by the type of workload that is requested by the service:

- *Native Charm*: Are much easier to be implemented. Only the Juju controller (that will be explained in the next section) is hosted inside the OSM host. In this case the Charm is running directly inside the VM.

- *Proxy Charms*: The OSM host contains both the Juju controller and the charm. All the complex code is running inside the OMS host and is sent to the targeted VM (in this case managed by OpenStack VIM) by means of ssh.

  They run outside the application like in a specific LXC container that can be configure depending by the VNFs that we want to obtain. This is a very powerful solution because enables the possibility of configuration properties to PNFs and HNFs.

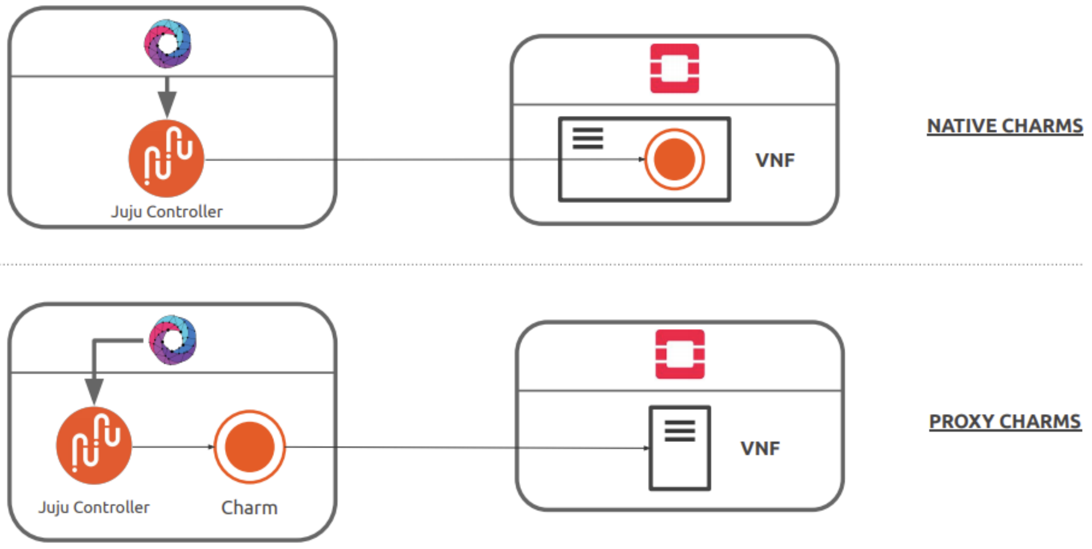
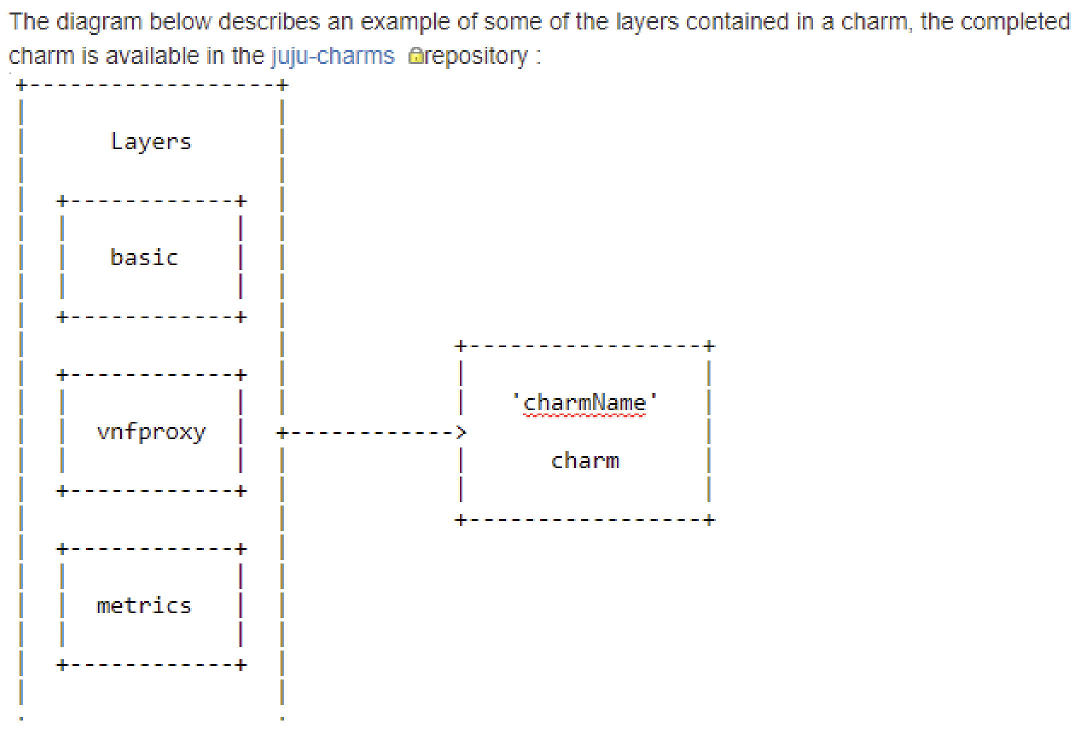


Figure 3.6: Differences between Proxy Charms and Native Charms

A proxy charms can be seen as a layered structure like the ISO/OSI layer structure. Each layer adds new functionalities underneath ones.



The diagram below describes an example of some of the layers contained in a charm, the completed charm is available in the juju-charms repository :

```
+-------------------+
|                   |
|      Layers       |
|                   |
|  +------------+   |
|  |            |   |
|  |   basic    |   |
|  |            |   |
|  +------------+   |
|                   |            +------------------+
|  +------------+   |            |                  |
|  |            |   |            |    'charmName'   |
|  |  vnfproxy  |   +----------->|                  |
|  |            |   |            |       charm      |
|  +------------+   |            |                  |
|                   |            +------------------+
|  +------------+   |
|  |            |   |
|  |  metrics   |   |
|  |            |   |
|  +------------+   |
|                   |
```

Figure 3.7: Example of layers contained in a Charm [18].

To enter more in detail, there some basic layers that are common in all the charms:

- Basic layer → Contains the core needed for other layers for function properly.
-
- Vnfproxy → Imports the required functions to run actions in the VNF via SSH. This layer has been designed to aid in the development of proxy charms.

- Metrics → Imports the required functions to get metrics from the VNF.
- RESTAPI → Imports the required functions to run actions in the VNF via REST API.
- Netconf → imports the required functions to run actions in the VNF via Netconf primitives.

To deploy a NS could be necessary to use multiple Charms. By interconnecting multiple Charms and allow them to interoperate each other is possible to obtain the wanted functionality.

As shown in Figure 3.9, each charm offers one or more matching integration points which are used to interconnect two different charms and exchange data and information.

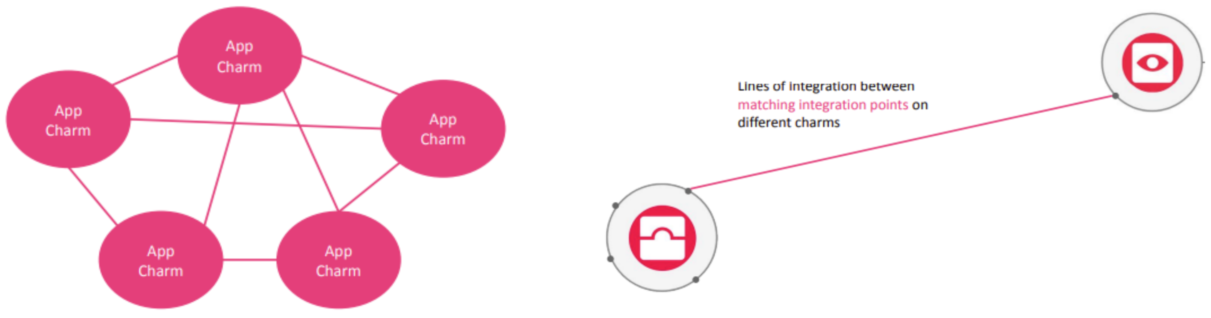

App Charm
App Charm
App Charm
App Charm
App Charm
Lines of integration between matching integration points on different charms

Figure 3.8: Interconnection between Charms

3.5 **JUJU**

Juju is an open-source application developed by Canonical. It is used as modelling tool for the deployment, configuration, scaling and operates cloud infrastructures quickly and efficiently on public clouds such as AWS, GCE, and Azure along with private ones such as MAAS, OpenStack, and vSphere [19].

Its role in the OSM ecosystem is to manage and orchestrate cloud resources. To compare it with other existing technologies, can be classified as an IaaS like in OpenStack.

To understand better what the role of Juju is, is betted to define first what are the modules that we can find inside the Juju architecture:

- *Juju Deployer/Controller* → Allows to interact with the cloud software through some APIs.

- *Juju CLI* → Is the command line interface that allow to access and interact with the Controller.

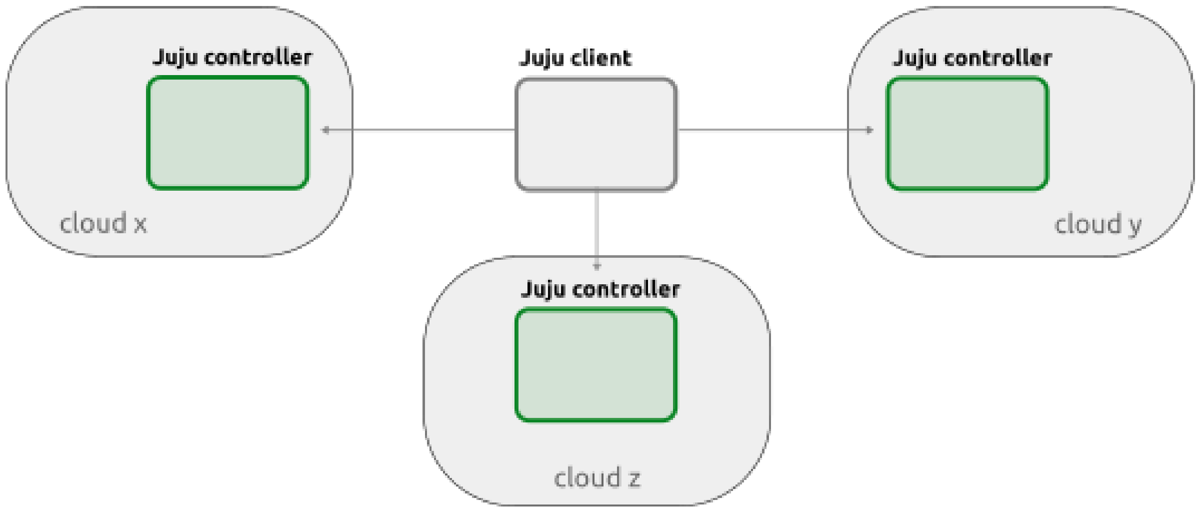


Figure 3.9: Juju architecture

- *Juju GUI* → Is the same concept of the Juju CLI but in this case the software also provides a graphical interface with some useful tools.
- *Juju State Server* → It is the entity in charge of the management of the data that is uploaded on the cloud by the software.

- *Charm Store* → It is a sort of repository where preconfigured bundles and scripts can be downloaded and used.

- *Juju Agent* → It is a software that is execute inside every Juju machine and operate at the application level. Usually is responsible of tracking state changes, respond to those changes, and pass updated information back to the controller.

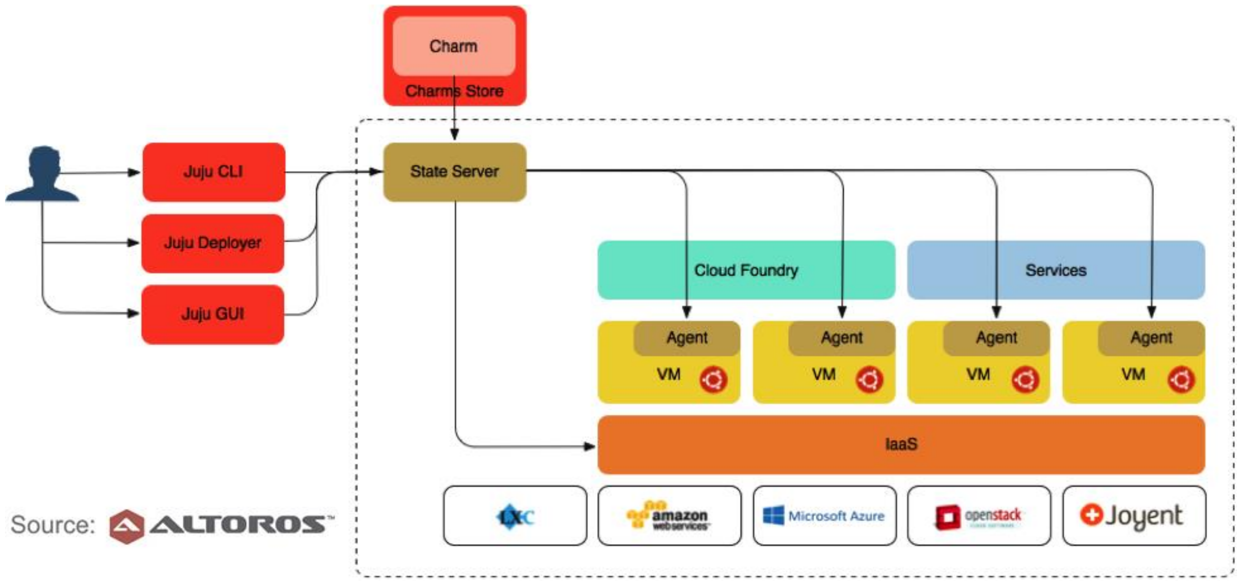


Figure 3.10: Example Juju

Represented in Figure 3.11, there is an example of the deployment of an IaaS service. The Controller send all the instructions to the State Server that download from the cloud (Charm Store) the platform needed.

Alter that, it is possible to instantiate 4 Virtual Machines that are running Ubuntu as OS.

3.5.1 *Juju controller*

The Juju Controller is a core component inside the OSM architecture because can be seen as a lifecycle manager.

The VCA functional block is represented by the Juju Controller which deploy all the Charms and actions that are needed.

3.6 **Juju and Proxy Charms**

Referred to the OSM ecosystem, Juju software act as an orchestrator. Through the data and functionalities that are described inside Charms, can manage the VNF in terms of configuration and life-cycle management. All the resources that are needed for a certain VNF are managed by this application. This means that for what concern the MANO framework, all functionalities of the VNF Manager and Element Manager are provided by Juju.

Only instantiation and termination operations are not controlled by it.

This software is installed inside the VCA module as shown in Figure 3.12.

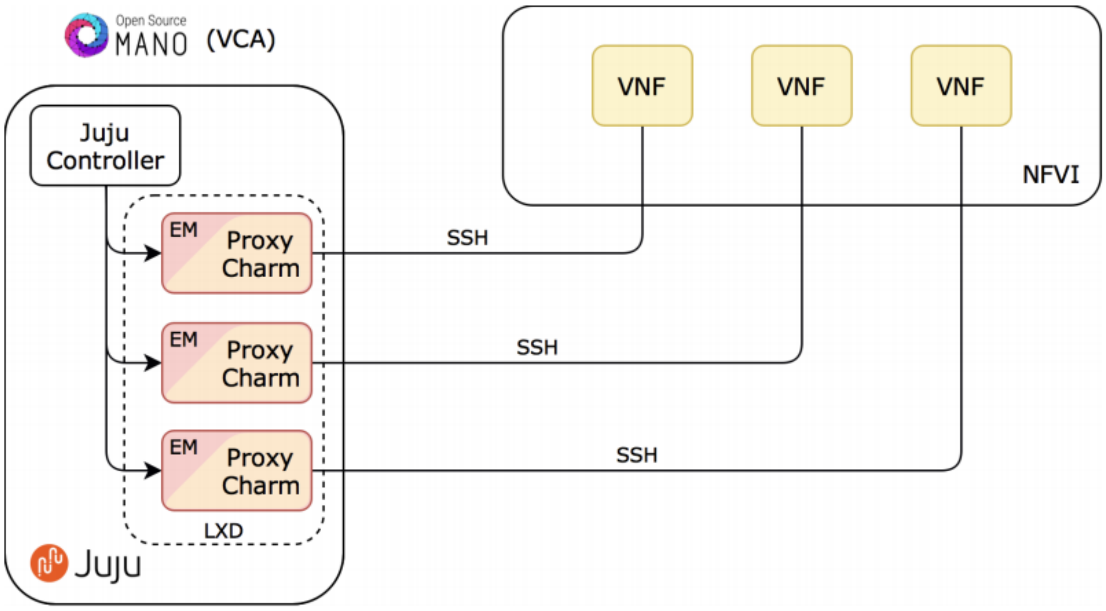


Figure 3.11: Deployment of a VNF through Juju and Proxy Charms

All the Charms are supported by the Juju software.

Charms can be seen as a piece of codes to be executed to setup a certain service required by a vendor operator.

In the real applications, both Proxy and Native Charms are used.

In case we want to develop a PNF, the operator code cannot be inside the NF. In this case is necessary to use Proxy Charms that are running outside the actual NF.

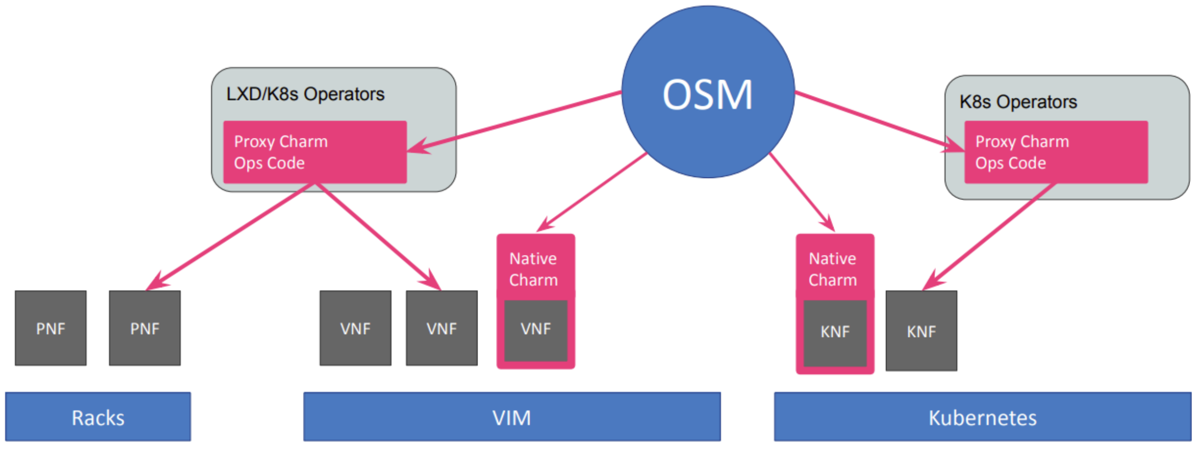


Figure 3.12: Implementation using Proxy and Native Charms

In case we want to deploy a VNF we can use both approach:

- In case the VNF is already configure with the software it needs, is possible to operate using Proxy Charms. By means of APIs offered by the VNF is possible to execute the wanted functionalities from outside.

- In case the code of the operator is in the same VNF is possible to operate by means of Native Charms.

In general, charms are installed into a container that is responsible for day-1 and day-2 configuration. These actions are typically executed remotely via ssh.

All the commands and steps that are needed for the deployment of the VNF are transformed into Juju actions.

Juju actions are the set of scripts and commands that need to be executed to provide the wanted service.

OSM primitives are Charm Action scripts. Each primitive is a charm action script that produce an output given some parameters in input.

Generally speaking, the functioning of Juju Actions is similar to that one of a software automation tools (like Ansible). In the case of Juju all the action parameters are defined inside a YAML file called *action.yaml*.

The following primitives represents Charm actions scripts:

- actions: list actions defined for a service.
- run-action: queue an action for execution
- show-action-output : show output of an action by ID
- show-action-status : show status of all actions filtered by optional ID
- Backup
- Monitor
- Debug
- Add users, policies, rules, etc.
- Manage certificates, keys, etc.
- Rotate logs.

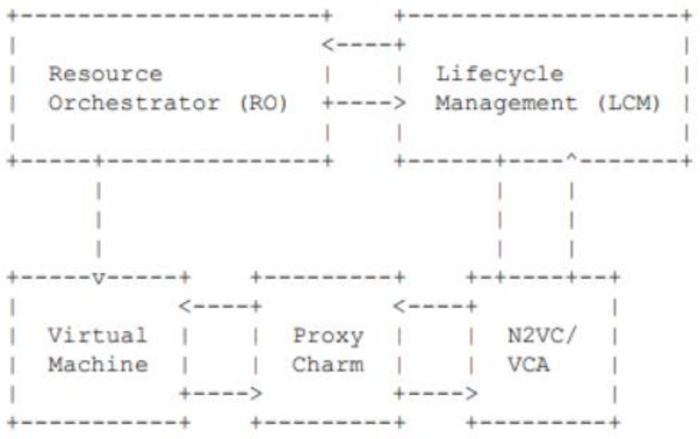


Figure 3.13: OSM workflow for the deployment of a VNF [20]

In Figure 3.14 is shown an example of how a proxy charm fits into the OSM workflow:

- A VNF package is instantiated via the LCM.
- The LCM requests a virtual machine from the RO.
- The RO instantiates a VM with your VNF image.
- The LCM instructs N2VC, using the VCA, to deploy a VNF proxy charm and tells it how to access your VM (hostname, username, and password).

3.7 **Virtualized Network Function Descriptor (VNFD)**

In the previous chapter I have discussed about the role of Descriptors.

Now I want to examine more in detail what is the role of the VNFD in the OSM architecture.

Generally speaking, a VNFD is a component that has inside:

- Metadata
- Charms

The integration between these two, allow the definition of some *actions* and *primitives*.

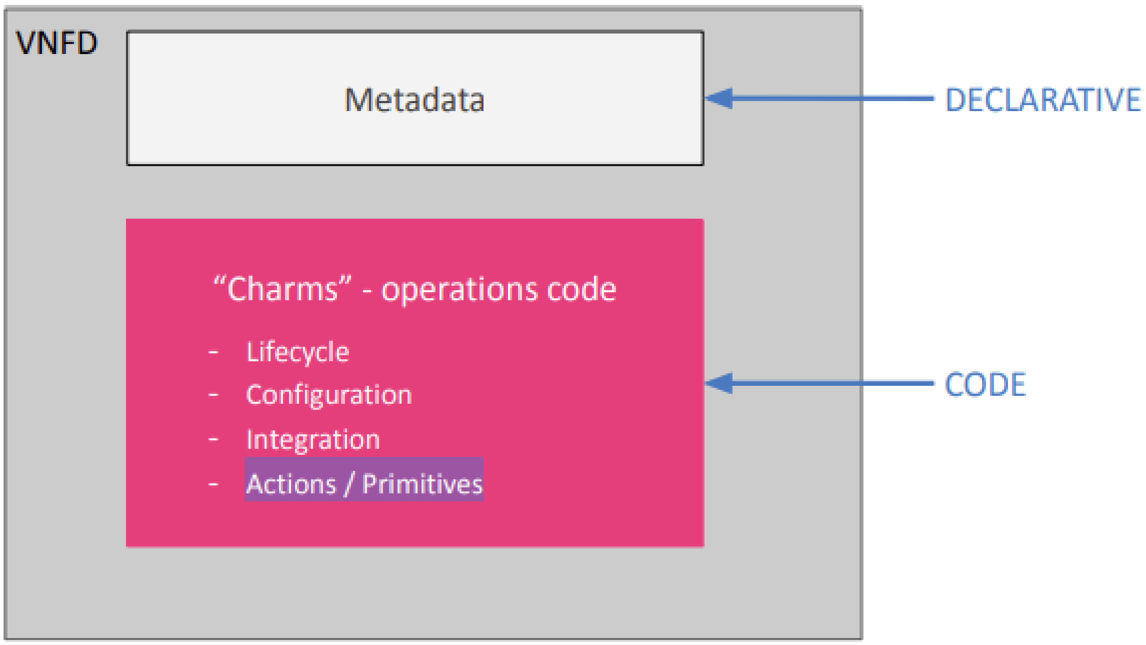


Figure 3.14: VNFD Structure

Inside the metadata are contained all the description of the network service like connection points, VDUs etc.

Inside charms instead is contained all the code for the real deployment of the NF.

Primitive are actions exposed by the operator. In other world, *OSM primitives are actions exposed by the Charm*.

3.8 **VNF Workflow**

Juju and cloud-init software have to work together for the management of the VNF configuration.

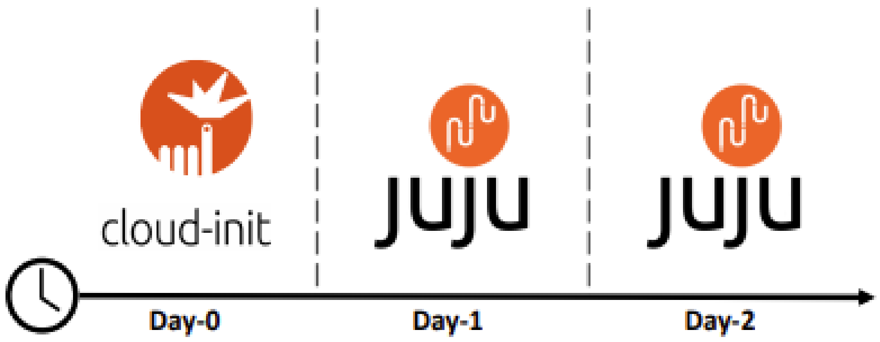


Figure 3.15: VNF steps

Starting from the VNF Package (which contains all the details of the service that we want to deploy), the main steps that we have to follow are:

0. Onboarding Requirements
1. Day 0: VNF Instantiation & management setup
2. Day 1: VNF Services initialization
3. Day 2: VNF Runtime operations
4. Termination

3.8.1 *On-boarding*

In this stage we have the validation of the descriptors of the NS. If they pass the validation of the OSM dashboard, it means that they can be used for the deployment of the service ad are added to the OSM catalogue.

As previous mentioned, for each NS we need two descriptor packages:

-VNFD packages

-NSD packages

3.8.2 *Day 0*

At this phase we have a first initialization of all the VNFs. By using all the packages that are stored in the catalogue during the on-boarding phase, it is possible to create an initial configuration for the VNF. All these configurations are used by the VDUs for the first deployment of the VNFs.

All the cloud-init scripts, functions and parameters that are needed for the deployment of the service that is described inside the descriptors are aggregated.

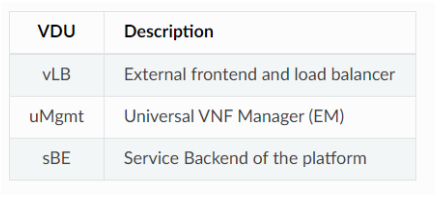
At this step, the VNFs that are created do not correspond to the final one. They are just a “first release” that will be modified in the following steps. All functions and scripts identified by the cloud-init software can be changed or adapted in later stages by different infrastructures.

Just for completeness, I want to mention that in the last release of OSM, native Charms can be used during the Day-0 configuration replacing the cloud-init scripts.

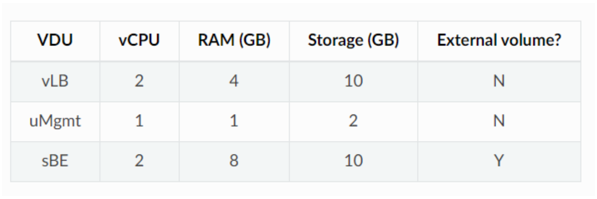
3.8.2.1 Requirements

In all these stages we have some minimum requirements that need to be satisfied.

Not all the NSs require the same amount of resources. Depending by the VNF that we need for the realization of the NS, we need some physical and hardware resources that must be present in the infrastructure.



VDU	Description
vLB	External frontend and load balancer
uMgmt	Universal VNF Manager (EM)
sBE	Service Backend of the platform



VDU	vCPU	RAM (GB)	Storage (GB)	External volume?
vLB	2	4	10	N
uMgmt	1	1	2	N
sBE	2	8	10	Y

Figure 3.16: VNF Requirements

In Figure 3.17 is reported an example that shows the definition of requirements of different VDUs (VDU is main function of every VNF component).

3.8.3 *Day 1*

In this phase we have the first real deployment of the service. At the end of this stage, all the services defined inside the VNF are automatically initialized.

All the steps that are necessary to do that, are defined inside a proxy Charm that need to be properly created for the scope in a Linux machine. In this way it is possible to provide all the steps and instructions that are needed for the creation of the VNFs that are required by the NS.

As example some actions that can be install packages, edit config files, execute commands, etc.

At the end of this step the machine is configured for providing services

3.8.3.1 Requirements

At this stage, requirements are:

- Identifying dependencies between components.
- Defining the required configuration for service initialization.
- Identifying the need for instantiation parameters: Non all the parameters have the same importance. Some of them are more important than others and can be requested immediately during the Day-1 phase, while others can be requested after. For this reason, it is important to identify those parameters that are require immediately for the instantiation of the service.

3.8.4 *Day 2*

The scope of this phase is to provide all the elements and components in the VNF package to allow the re-configuration of the VNF. In this way its behaviour can be modified during runtime, being able to monitor its main KPIs, and running scaling or other closed-loop operations over it.

To achieve that level or runtime reconfigurability, the OSM platform creates a Proxy Charms and include it in the descriptor.

Some on demand actions are performed like dump logs, backup mysql database, etc.

3.8.4.1 Requirements

At this stage, requirements are:

- Identifying dependencies between components: It is necessary to identify if a certain VNF component need to access to a specific parameter that is coming from another component. This is essential to understand what is the correct workflow that permit the runtime operations.

- Defining all possible configurations for runtime operations

- Defining KPIs: some relevant metrics need to be collected.
  Some examples of them could be:
  - Metrics collected from the VIM/NFVI:
    - CPU Usage.
    - Memory Usage.
    - Network activity indicators like bandwidth usage or packet drop rate.

  - Metrics collected from the VNF/RM:
    - Active users.
    - Size of the database.
    - Application status.

3.8.5 *Termination*

This is the last step on the Network Service lifecycle. All the resources that were allocated by the VIM are freed (both virtual and hardware resources).

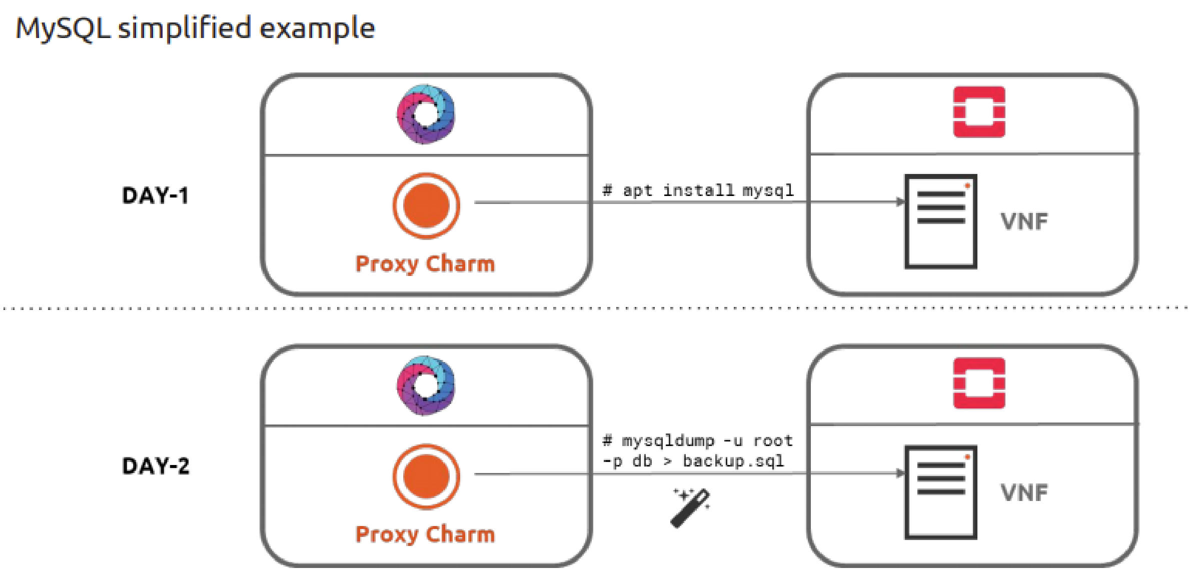


Figure 3.17: Example MySQL Day-1 and Day-2 [21]

3.9 **OSM Primitives for PNFs**

Before proceeding to the Primitive concept is necessary to analyse more in detail the difference between PNF and VNF.

PNF → Is a fixed asset and it is not possible to manage the lifecycle. We can define it as a high-cost unit.

An example of a PNFs are all the purpose build appliances, like a firewall. When you need a firewall in your network, you order it and the vendor that produce it will ship to you a piece of hardware with the software already installed on it. In this case it is not possible to have the control over that software. It is only possible to interact with the firewall by using a set of APIs which are defined by the vendor.

VNF → We can define it as a low-cost unit. OSM can create and destroy a VNF on demand. In this way it is possible to make VNFs recyclable. It is possible to instantiate VNFs depending by the service required or by the overload state of the network.

There is another category which is called Hybrid Network Function (**HNF**) which is the composition of both physical and virtual elements.

After having analysed the difference between PNF and VNF, a question naturally arises: *Does a PNF always mean pure Bare Metal?*
If we remain stick to what is written above, the answer could seem *yes*.
That is not completely true.
A PNF is a network function that is tightly coupled between the software and the hardware that it's running on.
It is a fixed function that is executed over a specialized hardware but there can be the possibility that the PNF is a Virtual Machine.
This means that a PNF can be seen as black box that is executed in a VM without the possibility to have any lifecycle control over it.

As stated before, the trend today is to try to visualize as much as possible all the network infrastructure, but we have to consider also the real-world scenario that could be messy and crowded.
There can be some situations where the ability of orchestration the full lifecycle of the PNF is unwanted.

To clarify what I mean, I'll make an example.
We take as reference a company that is quite small and has an internal network that is composed by a server and some hosts interconnected. This company requires a firewall which can protect the exchange the access and the exchange of data between the internal network and the rest of the IT network.
In this case the company is quite small and does not want to use OSM to manage the entire lifecycle of the Virtual Firewall with the risk of creating a second instance of it and perhaps mess up network firewall rules which were previously carefully defined.

This is an example just to make understand that PNFs can be Network appliances and can be executed over VMs.

Now that we have analysed the PNFs in detail, a new question arises: *VNFs are always a VNF or a Container?*
The answer is *NO*. VNF does not always mean running in VM or a Container.

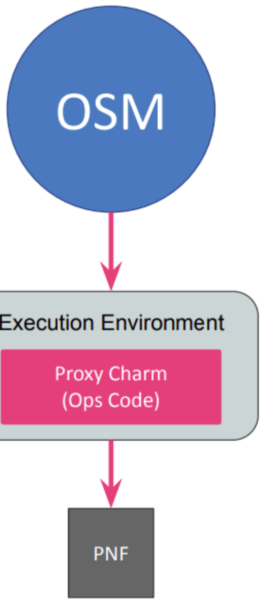
Inside OpenStack (a software which will be treated in a next Chapter) there is a functional block called *Ironic* which takes bare-metal servers which are previously commissioned (can be seen as black boxes) and it exposes them to OpenStack as if they were VMs.
From an external point of view, it looks like a VM. Actually, Ironic is taking the image that was stored in OpenStack and copy it to the physical disk and boot the bare metal server based on that image.
The bare metal server can be seen as a sort of VM but is running it without a hypervisor.

At the end of the day, is possible to understand that real world networks are messy.
It is possible to have VMs running on bare metal, but also PFNs running in virtualized environments.

- In a PNF the software is already installed.
- In a VNF OSM has the role to install the software in a new VM that is created on demand for that software to live with it on runtime.

The role of OSM is to manage this complexity. From the point of view of OSM, there are no difference between VNF, or PNF or HNF.

Now that we have understood what the real difference between VNF is and PNF is possible to analyse more in detail how OSM manage the lifecycle of PNFs.
When OSM need to create a PNF it simply creates a model (execution environment) of the network service. After launching the model which represent the PNF, we can communicate with it as if it were a VM that OSM launched.
This execution environment could be in LXC Container or Kubernetes pod. Inside this environment the proxy code is injected. This code can communicate with the PNF.
This approach is simply building up an environment where the code can act over the PNF but keeping it pure without any modification.



3.10 **The role of Kubernetes inside OSM**

Kubernetes can be seen as a system which provide some APIs which can be used for the management of the lifecycle of NFs.

Those APIs can be seen as a high-level service's objects which are modelled with K8s manifest files in YAML format. Those objects can be quite complex, and it is common the creation of packages which contain all the necessary software.

Two types of package format can be present:

- **Helm Charts**: Indirect call to the K8s API via helm
- **Juju charms and Bundles**: Indirect call to the K8s API via Juju.

The major functionalities offered by these high-level object components can be:

- *Pod sets*: Deployment of containers (called Pod in K8s) either stateless or stateful.

- *Services*: ClusterIP. NodePort, LoadBalancer

- *Storage of persistent volumes*.

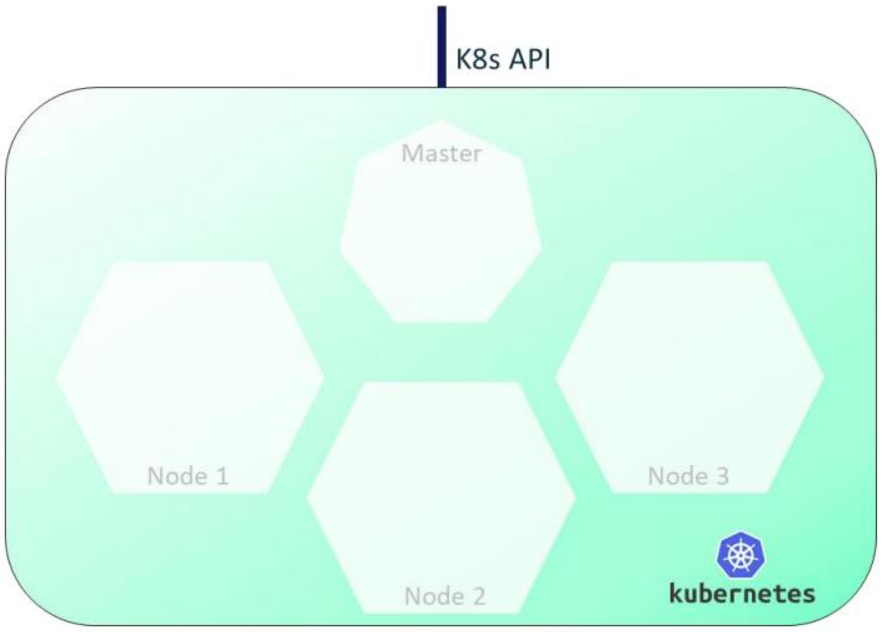


Figure 3.18: K8s APIs

From the Figure 3.19, it is possible to see that in the K8s architecture there can be some Master nodes which offer the APIs previously mentioned.
There are also a set of nodes (workers) which are running in the workload, like containers.
To run a service on K8s is necessary to run a K8s Cluster which can created in different ways:

- Standalone: Using some specific software like OpenShift, Charmed K8s or Ericsson CCD.
- Public cloud: Is possible to use the APIs of a public cloud like AWS or VMware Cloud PKS to deploy a cluster.
- Can run over a Bare Metal or in VMs running in a VIM.

After the cluster is created, independently by the way it is done, it provides all the K8s APIs. The full catalogue of K8s objects is entirely incorporated.

- All the Helm Charts are stable applications that are ready to be used.
- Juju bundles are widely used for inter-object configuration.

In OSM there are different kind of deployment unit which are used for the composition of the wanted Network Function:

- **Virtual Deployment Unit** (**VDU**): Are VMs
- **Physical Deployment Unit** (**PDU**): Are Physical nodes
- **Kubernetes Deployment Unit** (**KDU**): Are K8s applications.

K8s, like OSM is a model-driven platform. To be used and configured, are necessary some files or descriptors which contains all the data and steps necessary for the deployment of the K8s applications.
Inside each descriptor, there are some sections where is possible to define some fields related to the network function.
In general, a network function can require:

- KDUs that are based on helm charts or juju bundles.
- K8s cluster requirements
- Link cluster networks to external connection points

All the OSM lifecycle operations can be done also by K8s applications as shown in Table 2.

OSM operations	**K8s**
NS instantiate	Install
NS primitive	Update and rollback
NS termination	Delete

Table 2 : Translation OSM primitives

3.10.1 *Association of K8s cluster to a VIM*

Every time we deploy a Network Function or a Network Service, we need to specify the VIM where they are going to be deployed.
To do that we can have two possibilities:

1) The K8s cluster is running inside the VIM itself, as shown in Figure 3.20.

2) The K8s cluster is outside the VIM. In this case all the nodes are interconnected though an external network to the VIM, as shown in Figure 3.21.

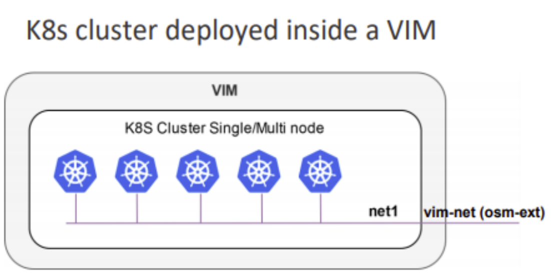

K8s cluster deployed inside a VIM
VIM
K8S Cluster Single/Multi node
net1
vim-net (osm-ext)

Figure 3.19: K8s cluster inside the VIM

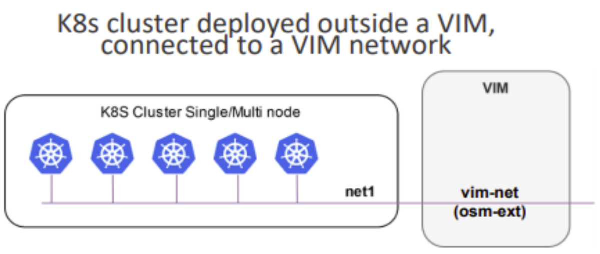
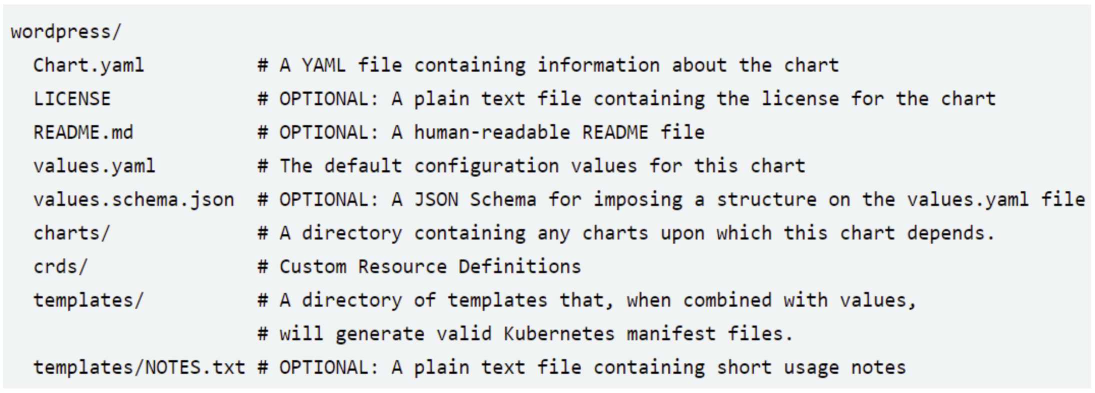

K8s cluster deployed outside a VIM, connected to a VIM network
K8S Cluster Single/Multi node
net1
VIM
vim-net
(osm-ext)

Figure 3.20: K8s cluster outside the VIM

3.10.2 *Helm charts*

Kubernetes uses the helm-chart concept to manage and deploy applications.
Helm a package manager for K8s applications which uses a package format called *Charts*.
A *Helm-chart* is a packaged format for deploying of K8s applications.

All the packages that are available for K8s applications are stored inside some repositories. The most used repositories are called *Stable*, *Incubator* and *Bitnami*.
All the updated repositories can be found at [22].



```
wordpress/
  Chart.yaml          # A YAML file containing information about the chart
  LICENSE             # OPTIONAL: A plain text file containing the license for the chart
  README.md           # OPTIONAL: A human-readable README file
  values.yaml         # The default configuration values for this chart
  values.schema.json  # OPTIONAL: A JSON Schema for imposing a structure on the values.yaml file
  charts/             # A directory containing any charts upon which this chart depends.
  crds/               # Custom Resource Definitions
  templates/          # A directory of templates that, when combined with values,
                      # will generate valid Kubernetes manifest files.
  templates/NOTES.txt # OPTIONAL: A plain text file containing short usage notes
```

Figure 3.21: Helm chart file structure

Figure 3.22 reports all the structure of an Helm Chart.
There are some fields inside the structure that are important:

- **Chart.yaml**: is a sort of README file that contains all the information about the Helm Chart.

- **Values.yaml**: contains all the values of the variable that are needed for the correct deployment of the application. Those values can be updated to create real-time changes inside the application.

- **Templates**: is a directory that is necessary for the creation of the K8s manifest file.

3.11 **Summary OSM and concepts**

In this chapter were introduced many concepts and arguments. This final section contains a sort of summary of how the OSM platform works.

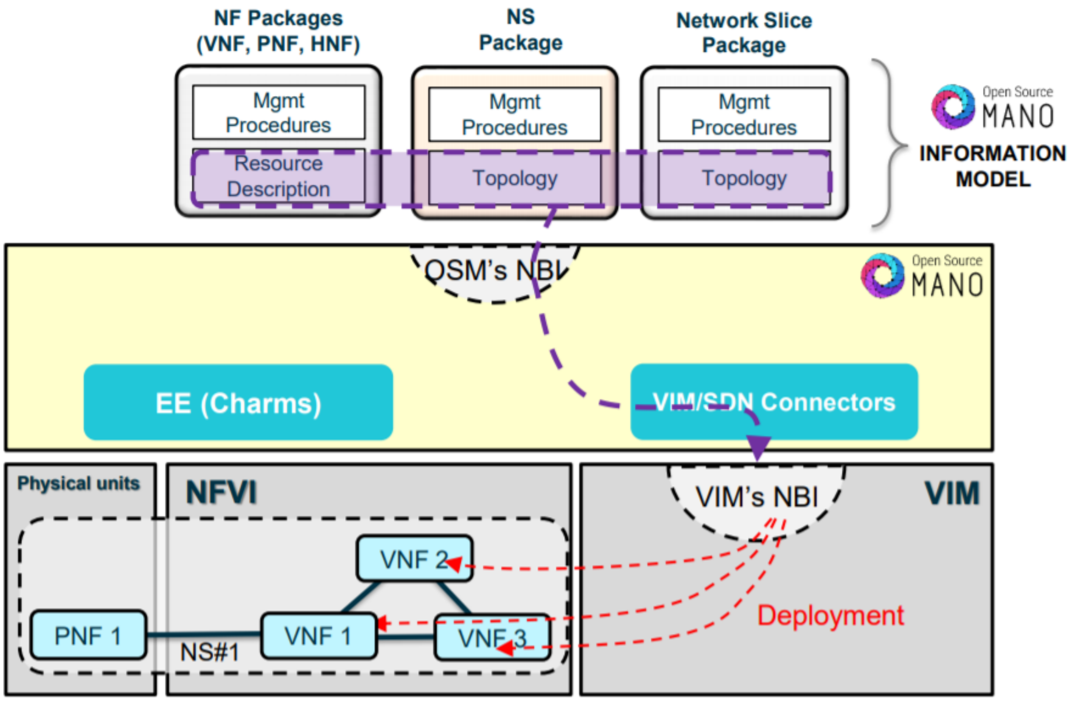


Figure 3.22: OSM packages

OSM provides a platform to create Naas and to manage them.

From this platform, we can create Network Functions or Network Services or Network Slices by aggregating multiple NSs.

To achieve this result there are some files, called packages, which contains all the information and resource descriptors (written in YANG language – SOL006) about the three services previously mentioned: NSs, NFs and NSI.

This the reason why OSM is called model-driven platform. It allows to make VNFs and NSs as portable and reusable as possible.

Those packages are provided by the vendor and fully described the function:

- Topology
- Parameters
- Actions for Day-0, Day-1, Day-2.

All packages just describe the service from a functional point of view. They do not care about the target infrastructure or the hardware components on the top if it they would be deployed.

Thanks to a set of VIM connectors, OSM can deploy the device described in the descriptors inside a NFVI. This phase is called instantiation and deployment of the services.

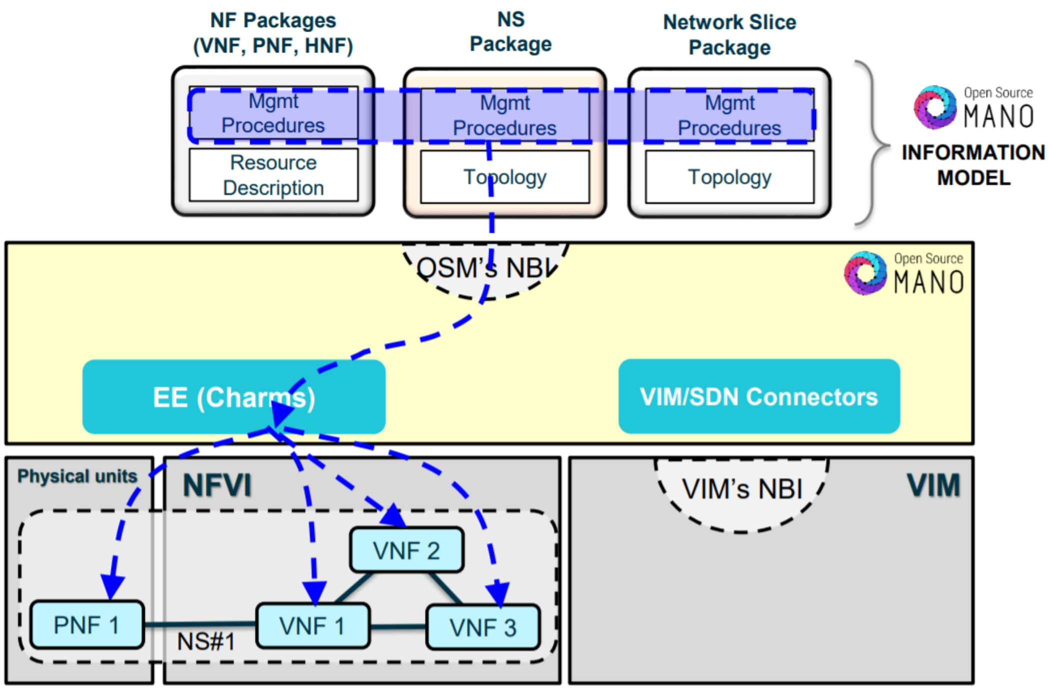


Figure 3.23: OSM actions and procedures

In addition to the information contained in the packages, OSM require some actions to be performed over the VNFs that are instantiated. All these actions are performed by means of Charms.

This is almost all the theory that is behind the OSM model.

During the development phase of a Network Service, we can identify 3 different stages:

1) **Design time**

   Is the stage where are build all the NF and NS descriptors, but also all Day-1 and Day-2 logic to complete all the packages.

2) **Provisioning time**

Is where the onboarding happens. All the packages are onboarded into the OSM platform.

3) **Run Time**

Is the time where being possible to see in action the VNFs and NSs.

Two phases can be identified:

- *Instantiation*: Is the phase where is required the choice in which VIM to deploy the NS and which instantiation parameters to use.

  All these tasks are performed by Day-0 and Day-1 operations.

  - Day-0: Is the minimal configuration that allow to a VNF to be reachable through a management interface.
  - Day-1: Action that are done once the NF is reachable.

- *Operation phase* (Day-2)

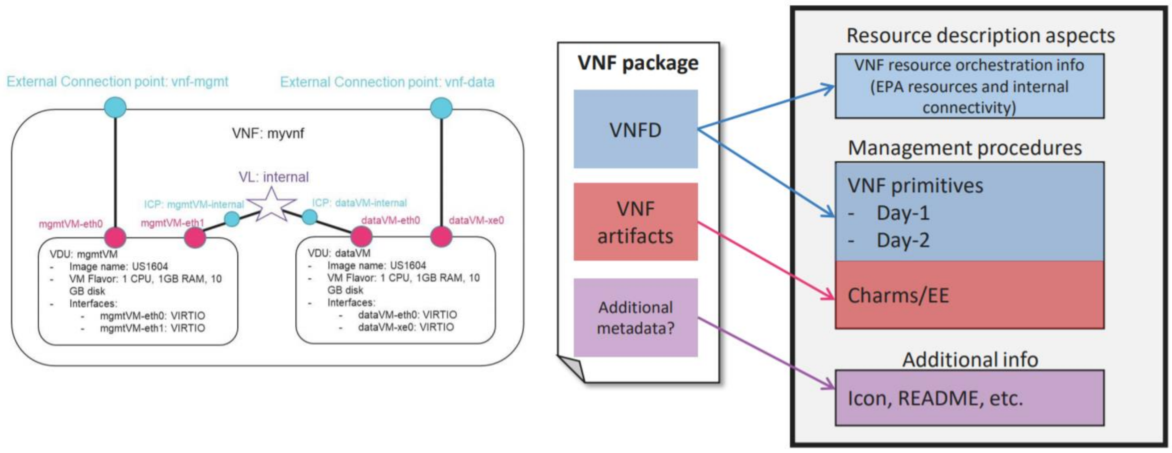


Figure 3.24: Example modelling a NF through a NF package

3.11.1 *Modelling NFs though packages*

As previously mentioned, a package contains all the information for the correct deployment of the NS or NF.

A NF package is mainly composed by two elements:

1) **Descriptors**: are written in YAML

2) **Other files**:

- Charms
- KDU objects: Juju bundles and helm chart
- Day-0 configuration files for cloud-init
- Checksums

An example of that structure is represented in Figure 3.25.

CHAPTER 4: MEC PLATFORM

In the previous chapters I have introduced the principal concepts of the NFV and MANO frameworks.
In this chapter is presented the Multi-access Edge Computing (MEC) concept. It can be one of the possible solutions that allow to achieve the requirements previously mentioned.
MEC initiative is an Industry Specification Group (ISG) within ETSI and can offer application developers and content providers cloud-computing capabilities and an IT service environment at the edge of the network.

In this thesis, MEC platform functionalities are used with the aim of unifying the orchestration of heterogeneous fog and edge resources. In this way all the advantages of a MEC-based solutions can be used for the development and the deployment of IIoTaaS applications.

4.1 **Introduction**

The MEC Platform is an environment where MEC applications can register and advertise their services.
It can be seen as a DNS proxy/server that contains all the endpoints of all the MEC applications that has registered their services.
The platform can also advertise and offer MEC services to other MEC applications that request endpoint information how to reach certain services that were previously registered to it.
All the reference points between the MEC platform and the MEC applications, as defined in ETSI GS MEC 003 [23].

Some of the most import functions that are implanted by the platform are:

- MEC service assistance:
    - Authentication and authorization of producing and consuming MEC services.
    - A means for service producing MEC applications to register/deregister towards the MEC platform the MEC services they provide, and to update the MEC platform about changes of the MEC service availability.
    - A means to notify the changes of the MEC service availability to the relevant MEC application.
    - discovery of available MEC services.

- MEC application assistance:
    - MEC application start-up procedure.
    - MEC application graceful termination/stop.

- Traffic routing: traffic rules update, activation, deactivation.

- DNS rules: DNS rules activation and deactivation.

- Timing: provides access to time-of-day information.

- Transport information: provides information about available transports.

4.2 **Multi-access Edge Computing framework**

Multi-access Edge Computing is the software entity that allow to all the MEC applications to run over a virtualized infrastructure.

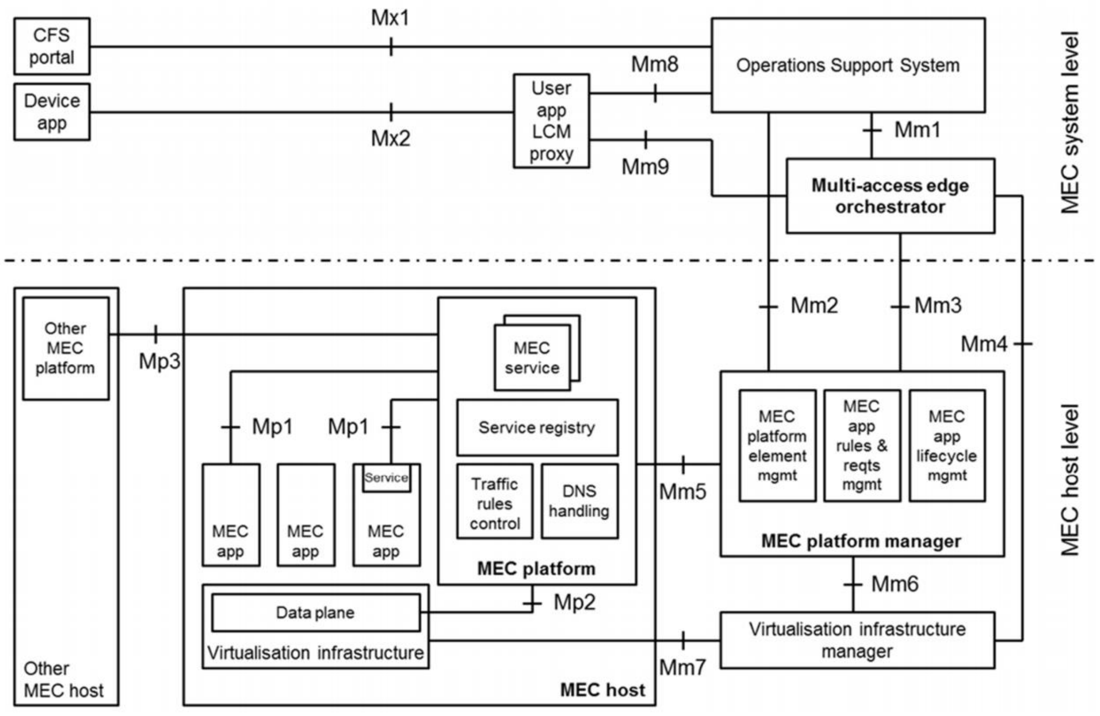


Figure 4.1: MEC system reference architecture

In Figure 4.1 is represented the MEC framework. All the entities can be grouped into two groups: **System level** and **Host level**.

Inside those two levels, there are some functional blocks that I’ll briefly comment how they are organized from the architectural point of view, because some of them are already described in Chapter 1 and Chapter 2.

4.2.1 *System level*

- ***Multi-access Edge orchestrator (MEAO)***: is the core of the entire MEC system level because is responsible to maintain an overall view of the MEC system.
  Relies on the NFV Orchestrator (NFVO) for resource orchestration and for management of the set of MEC application.
  In particular, the main functionalities of it are:
    - On-boarding of application packages (checking the integrity and authenticity of them, validating application rules and requirements etc.)
    - Depending by the application constraints and available resources, selects the appropriate MEC host for the instantiation of the applications.
    - Management of instantiation and termination of application

4.2.2 *Host level*

- ***MEC Host*****:** Is the entity which offers all the resources (compute, storage, and network) for the handling of the Virtualization Infrastructure (VI) and the MEC platform itself. In particular, the VI executes all the commands received by the MEC platform. This is made through the VI's data plane that routes the traffic to the corresponding service or application.

- ***MEC Platform****:* This term refers to all the services and functionalities that are required to all the MEC applications that are running on a certain VI.
  In particular, the main functionalities that this entity is in charge of, are:
    - Setup the environment where all the MEC applications can find or discover alle the MEC services that are provided by the platform.

    - Management of the traffic rules among all the different MEC applications. This is done by sending some commands to the data plane.

- ***MEC Applications****:* All the MEC applications that need to be executed are instantiated through virtual machines that are running inside the VI of the MEC host.

  All these applications have some requirements and rules that need to be respected (latency, resources etc). All these requirements need to be respected end validated by the MEC system level.
  In case some requirements are missing, they can be set to default values.

- ***MEC Platform manager*****:** Has the role of the management of the application's lifecycle. To do that, controls and manages all the applications requirements and rules.
  It is also responsible to receive from the VIM all the performance measurements to process them.

- ***Virtualization Infrastructure Manager***: I mention this entity just for completeness. The working principle of the VIM is already described in Section 2.1.1.

4.3 **MEC architecture in NFV**

The MEC concept can be overlapped to that one of NFV. From the point of view of the ETSI MANO framework, the MEC platform is deployed as a VNF.

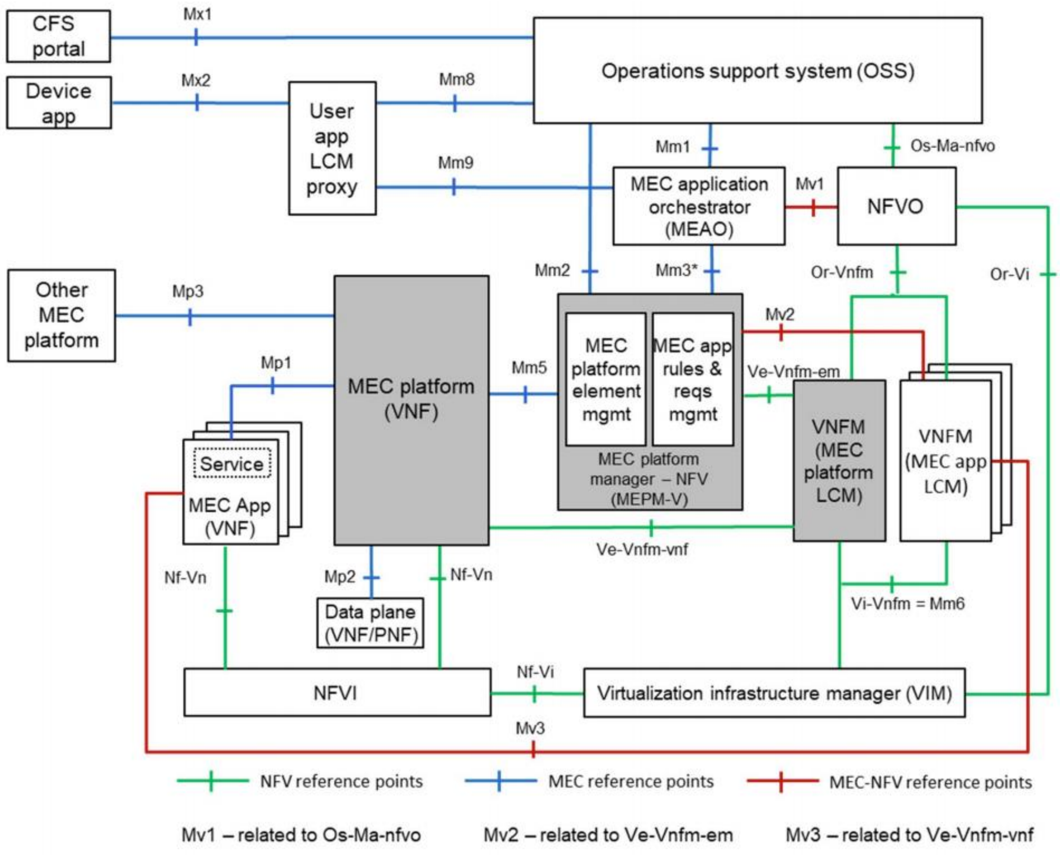


Figure 4.2 : MEC reference architecture

In Figure 4.2 is shown the MEC architecture. We can see it as an extension of the previously described MANO architecture because many components are the same and are previously described.

All the reference points inside the architecture are described in the reference RGS/MEC-0003v211Arch approved by ETSI. The document can be found at [24].

CHAPTER 5:
OPENSTACK SOFTWARE

So far there has been talk of the OSM software that is responsible of the management of the VNFO and the VNFM.

The MANO framework also includes the VIM. For the management of it a new software is required. In this case OpenStack software comes to our aid.

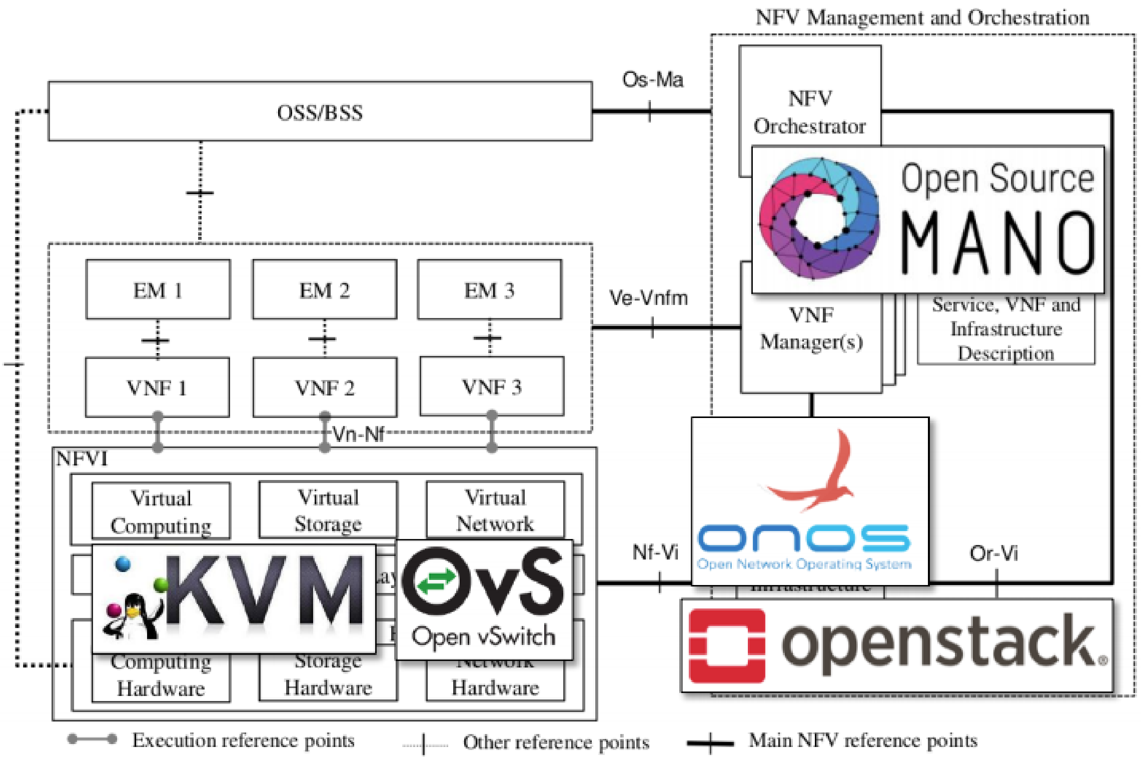


Figure 5.1: ETSI Architecture Services and Vendors

5.1 **OpenStack**

OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacentre. It is possible to see it as Infrastructure as a Service (IaaS) where a set of cloud resources are all managed and provisioned through Application Programmable Interfaces (APIs) with common authentication mechanisms [25].
Resources and functionalities are abstracted by means of virtualization techniques (like hypervisors, LXC or VMS).

It is an open-source software that is widely used in all the Infrastructure as a Service projects.
GSMA and Linux Foundation work together for the definition of the Common NFVI Telecommunications Taskforce (CNTT). The scope of the CNTT is to identify all the properties of a Telco Cloud. They identify OpenStack as the reference architecture form the management of virtual machines.

It is a very powerful tools that is widely used in Network Function Virtualization. In our case-study the management system offered by this software can be used inside the MANO Framework for the control of the cloud infrastructure. At the end of the day, we can say that OpenStack software acts as a VIM.

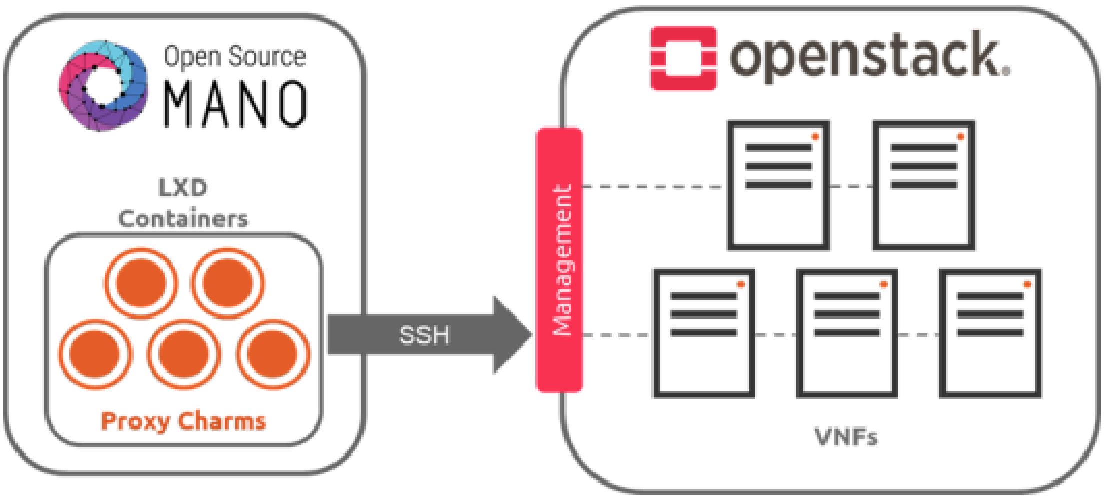


Figure 5.2: OpenStack as VIM inside the MANO architecture [26]

5.1.1 *OpenStack to deploy a VNF*

From the introduction section of this chapter, it is easy to understand that OpenStack software is used by OSM every time a new VNF (which contains some code to be executed like Native or Proxy charms) needs to be deployed.

As reference example suppose that a Proxy charm needs to be executed. This charm contains all the code necessary to create the wanted NF.
To do that, OSM tells OpenStack to launch a new VM based un a basic Ubuntu image installing only shh and some essential packages. All the code contained in the Proxy charm is executed inside this VM.

Now is possible to undusted better what is the role of a proxy charm.
A charm has the role to perform operations on a VNF and install the needed software on demand. The base image of the VM (Ubuntu in this example) can be changed at any time.

5.2 **OpenStack Logical architecture**

The logical architecture of the software [27] can be seen as an interconnection of a huge number of independent blocks, that are called *OpenStack Shared Services*.
All these running processes, need to communicate each other to build up the required service.

To do that, Advanced Message Queuing Protocol (AMQP) message broker is used.

This means that the list of services with the correspondent state is stored inside a database. Depending by the service that is required, the broker can choose those ones that are needed.
There is an administrator (controller node) that has the role to mage all these services. To distinguish between them, each one authenticates itself by means of an Identity service (that work in an independent way with respect to all the other).
For each service there is one or more services/processes/nodes that are running in the background.

In particular, there can be three types of nodes:

- *Computing nodes* → Represent the real computing and storage capability.

- *Network nodes* → Are responsible of the network configuration and management.

- *Storage nodes*

An API process is running, waiting for API requests. In case a request is receive, it is immediately forwarded internally to the related service to be accomplished.
From the user view point it is very easy to interact with OpenStack because the software provides a web-based user interface (using Horizon Dashboard). The user can interact with the Dashboard by inserting some commands or issuing API requests through tools like browser plug-ins or url.

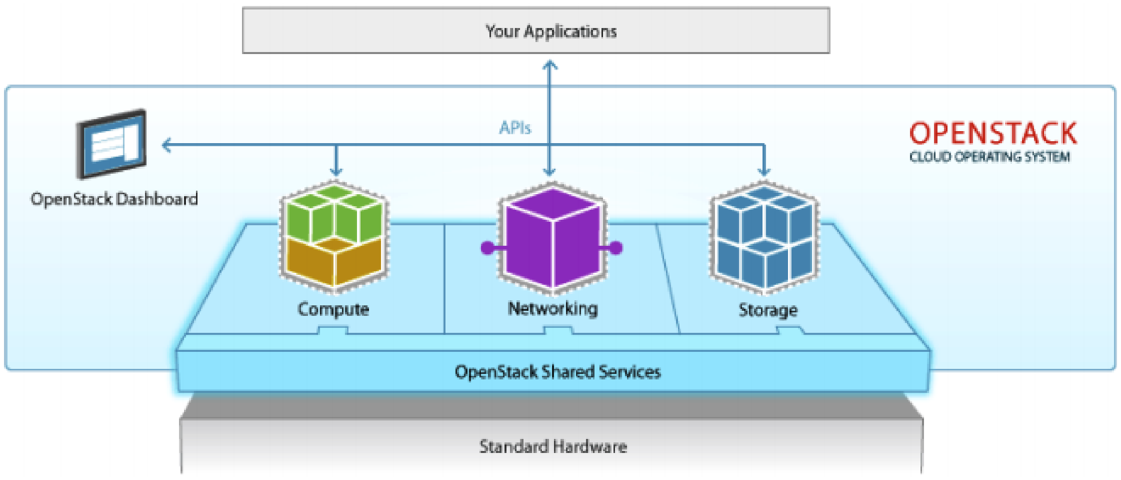


Figure 5.3: OpenStack logical architecture

The functionalities that OpenStack software can provide are:

- *Self-service*:

  The OpenStack cloud service that is offered should be able to auto-configure itself without the need the human interaction.

- *Bidirectional Compatibility*:

  All the software components should be compatible independently from the version of the cloud architecture that is used.

- *Cross-Project Dependencies*:

  All the operators have to choose as much as possible well-known functionalities that are already presented and used. In this way is possible to focus the attention of few software and functionalities and keep them updated. In this way is possible to avoid a large variety of ad-hoc solution or proprietary software.

5.3 **OpenStack architecture**

Inside the OpenStack architecture, is possible to identify some functional blocks that perform all the functions that are needed.

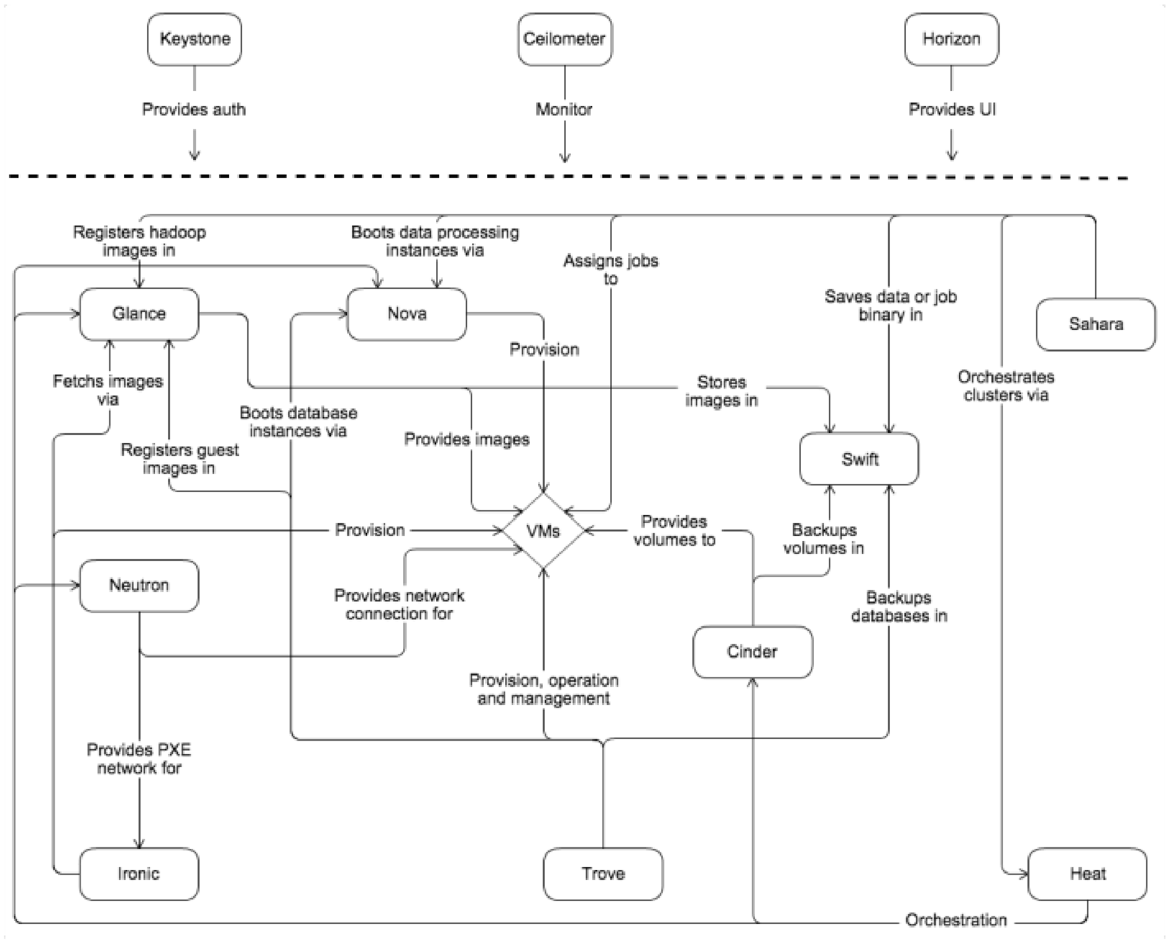


Figure 5.4: OpenStack architecture [28]

OpenStack architecture is reported in Figure 5.4.

The core components of the OpenStack's architecture are:

5.3.1 *Nova*

It is the component that is responsible of the creation and management of VMs and Containers [29]. All the functionalities that are provided by this component are executed by means of a certain number of daemons that are running on the top of Linux servers.

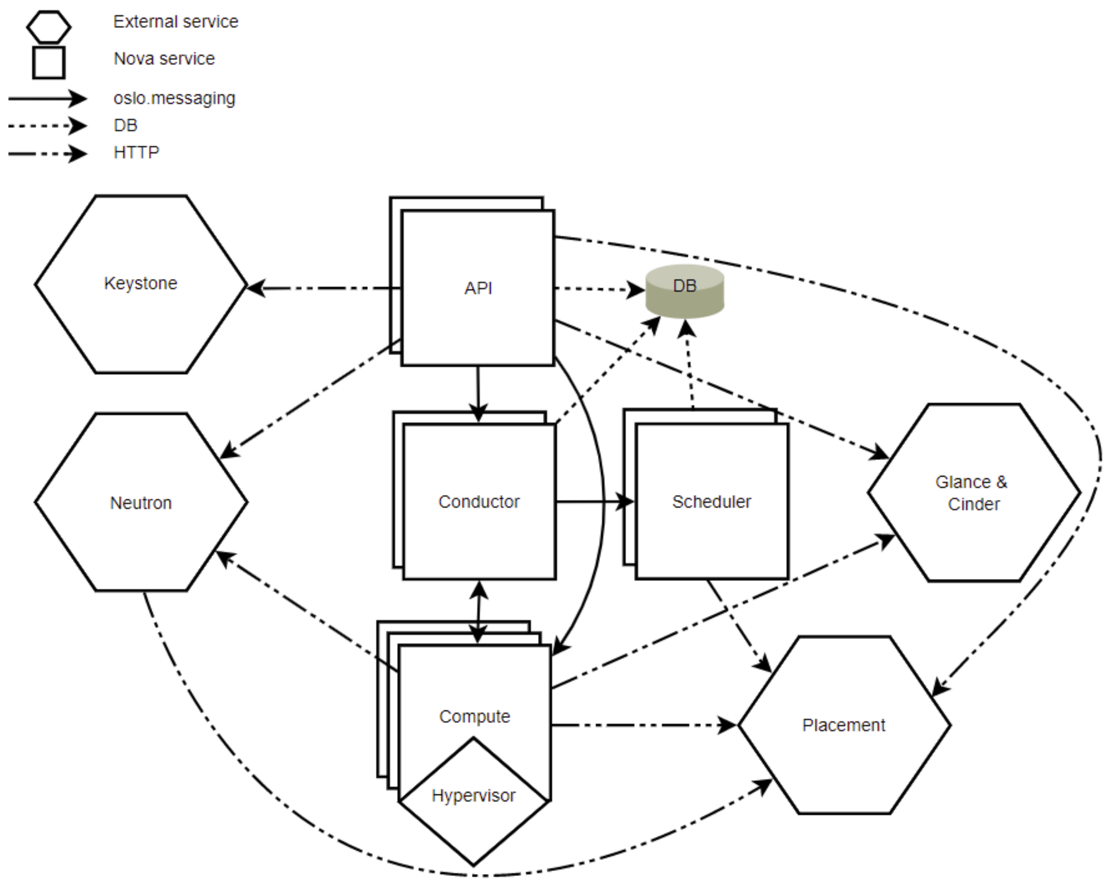


Figure 5.5: Nova architecture [30]

From Figure 5.5, we can see some of functional blocks that are working together to achieve the wanted functionalities:

- *DB*: sql database for data storage.
- *API*: component that receives HTTP requests, converts commands and communicates with other components via the *oslo.messaging* queue or HTTP.
- *Scheduler*: decides which host gets each instance.
- *Compute*: manages communication with hypervisor and virtual machines.

- *Conductor*: handles requests that need coordination (build/resize), acts as a database proxy, or handles object conversions.
- *Placement*: tracks resource provider inventories and usages.

5.3.2 *Cinder*

Is the OpenStack Block Storage service [31] for providing volumes to other components like Nova virtual machines, Ironic bare metal hosts, etc.

In particular, volumes functionalities that provided are:

- Create a volume.
- Attach a volume to an instance.
- Detach a volume from an instance.
- Create a snapshot from a volume.
- Edit a volume.
- Delate a volume.

From the user perspective, is possible to create and manage volumes using Horizon interface or directly by means of REST API.

5.3.3 *Horizon:*

Is the component in charge of providing a user web dashboard to manage and control all the components of the OpenStack architecture.

5.3.4 *Neutron*

Neutron is an OpenStack project [32] that can provide Networking as a Service (NaaS) between interfaces and devices managed by other OpenStack services [33] (e.g., nova).

To do that, a series of APIs and plug-ins are used. The logical architecture, has some functional blocks inside it:

- *Neutron-server*: Act as a controller that receives all the APIs and route them to the correspondent plug-in.

- *Networking Pug-ins and agents*: Are responsible of create networks or subnetworks with the corresponding IP addresses. Different types of plugins and agents depend by the vendor and cloud technology we are working on.

- *Messaging queue*: Has mainly two jobs to handle:
  - Route information between the neutron-server and all the agents.
  - Act as database to store information regarding networking states and plug-ins.

Neutron is also responsible of the support to the *multi-tenancy isolation*.

Multitenancy is a very important concept in cloud computing. It is referred to the case where multiple costumers of a cloud vendor need to access to the same computing resources. In case there are multiple users in a cloud infrastructure there can be the possibility that they are requesting a certain common resource. This kind of technology allows to share the resource between them all. In the meanwhile, cloud customers are not aware of each other, because the data is kept totally separated.

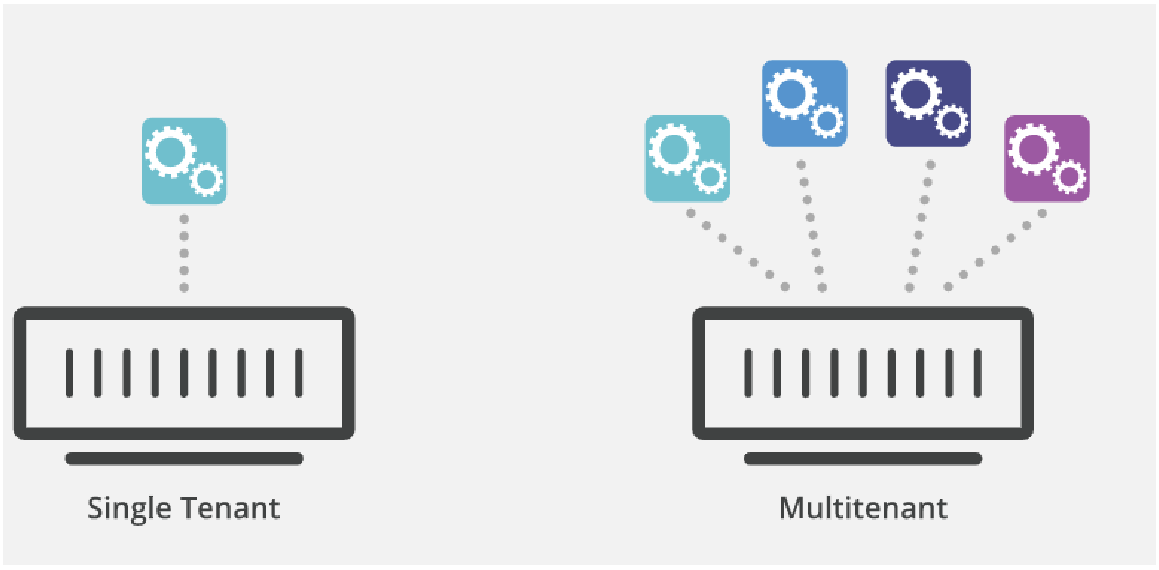


Figure 5.6: Multitenant vs Single Tenant [34]

To achieve this, are used some techniques to isolate the incoming traffic from the different vendors and redirect it to the correspondent resource. Protocols like VLAN in trunk mode or VXLAN tunnels are used to do that.

At the end of the day, all these functionalities are needed to maintain traffic isolation across Layer 3 devices.

This is also called Virtual Routing and Forwarding (VRF).

5.3.5 *Heat*

Is the component responsible of the orchestration of cloud applications [35]. This is achieved by means of some templates that can be seen as a sort of Network Descriptors which contain the necessary information for the deployment of the service.

A Heat template describes the infrastructure for a cloud application in text files which are readable and writable by humans and can be managed by version control tools.

5.3.6 *Tacker*

Is the component that behave as a VNF Manager and NFV Orchestrator in order to take control of Network services and VNFs [36].

NFV Infrastructures can be Kubernetes or OpenStack.

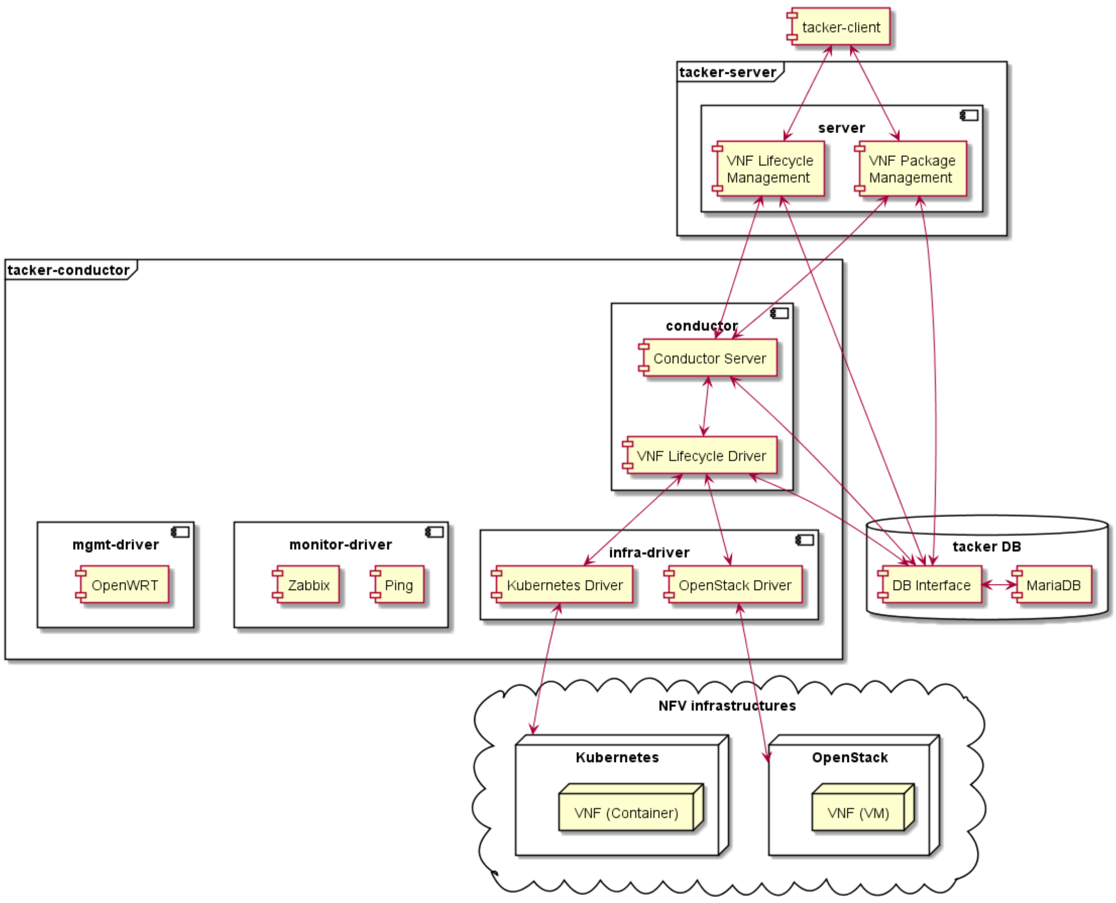


Figure 5.7: Tacker architecture [37]

As we can see from Figure 5.7 about the architecture, there are some functional blocks and packages:

- Packages:
  - **python-tackerclient** - is the package for CLI and *REST API* calling.
  - **tacker** - is the main package for Tacker project.
- Components:
  - **tacker-client** - provides CLI and communication with Tacker via *REST API*.
  - **server** - provides *REST API* and calls conductor via RPC.

- **conductor** - implements all logics to operate VNF and call required drivers providing interface to NFV infrastructures.
- **infra-driver** - is responsible for exact actions to operate OpenStack or Kubernetes resources.
- **vim-driver** - is responsible for registration of VIM.
- **mgmt-driver** - is responsible for exact actions to configure VNFs.
- **monitor-driver** - is responsible for exact actions to monitor VNFs.
- **policy-driver** - is responsible for policy based VNF operations.

5.3.7 *Glance*

It is an image service inside the OpenStack architecture [38]. Using a REST API interaction, each user can perform operations that involves images like discovering, registering and retrieving virtual machine images.

This is a central component for OpenStack because lays the foundations of IaaS. Is able to accept APIs requests but also supports the storage of disk or server images that can come from different repositories.

5.3.8 *Swift*

It is the component of the OpenStack architecture which provides multi-tenant Object Storage system [39].

Can offer a high scalability. For this reason, can manage large amounts of unstructured data.

5.3.9 *Ironic*

Ironic [40] is the functional block dedicated for the configuration of the Ironic Python Agent (IPA) for the deployment of bare metal nodes instead of virtual machines.

The bare metal concept is the opposite with respect to the virtualization one. I have previously stated that the virtualization allows to run every type of function over a general-purpose hardware.

This solution does not fit in all allocative fields. There are some specific tasks where the virtualization approach cannot be taken. Here is where the bare metal solution can be useful.

The bare metal solution provides a specialized hardware for the execution of specific tasks.

CHAPTER 6: OPC UA PROTOCOL

As previously discussed in the introductory section, industry 4.0 scenarios require new types of protocols and network architecture to be able to satisfy requirement of services.

All the interconnected intelligent devices that are installed over machines lead to the IIoTaaS concept.

This is also called Cyber-Physical System (CPS) because can combine statistics, computer modelling, and real-time data measured on physical systems. The final scope of this approach is to create a model that can be efficient and responsive under multiple working scenarios.

To do that, is necessary to deploy a distributed network infrastructure. In this new kind of architecture there are some devices in the premises of sensors that act as collectors and brokers. The intelligence and the storage instead, is localized in a cloud server that is capable of high performance in terms of storage and computational power.

Those solutions are respectively called cloud and edge computing.

In addition to that, there are all the paradigms of virtualization so far described. Resources and services are virtualized and offered as a service.

An example of a possible IIoTaaS could be the following as described in the Figure 6.1:

- **The things**: Are all the sensors that we have in the premises of machines for monitoring and control them.

- **Sensors and actuators** & **Data aggregation and Gateways**:
  They represent the Fog cloud. Are placed near machines to collect all the data coming from IoT sensors and manage all communication and commands to send to them.

- **Edge IT:** It is capable of a certain computational power for the deployment of a quick data analytics based on data coming from IoT devices. This local processing is essential for low latency services.

- **Data centre** & **Cloud IT**: Are servers with huge computational power and large amount of storage available. They can deeply analyse long-term data received from the factory premises and all the data coming from the IoT sensors.

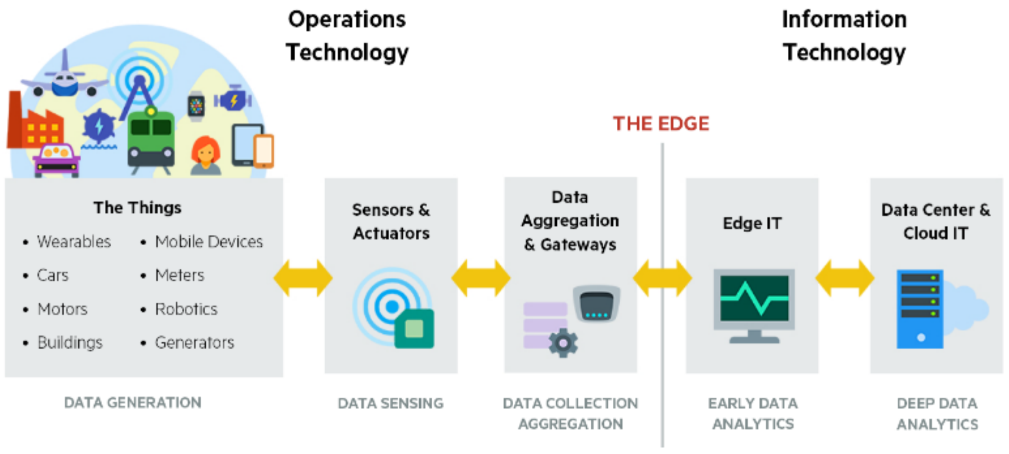


Figure 6.1: IIoTaaS example approach [41]

To implement these concepts, is required the use of an IoT messaging protocols.
Well known protocols that could fit these requirements are:

-**MQTT**: Message Queuing Telemetry Transport

-**MAQP**: Advanced Message Queuing Protocol

-**COAP**: Constrained Application Protocol

-**DDS**: Data distribution Service

-**OPC UA**: Open Platform Communication Unified Architecture

All those protocols above mentioned could satisfy requirements of Industry 4.0 and implement an IIoTaaS.

This thesis will be focused on the use of the OPC UA protocol for the implementation of a VNF service based on that protocol.

6.1 **OPC Protocol**

OPC stands for Open Platform Communication. With the term OPC we are not referring to a specific protocol but more in general we are referring to a multitude of different protocols.

I report here some history to contextualize the OPC UA protocol that is used nowadays.

In 1996 was launched the OPC Foundation and Monitoring of Industry Equipment. The aim of this project is to find a way to solve all the interoperability and communication problems that are always present between devices of different vendors, which could also have different functionalities.

This first protocol that was launched in 1996 was based over a Client/Server Architecture where an OPC-Client and an OCP-Server exchange between each other data and information.

This approach is no more usable nowadays because of several problems that I will not deeply explain in this thesis.

Just to mention, one of the big problems of this protocol is security in the flow of information between OPC-client and OPC-server (and vice versa).

6.2 **OPC UA Protocol**

The main scope of this project is to find a way to standardize the exchange of data and information inside an industry 4.0 scenario.
In 2008 the OPC UA (Unified Architecture) was released. It is possible to see it as an improved version of the original OPC Client/Server. This new release can fit all the requirements requested in the modern industrial field.
This new protocol is now based on a *service-oriented architecture* (SOA). This means that all the software components can be used and assessed through service interfaces that are using common standards.
This is a key aspect because every type of software component can be easily integrated in new applications without any problem of compatibility.
For this reason, starting from 2015 the new protocol OPC UA (Unified Architecture) was recommended by International Electrotechnical Commission (IEC) as a reference architecture model to be followed in the industry 4.0 [42].
OPC UA is a standardized client-server-architecture. A big improvement with respect to the original OPC protocol is related to the security aspect. For the reliable end to end transmission of data from the IoT (generally sensor placed inside the machine) to the decentralized cloud server, is used TCP and HTTP protocol. For what concerns the authentication process done by each client and server is used OpenSSL protocol.

The main features that are offered by this protocol are [43]:

- Secure, reliable and vendor-independent exchange of information

- **Vendor-independent platform**: The OPC-UA offers no dependency from any vendor that is producing a certain application. The communication is independent from any programming language or operative systems on which the application is executed.

- Secure TCP and HTTP as previously mentioned.

- Protection against unauthorized access: Some standards like SSL, TSL or AES are used to guarantee security.

- Accessibility and Reliability: The architecture can offer a reliable communication mechanism capable of automatic error detection (by means of redundancy functions). In this way it is possible a reliable communication between the OCP-Client and OPC-Server.

6.3 **OPC UA Architecture**

A Service-Oriented Architecture is a sort of service provider that offers a certain number of *services* that process some requests and send back the response to the client that requested it.
All these services are running inside a server. A generic client can invoke a service by means of *service calls*. Any time a client requires a new service, a new session in established and is based on a secure channel that can guarantee integrity and confidentiality for the reasons previously explained.

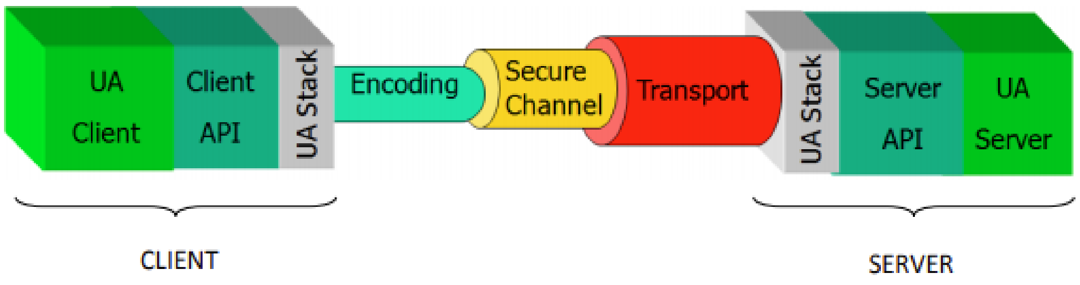


Figure 6.2: OPC UA Stack Architecture

In Figure 6.2, is represented a first overview of the stack architecture of this protocol. As previously mentioned, the OPU UA is based on a client-server architecture. In the real implementations instead, it is possible that a single application can act both as cloud and server at the same time.
The entire software stack could be implemented in many different programming languages to conciliate the vendor-independent platform concept previously mentioned.

In general, the most common programming languages used for the Software stack can be Java, C, C++ or Python.
The implementation of applications can be done through the OPC UA Software Development Kit (OPC UA SDK).

6.4 **Information modelling and Address Space**

An important aspect to a vendor-independent platform is the necessity to design an information model that can fit all the requirement from different vendors applications.
OPC UA provides only a simple infrastructure of the information model, but it is possible to adapt it by means of extensions. This means that each vendor could implement the information model (or object) that is needed inside its application. Starting from the base information model offered by the OPC UA is possible to model some extension models to obtain the wanted result.
When a certain vendor has produced its own extensions could share it to all clients that are requesting the application implemented by it.
In this way is possible to obtain an address space that allow to OPC UA-Servers to represent objects in a standard way to all OPC UA-Clients.
Those objects are container for Variables and Method as shown in Figure 6.3

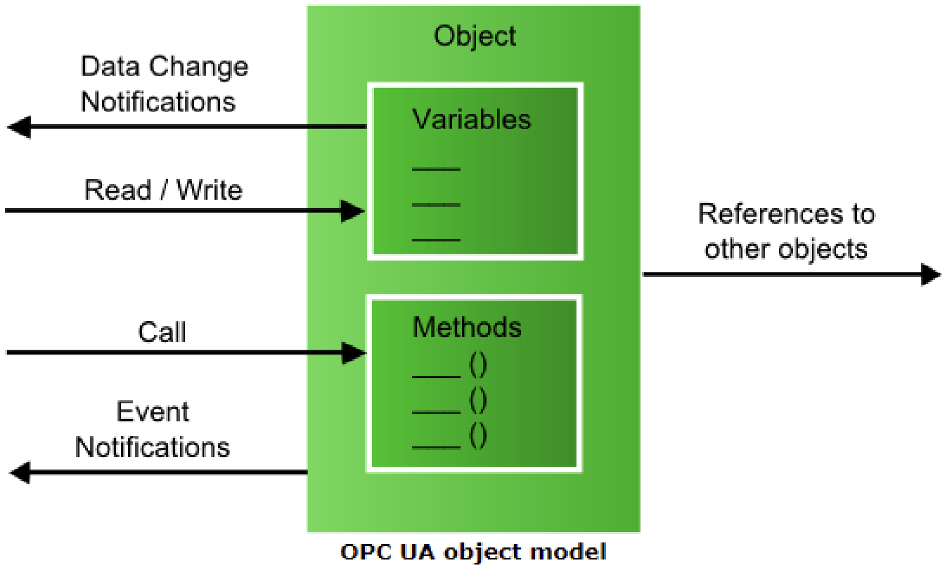


Figure 6.3: OPU UA Objects [44]

6.4.1 *Nodes*

Nodes are the base component of the information model. Objects and all components that compose them, are described in the address space by means of interconnected nodes.

Each node is identified by:

- ***node-id***: Identify in a unique way a certain node. OPC UA utilize two types of variables:
  - *name*: Is the real name of the node.
  - *namespace*: It is the URL that identify that node.

  In a next section will be developed deeply the concept of *node-id*

- ***attributes***: Are descriptors and contain all the information of the node. Attributes are different depending by the *NodeClass* of the node. (*NodeClasses* will be explained in the next section).

  Inside attribute descriptions are stored some fields that to describe a certain node:
  - *Attribute id*
  - *Name*
  - *Description*
  - *Type of data*
  - *Mandatory/optional*: Additional field that indicate if the attribute needs to be applied when the node is executed.

A client can access to attributes using services: *Read*, *Write*, *Query*, *Subscription*, *Monitored* etc.
An important aspect is related to the fact that the set of attributes of a certain node cannot be modified or extended by clients or servers. Once they are defined, they cannot be modified anymore.
All the *attributes*, *nodes* and *references* are assigned as soon a *Node* is instantiated inside the *Address Space*.

The *address space* can be seen as a sort of catalogue that contains all the information of a certain information model. This structure is stored in the OPC UA-Server.
An OPC UA-Client can access to it for gathering information about the model.

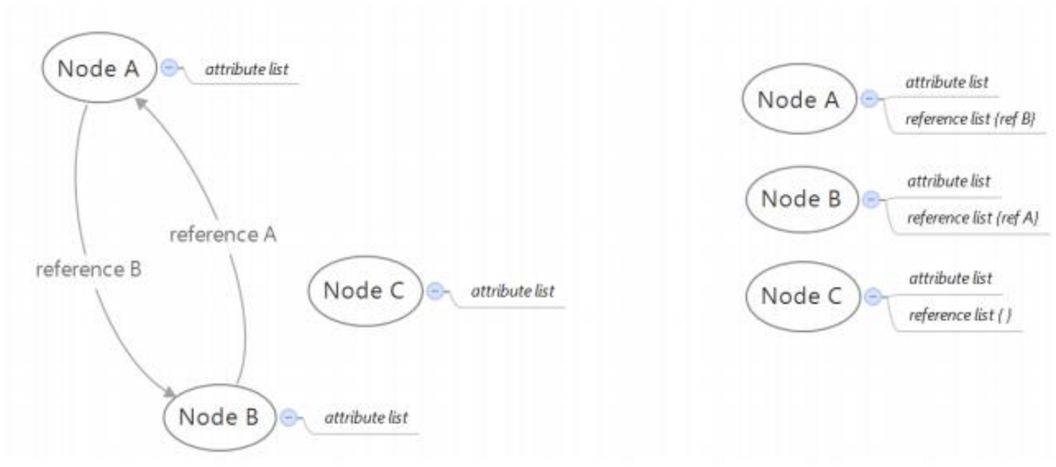


Figure 6.4: OPC UA Nodes, Attributes and References

6.4.2 *References*

References are the interconnections between *nodes* as shown in Figure 6.4.
The OPC UA defines a hierarchical structure of *nodes*. By means of *references,* is possible to define this structure.
To achieve this, a single reference is represented through a triad of parameters:

- *Source node*: Is the source of this process and keeps stored inside all the data referred to the reference.

- *Target node*: Generally, is in the same address space of the OPC-Server, but it is possible to be located inside a different one.

- *Reference type node*: Are *NodeClasses* (will be explained soon in the next section).

6.4.3 *NodeClasses*

The OPC UA uses *NodeClasses* to define attributes and references for different nodes [45].
As for *nodes*, *NodeClasses* cannot be redefined or extended by Client or Servers.

OPC UA define two macro categories that identify eight different node classes:

1) ***Access classes***:
   - *Object*: used to represent systems, system components, real-world objects, and software objects.
   - *Variable*: used to represent the content of an *Object*.
   - *Method*: used to represent a Method in the server *address space*.
   - *View*: used to restrict the number of visible *Nodes* and *References* in a large *Address Space*. By using *Views* servers can organize their *Address Space* and provide *views* on it tailored to specific tasks or use cases.

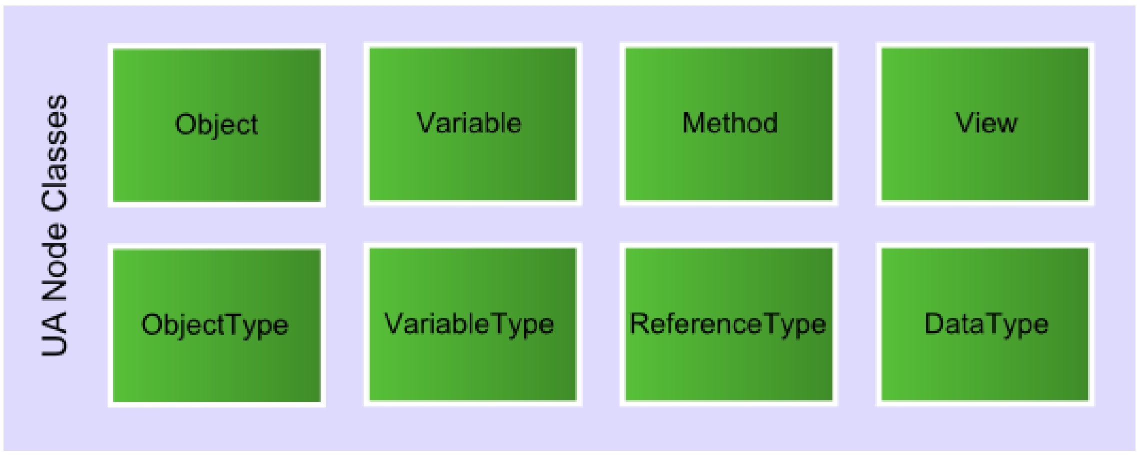


Figure 6.5: OPC UA Node Classes

2) **Type classes:**

- *ObjectType*: represents a type of node for objects in the server address space. ObjectTypes can be seen as classes in object-oriented languages.

- *VariableType*: represents a type of node for variables in the server address space. Are typically used to define which properties are available on the Variable instance.

- *ReferenceType*: used to represent the type of references used by the server.

- *DataType*: represented as *Nodes* of the *NodeClass* in the *Address Space*.

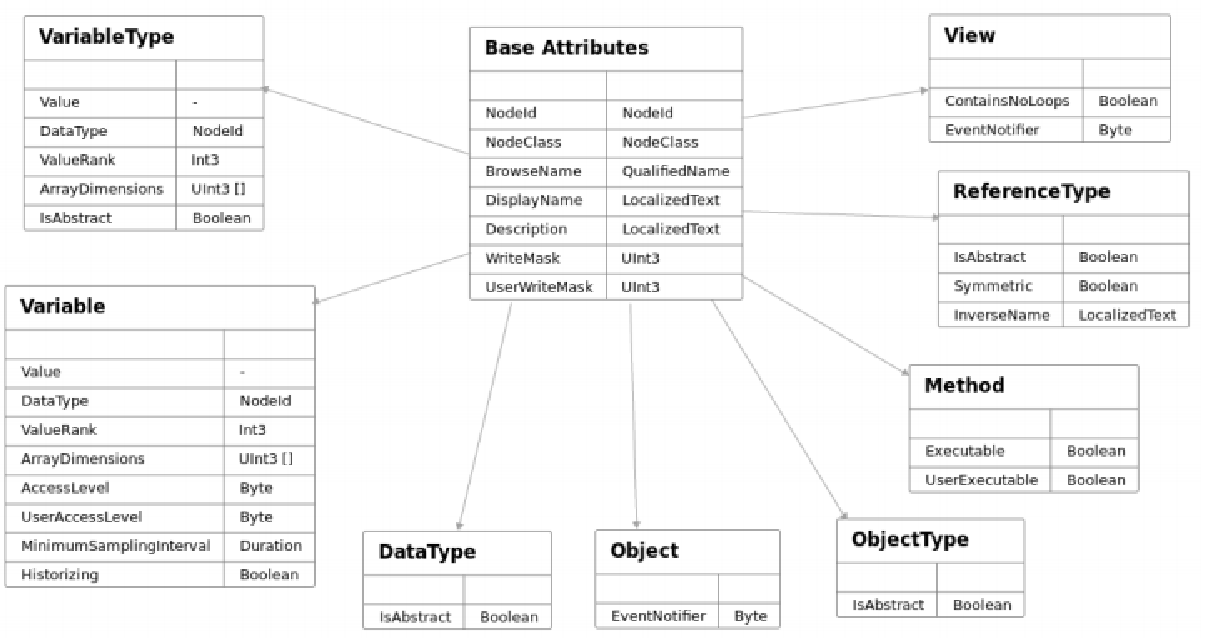


Figure 6.6: Attributes of OPC UA *NodeClasses*

6.4.4 *Variables*

Variables are of course used for the representation of certain values.
Inside the OPC UA protocol are defined two different types of variables:

- *Properties*: Like *attributes*, *properties* can be seen as a sort of descriptors. They can contain server-defined meta data of objects, data variables. With respect to *attributes*, *properties* can be defined and modified by the OPC-Server.

- *Data Variables*: Are the content of an object.

6.4.5 *OPC UA NodeID Concept*

As mentioned in a previous section, to uniquely identify a *Node* inside the OPC UA address space we use the *NodeID*.

The *Namespace* act as a *URI* that is responsible of the assignment of certain identifiers element which identify in a unique way each *NodeId*. In this way, multiple OPCUA Servers can use the same identifier to identify the same *Object*. This enables *Clients* that connect to those *Servers* to recognise *Objects* that they have in common.
Namespace URIs, like *Server* names, are identified by numeric values in OPC UA *Services* (this allows an easy and more efficient table lookups and transfer of it). The numeric values used to identify namespaces correspond to the index into the *NamespaceArray*.
A basic example of a possible URI an OPC UA namespace is:

```
http://opcfoundation.org/UA/”
```

whose corresponding index in the namespace table is 0, as shown in Figure 6.7.

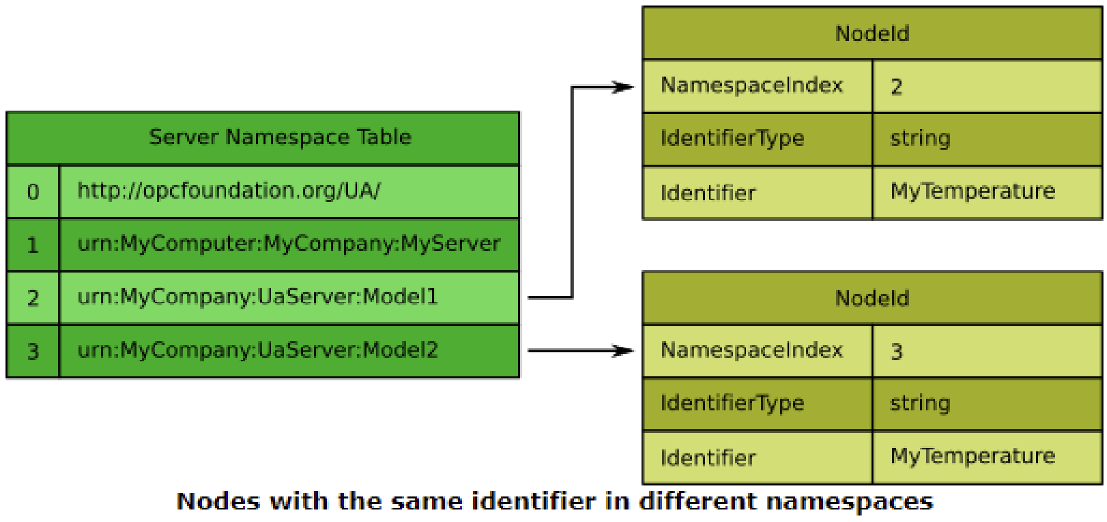

NodeId
NamespaceIndex
2
IdentifierType
string
Identifier
MyTemperature
Server Namespace Table
0
http://opcfoundation.org/UA/
1
urn:MyComputer:MyCompany:MyServer
2
urn:MyCompany:UaServer:Model1
3
urn:MyCompany:UaServer:Model2
NodeId
NamespaceIndex
3
IdentifierType
string
Identifier
MyTemperature
Nodes with the same identifier in different namespaces

Figure 6.7: Namespace Server and NodeID

To enter more in detail, each *NodeID* is composed by three elements:

- **Namespace Index**: it is a number used to identify the namespace URI. As shown in Figure 6.7 all the values of the Index are stored inside the Server Namespace Table.

- **Identifier Type**: represent the format and data type of the identifier. The possible types of data that can be represented are:
    - *Numeric value*
    - *String*
    - *Globally Unique Identifier* (GUID)
    - *Opaque*: that is referred to a specific format in *ByteString*

  The type of data to be used depend by the requirement to satisfy in our service.

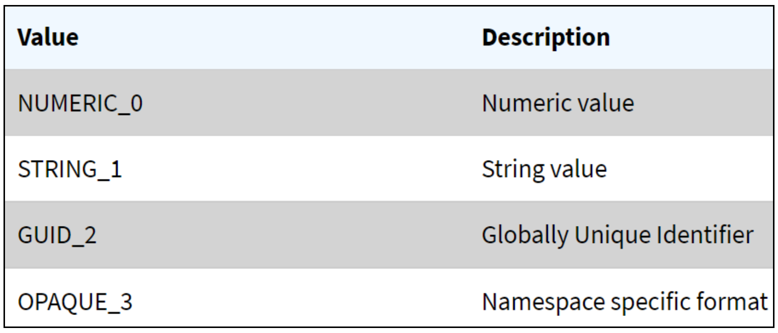
- **Identifier**: The identifier for a node in the address space of an OPC UA server. It is possible to use the same *Identifier* for different *nodes* that are part of a different *namespace* as shown in Figure 6.7.

6.4.6 *XML Notation*

There can be also a different XML notation:

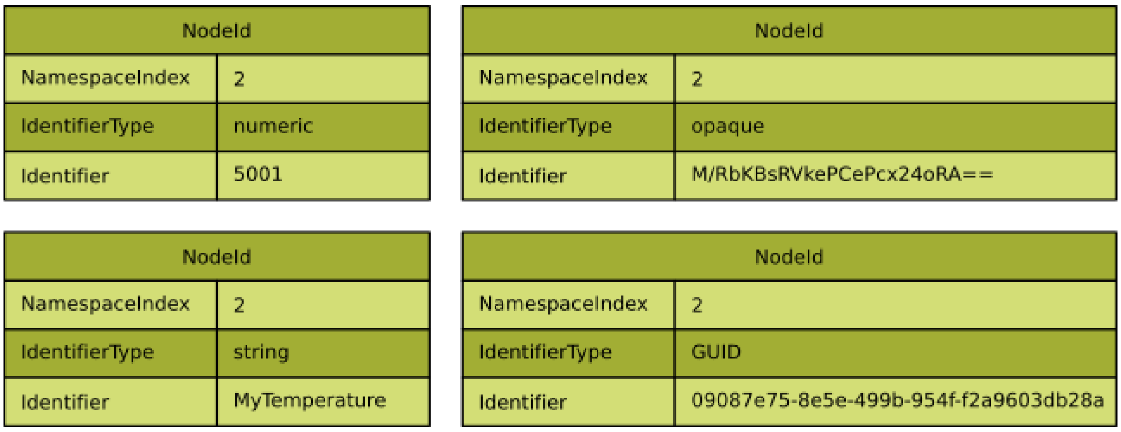
ns=<namespaceIndex>;<identifiertype>=<identifier>

- **<namespace index>** → The namespace index formatted as a base 10 number. If the index is 0, then the entire "ns=0;" clause is omitted
- **<identifier type>** → A flag that specifies the identifier type as shown in Figure 6.8



Value	Description
NUMERIC_0	Numeric value
STRING_1	String value
GUID_2	Globally Unique Identifier
OPAQUE_3	Namespace specific format

Figure 6.8: Identifier Types



NodeId	
NamespaceIndex	2
IdentifierType	numeric
Identifier	5001

NodeId	
NamespaceIndex	2
IdentifierType	opaque
Identifier	M/RbKBsRVkePCePcx24oRA==

NodeId	
NamespaceIndex	2
IdentifierType	string
Identifier	MyTemperature

NodeId	
NamespaceIndex	2
IdentifierType	GUID
Identifier	09087e75-8e5e-499b-954f-f2a9603db28a

Figure 6.9: Examples of possible NodeIDs representation

In Figure 6.9 are reported some possible examples of NodeID's representation using different Identifiers and Identifier Types.

In Figure 6.10 is shown a particular case where are defined in the same way for what concerns *Identifier and IdnetifierType*. We can distinguish them because they belong to a NamespaceIndex.

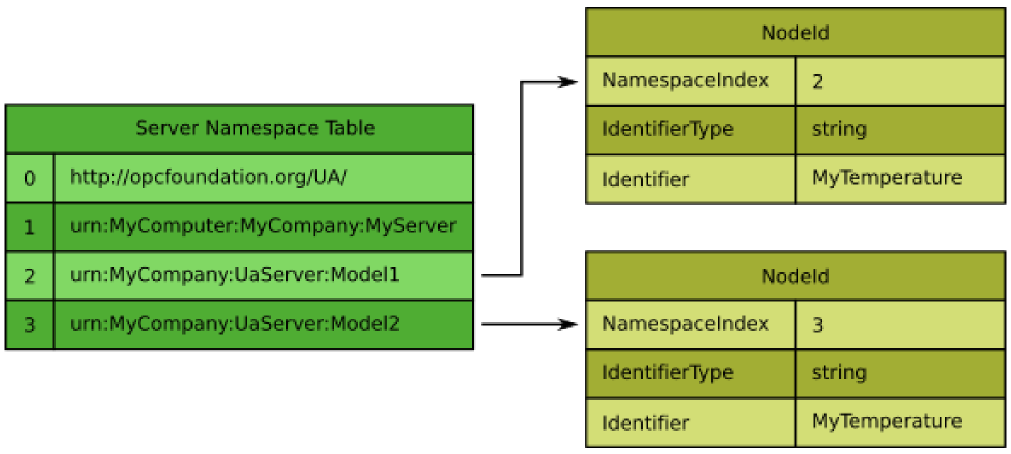


Figure 6.10: Nodes with the same identifier in different namespaces

6.5 **Access to the Namespace of a Server**

So far, I have introduced the structure of the information model that is used to store data and variables inside a Server.

Now I need to explain how a client can get access to that model.

To make possible to get access to the *namespace* of the server, the client must keep update his *namespace*. In this way the two *namespaces* of the two entities can match, allowing the client to know how to access the data in real time.

Figure 6.11 is presented an example with all the steps that are implemented to carry out the procedure.

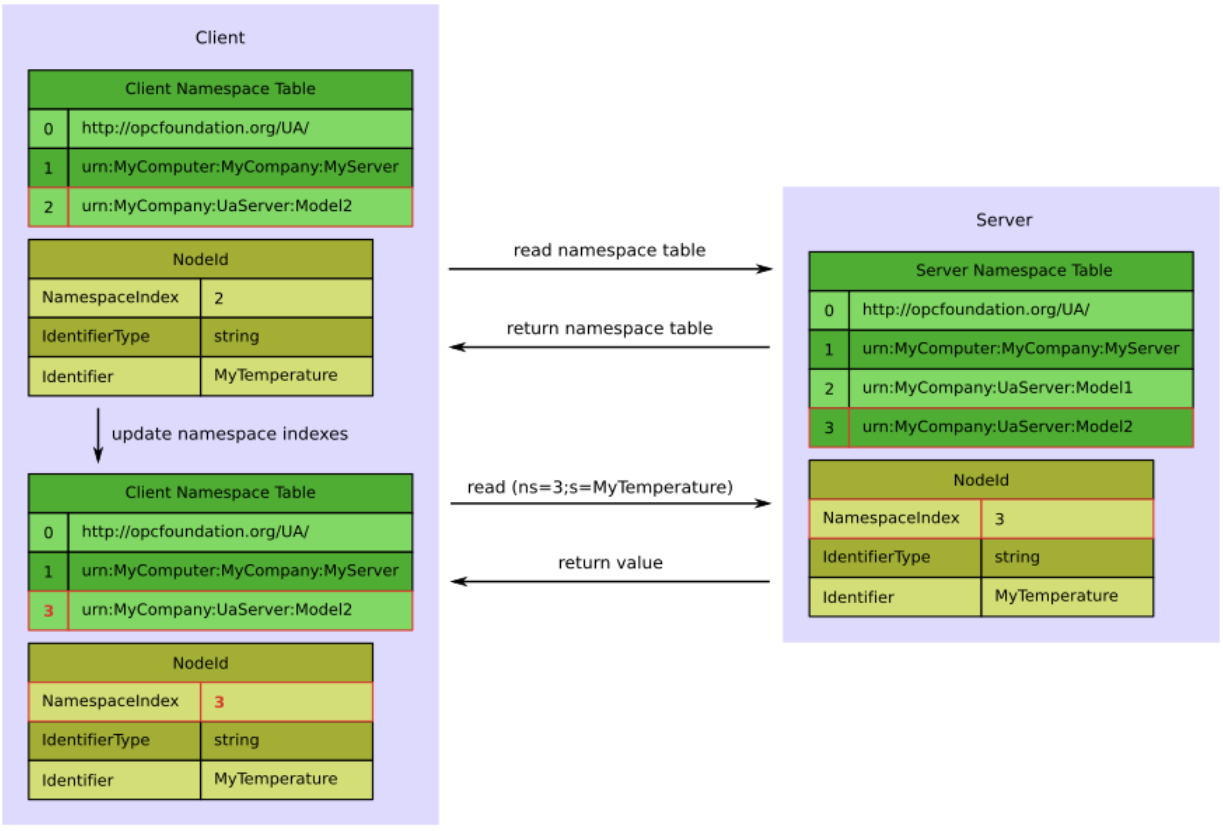


Figure 6.11 Read the namespace of the Server

In this example, the client wants to read the Node represented by the identifier "*MyTemperature*" which belongs to the namespace identified by the URI "*urn:MyCompany:UaServer:Model2*".

From the point of view of the client, the Node "*MyTemperature*" has a namespace index "2" but doesn't yet know the corresponding namespace index in the Server namespace table (which is "*3*").
To correctly access to the Node, the client must get access to the namespace of the server (*read namespace table*).
The Server send back the namespace index "*3*" which correspond to the namespace URI "*urn:MyCompany:UaServer:Model2*" inside his namespace.
The client can now update his namespace index whit the current value "3".

From this moment the Client has all information that is needed to access the correct node inside the namespace of the server, in our example "*ns=3;s=MyTemperature*" in XML notation.

6.6 **OPC UA Subscription**

The OPC UA protocol offer a mechanism called *subscription* that allow to an OPC UA-Clients to get access to information and services. In particular, a Client can request the subscription to a group of *Nodes* which are providing some important services, necessary for the correct deployment of a certain application.
After the subscription, the Server manages all these items, meaning that, it notifies to Clients only when changes occur. This mechanism is implemented to reduce the amount of data exchanged and bandwidth utilization.
This means that if the OPC UA-Client does not receive any status change from the server, nothing is changing inside the Classes.
To accomplish the subscription there is an entity called *Monitored Item* (MI) that is created as soon as the session is created and has the role of notify status changes that are happening in the classes.
The subscription process starts from the Client side, by add a Monitor Item to the subscription.

The MI can notify only three different types of “changes”, as reported in Figure 6.12:

- Subscribe to data changes of Variable Values (Value attribute of a Variable)
- Subscribe to Events of Objects (EventNotifier attribute of an Object & EventFilter Set).
- Subscribe to aggregated Values, which are calculated in client-defined time intervals, based on current Variable Values.

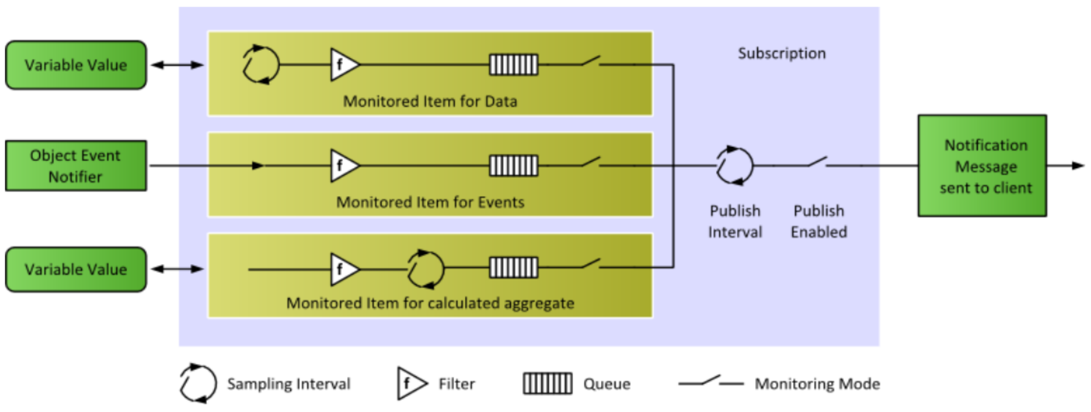


Figure 6.12: Subscription Process

It is also necessary to establish a *session* between OPC UA-Server and OPC UA-Client. To do that a *Secure Channel* is established between the two entities.

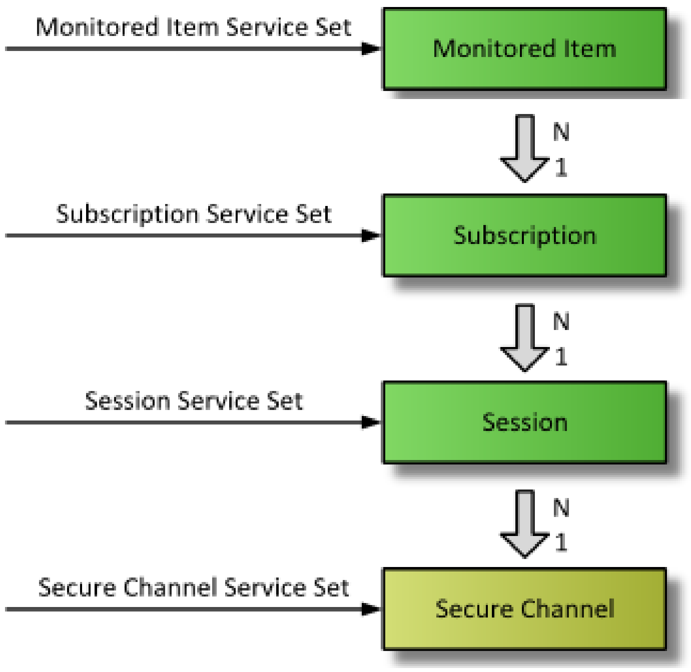


Figure 6.13: Subscription process and Secure Channel

There are other parameters that characterize the Subscription:

- **Sampling interval**: Its value must be defined for each MI and represent the frequency at which the server checks the data source for changes.

- **Publishing interval** (reporting): When a change is detected, the correspondent update is not sent immediately. This because also the Notification Message (reporting) is sent periodically. It can happen that the sampling interval is much smaller than the periodicity of the reporting. In this case the server stores inside a queue all the detection and wait the next reporting. The size of the queue is not fixed and require to be set for each MI depending by the application requirements. In case the queue is full a discard policy is applied to the oldest report in the queue.

- **KeepAlive**: In case no notification changes have to be reported, the Server send a *KeepAlive* notification to the Client. In this way the Server make the Client aware of the fact that nothing is changed in the status of the node, but the service is up and running.
  If nothing is received at the Client side that could be the possibility that the Secure Channel is no more active.

6.7 **Certification**

All the OPC UA devices to be identifiable from the other devices need the so-called *Application Instance Certificate* associated with a public/private pair of keys.

- The private key is secret and is used for encrypting messages.
- The public key is used to verify the trust relation and encrypt messages.

A digital certificate is used for the certification of the ownership of a public key. Everything is managed by the **Certificate Authority** (CA) that acts as a trusted third party. The CA is aware of the entire list of public keys and can verify the ownership of them. The CA certificate is added to the trust list of all the applications.

This is an essential point because allow to all applications that are signed by the CA to communicate each other.
All the certificates are located inside the *Certificate Store*:

- *Trusted certificates*: Self-signed certificates of trusted OPC UA application or CA certificates from trusted CAs. In general, are used for secure connection to a server.

- *Own certificates*: Application Instance certificate and private key

- *Issuers*: All the certificates that are not directly trusted. In this case they require a chain of CA certificates.

- *Self-certificates*: All the certificates that are released by the application itself. In general, are created as soon as the application is stared. During this first procedure, is necessary to establish a trust connection between Client and Server. To do that the client certificates are added to the trusted list inside the server and vice versa.

- *Rejected certificates*: All the certificate referred to applications that are not able to establish trusted connection are classified as rejected.

6.8 **Discovery process**

The discovery is the process performed by the OPC UA-Client for the research of the necessary information to setup a subscription to a certain OPC UA-Server that contains one or more classes and variables [46].
A *Discovery Server* (DS) contains the list of Servers and the related information of them. Inside this list are specified all the *endpoints* that are necessary for the connection to a certain server. OPC UA Protocol ensure that all servers have at least one endpoint. This is essential, otherwise clients were not able to connect to it.

When an OPC UA-Client wants to implement a certain application, must search it inside the DS and find all the information to establish a connection with the wanted server.
This last procedure is done through the *GetEndpoints or Discovery URL* service for gathering information from a server. After this procedure, the client can establish a connection with the Server.

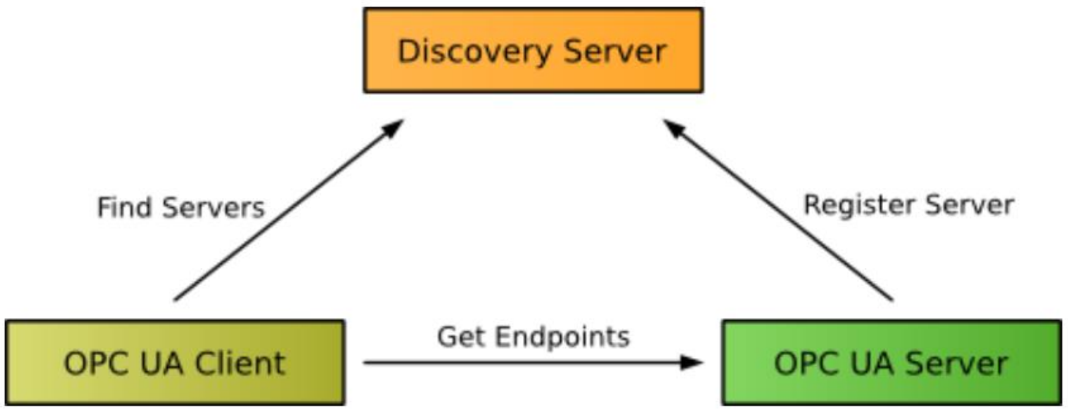


Figure 6.14: Subscription Process

The endpoints that are required for the correct subscription procedure are:

- *Endpoint URL*: Protocol and network address.
- *Security Policy Name*: For a set of security algorithms and key length.
- *Message Security Mode*: Security level for exchanged messages.
- *User Token Type*: Types of user authentication supported by the server.

The discovery process so fare described can be done at three different levels:

- **Local Discovery**
- **Multicast subnet Discovery**
- **Global discovery**

6.8.1 *Local Discovery*

Is used when a Client has the knowledge that the server he is looking for, is located on a certain host.
The standard OPC UA defines that *Local Discovery Server (LDS)* service is up and running on default port 4840.
All servers must be registered to the LDS service which also include security configuration by adding the server certificate in the trust list of it.

6.8.2 *Multicast Subnet Discovery*

Is used when the Client has only a restricted knowledge where to find the Server. In this specific case the OPC UA foresees the use of multicast DNS (mDNS).
The mDNS is a service used in small networks or subnets like in the case of OPC UA ecosystem. The working principle is the same of the DNS but instead of sending the request to the server DNS, it is sent in broadcast to all the devices connect to the network by means of a multicast request. When the wanted Server receives the request, responds directly to the Client.
Every time we send a multicast request in the network, we increase a little by the traffic inside the network, but the objective of this approach is to reduce as much as possible the complexity of networks.
The overall result is to obtain Zero Configuration Network (Zeroconf). In this approach all the devices of the network can find and to talk each other without the need of a network configuration because the multicast procedure is embedded inside the TCP/IP protocol.
This functionality so far discussed is implemented by LSD with Multicast Extension (LSD-ME).
It is important to notice that this procedure is not directly implemented by the OPC UA-Clients. They simply send the *FindServerOnNetwork* request to the LSD-ME. The latter sends the multicast to all the other LSD-MEs to find the wanted OPC UA-Server.
All the procedure is schematized in the Figure 6.15 down below.

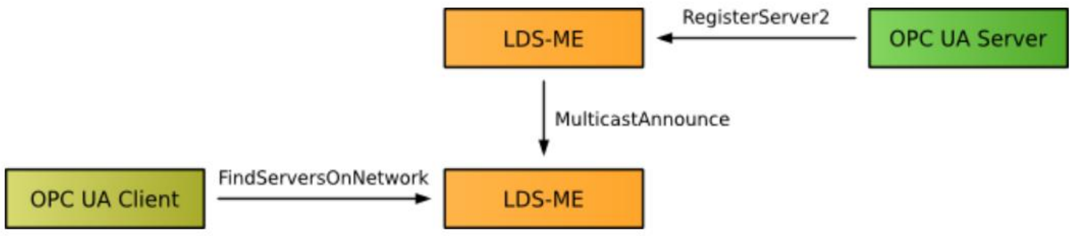


Figure 6.15: Multicast Subnet Discovery

6.8.3 *Global Discovery*

This approach is used when the discovery of the server is no more restricted to a small subnet (solved through mDNS) but is referred to wide networks composed by a huge number of Servers.
For these cases, the *OPC UA Global Discovery Server* (GDS) act as a sort of DNS server. Is in fact responsible of the central certificate management and the distribution of CA certificates.
To make possible that all the OPC UA-Servers will be included in the research done by the GDS, is necessary that they are registered as application to the GDS.

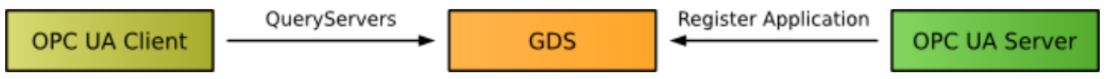


Figure 6.16: Global Discovery + Client and Server are in the same Subnet

In case the GDS is not in the same subnet of the OPC UA-Client it is necessary to pass through the LSD-ME suing the previously explained *FindServerOnNetwork* function.

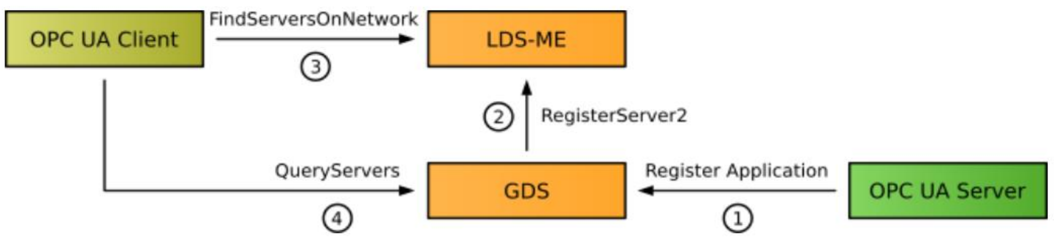


Figure 6.17: Global Discovery + Client and Server are into two different subnets

6.8.4 *Certificate Management with GDS*

The GDS does not only act as discovery entity but plays an important role also in the process of certification management of all the OPC UA applications that are registered. This because all the applications that are registered in the GDS require administrative rights. Generally, this operation is handled by the CA, but there can be some cases where the GDS take its place.

It can happen that the GDS is responsible of:

- Management of self-signed certificates.
- Generation of CA signed Application Instance Certificates.
- Distribution of the CA related Certificate Revocation Lists (CRL).

The procedure of the generation of administrative rights of the application registered inside the GDS is composed by two steps shown in Figure 6.18:

1) All the clients and servers are registered through *DirectoryType RegisterApplication.*

2) A call to *CertificateDirectoryType StartSigningRequest* for the creation of the CA signed certificate. In this way the private key of the entity (could be client or server) is only used to sign the request and is not revealed outside.
   The CA can now sign and create the public key using the request of the entity.

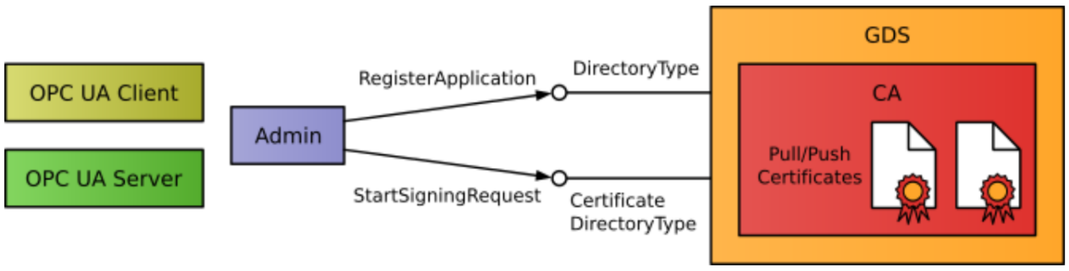


Figure 6.18: Registration of applications

6.9 **Configuration process**

Now that I have introduced all the principal concepts of the OPC UA Protocol is possible to see more in detail the procedure for the correct setup of the service. This procedure starts from the configuration of server and client and finishes with the connection between them.

6.9.1 *Server Configuration*

The configuration at the server side is done through:

- An application instance certificate identifying the server installation.
- A certificate store, including a list of trusted and rejected application instance certificates.
- At least one endpoint

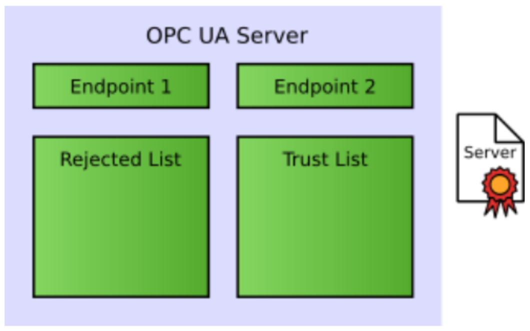


Figure 6.19: Server Configuration

6.9.2 *Client configuration*

The configuration at the client side is done through:

- An application instance certificate to identify the client application.
- A certificate store, including a list of trusted certificates.
- The list of server endpoints

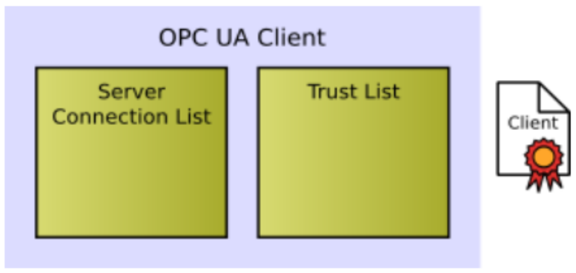


Figure 6.20: Client configuration

6.9.3 *Connection between client and Server*

A client is looking for the wanted service. As previously mentioned, this operation is done using the Discovery Server that provides all the necessary tools:

- Endpoints to reach the wanted Server
- Public key of the server
- Certificate of the server

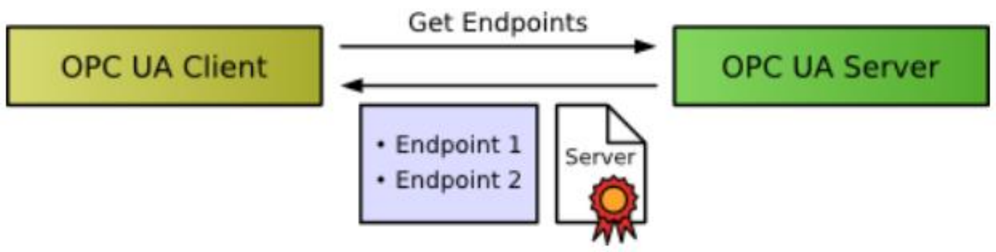


Figure 6.21: Exchange of data during the configuration between Client and Server

The client has at disposal the public key of the server and its certificate. These two elements are necessary to add the server inside the Client’s trust list at the Client side.
Once this operation is done, the secure connection can be established from the Client to the server.
It is important to make notice that the secure channel that is established so fare is not bidirectional. It is only a secure channel from the Client to the Server, but on the other side is missing the secure channel because at this step the Server already does not trust on the Client.
To do that, is often required the direct intervention from an administrator that moves the client certificate form the list of rejected to that one of trusted certificates as shown in Figure 6.22.

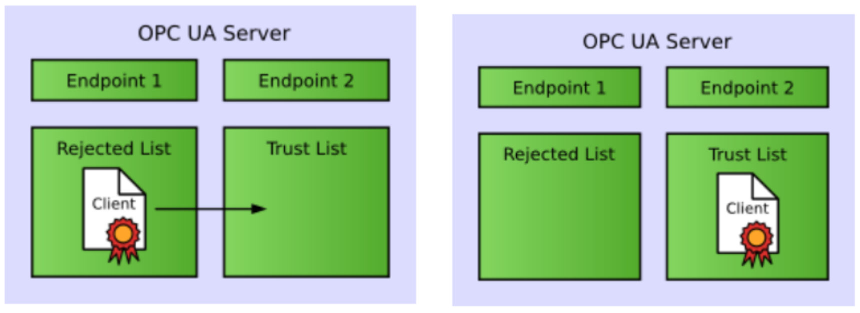


Figure 6.22: Adding the Client to the trusted list at the Server side.

Now the procedure is completed. A bidirectional channel is established between client and server.

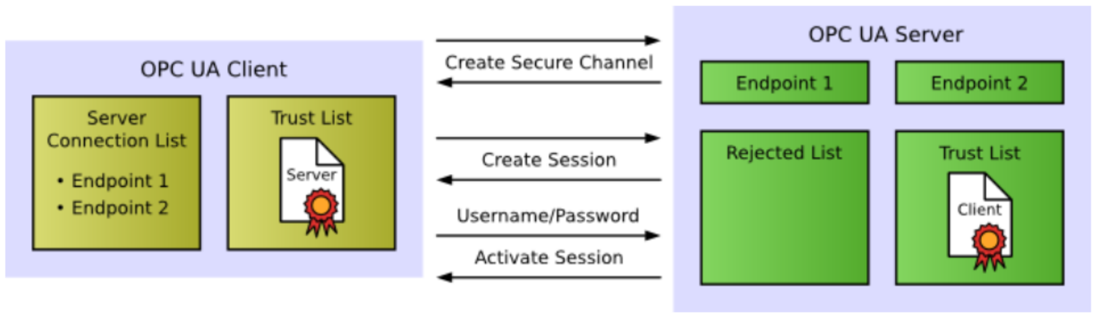


Figure 6.23: Bidirectional channel established between Client and Server

6.10 **MQTT Digression**

The final aim of this thesis is the orchestration of an IIoTaaS environment, where all the services are offered by entities that use OPC UA or MQTT as communication protocol. So far, I describe in detail the OPC UA protocol, because my work is focused on that. This section is dedicated to a small digression on the MQTT protocol.

As mentioned in the introduction part of this chapter, MQTT is one of the most used communication protocols in an IoT environment. Is a topic-base protocol that runs over TCP/IP and is based on a publish-subscribe mechanism.

All the entities that are defined by the MQTT protocol can be partially overlapped with those of the OPC UA.

The entities defined by the protocol are:

- **Publisher**: Act as the OPC UA Server. Is responsible of offer a certain service.
- **Subscriber/Client**: Is the entity that is look for a certain service offered by a certain server.
- **Broker**: Can be seen as a sort of Discovery Server of the OPC UA architecture.

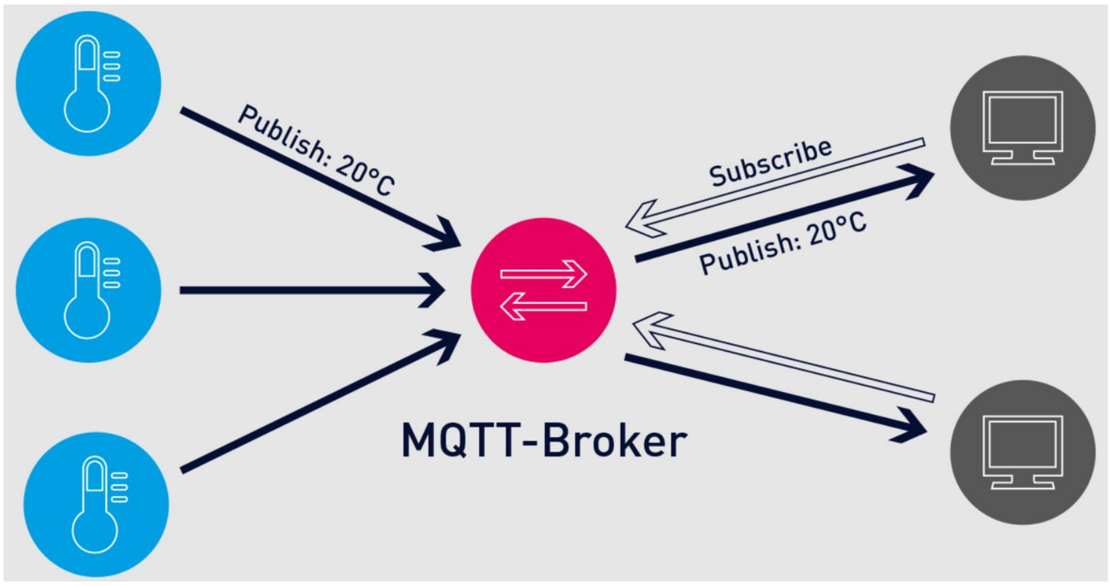


Figure 6.24: MQTT entities

In this case the information model is organized in a hierarchy of topics. Each publisher must send to the Broker all the items that it wants to distribute. The Broker is now aware of all the topics that are offered by the Publishers and make them available to all the clients. In this way clients can subscript to the topics they are interested in to receive updates of their state.

When a Publisher has an update from a certain topic, notify it to the Broker.

The broker finally distributes the information with all the clients that are subscribed to that specific topic.

It is evident, that the major difference with respect to the OPC UA case is that Publishers and Clients/Subscribers don't interact directly each other. Clients/Subscribers are not aware of the location or the configuration of Publishers because everything is managed by an intermediate entity that is represented by the Broker.

This is a big difference with the OPC UA protocol, where clients need to know all the endpoint information of the service, they are interested in.

MQTT is also a bidirectional communication protocol. This means that Publisher and Clients (but also Broker) are not constrained entities, and they are able to publish data and subscribe to certain services at the same time. This possibility can happen also with the OPC UA Protocol.

CHAPTER 7: OPC UA IMPLEMENTATION

For the implementation of OPC UA, I have evaluated two possible solutions to find which one is the best for my study case.

These two possible implementations of the OPC UA functionalities are Open62541 or the utilization of Python Libraries.

7.1 **Open62541**

Open62541 is an open-source C (C99) implementation of OPC UA licensed under the Mozilla Public License v2.0 [47].

C99 (previously known as C9X) is an informal name for ISO/IEC 9899:1999, a past version of the C programming language standard.

This software implements the OPC UA binary protocol stack as well as a client and server SDK.

7.1.1 *Channel establishment between client and Server*

In this paragraph is shown an experimental scenario where is established the connection between a OPC UA Server and Client.

For the deployment of the scenario, two virtual machines are created and handled by OpenStack.

As shown in Figure 7.1, the two machines are connected to the same mgmtNet. In this way they can see each other.

The IP addresses of the two machines are:

- Server → 10.15.2.53
- Client → 10.15.2.19

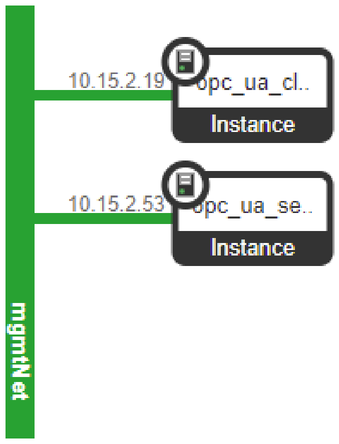
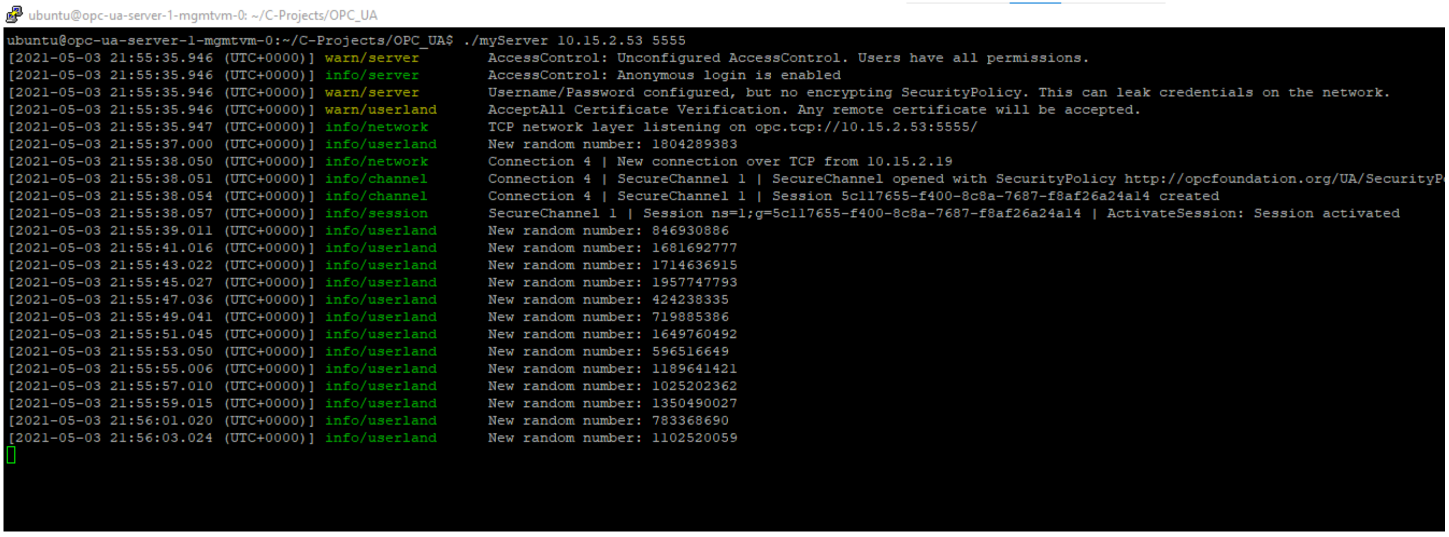


Figure 7.1: OPC UA Server and Client VMs

As application example, I created a Server that generates a random number every second. This service is exposed on a certain port that can be defined by the user. In this case the service endpoint is the *ens3* interface of the Server (with IP address 10.15.2.53) exposed on port 5555.

From the Server side, the service can be run through the following command: ./myServer *ip_address port_number*

```
$ ./myServer 10.15.2.53 5555
```



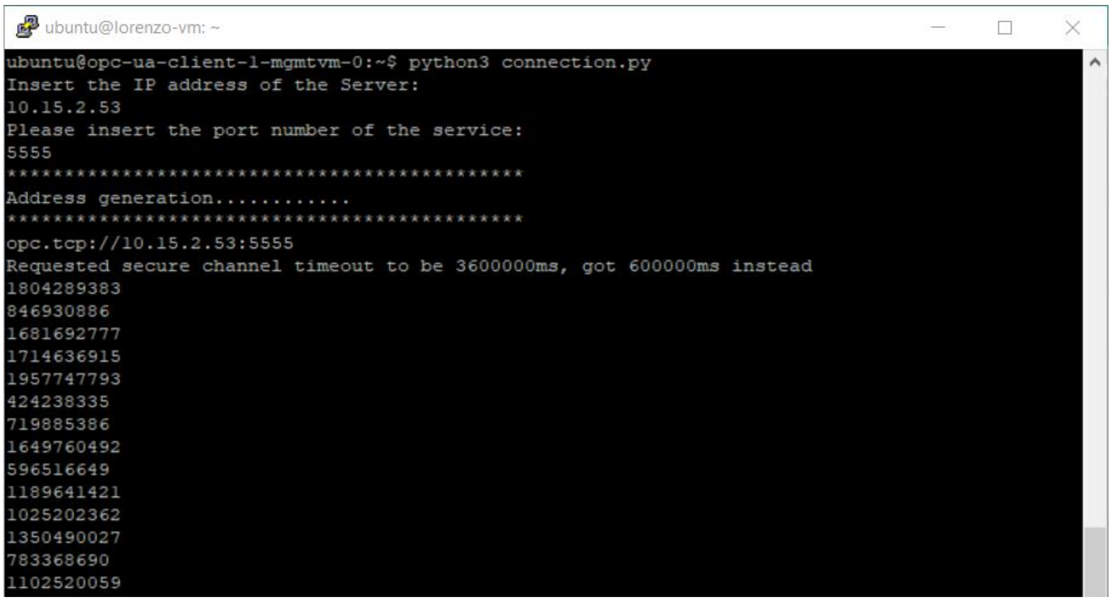
```
ubuntu@opc-ua-server-1-mgmtvm-0: ~/C-Projects/OPC_UA
ubuntu@opc-ua-server-1-mgmtvm-0:~/C-Projects/OPC_UA$ ./myServer 10.15.2.53 5555
[2021-05-03 21:55:35.946 (UTC+0000)] warn/server      AccessControl: Unconfigured AccessControl. Users have all permissions.
[2021-05-03 21:55:35.946 (UTC+0000)] info/server      AccessControl: Anonymous login is enabled
[2021-05-03 21:55:35.946 (UTC+0000)] warn/server      Username/Password configured, but no encrypting SecurityPolicy. This can leak credentials on the network.
[2021-05-03 21:55:35.946 (UTC+0000)] warn/userland    AcceptAll Certificate Verification. Any remote certificate will be accepted.
[2021-05-03 21:55:35.947 (UTC+0000)] info/network     TCP network layer listening on opc.tcp://10.15.2.53:5555/
[2021-05-03 21:55:37.000 (UTC+0000)] info/userland    New random number: 1804289383
[2021-05-03 21:55:38.050 (UTC+0000)] info/network     Connection 4 | New connection over TCP from 10.15.2.19
[2021-05-03 21:55:38.051 (UTC+0000)] info/channel     Connection 4 | SecureChannel 1 | SecureChannel opened with SecurityPolicy http://opcfoundation.org/UA/SecurityP
[2021-05-03 21:55:38.054 (UTC+0000)] info/channel     Connection 4 | SecureChannel 1 | Session 5c117655-f400-8c8a-7687-f8af26a24a14 created
[2021-05-03 21:55:38.057 (UTC+0000)] info/session     SecureChannel 1 | Session ns=1;g=5c117655-f400-8c8a-7687-f8af26a24a14 | ActivateSession: Session activated
[2021-05-03 21:55:39.011 (UTC+0000)] info/userland    New random number: 846930886
[2021-05-03 21:55:41.016 (UTC+0000)] info/userland    New random number: 1681692777
[2021-05-03 21:55:43.022 (UTC+0000)] info/userland    New random number: 1714636915
[2021-05-03 21:55:45.027 (UTC+0000)] info/userland    New random number: 1957747793
[2021-05-03 21:55:47.036 (UTC+0000)] info/userland    New random number: 424238335
[2021-05-03 21:55:49.041 (UTC+0000)] info/userland    New random number: 719885386
[2021-05-03 21:55:51.045 (UTC+0000)] info/userland    New random number: 1649760492
[2021-05-03 21:55:53.050 (UTC+0000)] info/userland    New random number: 596516649
[2021-05-03 21:55:55.006 (UTC+0000)] info/userland    New random number: 1189641421
[2021-05-03 21:55:57.010 (UTC+0000)] info/userland    New random number: 1025202362
[2021-05-03 21:55:59.015 (UTC+0000)] info/userland    New random number: 1350490027
[2021-05-03 21:56:01.020 (UTC+0000)] info/userland    New random number: 783368690
[2021-05-03 21:56:03.024 (UTC+0000)] info/userland    New random number: 1102520059
```

Figure 7.2: OPC UA Server random number generation

In Figure 7.2 is reported the creation of the server and the generation of random numbers.

Now that the service is up and running, the client can access to that service by accessing the server on that port (5555).
To do that I created a python scritp that ask to the user to inser ip address and port number for the connection with the server.

It is possible to see from Figure 7.3, that the application is working correctly. All the numbers generated by the server are correctly received by the client.



```
ubuntu@opc-ua-client-1-mgmtvm-0:~$ python3 connection.py
Insert the IP address of the Server:
10.15.2.53
Please insert the port number of the service:
5555
*********************************************
Address generation...........
*********************************************
opc.tcp://10.15.2.53:5555
Requested secure channel timeout to be 3600000ms, got 600000ms instead
1804289383
846930886
1681692777
1714636915
1957747793
424238335
719885386
1649760492
596516649
1189641421
1025202362
1350490027
783368690
1102520059
```

Figure 7.3: OPC UA Client

An additional confirmation that everything is working correctly can be spotted in Figure 7.4 at the Server side. After the correct establishment of the channel, we have the following output in the bash:

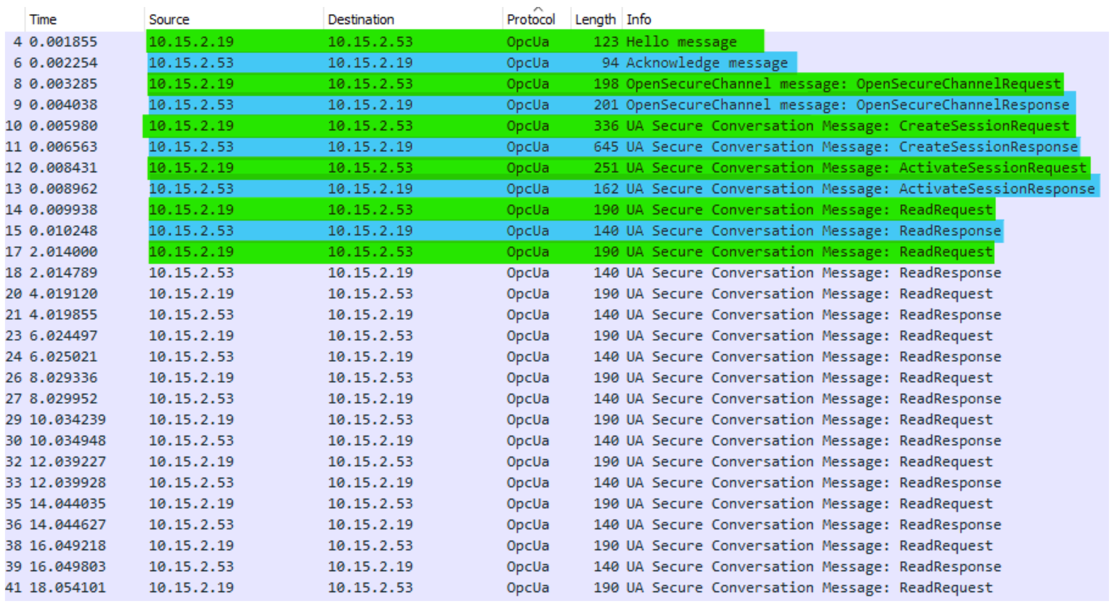
```
Connection 4 | New connection over TCP from 10.15.2.19

Connection 4 | SecureChannel1 | Secure channel1 opened with securityPolicy http://opcfoundation.org

Connection 4 | Session 5c117655-f400-8c8a-7687-f8nf26a24a14
```

Figure 7.4: OPC UA Server channel established message

To see what is the set of messages that are exchanged between the two entities for setup the service, I captured the traffic on the server port 5555.
In Figures 7.5 and 7.6 are shown all the messages that are exchanged to properly create the session between the Client and the Server.

Once the session is established, the client asks to the server the data to read through a *ReadRequest*.

The server sends the value through a *ReadResponse*.



```
   Time         Source          Destination     Protocol  Length  Info
 4 0.001855     10.15.2.19      10.15.2.53      OpcUa     123 Hello message
 6 0.002254     10.15.2.53      10.15.2.19      OpcUa      94 Acknowledge message
 8 0.003285     10.15.2.19      10.15.2.53      OpcUa     198 OpenSecureChannel message: OpenSecureChannelRequest
 9 0.004038     10.15.2.53      10.15.2.19      OpcUa     201 OpenSecureChannel message: OpenSecureChannelResponse
10 0.005980     10.15.2.19      10.15.2.53      OpcUa     336 UA Secure Conversation Message: CreateSessionRequest
11 0.006563     10.15.2.53      10.15.2.19      OpcUa     645 UA Secure Conversation Message: CreateSessionResponse
12 0.008431     10.15.2.19      10.15.2.53      OpcUa     251 UA Secure Conversation Message: ActivateSessionRequest
13 0.008962     10.15.2.53      10.15.2.19      OpcUa     162 UA Secure Conversation Message: ActivateSessionResponse
14 0.009938     10.15.2.19      10.15.2.53      OpcUa     190 UA Secure Conversation Message: ReadRequest
15 0.010248     10.15.2.53      10.15.2.19      OpcUa     140 UA Secure Conversation Message: ReadResponse
17 2.014000     10.15.2.19      10.15.2.53      OpcUa     190 UA Secure Conversation Message: ReadRequest
18 2.014789     10.15.2.53      10.15.2.19      OpcUa     140 UA Secure Conversation Message: ReadResponse
20 4.019120     10.15.2.19      10.15.2.53      OpcUa     190 UA Secure Conversation Message: ReadRequest
21 4.019855     10.15.2.53      10.15.2.19      OpcUa     140 UA Secure Conversation Message: ReadResponse
23 6.024497     10.15.2.19      10.15.2.53      OpcUa     190 UA Secure Conversation Message: ReadRequest
24 6.025021     10.15.2.53      10.15.2.19      OpcUa     140 UA Secure Conversation Message: ReadResponse
26 8.029336     10.15.2.19      10.15.2.53      OpcUa     190 UA Secure Conversation Message: ReadRequest
27 8.029952     10.15.2.53      10.15.2.19      OpcUa     140 UA Secure Conversation Message: ReadResponse
29 10.034239    10.15.2.19      10.15.2.53      OpcUa     190 UA Secure Conversation Message: ReadRequest
30 10.034948    10.15.2.53      10.15.2.19      OpcUa     140 UA Secure Conversation Message: ReadResponse
32 12.039227    10.15.2.19      10.15.2.53      OpcUa     190 UA Secure Conversation Message: ReadRequest
33 12.039928    10.15.2.53      10.15.2.19      OpcUa     140 UA Secure Conversation Message: ReadResponse
35 14.044035    10.15.2.19      10.15.2.53      OpcUa     190 UA Secure Conversation Message: ReadRequest
36 14.044627    10.15.2.53      10.15.2.19      OpcUa     140 UA Secure Conversation Message: ReadResponse
38 16.049218    10.15.2.19      10.15.2.53      OpcUa     190 UA Secure Conversation Message: ReadRequest
39 16.049803    10.15.2.53      10.15.2.19      OpcUa     140 UA Secure Conversation Message: ReadResponse
41 18.054101    10.15.2.19      10.15.2.53      OpcUa     190 UA Secure Conversation Message: ReadRequest
```

Figure 7.5: Traffic captured between Client and Server

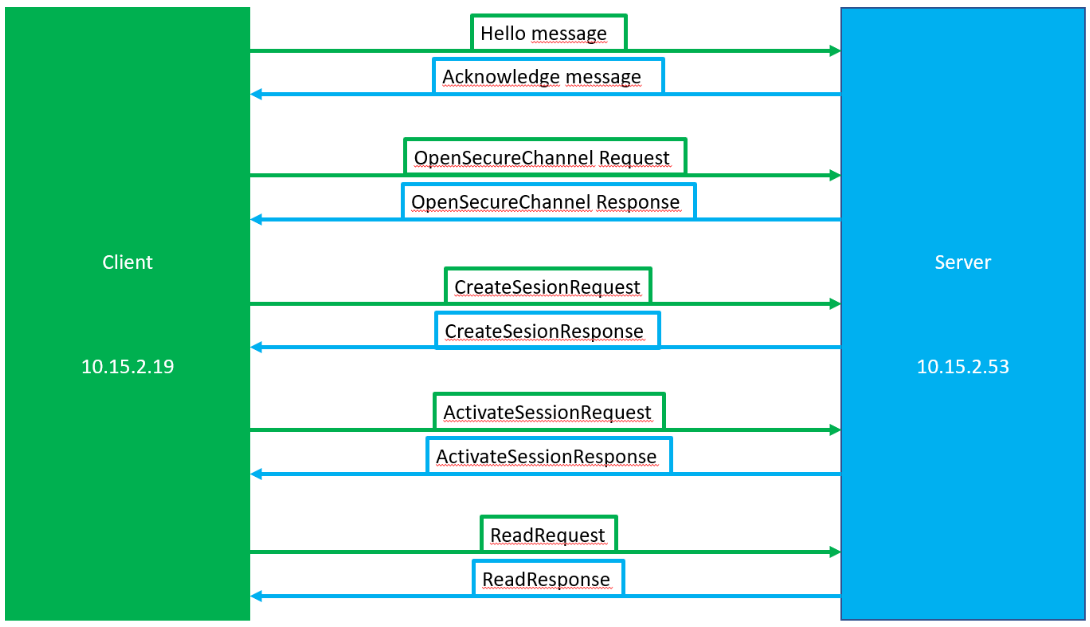


Figure 7.6: OPC UA Session creation

7.1.2 *Creation of a Docker Container*

The case study of this thesis requires that all the application that are implemented, are running over a container. For this reason, is necessary that both Client and Server are running on a Docker container.

I created two Docker images and uploaded them into the Docker Hub. They contain all the necessary software for the Deployment of the two applications.

The image of the server is made with an environment variable called *time*. It represents the periodicity at which the server generates random numbers.

This variable can be set during the launching phase of the container using the command:

```
sudo time docker run --net=host --rm -e time=X lorenzobassi/opcua_server_reconfigurable
```

where X represents the periodicity in seconds that we want to set.

After the command is launched, it requires some time before the server is effectively up and running. This is because the server is written in C99 code. It means that it must be compiled and built to generate an executable file that is launched.

The entire operation requires around 30 seconds before the server is up and running.

Unfortunately, this time required for launch a container is too high and cannot match the low delay requirements that are requested for the scope.

For this reason, I choose to not proceed with the software Open6254 because requires compiling and rebuild the server every time a small change is applied inside the lines of code of that are controlling it (and the same is for the client side).

7.2 **OPC UA with Python Libraries**

Python libraries for the implementation of OPC UA [48], can solve some of the problems that are shown by Open6254 software. It is a more flexible and easier solution to be implemented. All the software is written in Python scripts that can be executed immediately without the need build an executable file required by the C99 solution.

All the software needed is contained in a package called "*opcua*" that follow IEC 62541 standard and can be installed using pip command. It contains some libraries called "*client*", "*server*" and "*ua*" that must be imported inside the code. This python implementation of the OPC UA is quasi complete and has been tested against many different OPC UA stacks.

All these properties are compliant with the application I am looking for. At the end of the day, this second approach represent a good solution for my scope, and I choose to proceed in this direction.

The installation of this software on Ubuntu is very easy and can be made through these commands:

```
pip install opcua
```

In this way the Python package *opcua* is installed. It contains all the functions that are needed for the execution of the OPC UA software.

```
apt install python-opcua        # Library

apt install python-opcua-tools  # Command-line tools  install opcua
```

With these latter, are installed some Python libraries and some command line tools needed.

7.2.1 *Analysis of the traffic exchanged between client and server*

As in the case of Open6254, I have evaluated the traffic exchanged between client and server for the establishment of the channel. With respect to the case of Open6254 I implemented additional functionalities. It is possible for the OPCUA Client to ask for subscription of a variable inside the server.

That is why new types of packets are used in the communication.

In this case, the protocol used for the establishment of the secure channel is **OPC UA Secure Conversation (UASC)** that uses binary encoded Messages.

This protocol is used in several applications where there is the need to operate with transport protocols that have a limited buffer size.

In this specific case, the protocol does not operate directly over the entire OPC UA message. The packet is instead segmented into multiple chunks called *MessageChunks*. This is an essential step because all the chunks must not overcome the buffer size that is set at 8192 bytes for the *TrasportProtocol*.

The security is achieved over the entire OPC UA message by acting over all the single chunks of which it is spitted.

At the receiver side, all the chunks are received and need to be reordered to reconstruct the original OPC UA message.

This operation can be done thanks to a 4-byte sequence that is assigned to each *MessageChunk* as shown in Figure 7.7. These bytes are sequence numbers used to detect replay attacks and possibly avoid them.

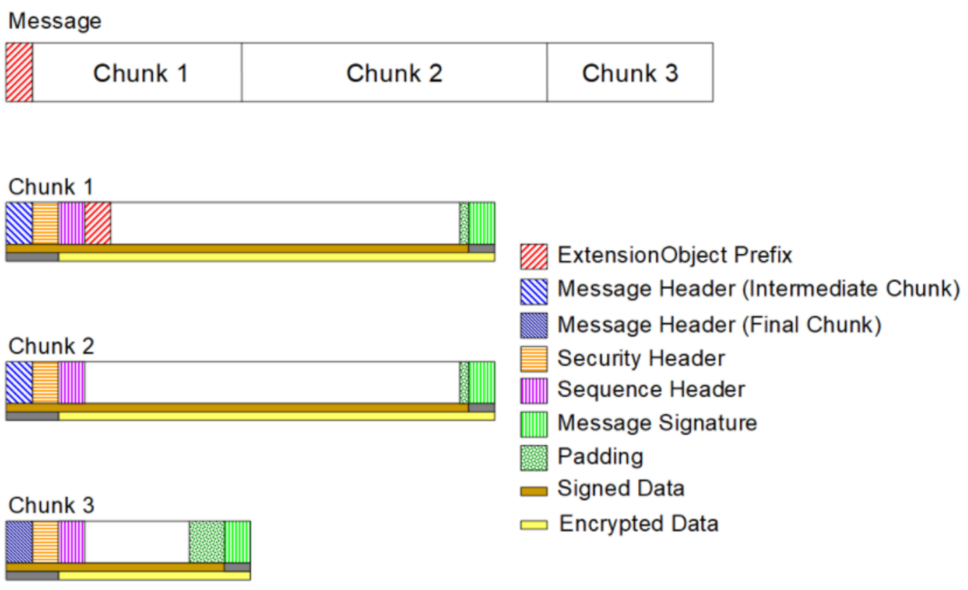


Figure 7.7 : UASC Message Structure

During the establishment of the secure channel some messages are exchanged. Down below I briefly describe the most important fields inside them (the full description of the content of the packets can be found at: [49].

- **Hello message**.
    - *ReceiveBufferSize*: The largest *MessageChunk* that the sender can receive.
    - *SendBufferSize*: The largest *MessageChunk* that the sender will send.
    - *MaxMessageSize*: The maximum size for any response *Message*.
    - *MaxChunkCount*: The maximum number of chunks in any response *Message*.
    - *EndpointUrl*: The URL of the *Endpoint* which the *Client* wished to connect to. The encoded value shall be less than 4096 bytes.
- **Acknowledge message**
    - *ReceiveBufferSize:*
    - *SendBufferSize:*
    - *MaxMessageSize:*
    - *MaxChunkCount:*

The conversation for the creation of the secure channel between the two entities can be made starting either from client or server by creating a *TransportConnection*.
UASC uses TCP/IP protocol for the communication between client and server, because can guarantee full-duplex communication between the two entities. A socket of the TCP/IP can be seen as a *TransportConnection* in this new paradigm.

- If the client starts the *TransportConnection*, the first message that is exchanged is a *Hello message* (sent by the client itself) used to specify the maximum buffer size handled by the Client.
  The server responds with an *Acknowledge Message* to start the negotiation for the buffer size.
  At the end of the negotiation, the *SendBufferSize* field specifies the size of the *MessageChunk* that will be used during the communication between client and server.
  At the end of the negotiation, the client sends a *OpenSecureChannelRequest*. In case the server accepts it, a new channel related to the TransportConnection is created with a *SecureChannelId*.

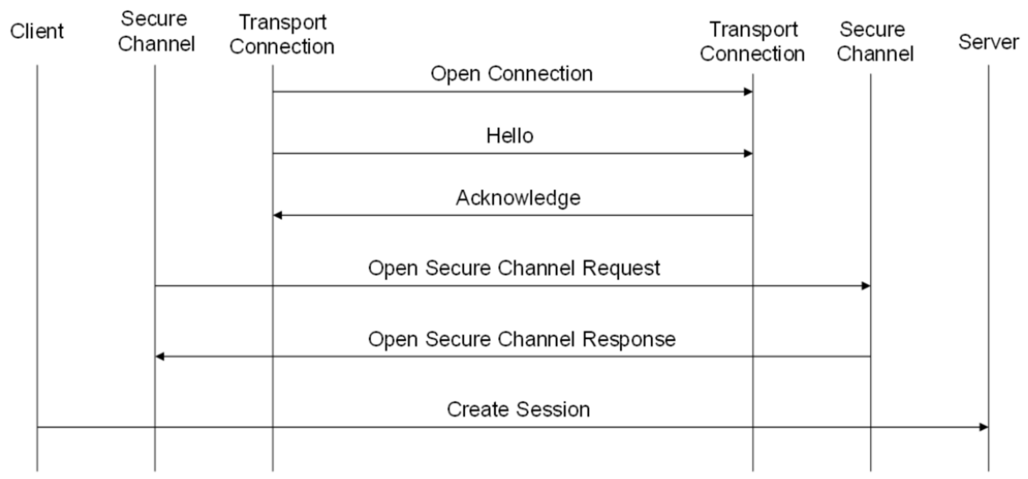


Figure 7.8 : Initiation of the *TransportConnection* by the Client

- If the server starts the procedure of the creation of the *TransportConnection*, the only difference with respect to the previous case is the first message exchanged. In this case, the server sends a *ReverseHello* to the client.

  When the *TransportConnection* is accepted by the client, it responds with an *Hello message*. At this point, the situation is the same described in the previous case. The negotiation procedure will start following the same steps described before.

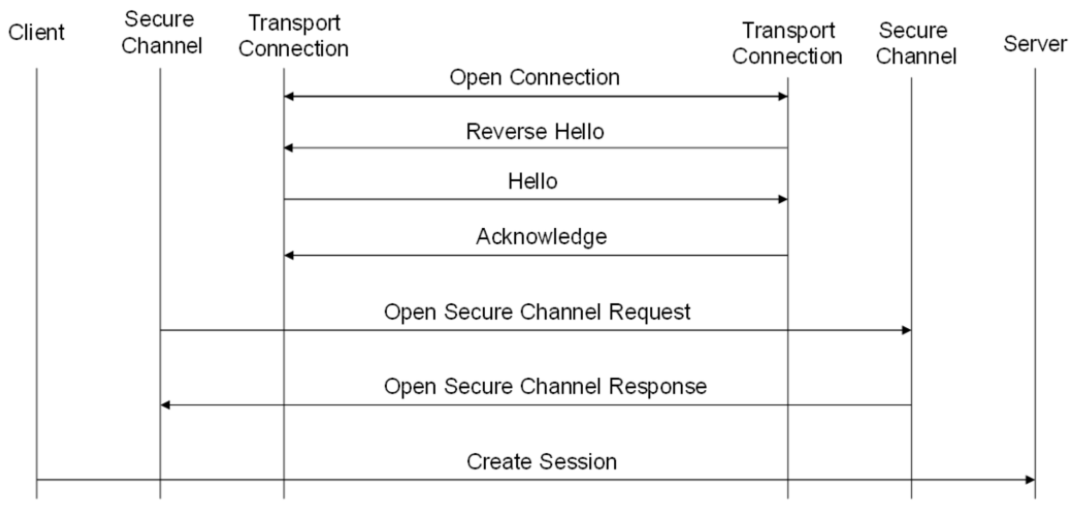
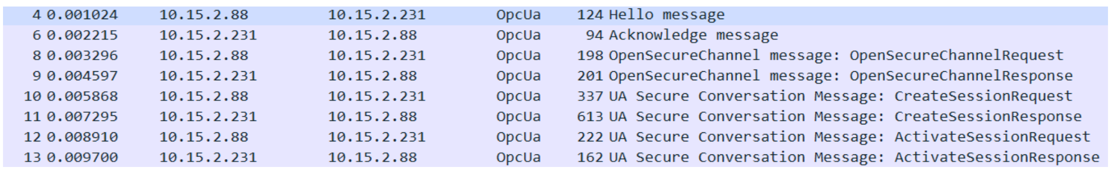


Figure 7.9 : Initiation of the *TransportConncetion* by the Server



```
 4 0.001024   10.15.2.88    10.15.2.231   OpcUa   124 Hello message
 6 0.002215   10.15.2.231   10.15.2.88    OpcUa    94 Acknowledge message
 8 0.003296   10.15.2.88    10.15.2.231   OpcUa   198 OpenSecureChannel message: OpenSecureChannelRequest
 9 0.004597   10.15.2.231   10.15.2.88    OpcUa   201 OpenSecureChannel message: OpenSecureChannelResponse
10 0.005868   10.15.2.88    10.15.2.231   OpcUa   337 UA Secure Conversation Message: CreateSessionRequest
11 0.007295   10.15.2.231   10.15.2.88    OpcUa   613 UA Secure Conversation Message: CreateSessionResponse
12 0.008910   10.15.2.88    10.15.2.231   OpcUa   222 UA Secure Conversation Message: ActivateSessionRequest
13 0.009700   10.15.2.231   10.15.2.88    OpcUa   162 UA Secure Conversation Message: ActivateSessionResponse
```

Figure 7.10 : Capture of the traffic related to creation of the secure channel.

In Figure 7.10 are reported the packet exchanged between client and server for the procedure of the creation of the secure channel.

The way the two entities are interacting each other can be seen as a WebSocket.
A WebSocket is a protocol that is used in case is needed a bi-directional communication between two entities like in the case of OPC UA.
The procedure for the setup of the web socket is represented in Figure 7.11 and starts with a HTTP GET request.

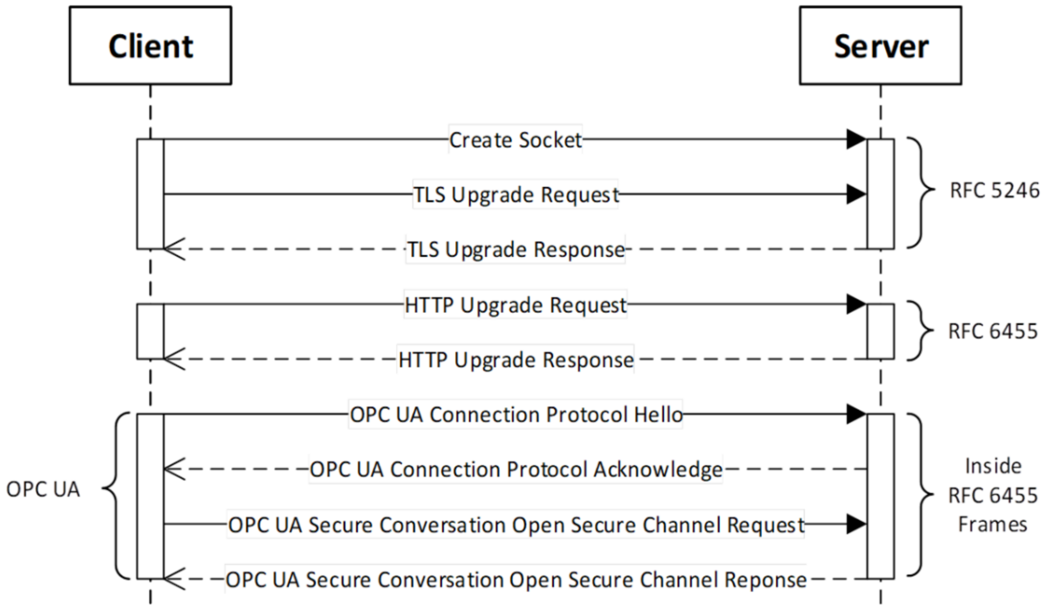

Client
Server
Create Socket
TLS Upgrade Request
TLS Upgrade Response
RFC 5246
HTTP Upgrade Request
HTTP Upgrade Response
RFC 6455
OPC UA Connection Protocol Hello
OPC UA Connection Protocol Acknowledge
OPC UA Secure Conversation Open Secure Channel Request
OPC UA Secure Conversation Open Secure Channel Reponse
OPC UA
Inside RFC 6455 Frames

Figure 7.11 : Steps for the creation of a WebSocket

At this point, the secure channel is created. Server and Client know how to send messages each other.
Now that the channel is created the other functionalities can be implemented.

In case the client requires the subscription to a variable inside the server, other types of messages are exchanged.

First, the client is not aware how to reach the variable inside the server. This means that it must discover the Reference of that specific node that the client is interested in the subscription. To do that, the Client uses the *browse* service to navigate inside the *AddressSpace* of the server and find references of that node. In general, are called

“*browse path*” and are composed by a starting *Node* and a *RelativePath*. The *Node* identify the *Node* itself from which the *RelativePath* is referred to.
This operation is achieved through a *BrowserRequest*/*BrowserResponse*.

After that the client has discovered the reference of the node a *TranslateBrowsePathsToNodeIds* is sent to the server for the translation of the paths previously mentioned in the corresponded *NodeIds*.

Now the client is aware of all the information needed to reach and identify in a unique way the variable inside the *NameSpace* of the server that it is interested to read. The client requests the subscription to that variable through a *CreateSubscriptionRe*quest/*CreateSubscriptionResponse*.

After the subscription, a *MonitoredItem* is needed to update changes of the variable as described in Section 6.6 by means of *Notifications*.
Each *MonitoredItem* sends a *Notification* by looking at the *Subscriptions* to a certain variable. A *Notification* is created at every occurrence of data changes or event which involve the subscribed variable.
All *Notifications* are not sent immediately but are packaged into *NotificationMessages* that are sent periodically to the client following a specific *Publishing interval*.

In Figure 7.12 are reported all the steps that are involved in the subscription.
There are four parameter that characterize each *MonitoredItem* [50]:

- **Monitoring Mode**
  This parameter is used for the activation and deactivation of a sampling or the reporting of Notifications.

- **Filter**
  It is a criterion that is applied to each sample generated by the server and evaluate if in necessary to generate the Notification related to it.

- **Sampling Interval**

  This period is referred to the fastest rate at which the server samples all the data changes of the items of the subscription.

  This parameter is established by the client as soon as the *MonitoredItem* is created. In case is set to a negative number, it refers to the default value that is defined during the instantiation.

  It is important to notice that does not represent the maximum threshold at which the server can sample the data. It defines the “best effort” rate of sample the data. It is possible that the server could go beyond that threshold and sample at a faster rate but would be useless with the type of item that the server is managing.

  A sample rate equal to 0 is used in two situations:

  - In case the client is interested only in the subscription of events.
  - The client needs the fastest sampling rate that the Server can handle.

  There can be the possibility that the client requests a sampling rate that is not supported. In this case the server assigns the most appropriate interval that can meet the request of the client.

  All the mechanism explained so far is referred to a *sampling model*.

  There can be the possibility to define an *exception-based model*. In this case there is not a definition of a sampling interval, and the server simply reports all data changes.

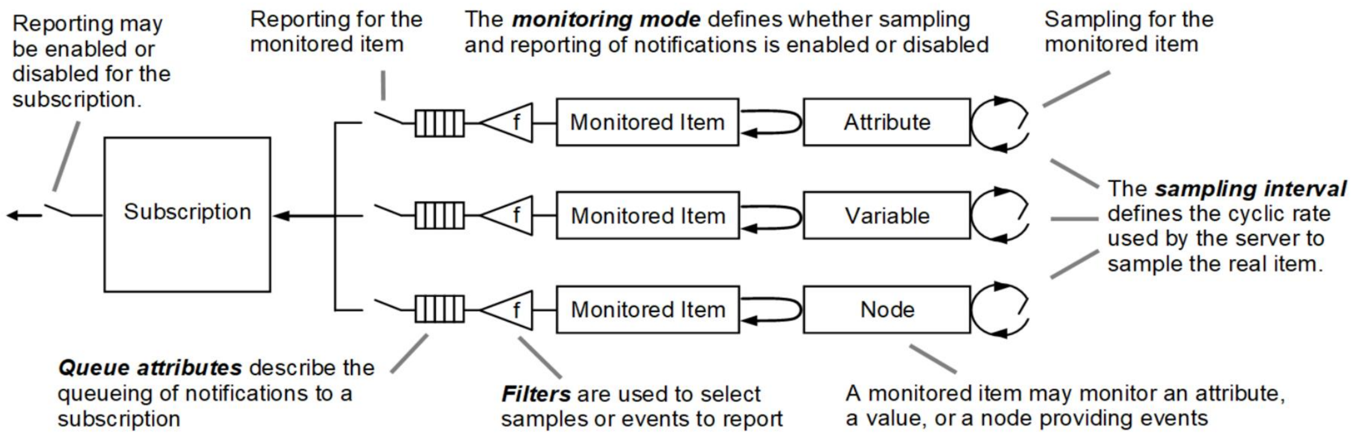

Reporting may be enabled or disabled for the subscription.
Reporting for the monitored item
The monitoring mode defines whether sampling and reporting of notifications is enabled or disabled
Sampling for the monitored item
Subscription
f
Monitored Item
Attribute
Variable
Node
The sampling interval defines the cyclic rate used by the server to sample the real item.
Queue attributes describe the queueing of notifications to a subscription
Filters are used to select samples or events to report
A monitored item may monitor an attribute, a value, or a node providing events

Figure 7.12 : Monitored Item block diagram

- **Queue parameters**

  After that a sample has passed the criteria previously described, a *Notification* is generated. All the notifications that are generated in the sampling interval are stored inside a queue, waiting the end of the sampling interval.

  The queue size is determined at the creation of the *MonitoredItem*, also including the FIFO or LIFO discard policy to be applied in case of queue saturation.

  This mechanism is schematized in Figure 7.13.

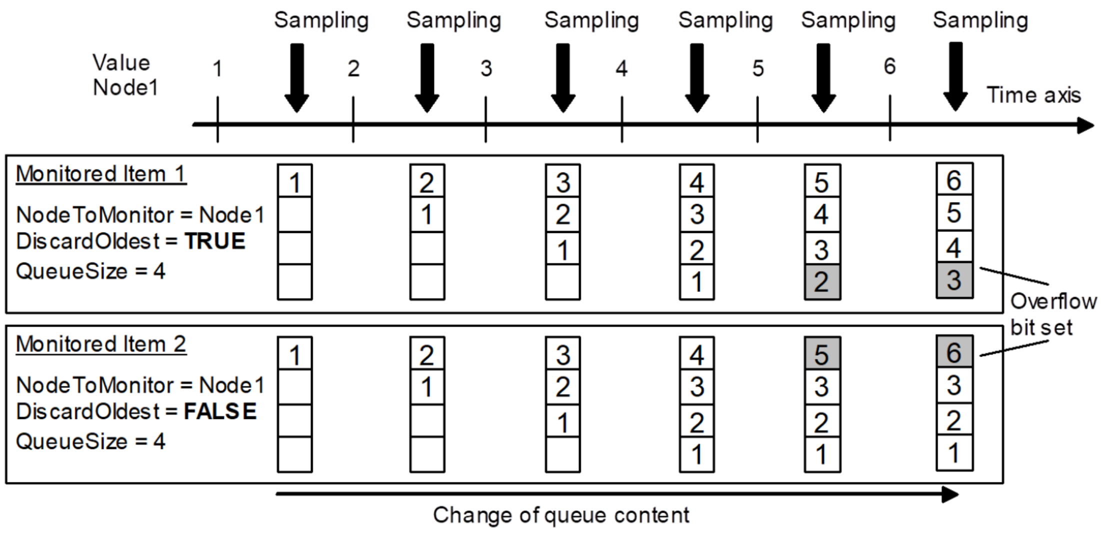


Figure 7.13 : Handling of the queue saturation

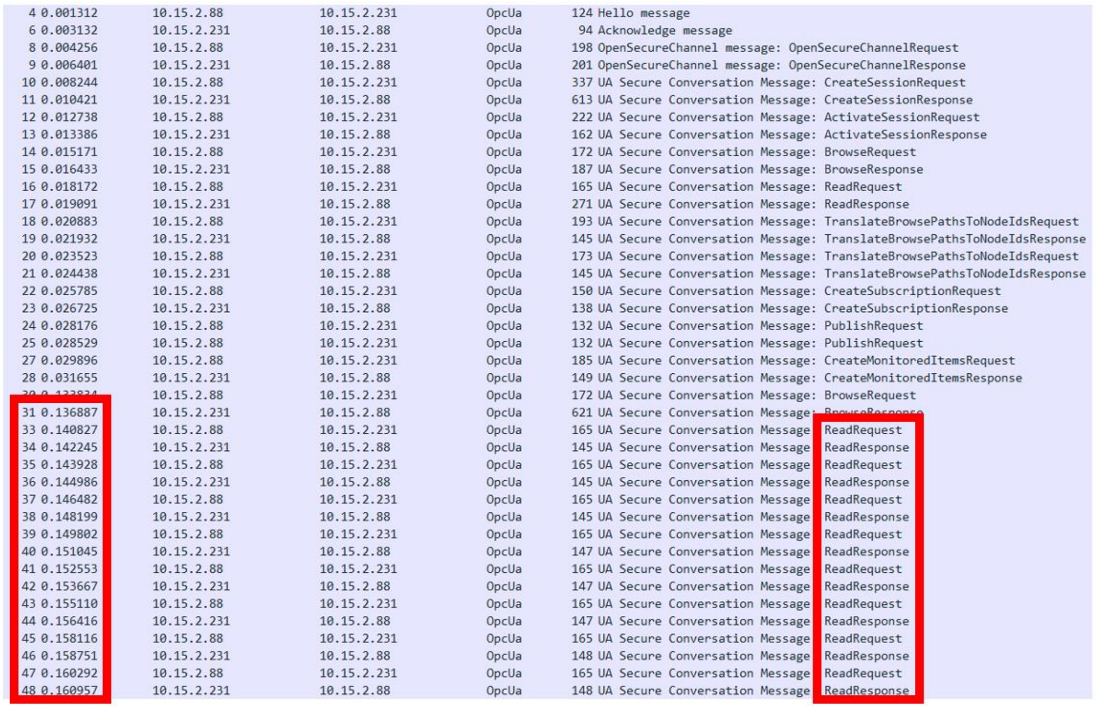
Now that the subscription procedure is finally completed, client and server can exchange packets containing values of the variable subscribed by the client.

The client sends a *PublishRequest* to the server. When the server will receive it, is allowed to transit a *PublishResponse* for the notification of an event over the subscribed variable. The *PublishResponse* is not immediate. As previously mentioned, the instant of transmission depends on several factors, like the *Sampling interval* or the instant at which there is a change in the status of the variable.

This procedure allows to achieve the best performance in terms bandwidth consumption.

In case the Client does not require any subscription, variables are reported with a polling method. This means a certain variable is reported cyclically using *ReadRequest*/*ReadResponse* like in the case of Open62541.

This case is shown in Figure 7.14, where alle the *ReadRequest*/*ReadResponse* starts to be exchanged starting from packet 31, even if there are no changes in the variable and with a very high periodicity.



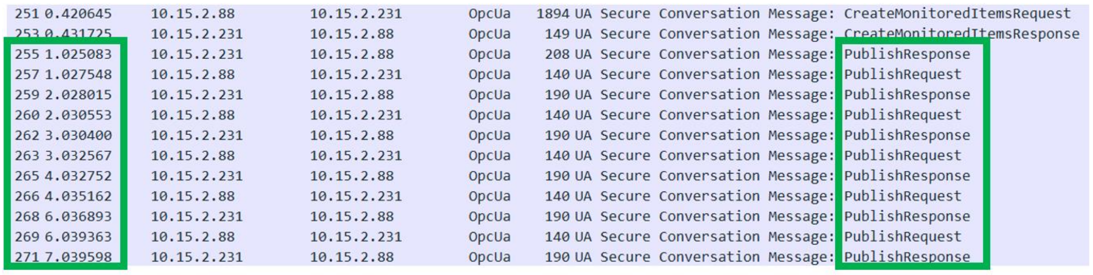
```
 4 0.001312    10.15.2.88     10.15.2.231    OpcUa    124 Hello message
 6 0.003132    10.15.2.231    10.15.2.88     OpcUa     94 Acknowledge message
 8 0.004256    10.15.2.88     10.15.2.231    OpcUa    198 OpenSecureChannel message: OpenSecureChannelRequest
 9 0.006401    10.15.2.231    10.15.2.88     OpcUa    201 OpenSecureChannel message: OpenSecureChannelResponse
10 0.008244    10.15.2.88     10.15.2.231    OpcUa    337 UA Secure Conversation Message: CreateSessionRequest
11 0.010421    10.15.2.231    10.15.2.88     OpcUa    613 UA Secure Conversation Message: CreateSessionResponse
12 0.012738    10.15.2.88     10.15.2.231    OpcUa    222 UA Secure Conversation Message: ActivateSessionRequest
13 0.013386    10.15.2.231    10.15.2.88     OpcUa    162 UA Secure Conversation Message: ActivateSessionResponse
14 0.015171    10.15.2.88     10.15.2.231    OpcUa    172 UA Secure Conversation Message: BrowseRequest
15 0.016433    10.15.2.231    10.15.2.88     OpcUa    187 UA Secure Conversation Message: BrowseResponse
16 0.018172    10.15.2.88     10.15.2.231    OpcUa    165 UA Secure Conversation Message: ReadRequest
17 0.019091    10.15.2.231    10.15.2.88     OpcUa    271 UA Secure Conversation Message: ReadResponse
18 0.020883    10.15.2.88     10.15.2.231    OpcUa    193 UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsRequest
19 0.021932    10.15.2.231    10.15.2.88     OpcUa    145 UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsResponse
20 0.023523    10.15.2.88     10.15.2.231    OpcUa    173 UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsRequest
21 0.024438    10.15.2.231    10.15.2.88     OpcUa    145 UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsResponse
22 0.025785    10.15.2.88     10.15.2.231    OpcUa    150 UA Secure Conversation Message: CreateSubscriptionRequest
23 0.026725    10.15.2.231    10.15.2.88     OpcUa    138 UA Secure Conversation Message: CreateSubscriptionResponse
24 0.028176    10.15.2.88     10.15.2.231    OpcUa    132 UA Secure Conversation Message: PublishRequest
25 0.028529    10.15.2.88     10.15.2.231    OpcUa    132 UA Secure Conversation Message: PublishRequest
27 0.029896    10.15.2.88     10.15.2.231    OpcUa    185 UA Secure Conversation Message: CreateMonitoredItemsRequest
28 0.031655    10.15.2.231    10.15.2.88     OpcUa    149 UA Secure Conversation Message: CreateMonitoredItemsResponse
30 0.133834    10.15.2.88     10.15.2.231    OpcUa    172 UA Secure Conversation Message: BrowseRequest
31 0.136887    10.15.2.231    10.15.2.88     OpcUa    621 UA Secure Conversation Message: BrowseResponse
33 0.140827    10.15.2.88     10.15.2.231    OpcUa    165 UA Secure Conversation Message ReadRequest
34 0.142245    10.15.2.231    10.15.2.88     OpcUa    145 UA Secure Conversation Message ReadResponse
35 0.143928    10.15.2.88     10.15.2.231    OpcUa    165 UA Secure Conversation Message ReadRequest
36 0.144986    10.15.2.231    10.15.2.88     OpcUa    145 UA Secure Conversation Message ReadResponse
37 0.146482    10.15.2.88     10.15.2.231    OpcUa    165 UA Secure Conversation Message ReadRequest
38 0.148199    10.15.2.231    10.15.2.88     OpcUa    145 UA Secure Conversation Message ReadResponse
39 0.149802    10.15.2.88     10.15.2.231    OpcUa    165 UA Secure Conversation Message ReadRequest
40 0.151045    10.15.2.231    10.15.2.88     OpcUa    147 UA Secure Conversation Message ReadResponse
41 0.152553    10.15.2.88     10.15.2.231    OpcUa    165 UA Secure Conversation Message ReadRequest
42 0.153667    10.15.2.231    10.15.2.88     OpcUa    147 UA Secure Conversation Message ReadResponse
43 0.155110    10.15.2.88     10.15.2.231    OpcUa    165 UA Secure Conversation Message ReadRequest
44 0.156416    10.15.2.231    10.15.2.88     OpcUa    147 UA Secure Conversation Message ReadResponse
45 0.158116    10.15.2.88     10.15.2.231    OpcUa    165 UA Secure Conversation Message ReadRequest
46 0.158751    10.15.2.231    10.15.2.88     OpcUa    148 UA Secure Conversation Message ReadResponse
47 0.160292    10.15.2.88     10.15.2.231    OpcUa    165 UA Secure Conversation Message ReadRequest
48 0.160957    10.15.2.231    10.15.2.88     OpcUa    148 UA Secure Conversation Message ReadResponse
```

Figure 7.14 : Traffic before the creation of the subscription

After completing the subscription things are improves a lot in terms of bandwidth consumption. In case a certain variable that the client is subscribed in, is changing slowly, there is a big reduction the packet exchanged. There is no more the need of cyclically send and receive *ReadRequest*/*ReadResponse*. All the variables that are included in the subscription are included in the same *PublishRequest*/*PublishResponse* packet.

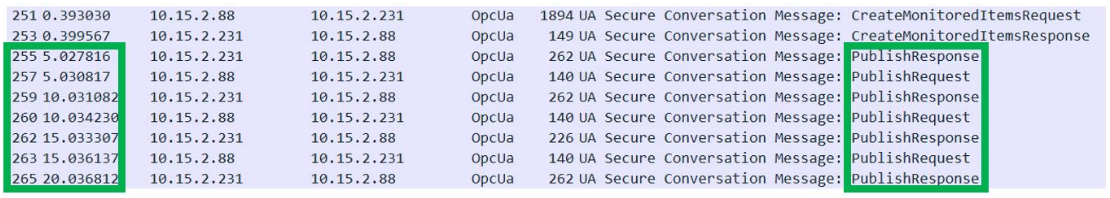
This means that in case of subscription to multiple variables, the saving in terms of bandwidth is bigger, because all the notification changes are grouped inside a single *PublishRequest*/*PublishResponse*.

This mechanism can be verified in different traffic captures reported from Figures 7.15 to 7.18. The first column represents the time passed since the capture is started. It is possible to see that by increasing the Sampling interval, also the *PublishRequest*/*PublishResponse* periodicity increases.



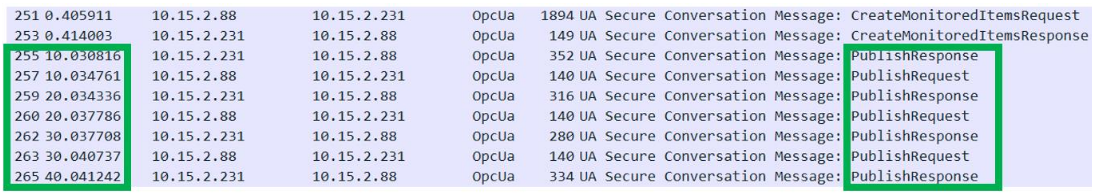
```
251 0.420645    10.15.2.88      10.15.2.231     OpcUa   1894 UA Secure Conversation Message: CreateMonitoredItemsRequest
253 0.431725    10.15.2.231     10.15.2.88      OpcUa    149 UA Secure Conversation Message: CreateMonitoredItemsResponse
255 1.025083    10.15.2.231     10.15.2.88      OpcUa    208 UA Secure Conversation Message: PublishResponse
257 1.027548    10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
259 2.028015    10.15.2.231     10.15.2.88      OpcUa    190 UA Secure Conversation Message: PublishResponse
260 2.030553    10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
262 3.030400    10.15.2.231     10.15.2.88      OpcUa    190 UA Secure Conversation Message: PublishResponse
263 3.032567    10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
265 4.032752    10.15.2.231     10.15.2.88      OpcUa    190 UA Secure Conversation Message: PublishResponse
266 4.035162    10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
268 6.036893    10.15.2.231     10.15.2.88      OpcUa    190 UA Secure Conversation Message: PublishResponse
269 6.039363    10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
271 7.039598    10.15.2.231     10.15.2.88      OpcUa    190 UA Secure Conversation Message: PublishResponse
```

Figure 7.15 : (Server publication time = 1s) (Sampling interval = 1s)



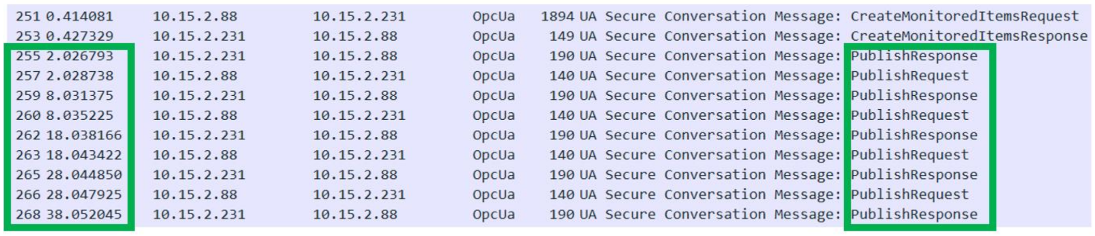
```
251 0.393030    10.15.2.88      10.15.2.231     OpcUa   1894 UA Secure Conversation Message: CreateMonitoredItemsRequest
253 0.399567    10.15.2.231     10.15.2.88      OpcUa    149 UA Secure Conversation Message: CreateMonitoredItemsResponse
255 5.027816    10.15.2.231     10.15.2.88      OpcUa    262 UA Secure Conversation Message: PublishResponse
257 5.030817    10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
259 10.031082   10.15.2.231     10.15.2.88      OpcUa    262 UA Secure Conversation Message: PublishResponse
260 10.034230   10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
262 15.033307   10.15.2.231     10.15.2.88      OpcUa    226 UA Secure Conversation Message: PublishResponse
263 15.036137   10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
265 20.036812   10.15.2.231     10.15.2.88      OpcUa    262 UA Secure Conversation Message: PublishResponse
```

Figure 7.16 : (Server publication time = 1s) (Sampling interval = 5s)



```
251 0.405911    10.15.2.88      10.15.2.231     OpcUa   1894 UA Secure Conversation Message: CreateMonitoredItemsRequest
253 0.414003    10.15.2.231     10.15.2.88      OpcUa    149 UA Secure Conversation Message: CreateMonitoredItemsResponse
255 10.030816   10.15.2.231     10.15.2.88      OpcUa    352 UA Secure Conversation Message: PublishResponse
257 10.034761   10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
259 20.034336   10.15.2.231     10.15.2.88      OpcUa    316 UA Secure Conversation Message: PublishResponse
260 20.037786   10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
262 30.037708   10.15.2.231     10.15.2.88      OpcUa    280 UA Secure Conversation Message: PublishResponse
263 30.040737   10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
265 40.041242   10.15.2.231     10.15.2.88      OpcUa    334 UA Secure Conversation Message: PublishResponse
```

Figure 7.17 : (Server publication time = 1s) (Sampling interval = 10s)



```
251 0.414081    10.15.2.88      10.15.2.231     OpcUa   1894 UA Secure Conversation Message: CreateMonitoredItemsRequest
253 0.427329    10.15.2.231     10.15.2.88      OpcUa    149 UA Secure Conversation Message: CreateMonitoredItemsResponse
255 2.026793    10.15.2.231     10.15.2.88      OpcUa    190 UA Secure Conversation Message: PublishResponse
257 2.028738    10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
259 8.031375    10.15.2.231     10.15.2.88      OpcUa    190 UA Secure Conversation Message: PublishResponse
260 8.035225    10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
262 18.038166   10.15.2.231     10.15.2.88      OpcUa    190 UA Secure Conversation Message: PublishResponse
263 18.043422   10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
265 28.044850   10.15.2.231     10.15.2.88      OpcUa    190 UA Secure Conversation Message: PublishResponse
266 28.047925   10.15.2.88      10.15.2.231     OpcUa    140 UA Secure Conversation Message: PublishRequest
268 38.052045   10.15.2.231     10.15.2.88      OpcUa    190 UA Secure Conversation Message: PublishResponse
```

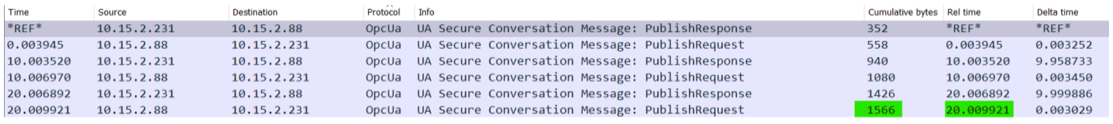
Figure 7.18 : (Server publication time = 10s) (Sampling interval = 2s)

Just to make a rough estimation I also report the amount of data saved comparing the cases shown in Figure 7.14 and Figure 7.17.

Figure 7.14 represents the case before the subscription where *ReadRequest*/*ReadResponse* are exchanged with an approximate periodicity of 0.0027 seconds. A *ReadRequest* has a size of 165 bytes, while a *ReadResponse* only 145 bytes. Considering a period of 0.05 seconds, the total amount of data exchanged using these two types of packets is 6708 bytes.

In this case if we consider 20 seconds an approximate data flow between client and server is around 134160 bytes

Figure 7.17 instead, is the case of subscription and the MonitoredItem procedure already completed with a Sampling interval equal to 10 seconds.



Time	Source	Destination	Protocol	Info	Cumulative bytes	Rel time	Delta time
REF	10.15.2.231	10.15.2.88	OpcUa	UA Secure Conversation Message: PublishResponse	352	*REF*	*REF*
0.003945	10.15.2.88	10.15.2.231	OpcUa	UA Secure Conversation Message: PublishRequest	558	0.003945	0.003252
10.003520	10.15.2.231	10.15.2.88	OpcUa	UA Secure Conversation Message: PublishResponse	940	10.003520	9.958733
10.006970	10.15.2.88	10.15.2.231	OpcUa	UA Secure Conversation Message: PublishRequest	1080	10.006970	0.003450
20.006892	10.15.2.231	10.15.2.88	OpcUa	UA Secure Conversation Message: PublishResponse	1426	20.006892	9.999886
20.009921	10.15.2.88	10.15.2.231	OpcUa	UA Secure Conversation Message: PublishRequest	1566	20.009921	0.003029

Figure 7.19 : Data saving with (Server publication time = 1s) (Sampling interval = 10s)

In this case the amount of data exchanged in 10 seconds is 1566 bytes (as show in Figure 7.19). This rapresent an improvement of almost 98% of the previuos case.

No subscription Pubblication time = 1s	Subscription Pubblication time = 1s Sampling interval = 1s	Subscription Pubblication time = 1s Sampling interval = 5s	Subscription Pubblication time = 1s Sampling interval = 10s
134160 bytes	8194 bytes	2304 bytes	1566 bytes

Table 3 : Traffic referred to a Server publication time = 1s

In Table 3 are reported all the different traffic size in different conditions.

In Figure 7.20 is reported the trend of the traffic size. As previously mentioned, by increasing the Sampling interval, the traffic size decreases because all the new changes of a variable are grouped and reported inside a single packet.

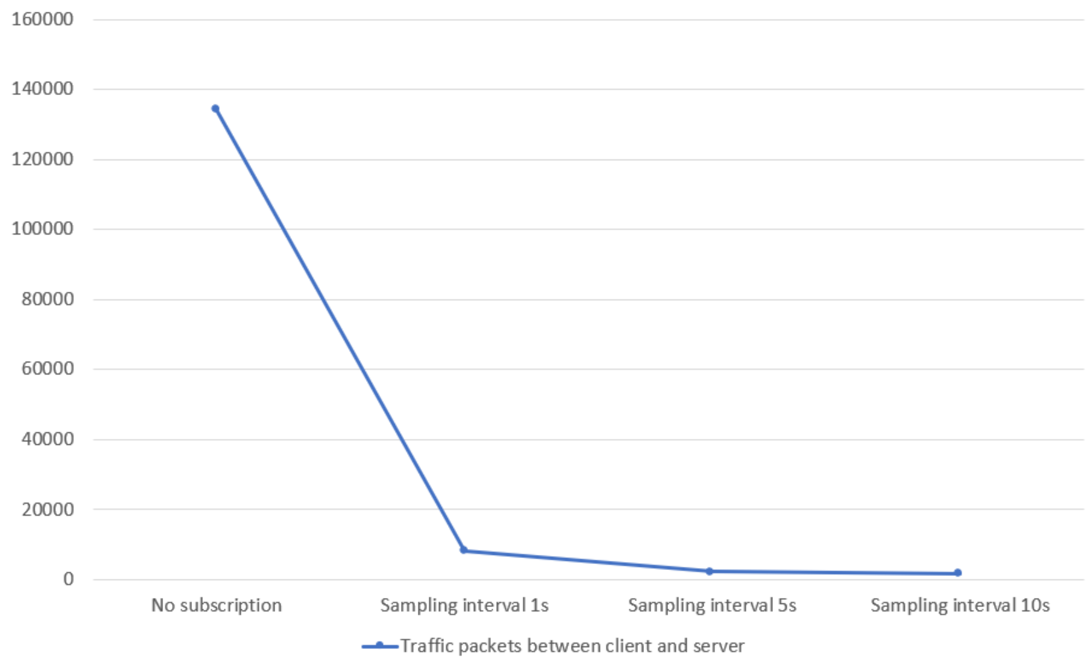


Figure 7.20 : Traffic packets between client and server

7.3 **Code OPC UA Client and Server**

In this section I show a simple example with a brief explanation of the Python code used for the interaction between an OPC UA Client and Server including the possibility of subscription.

7.3.1 *Python code OPC UA Server*

OPC UA Server script

```
import uuid
from threading import Thread
import copy
import logging
from datetime import datetime
import time
from math import sin
import sys
import random
from opcua.ua import NodeId, NodeIdType

sys.path.insert(0, "..")

try:
    from IPython import embed
except ImportError:
    import code
    def embed():
        myvars = globals()
        myvars.update(locals())
        shell = code.InteractiveConsole(myvars)
        shell.interact()

from opcua import ua, uamethod, Server

class SubHandler(object):

    def datachange_notification(self, node, val, data):
        print("Python: New data change event", node, val)

# method to be exposed through server

def func(parent, variant):
    ret = False
    if variant.Value % 2 == 0:
        ret = True
    return [ua.Variant(ret, ua.VariantType.Boolean)]

@uamethod
def multiply(parent, x, y):
    print("multiply method call with parameters: ", x, y)
    return x * y

class VarUpdater(Thread):
    def __init__(self, var):
        Thread.__init__(self)
        self._stopev = False
        self.var = var

    def stop(self):
        self._stopev = True

    def run(self):
        while not self._stopev:
            v = sin(time.time() / 10)
            self.var.set_value(v)
            time.sleep(0.1)

if __name__ == "__main__":
    # optional: setup logging
    logging.basicConfig(level=logging.WARN)
    # now setup our server
    server = Server()

    hostname = str(sys.argv[1])
    port = str(sys.argv[2])

    domain = "opc.tcp://"
    ddd = ":"
    final_address = (domain+hostname+ddd+port)
    server.set_endpoint(final_address)

    server.set_server_name("FreeOpcUa Example Server")

    server.set_security_policy([
                ua.SecurityPolicyType.NoSecurity,
                ua.SecurityPolicyType.Basic256Sha256_SignAndEncrypt,
                ua.SecurityPolicyType.Basic256Sha256_Sign])

    # setup our own namespace
    uri = str(sys.argv[4])
    idx = server.register_namespace(uri)
    # create a new node type we can instantiate in our address space
    dev = server.nodes.base_object_type.add_object_type(idx, "MyDevice")
    dev.add_variable(idx, "sensor1", 1.0).set_modelling_rule(True)
    dev.add_property(idx, "device_id", "0340").set_modelling_rule(True)
    ctrl = dev.add_object(idx, "controller")
    ctrl.set_modelling_rule(True)
    ctrl.add_property(idx, "state", "Idle").set_modelling_rule(True)

    # populating our address space

    # First a folder to organise our nodes
    myfolder = server.nodes.objects.add_folder(idx, "myEmptyFolder")
    # instanciate one instance of our device
    mydevice = server.nodes.objects.add_object(idx, "Device0001", dev)
    mydevice_var = mydevice.get_child(["{}:controller".format(idx), "{}:state".format(idx)])
 # get proxy to our device state variable

    # create directly some objects and variables
    #Definition of the new object
    obj = str(sys.argv[5])
    myobj = server.nodes.objects.add_object(idx, obj)

    #Definition of sensors
    sensore1 = str(sys.argv[6])
    sensore2 = str(sys.argv[7])
    myvar = myobj.add_variable(idx, sensore1, 6.7)
    mysin = myobj.add_variable(idx, sensore2, 0, ua.VariantType.Float)
    myvar.set_writable()    # Set MyVariable to be writable by clients
    mystringvar = myobj.add_variable(idx, "MyStringVariable", "Really nice string")
    mystringvar.set_writable()  # Set MyVariable to be writable by clients
    myguidvar = myobj.add_variable(NodeId(uuid.UUID('1be5ba38-d004-46bd-aa3a-
b5b87940c698'), idx, NodeIdType.Guid),
                                   'MyStringVariableWithGUID', 'NodeId type is guid')
    mydtvar = myobj.add_variable(idx, "MyDateTimeVar", datetime.utcnow())
    mydtvar.set_writable()    # Set MyVariable to be writable by clients
    myarrayvar = myobj.add_variable(idx, "myarrayvar", [6.7, 7.9])
    myarrayvar = myobj.add_variable(idx, "myStronglytTypedVariable", ua.Variant([], ua.Varian
tType.UInt32))
    myprop = myobj.add_property(idx, "myproperty", "I am a property")
    mymethod = myobj.add_method(idx, "mymethod", func, [ua.VariantType.Int64], [ua.VariantTyp
e.Boolean])
    multiply_node = myobj.add_method(idx, "multiply", multiply, [ua.VariantType.Int64, ua.Var
iantType.Int64], [ua.VariantType.Int64])

    # import some nodes from xml
    server.import_xml("custom_nodes.xml")
    myevgen = server.get_event_generator()
    myevgen.event.Severity = 300

    # starting!
    server.start()
    print("Available loggers are: ", logging.Logger.manager.loggerDict.keys())
    vup = VarUpdater(mysin)  # just  a stupide class update a variable
    vup.start()
    try:
        var = myarrayvar.get_value()
        var = copy.copy(var)
        var.append(9.3)
        myarrayvar.set_value(var)
        mydevice_var.set_value("Running")
        myevgen.trigger(message="This is BaseEvent")
        server.set_attribute_value(myvar.nodeid, ua.DataValue(9.9))

        sleep = int(sys.argv[3])

        while 15 == 15:
              temp = random.randint(1,10)
              server.set_attribute_value(myvar.nodeid, ua.DataValue(temp))
              print("Nuovo valore myvar : ", temp )
              time.sleep(sleep)

        embed()
    finally:
        vup.stop()
        server.stop()
```

The script of the Server requires 7 paramters to be executed:

- The *ip addres* where the server is executed

```
hostname = str(sys.argv[1])
```

- The *port* where the service is exposed

```
port = str(sys.argv[2])
```

- The *periodicity* between a publication and the following ones

```
sleep = int(sys.argv[3])
```

- The *uri* to be set in the namespace containing the variables of the server.

```
uri = str(sys.argv[4])
idx = server.register_namespace(uri)
```

- The name of the object which contains the variables.

```
obj = str(sys.argv[5])
```

- The names of the variables.

```
sensore1 = str(sys.argv[6])
sensore2 = str(sys.argv[7])
```

This script is capable of the creation of an object with two variables inside the namespace of the Server.

```
# create directly some objects and variables
#Definition of the new object
  myobj = server.nodes.objects.add_object(idx, obj)

#Definition of sensors
 myvar = myobj.add_variable(idx, sensor1, 6.7)
 myvar.set_writable()    # Set MyVariable to be writable by clients
```

After that the server can be started:

```
# now setup our server
    server = Server()
```

As example the variable *sensor1* is added inside the object of the server's namespace (referenced through the variable *myvar*). Is also initialized at an initial value 6.7 and set as writable by the user.

```
myvar = myobj.add_variable(idx, sensore1, 6.7)
myvar.set_writable()
```

Then a random number is generated and associated to that variable (still referenced through *myvar*) using an infinite loop while.

```
while 15 == 15:
              temp = random.randint(1,10)
              server.set_attribute_value(myvar.nodeid, ua.DataValue(temp))
              print("Nuovo valore myvar : ", temp )
              time.sleep(sleep)
```

The periodicity of publication is respected through the parameter **sleep** defined by the user.

Inside the server is also defined a method called *multiply* that can be accessed by the client using command line. This method takes in input two variables and gives back their product.

```
@uamethod
def multiply(parent, x, y):
    print("multiply method call with parameters: ", x, y)
    return x * y
```

As example, the server can be executed using the command:

```
python3 server-example.py 0.0.0.0 4840 1 http://examples.freeopcua.github.io object1 sensor1 sensor2
```

In this case the parameters are:

- `hostname = 0.0.0.0`

  Using 0.0.0.0 we are executing the server in all the addresses available.
- `port = 4840`
- `sleep = 1 second`
- `uri = http://examples.freeopcua.github.io`
- `obj = object1`
- `sensore1 = sensor1`
- `sensore2 = sensor2`

7.3.2 *Python code OPC UA Client*

OPC UA Client script

```
import sys
sys.path.insert(0, "..")
import logging
import time

try:
    from IPython import embed
except ImportError:
    import code

    def embed():
        vars = globals()
        vars.update(locals())
        shell = code.InteractiveConsole(vars)
        shell.interact()
```

```
from opcua import Client
from opcua import ua


class SubHandler(object):


    def datachange_notification(self, node, val, data):
        print("Python: myvar is: ", val)

    def event_notification(self, event):
        print("Python: New event", event)


if __name__ == "__main__":
    logging.basicConfig(level=logging.WARN)

    hostname = str(sys.argv[1])
    port = str(sys.argv[2])
    domain = "opc.tcp://"
    ddd = ":"
    final_address = (domain+hostname+ddd+port)

    client = Client(final_address)

    try:
        client.connect()
        client.load_type_definitions()  # load definition of server specific
structures/extension objects

        # Client has a few methods to get proxy to UA nodes that should always be in address
space such as Root or Objects
        print("---------------------------------------")
        root = client.get_root_node()
        print("Root node is: ", root)
        objects = client.get_objects_node()
        print("Objects node is: ", objects)
        print("---------------------------------------")

        # gettting our namespace idx
        uri = str(sys.argv[4])
        idx = client.get_namespace_index(uri)

        # Now getting a variable node using its browse path
        print("---------------------------------------")
        print("Informazioni variabile myvar")
        myvar = root.get_child(["0:Objects", "{}:object1".format(idx),
"{}:sensor1".format(idx)])
        obj = root.get_child(["0:Objects", "{}:object1".format(idx)])
        print("myvar is: ", myvar)

        print("---------------------------------------")

        # subscribing to a variable node
        handler = SubHandler()
        tempo_aggiornamento = 1000 * int(sys.argv[3])
        sub = client.create_subscription(tempo_aggiornamento, handler)
        handle = sub.subscribe_data_change(myvar)
        time.sleep(0.1)

        # we can also subscribe to events from server
        sub.subscribe_events()

        # calling a method on server
        print("---------------------------------------")
        res2 = obj.call_method("{}:multiply".format(idx), 3, 2)
        print("method result is: ", res2)
        print("---------------------------------------")

        embed()
    finally:
        client.disconnect()
```

The script of the Client requires 4 paramters to be executed:

- The *ip addres* where the server is executed.

```
hostname = str(sys.argv[1])
```

- The *port* where the service is exposed.

```
port = str(sys.argv[2])
```

- The *sampling rate* at which the data is reported to the client.

```
tempo_aggiornamento = 1000 * int(sys.argv[3])
```

- The *uri* of the server's namespace where are stored objects and variables.

```
uri = str(sys.argv[4])

idx = client.get_namespace_index(uri)
```

This piece of software allows the connection to the Server identified by the endpoint information passed as parameters.

Once the user set the *ip address* and the *port* of the server, the connection can be established:

```
try:
        client.connect()
```

Through the variable *myvar* is referenced the variable *sensor1* contained inside the object *object1* inside the server's namespace:

```
myvar = root.get_child(["0:Objects", "{}:object1".format(idx), "{}:sensor1".format(idx)])
obj = root.get_child(["0:Objects", "{}:object1".format(idx)])
```

Now the subscription to that variable can be requested specifying the *sampling interval*:

```
# subscribing to a variable node
        handler = SubHandler()
        tempo_aggiornamento = 1000 * int(sys.argv[3])
        sub = client.create_subscription(tempo_aggiornamento, handler)
        handle = sub.subscribe_data_change(myvar)
        time.sleep(0.1)
```

The subscription in controlled by the function *datachange_notification* that publish the value of the variable (or the list of values that are published during the *sampling interval*):

```
def datachange_notification(self, node, val, data):
        print("Python: myvar is: ", val)
```

Is also set the subscription to all the events of the server:

```
# we can also subscribe to events from server
        sub.subscribe_events()
```

Finally, two numbers are passed to the method *multiply* defined by the server:

```
# calling a method on server
print("---------------------------------------")
res2 = obj.call_method("{}:multiply".format(idx), 3, 2)
print("method result is: ", res2)
print("---------------------------------------")
```

As result the server responds with the result of their product.

In Tables 4 and 5 are reported all the output bash of the Server and the Client. It is possible to see that all the values published by the server are correctly reported by to the client.

OPC UA Server
```
ubuntu@lorenzo-vm:~/ $ python3 server-example.py 0.0.0.0 4840 1 http://examples.freeopcua.github.io object1 sensor1 sensor2

WARNING:opcua.server.server:Endpoints other than open requested but private key and certificate are not set.

WARNING:opcua.server.binary_server_asyncio:Listening on 0.0.0.0:4840

Nuovo valore myvar :  9

Nuovo valore myvar :  5

multiply method call with parameters:  3 2

Nuovo valore myvar :  3

Nuovo valore myvar :  5
```

```
Nuovo valore myvar :  4

Nuovo valore myvar :  10

Nuovo valore myvar :  10

Nuovo valore myvar :  2

Nuovo valore myvar :  10

Nuovo valore myvar :  5

Nuovo valore myvar :  2

Nuovo valore myvar :  8

Nuovo valore myvar :  5
```

Table 4: OPC UA Server bash

OPC UA Client

```
ubuntu@opc-ua-server-1-mgmtvm-0:~/$ python3 client-python.py 10.15.2.231 4840 1 http://examples.freeopcua.github.io

Root node is:  i=84

Objects node is:  i=85

---------------------------------------

Informazioni variabile myvar

myvar is:  ns=2;i=13

---------------------------------------

method result is:  6

---------------------------------------

Python 3.6.9 (default, Jan 26 2021, 15:33:00)

>>> Python: myvar is:  5

Python: myvar is:  3

Python: myvar is:  5

Python: myvar is:  4

Python: myvar is:  10

Python: myvar is:  2

Python: myvar is:  10

Python: myvar is:  5

Python: myvar is:  2

Python: myvar is:  8
```

Table 5: OPC UA Client bash

Server (Publish interval = 1s)	Client (Sampling interval = 1s)
Nuovo valore myvar : 9	
Nuovo valore myvar : 5	>>> Python: myvar is: 5
Nuovo valore myvar : 3	Python: myvar is: 3
Nuovo valore myvar : 5	Python: myvar is: 5
Nuovo valore myvar : 4	Python: myvar is: 4
Nuovo valore myvar : 10	Python: myvar is: 10
Nuovo valore myvar : 10	Python: myvar is: 2
Nuovo valore myvar : 2	Python: myvar is: 10
Nuovo valore myvar : 10	Python: myvar is: 5
Nuovo valore myvar : 5	Python: myvar is: 2
Nuovo valore myvar : 2	Python: myvar is: 8
Nuovo valore myvar : 8	

Table 6: Client-Server communication with sampling interval = 1s

Table 6 reports the side by side the output of Tables 4 and 5. These results are referred to a case were server and client are set respectively with the same *publish interval* and *sampling interval* to 1 second. In this case the client is continuously updated at every publication of the server.

In case the sampling interval is set to 5 seconds (as reported in table 7), all the publication of the server in that period are grouped and reported in a single message.

Server (Pubblish interval = 1s)	Client (Sampling interval = 5s)
Nuovo valore myvar : 7	
Nuovo valore myvar : 5	
Nuovo valore myvar : 8	
Nuovo valore myvar : 4	>>> Python: myvar is: 4
Nuovo valore myvar : 10	Python: myvar is: 10
Nuovo valore myvar : 8	Python: myvar is: 8
Nuovo valore myvar : 2	Python: myvar is: 2

Nuovo valore myvar : 5	Python: myvar is: 5
	>>>
Nuovo valore myvar : 10	Python: myvar is: 10
Nuovo valore myvar : 2	Python: myvar is: 2
Nuovo valore myvar : 8	Python: myvar is: 8
Nuovo valore myvar : 3	Python: myvar is: 3
Nuovo valore myvar : 5	Python: myvar is: 5
	>>>
Nuovo valore myvar : 3	Python: myvar is: 3
Nuovo valore myvar : 1	Python: myvar is: 1
Nuovo valore myvar : 10	Python: myvar is: 10
Nuovo valore myvar : 2	Python: myvar is: 2
	>>>
Nuovo valore myvar : 10	Python: myvar is: 10
Nuovo valore myvar : 3	Python: myvar is: 3
Nuovo valore myvar : 10	Python: myvar is: 10
Nuovo valore myvar : 7	Python: myvar is: 7
	>>>
Nuovo valore myvar : 10	Python: myvar is: 10
Nuovo valore myvar : 8	Python: myvar is: 8
Nuovo valore myvar : 1	Python: myvar is: 1
Nuovo valore myvar : 6	Python: myvar is: 6
Nuovo valore myvar : 6	
Nuovo valore myvar : 1	

Table 7: Client-Server communication with sampling interval = 5s

It is important to notice from Table 7 that even if the server publication time is 1s and the client's sample interval is 5 seconds, not all the publication grouped are composed by 5 samples (highlighted in green). There are some cases (highlighted in red) where the samples reposted are only 4. This is because we need consider the network in between the two entities. There can be some packets lost that need to be retransmitted or maybe some packets follow different routes which translates in different RTTs.

7.4 **OPC-UA GUI**

There is also the possibility to use a graphical user interface. This GUI is available for client and there is also a *Modeler* version that allow to create a custom namespace without the need to write python code. It can be installed on Windows or Linux. Here I report how to install and use the GUI only in Linux OS.

7.4.1 *Client GUI*

Is written using *freeopcua* python *api* and *pyqt* [51]. This GUI can implement the most needed functionalities including subscribing for data changes and events, write variable values listing attributes and references, and call methods.

It can be easily installed and launched with the following commands (Python 3.6+ and python-pip are required):

```
1.  pip3 install opcua-client
2.  pip3 install --user pyqt5
3.  apt-get install python3-pyqt5
4.  apt-get install pyqt5-dev-tools
5.  opcua-client          #Run the GUI
```

As example an OPA UA server generates random number with a periodicity of 1 second, is running on address 10.0.2.15. The service is exposed on port 5555.

Once the GUI is launched, is necessary to insert the *ip address* and the *port* of the server where the service is running as shown in Figure 7.21 with the red line.

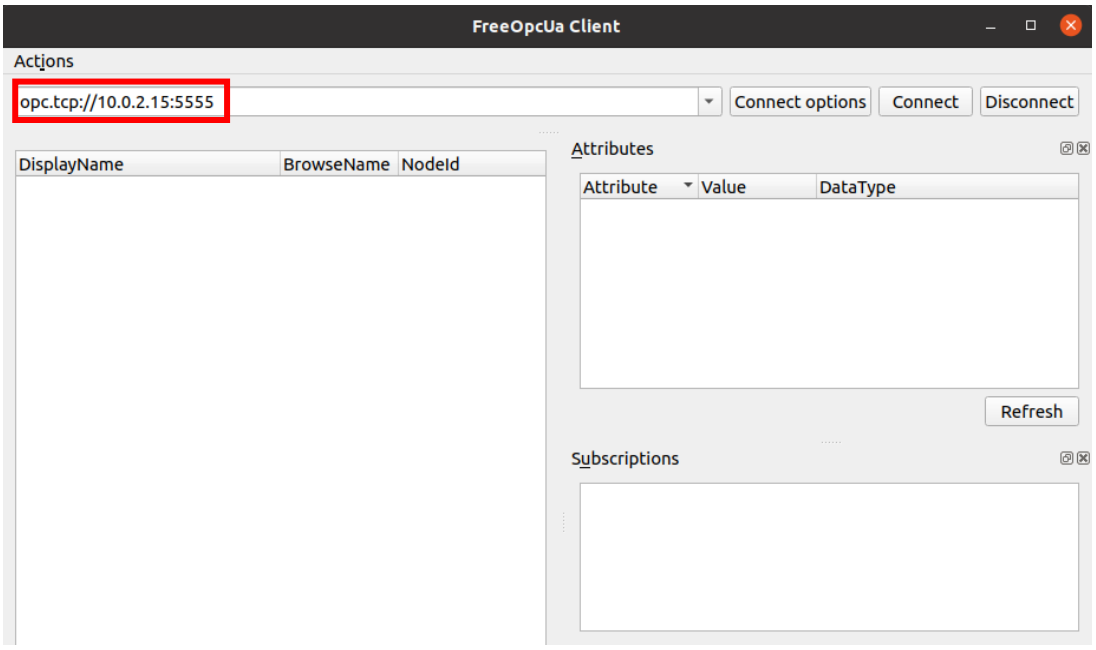


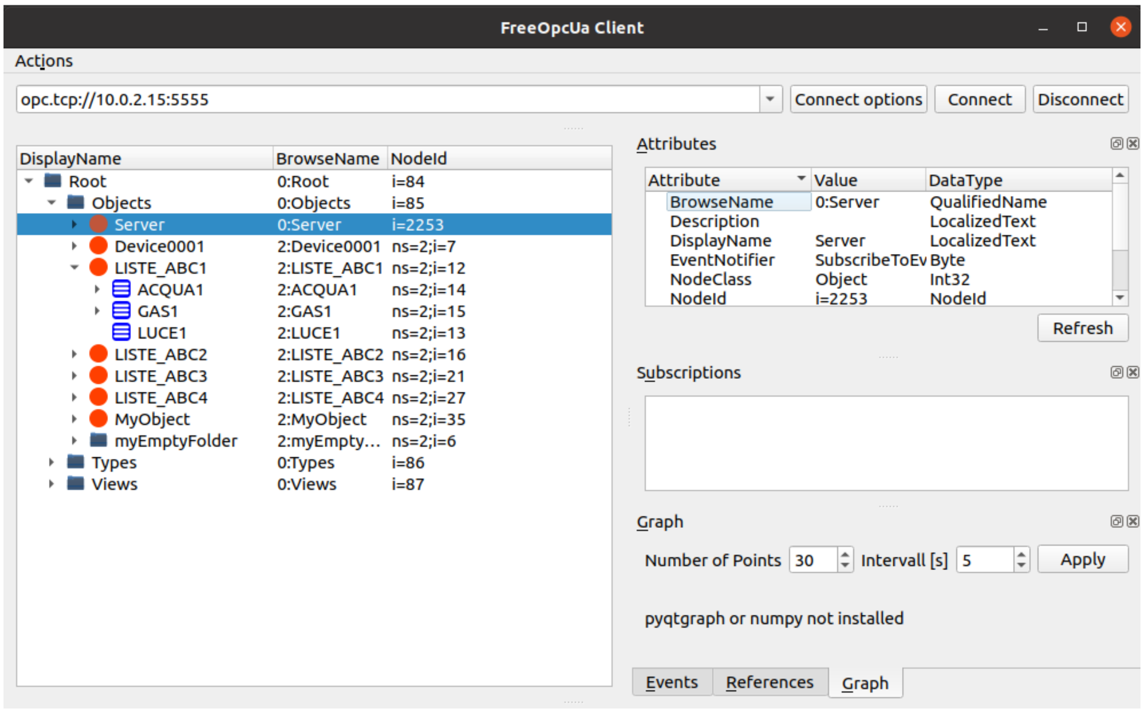
Figure 7.21: FreeOpcUa Client connection with the Server

Once the connection is established, on the Client GUI are shown all the variables and objects which populate the namespace of the server, as shown in Figure 7.22.

It is possible to see that all the objects and variables which populate the namespace of the server match perfectly with their definition.

All the objects and variable are defined inside a Json file that is accessed by the Server to properly build its namespace.

```
{
  "opcua": [
    {
      "port": "5555",
      "uri": "http://examples.freeopcua.github.io",
      "objects": [
        {
          "object_name": "LISTE_ABC1",
          "variables": [
            "LUCE1",
            "ACQUA1",
            "GAS1"
          ]
        },
        {
        "object_name": "LISTE_ABC2",
        "variables": [
            "LUCE2",
            "ACQUA2",
            "GAS2",
            "TERRA2"
          ]
        },
        {
        "object_name": "LISTE_ABC3",
        "variables": [
            "LUCE3",
            "ACQUA3",
            "GAS3",
            "TERRA3",
                      ]
        },
        {
        "object_name": "LISTE_ABC4",
        "variables": [
            "LUCE4",
            "ACQUA4",
            "GAS34",
            "TERRA4",
                     ]
        }

      ]
    }
  ]
}
```


FreeOpcUa Client

Actions

opc.tcp://10.0.2.15:5555 | Connect options | Connect | Disconnect

DisplayName	BrowseName	NodeId
Root	0:Root	i=84
Objects	0:Objects	i=85
Server	0:Server	i=2253
Device0001	2:Device0001	ns=2;i=7
LISTE_ABC1	2:LISTE_ABC1	ns=2;i=12
ACQUA1	2:ACQUA1	ns=2;i=14
GAS1	2:GAS1	ns=2;i=15
LUCE1	2:LUCE1	ns=2;i=13
LISTE_ABC2	2:LISTE_ABC2	ns=2;i=16
LISTE_ABC3	2:LISTE_ABC3	ns=2;i=21
LISTE_ABC4	2:LISTE_ABC4	ns=2;i=27
MyObject	2:MyObject	ns=2;i=35
myEmptyFolder	2:myEmpty...	ns=2;i=6
Types	0:Types	i=86
Views	0:Views	i=87

Attributes

Attribute	Value	DataType
BrowseName	0:Server	QualifiedName
Description		LocalizedText
DisplayName	Server	LocalizedText
EventNotifier	SubscribeToEv	Byte
NodeClass	Object	Int32
NodeId	i=2253	NodeId

Refresh

Subscriptions

Graph

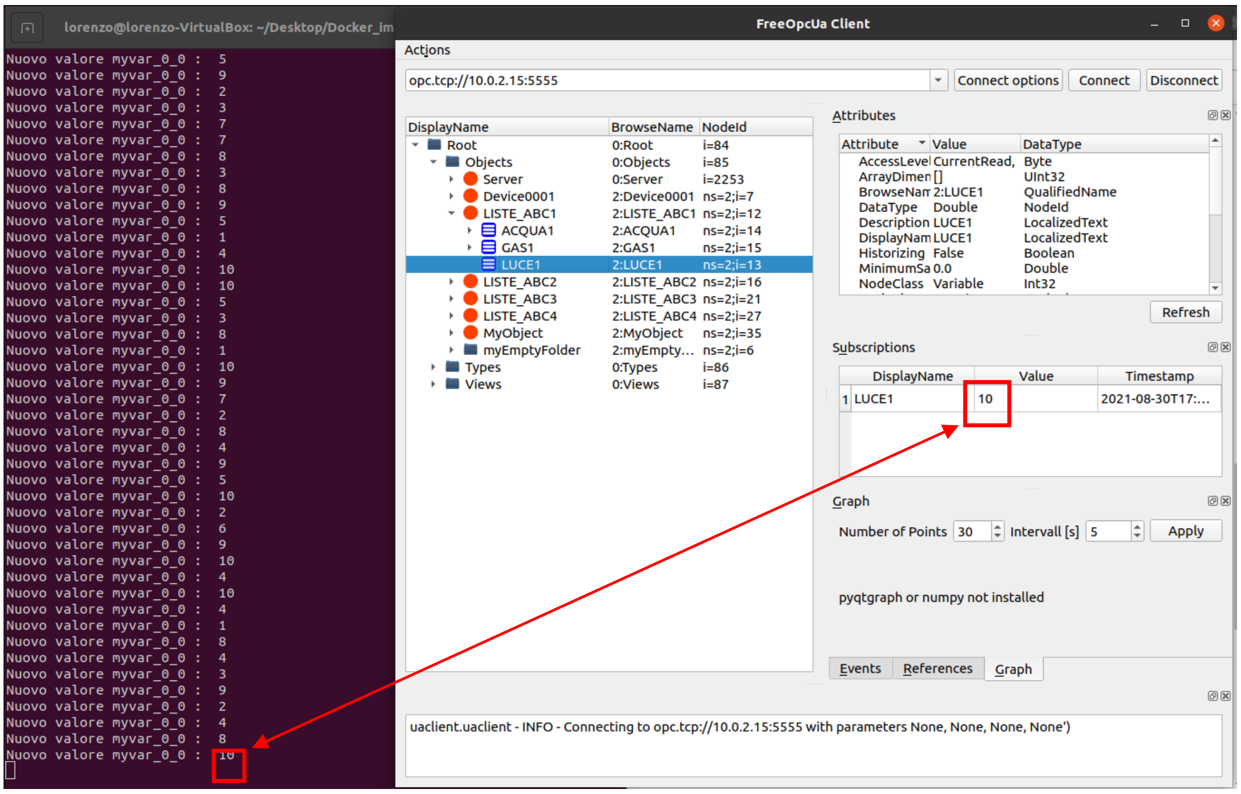
Number of Points 30 | Intervall [s] 5 | Apply

pyqtgraph or numpy not installed

Events | References | Graph

Figure 7.22: FreeOpcUa: connection with the server established

Now is possible to ask the subscription to a certain variable to see all the changes of that variable. The server is publishing numbers over the variable “LUCE1”.


lorenzo@lorenzo-VirtualBox: ~/Desktop/Docker_im

```
Nuovo valore myvar_0_0 :  5
Nuovo valore myvar_0_0 :  9
Nuovo valore myvar_0_0 :  2
Nuovo valore myvar_0_0 :  3
Nuovo valore myvar_0_0 :  7
Nuovo valore myvar_0_0 :  7
Nuovo valore myvar_0_0 :  8
Nuovo valore myvar_0_0 :  3
Nuovo valore myvar_0_0 :  8
Nuovo valore myvar_0_0 :  9
Nuovo valore myvar_0_0 :  5
Nuovo valore myvar_0_0 :  1
Nuovo valore myvar_0_0 :  4
Nuovo valore myvar_0_0 :  10
Nuovo valore myvar_0_0 :  10
Nuovo valore myvar_0_0 :  5
Nuovo valore myvar_0_0 :  3
Nuovo valore myvar_0_0 :  8
Nuovo valore myvar_0_0 :  1
Nuovo valore myvar_0_0 :  10
Nuovo valore myvar_0_0 :  9
Nuovo valore myvar_0_0 :  7
Nuovo valore myvar_0_0 :  2
Nuovo valore myvar_0_0 :  8
Nuovo valore myvar_0_0 :  4
Nuovo valore myvar_0_0 :  9
Nuovo valore myvar_0_0 :  5
Nuovo valore myvar_0_0 :  10
Nuovo valore myvar_0_0 :  2
Nuovo valore myvar_0_0 :  6
Nuovo valore myvar_0_0 :  9
Nuovo valore myvar_0_0 :  10
Nuovo valore myvar_0_0 :  4
Nuovo valore myvar_0_0 :  10
Nuovo valore myvar_0_0 :  4
Nuovo valore myvar_0_0 :  1
Nuovo valore myvar_0_0 :  8
Nuovo valore myvar_0_0 :  4
Nuovo valore myvar_0_0 :  3
Nuovo valore myvar_0_0 :  9
Nuovo valore myvar_0_0 :  2
Nuovo valore myvar_0_0 :  4
Nuovo valore myvar_0_0 :  8
Nuovo valore myvar_0_0 :  10
```

opc.tcp://10.0.2.15:5555

DisplayName	BrowseName	NodeId
Root	0:Root	i=84
Objects	0:Objects	i=85
Server	0:Server	i=2253
Device0001	2:Device0001	ns=2;i=7
LISTE_ABC1	2:LISTE_ABC1	ns=2;i=12
ACQUA1	2:ACQUA1	ns=2;i=14
GAS1	2:GAS1	ns=2;i=15
LUCE1	2:LUCE1	ns=2;i=13
LISTE_ABC2	2:LISTE_ABC2	ns=2;i=16
LISTE_ABC3	2:LISTE_ABC3	ns=2;i=21
LISTE_ABC4	2:LISTE_ABC4	ns=2;i=27
MyObject	2:MyObject	ns=2;i=35
myEmptyFolder	2:myEmpty...	ns=2;i=6
Types	0:Types	i=86
Views	0:Views	i=87

Attributes

Attribute	Value	DataType
AccessLevel	CurrentRead,	Byte
ArrayDimen	[]	UInt32
BrowseNam	2:LUCE1	QualifiedName
DataType	Double	NodeId
Description	LUCE1	LocalizedText
DisplayNam	LUCE1	LocalizedText
Historizing	False	Boolean
MinimumSa	0.0	Double
NodeClass	Variable	Int32

Subscriptions

	DisplayName	Value	Timestamp
1	LUCE1	10	2021-08-30T17:...

Graph

Number of Points 30 Intervall [s] 5

pyqtgraph or numpy not installed

```
uaclient.uaclient - INFO - Connecting to opc.tcp://10.0.2.15:5555 with parameters None, None, None, None')
```

Figure 7.23: FreeOpcUa: subscription to a variable

From Figure 7.23, it is possible to see that every time a new number is published by the server, is seen also in the Client's GUI with the correspondent timestamp of that value.

7.4.2 *Modeler GUI*

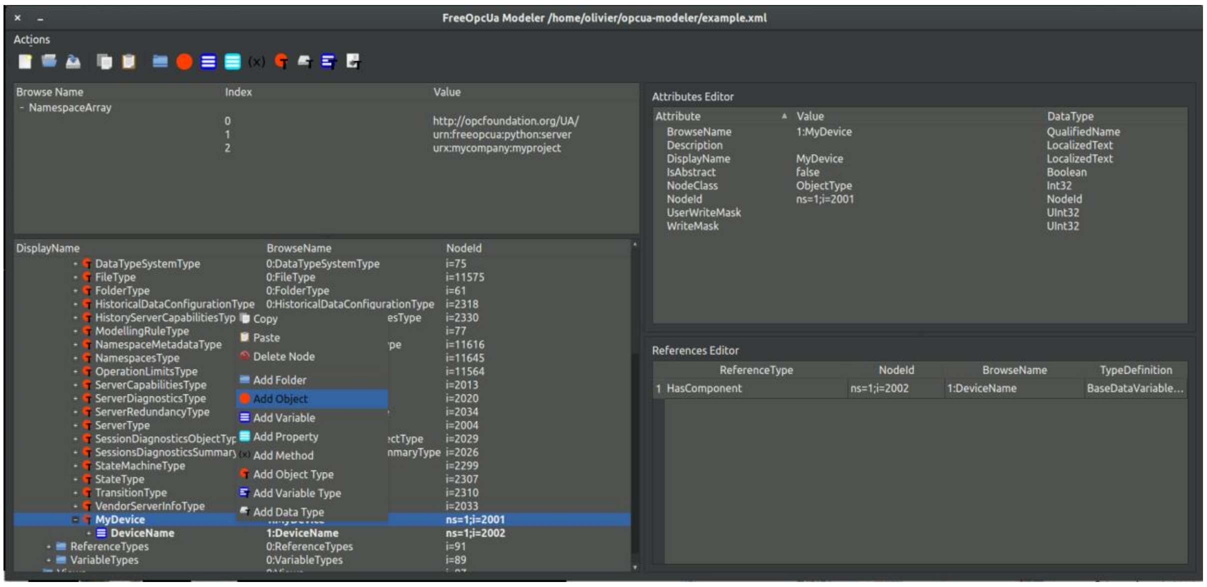
Free OPC UA Modeler [52] is a tool for the creation of a custom OPC UA address spaces using a simple GUI without the need to write Python code. It uses OPC UA specified XML format which allows the produced XML to be imported into any OPC UA SDK.

It can be easily installed and launched with the following commands (Python 3.6+ and python-pip are required):

```
1.  pip3 install opcua-modeler

2.  pip3 install --user pyqt5

3.  apt-get install python3-pyqt5

4.  apt-get install pyqt5-dev-tools
```

```
5.   opcua-modeler                    #Run the GUI
```

The modeler is basically an OPC UA Server (either python-opcua (default) or the C based open65421) that is running in background.



FreeOpcUa Modeler /home/olivier/opcua-modeler/example.xml

Browse Name	Index	Value
- NamespaceArray		
	0	http://opcfoundation.org/UA/
	1	urn:freeopcua:python:server
	2	urx:mycompany:myproject

DisplayName	BrowseName	NodeId
DataTypeSystemType	0:DataTypeSystemType	i=75
FileType	0:FileType	i=11575
FolderType	0:FolderType	i=61
HistoricalDataConfigurationType	0:HistoricalDataConfigurationType	i=2318
HistoryServerCapabilitiesTyp	esType	i=2330
ModellingRuleType		i=77
NamespaceMetadataType	pe	i=11616
NamespacesType		i=11645
OperationLimitsType		i=11564
ServerCapabilitiesType		i=2013
ServerDiagnosticsType		i=2020
ServerRedundancyType		i=2034
ServerType		i=2004
SessionDiagnosticsObjectTyp	ctType	i=2029
SessionsDiagnosticsSummary	nmaryType	i=2026
StateMachineType		i=2299
StateType		i=2307
TransitionType		i=2310
VendorServerInfoType		i=2033
MyDevice		ns=1;i=2001
DeviceName	1:DeviceName	ns=1;i=2002
ReferenceTypes	0:ReferenceTypes	i=91
VariableTypes	0:VariableTypes	i=89

Attributes Editor

Attribute	Value	DataType
BrowseName	1:MyDevice	QualifiedName
Description		LocalizedText
DisplayName	MyDevice	LocalizedText
IsAbstract	false	Boolean
NodeClass	ObjectType	Int32
NodeId	ns=1;i=2001	NodeId
UserWriteMask		UInt32
WriteMask		UInt32

References Editor

	ReferenceType	NodeId	BrowseName	TypeDefinition
1	HasComponent	ns=1;i=2002	1:DeviceName	BaseDataVariable...

Figure 7.24: FreeOpcUa Modelet GUI

This server is connected to the modeler and is customizable using action over the GUI as shown in Figure 7.24.

1. Create a new namespace (Only ONE namespace is required, namespace one will be used)
2. create a new data type under DataType / Structure
3. Populate data type with new variables using correct data type
4. Save

CHAPTER 8: ORCHESTRATION OF AN IOTAAS ENVIRONMENT

In the previous chapter I have described all the theoretical concepts that stand behind this thesis.

This chapter is focused on the description of the activity that was undertaken for the orchestration of an IIoTaaS scenario.

An initial version of this job was already presented during the 11th OSM Hackfest [53] "NF Onboarding Challenge" organized by the ETSI OSM group, and it was awarded with the "Best Use Case" award.

This work was further developed and incorporated inside another project that will be presented to be selected in an international conference. Which conference has yet to be decided. Inside this project, my work is mainly focused on the OPC UA side part.

This new implementation mixes all the theoretical concepts of NFV, OPC UA and MQTT that I have introduced and described so far.

The outcome of this study is to manage a possible IoT scenario, where a multitude of IoT sensors need to coexist, using their own communication protocol (in this case OPC UA or MQTT).

In this environment, all clients are not aware how to reach sensors in the network. The solution described in this chapter shows how is possible to register all services offered by all the servers/publisher in a sort of catalogue.

The MEC Platform (described in Chapter 4) is the software entity in charge of "helping" running applications by exposing a set of different APIs. Using MEC 011 APIs, all MEC applications can register the services they are exposing to the Service

Registry maintained by the MEC Platform. In this way other applications can discover these services and consume them by querying the registry.

An example of how the service discovery mechanism enabled by MEC 011 can be used for Industrial IoT applications can be found in [55].

In this way clients can know what are the endpoints to reach the desired service.

To do that, some software entities are required:

- Open Source MANO is the software entity that act as management and orchestration platform for all the NFV entities.

- The MEC Platform is used as “catalogue” where all services register the endpoints

- Kubernetes cluster to deploy all workloads for the MEC Platform and all the MEC Applications

- OpenStack for the management of the Kubernetes’s cluster.

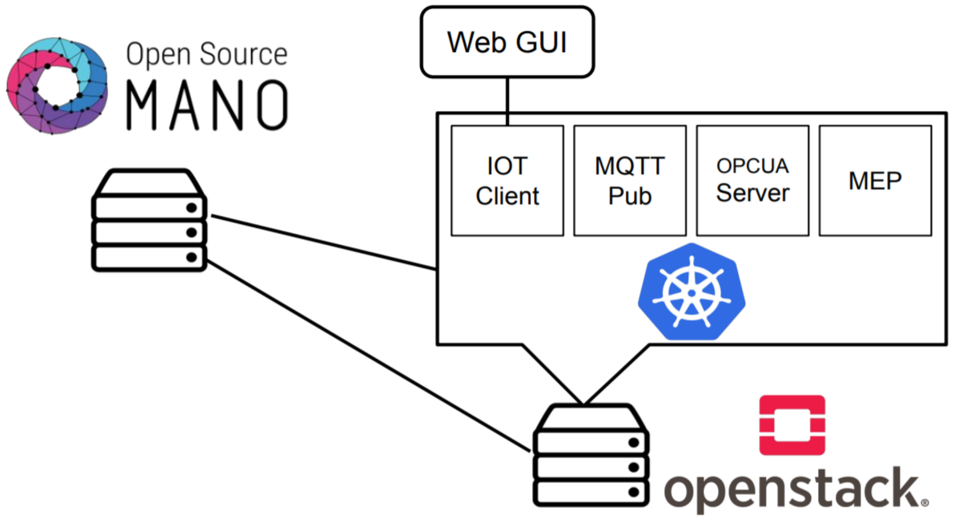


Figure 8.1: System overview

8.1 **Description of the testbed**

The testbed used for the development of the project is composed by two bare-metal servers hosted by the CloudLab [56] facilities. The first has installed OSM platform with the possibility of development all the functionalities described in Chapter 3.

The second server instead is dedicated to the handlining of all the virtualization functionalities. In particular, it hosts OpenStack Wallaby release, on top of that, three virtual machines host a Kubernetes cluster (one control node and two nodes for the workloads), each of them equipped with 8 Gb of RAM, 4 VCPUs and 80 Gb of disk.

All the pods managed by K8s are supported by some services directly offered by the platform:

- ClusterIP: Is the service used for the communication between pods (without the need to pass from an external network).

- NodePort: Is used when is required to expose a port of a container with an external network.

- LoadBalancer: Has the role to assign the IP address of a certain pod on an external network

This cluster is used to deploy all workloads for the MEC Platform and all the MEC Applications used for this project. All of these are custom made python applications packed inside Helm chart packages that will be described soon.

All the deployment is not hardware dependent. It can be replicated on whatever type of hardware. The only requirement is that hardware resources must be sufficient to handle all software and VMs that are used.

A single tenant is used for OSM and OpenStack.

The creation of a VIM account in the OSM dashboard is necessary. In particular, the VIM account is identified by the following parameters:

- Name: Identify the VIM inside the OSM platform

- VIM URL: http://10.15.3.1:5000/v3 (is the address and port to get access to the service)
- VIM Username and password
- Tenant name:
- Type: OpenStack (is the infrastructure)

In this way in OSM is possible to upload all the packages for the deployment of NSs.

OpenStack is used for the deployment of VMs. The VM from which I get access to the testbed can be access through username and password credentials and is running on Ubuntu 20.04.2 LTS.

8.2 **Description of the code**

The code is organized in three sections: Docker images, Helm Charts and OSM Descriptors.

The code related to the OPC UA part is described in this section. The full code can be found at [57].

8.2.1 *Docker images*

For each entity involved in the testbed, was necessary to build an ad-hoc Docker image. In this way was possible to run containers that are specialized in all the tasks required by the entity.

All the Dockerfile can be found in *Appendix A*. The entities involved are the following:

- *client-mec-docker* → Client that contains all the OPC UA and MQTT functionalities.
- *docker-img-mec* → Is the MEC Platform
- *mqtt-pub-docker* → MQTT Publisher
- *server-mec-docker* → Is the OPC UA Server with all the functionalities described in section 7.2.

8.2.2 *Helm Charts*

The deployment of the environment is made through OSM Descriptors and Helm Charts. They can be seen as a sort of packages that contains all command executed by K8s for the deployment of different containers.

All the Helm Charts descriptors are contained in *Appendix B*.

- *mec-iot-client*
- *mqtt-pub-docker*
- *mec-opcua_server*
- *mec-platform*

Here I report only a quick description about the Helm Chart descriptor of the OPC UA Server. All the other Charts are based on the same concepts.

mec-opcua_server

```yaml
---
{{ $randNum := randNumeric 3 }}
apiVersion: v1
kind: Pod
metadata:
  name: opcua-server-{{$randNum}}
  labels:
    app: opcua-server-{{$randNum}}
spec:
  #hostNetwork: true
  #dnsPolicy: ClusterFirstWithHostNet
  containers:
    - name: opcua-server-1
      image: lorenzobassi/opcua_server_mec-cnsm
      env:
        # MEC Platform endpoint
        - name: MEC_BASE
          value: "http://mec-platform"
        - name: INFRA
          value: "k8s"
        - name: MY_POD_NAME
          value: "opcua-server-{{$randNum}}"
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
      ports:
        - containerPort: 4840
      imagePullPolicy: Always
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: opcua-server-{{$randNum}}
spec:
  type: LoadBalancer
  selector:
    app: opcua-server-{{$randNum}}
  ports:
    - port: 4840
      targetPort: 4840
```

The role of this Helm Chart is to launch a new pod based on the Docker image *lorenzobassi/opcua_server_mec-cnsm*, which contains all the code for the deployment of the wanted service, as reported in section 8.2.1. The image is pulled from the Docker repository and extracted. This operation requires a different amount of time depending by the hardware over that the system is running.

Every time a new service is created, a new entry is created inside the DNS domain of K8s that correspond to the name of that service.

This operation is done inside the Helm Chart descriptor of all the services involved through the following code:

```
kind: Service
metadata:
  name: opcua-server-{{$randNum}}
```

8.2.3 *OSM Descriptors*

All the descriptors are reported in *Appendix C*.

As I did in the previous section I report here the description of only the OPC UA Server. All the other descriptor's entities are based on the same concepts.

VNFD Descriptor OPC UA Server

```
vnfd:
    description: KNF for deploying an OPC UA Server
    df:
```

```
    -   id: default-df
. . .
```

Most of the fields of this first part of the descriptors are reference names used chosen arbitrary. Through them is possible to reference to this descriptor inside other descriptors.

```
. . .
    ext-cpd:
    -   id: mgmt-ext
        k8s-cluster-net: mgmtnet
    id: mec-opcua-server_knf
    k8s-cluster:
        nets:
        -   id: mgmtnet
. . .
```

All the external connection points information and the network to reach K8s are defined

Because we want to deploy a K8s application is necessary to define the KDU:

```
. . .
kdu:
    -   helm-chart: chartmuseum/mec-opcua-server
        name: mec-opcua-server
    mgmt-cp: mgmt-ext
    product-name: mec-opcua-server_knf
    provider: Unibo
    version: '1.0'
```

As described in section 8.2.2, we are using Helm Charts. Inside the KDU is necessary to reference to the correspondent Helm Chart: “*chartmuseum/mec-opcua-server*”.

NSD Descriptor OPC UA Server

```
nsd:
    nsd:
    -   description: NS consisting of a single KNF mec-opcua-server_knf connected to mgmt network
        designer: OSM
        id: mec-opcua-server_ns
        name: mec-opcua-server_ns
        version: '1.0'
. . .
```

In the first part of the NSD are specified all the information about it.

```
. . .
df:
        -   id: default-df
            vnf-profile:
            -   id: mec-opcua-server
                virtual-link-connectivity:
                -   constituent-cpd-id:
                    -   constituent-base-element-id: mec-opcua-server
                        constituent-cpd-id: mgmt-ext
                    virtual-link-profile-id: mgmtnet
                vnfd-id: mec-opcua-server_knf
        virtual-link-desc:
        -   id: mgmtnet
            mgmt-network: 'true'
            vim-network-name: mgmt
        vnfd-id:
        - mec-opcua-server_knf
```

Then are defined all the information about connectivity: virtual link and connection points.

8.2.4 *OPC UA Server code*

The server used in this project has functionalities that can be overlapped to those ones described in 7.3. In this case the subscription is not implemented, simply because the aim of this project is not strictly related to the OPC UA functionalities but instead focused on the registration of this service in a multi-vendor scenario. The python code that implements the server is the following:

```
if __name__ == "__main__":
    # optional: setup logging
    logging.basicConfig(level=logging.WARN)

    # now setup our server
    server = Server()

    hostname = str(sys.argv[1])
    port = str(sys.argv[2])

    domain = "opc.tcp://"
    ddd = ":"
    final_address = (domain+hostname+ddd+port)
    server.set_endpoint(final_address)

    server.set_server_name("FreeOpcUa Example Server")

    uri = str(sys.argv[3])
    idx = server.register_namespace(uri)

    # create a new node type we can instantiate in our address space
    dev = server.nodes.base_object_type.add_object_type(idx, "MyDevice")
    dev.add_variable(idx, "sensor1", 1.0).set_modelling_rule(True)
    dev.add_property(idx, "device_id", "0340").set_modelling_rule(True)
    ctrl = dev.add_object(idx, "controller")
    ctrl.set_modelling_rule(True)
    ctrl.add_property(idx, "state", "Idle").set_modelling_rule(True)

    # create directly some objects and variables
    # READ OBJECTS AND VARIABLE FORM THE JSON FILE
    with open('opcua_fields.json') as file:
        data = json.load(file)
```

```
dim_obj_list = len(data["opcua"][0]["objects"])

for i in range(0, dim_obj_list):
  dim_var_list = len(data["opcua"][0]["objects"][i]["variables"])
  globals()[f"obj_{i}"] = data["opcua"][0]["objects"][i]["object_name"]

  globals()[f"myobj_{i}"] = server.nodes.objects.add_object(idx, globals()[f"obj_{i}"])


  for j in range(0, dim_var_list):
        globals()[f"variable_{i}_{j}"] = data["opcua"][0]["objects"][i]["variables"][j]
        globals()[f"myvar_{i}_{j}"].set_writable()

server.start()
print("Available loggers are: ", logging.Logger.manager.loggerDict.keys())
vup = VarUpdater(mysin)  # just  a stupide class update a variable
vup.start()
try:

    if sys.argv[4] == 'default_value':
        sleep = 1
    else:
        sleep = int(sys.argv[4])
    print("sleep in input: ",sleep)

    while 15 == 15:
     for k in range(0, dim_obj_list):
      num_variables = len(data["opcua"][0]["objects"][k]["variables"])

      for p in range(0, num_variables):
        temp = random.randint(1,10000)
        server.set_attribute_value(globals()[f"myvar_{k}_{p}"].nodeid, ua.DataValue(temp))
        print(" ")
        print("Nuovo valore myvar_",k,"_",p,":", temp)
        print("------------------------------------------------")
        time.sleep(sleep)

    embed()
finally:
    vup.stop()
    server.stop()
```

The code requires 4 parameters (defined arbitrary by the user) to function properly:

- The *ip addres* where the server is executed

```
hostname = str(sys.argv[1])
```

- The *port* where the service is exposed

```
port = str(sys.argv[2])
```

- The *uri* to be set in the namespace containing the variables of the server.

```
uri = str(sys.argv[3])
```

- The *periodicity* between a publication and the following ones

```
sleep = int(sys.argv[4])
```

To guarantee the vendor independent property of the protocol, the user have to define its own information model. This can be done modifying a json file (Reported in Table 8) inside a GitHub repository [58] before launching the server.

```
{
          "opcua": [
            {
              "port": "4840",
              "uri": "http://examples.freeopcua.github.io",
              "objects": [
                {
                  "object_name": "Device1",
                  "variables": [
                    "Light",
                    "Water",
                    "Gas"
                  ]
                },
                {
                "object_name": "Device2",
                "variables": [
                    "Temperature"
                  ]
                }
              ]
            }
          ]
        }
```

Table 8: Json list for the definition of the namespace of the server

Every time the OPC UA server is launched the json list of Table 8 is downloaded from the GitHub repository. In this way the server can get access to it:

```
with open('opcua_fields.json') as file:
        data = json.load(file)
```

Once the list is open, using a *for* cycle that spans through all the objects and variables is possible to store all of them inside the namespace of the server.

```
dim_obj_list = len(data["opcua"][0]["objects"])

    for i in range(0, dim_obj_list):
      dim_var_list = len(data["opcua"][0]["objects"][i]["variables"])
      globals()[f"obj_{i}"] = data["opcua"][0]["objects"][i]["object_name"]

      globals()[f"myobj_{i}"] = server.nodes.objects.add_object(idx, globals()[f"obj_{i}"])


      for j in range(0, dim_var_list):
            globals()[f"variable_{i}_{j}"] = data["opcua"][0]["objects"][i]["variables"][j]
            globals()[f"myvar_{i}_{j}"].set_writable()
```

At this point the server is ready to be started:

```
server.start()
```

A random number is assigned to all the variables. This happened with a periodicity defined by the user through the ***sleep*** parameter.

```
while 15 == 15:
         for k in range(0, dim_obj_list):
          num_variables = len(data["opcua"][0]["objects"][k]["variables"])

          for p in range(0, num_variables):
           temp = random.randint(1,10000)
            server.set_attribute_value(globals()[f"myvar_{k}_{p}"].nodeid, ua.DataValue(temp))
            print(" ")
            print("Nuovo valore myvar_",k,"_",p,":", temp)
            print("------------------------------------------------")
            time.sleep(sleep)
```

8.2.5 *OPC UA Client code*

The client of the project is a single entity which include both MQTT and OPC UA functionalities. I'll explain more in detail in section 8.4.

Here I report the code related to the OPC UA part:

```
if __name__ == "__main__":
    logging.basicConfig(level=logging.WARN)

    hostname = str(sys.argv[1])
    port = str(sys.argv[2])

    domain = "opc.tcp://"
    ddd = ":"
    final_address = (domain+hostname+ddd+port)

    client = Client(final_address)
    try:
        client.connect()
        client.load_type_definitions()

        root = client.get_root_node()

        objects = client.get_objects_node()

        # getting our namespace idx
        uri = str(sys.argv[3])
        idx = server.register_namespace(uri)


        myvar00 = root.get_child(["0:Objects", "{}:Device1".format(idx), "{}:Light".format(idx)])
        obj0 = root.get_child(["0:Objects", "{}:Device1".format(idx)])
        sensor_msg00 = myvar00.get_value()
        print("Device1_Light : ", sensor_msg00)

        myvar01 = root.get_child(["0:Objects", "{}:Device1".format(idx), "{}:Water".format(idx)])
        sensor_msg01 = myvar01.get_value()
        print("Device1_Water : ",sensor_msg01)

        myvar02 = root.get_child(["0:Objects", "{}:Device1".format(idx), "{}:Gas".format(idx)])
        sensor_msg02 = myvar02.get_value()
        print("Device1_Gas : ",sensor_msg02)
        myvar10 = root.get_child(["0:Objects", "{}:Device2".format(idx), "{}:Temperature".format(idx)])
        obj1 = root.get_child(["0:Objects", "{}:Device2".format(idx)])
        sensor_msg10 = myvar10.get_value()
        print("Device2_Temperature : ",sensor_msg10)

        embed()
    finally:
        client.disconnect()
```

As before, 3 parameters are to function properly:

- The *ip addres* where the server is executed

```
hostname = str(sys.argv[1])
```

- The *port* where the service is exposed

```
port = str(sys.argv[2])
```

- The *uri* to identity the namespace of the server.

```
uri = str(sys.argv[3])
```

These parameters need to match those defined by the user during the server start-up.

After that the connection is established, all the variables can be saved in a variable and printed out.

```
myvar00 = root.get_child(["0:Objects", "{}:Device1".format(idx), "{}:Light".format(idx)])
obj0 = root.get_child(["0:Objects", "{}:Device1".format(idx)])
sensor_msg00 = myvar00.get_value()
print("Device1_Light : ", sensor_msg00)
```

To get access to a variable, are required the object name and the name of the variable defined inside the Json list of Table 8.

8.3 **Registration of services inside the MEC Platform**

As mentioned before, in this implementation can coexist at the same time entities that are using OPC UA or MQTT as transport protocol. In this case all the OPC UA Servers and MQTT publishers that are part of the network need to register their service endpoints inside the MEC platform.

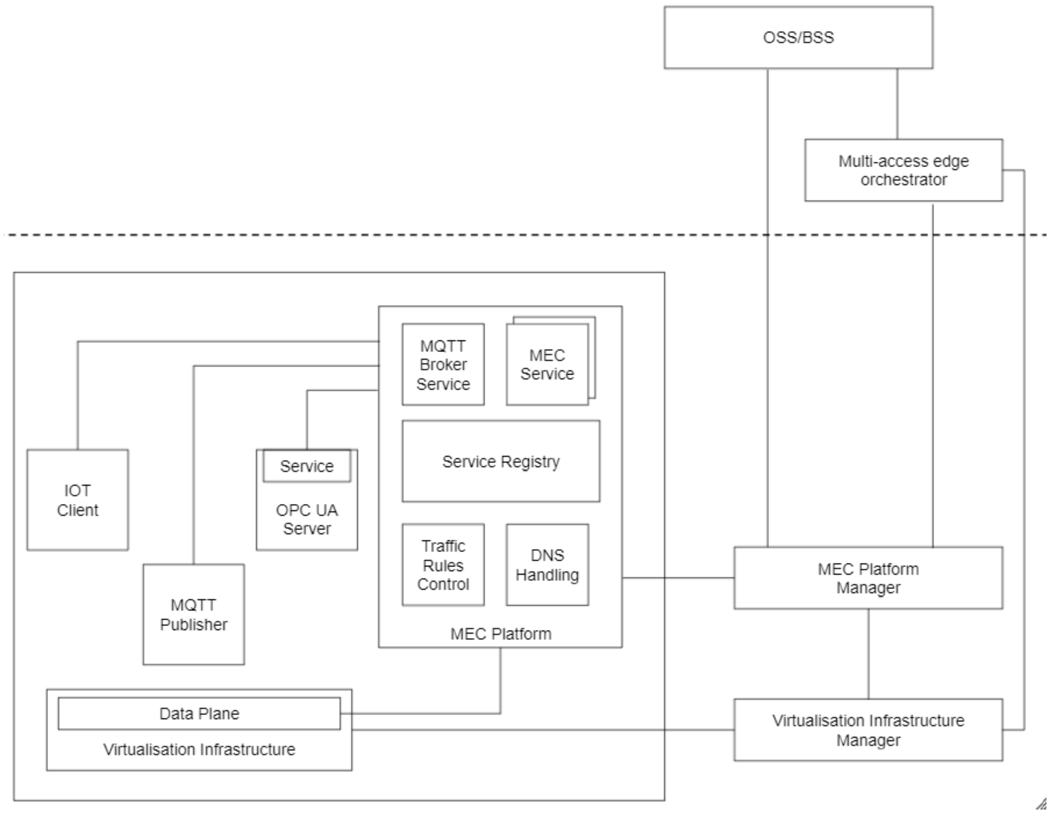


Figure 8.2: MEC Platform

As shown in Figure 8.2 the MEC Platform act as Broker MQTT for what concerns MQTT case. For the OPC UA protocol, instead, the platform act as Discovery server (located inside the Service Registry in Figure 8.2).

In this way, is possible to benefit of the standardized ETSI MEC framework functionalities. All the functionalities of the Discovery Server and the Broker MQTT are covered by the platform without the need to implement these two software entities.

By properly defining all the services following MEC 011 [59] standard to further describe what a specific IoT device is providing, an IoT client can discover the different available services of all the sensors in the network. In this way the client can know the type of data they are offering, and which type of transport protocol they are using.

MEC 011 standard defines the type “*EndPointInfo*” for the definition of transport endpoint. In particular, the flag “*alternative*” allow the definition a custom definition of it.

As result, this solution allows to coexist at the same time multiple services that are implemented with different technologies.

The IOT client in Figure 8.2 contains inside all the software to be able to interact with IoT sensors that are using both OPC UA and MQTT transport protocols.

8.3.1 *Registration of OPC UA Server service on the MEC Platform*

As already explained in Chapter 6, the OPC UA Server needs to specify some parameters and endpoints for the setup of the service.

As explained in section 8.2.4, in this project, the OPC UA Server acts as a sensor that outputs random numbers.

To register this kind of service some endpoints and parameters that are required:

- *Ip address* of the Server
- *Port number* where the service can be assessed
- *URI* of the namespace inside the server
- *Object name*
- *Variable name*

The IP address of the server is fixed and depends by the VM that is running the service.

Their registration on the MEC Platform is made through a JSON file (see Table 8). All these endpoints of the service are placed inside the field “*alternative*” of the MEC standard.

Here down below, is reported the entire standardized MEC 011 format for the OPC UA server:

```
{
        "serInstanceId": "OPC_UA_SERVER",
        "serName": "OPC_UA_SERVER-service",
        "serCategory": {
            "href": "/example/catalogue1",
            "id": "id12345",
            "name": "RNI",
            "version": "version1"
        },
        "version": "ServiceVersion1",
        "state": "ACTIVE",
        "transportInfo": {
            "id": "OPC_UA_SERVER---1",
            "name": "OPC_UA_SERVER",
            "description": "OPC_UA_SERVER",
            "type": "MB_TOPIC_BASED",
            "protocol": "OPCUA",
            "version": "2.0",
            "endpoint": {
                "alternative": {
                    "opcua": [
                        {
                            "port": "4840",
                            "uri": "http://examples.freeopcua.github.io",
                            "objects": [
                                {
                                    "object_name": "Device1",
                                    "variables": [
                                        "Light",
                                        "Water",
                                        "Gas"
                                    ]
                                },
                                {
                                    "object_name": "Device2",
                                    "variables": [
                                        "Temperature"
                                    ]
                                }
                            ],
                            "host": "opcua-server-865.6c391cef-2235-4589-8be7-0764e185f9e7.mec.host"
```

Table 9: OPC UA Server registration on the MEC Platform

Because of OPC UA is a vendor-independent platform. As explained in section 8.24, all these parameters are defined by the user during the Server start-up (using the JSON list of Table 8). The JSON list is downloaded from a project on GitHub and automatically registered on the MEC Platform. In this way is possible to easily create a namespace populated with all the fields defined by the user.

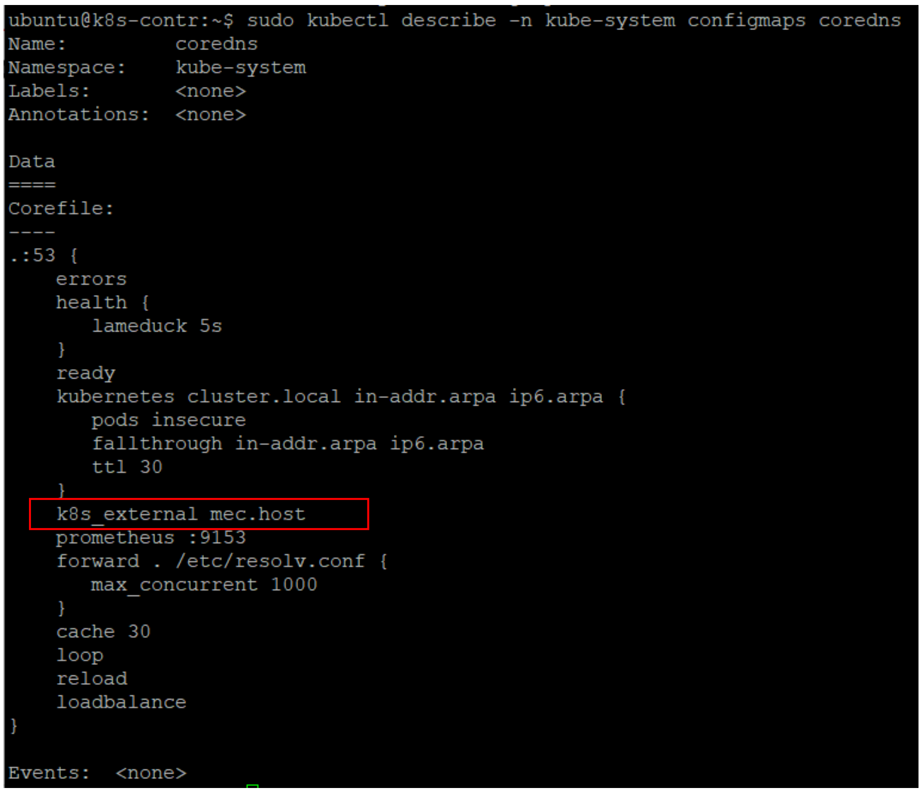
From Table 9 it is possible to see that the IP address of the OPC UA Server is not included in the list. That information is necessary to correctly reach the service but is not obtained directly from the JSON list of the service but instead using one of the K8s functionalities when a new pod is launched.

As reported in section 8.2.2, for each service that is created, is associated to a DNS name that can be resolved internally by K8s.

In this case is also created a subdomain DNS:

```
hostname = "{}.{}.mec.host".format(pod_name, pod_namespace)
```

This DNS subdomain is created using the service CoreDNS that is already included in the K8s software.



```
ubuntu@k8s-contr:~$ sudo kubectl describe -n kube-system configmaps coredns
Name:         coredns
Namespace:    kube-system
Labels:       <none>
Annotations:  <none>

Data
====
Corefile:
----
.:53 {
    errors
    health {
       lameduck 5s
    }
    ready
    kubernetes cluster.local in-addr.arpa ip6.arpa {
       pods insecure
       fallthrough in-addr.arpa ip6.arpa
       ttl 30
    }
    k8s_external mec.host
    prometheus :9153
    forward . /etc/resolv.conf {
       max_concurrent 1000
    }
    cache 30
    loop
    reload
    loadbalance
}

Events:  <none>
```

Figure 8.3: CoreDNS configuration

As shown highlighted with a red line in Figure 8.3, all the services which terminate with *mec.host* are resolved with the external addresses of K8s (*k8s_externals*).

In this way is possible to associate a DNS name to every service created (the OPC UA Server in this case). As example, a possible DNS name for the OPC UA Server could be: "*opcua-server-123.pod_name.pod_namespcae.mec.host*"

After having defined all these parameters, the registration of the service inside the MEC Platform is made through a *request.post*:

```
r = requests.post(query_base, data=json.dumps(service_data), headers=headers)
```

Another important fact that I want to point out, is the type of transport used for the OPC UA case. In the filed "*type*" is reported MB_TOPIC_BASED.

The definition of this field can be found at [59]: "*Topic-based message bus which routes messages to receivers based on subscriptions, if a pattern passed on subscription matches the topic of the message*."

This description matches perfectly the MQTT protocol. Even if is not totally true in the OPC UA case, is the one that best approximate the type of transport.

The list of transport type can be extensible. A future development of this study could be the definition of a new transport standard type that matches perfectly the OPC UA protocol.

8.3.2 *Registration of MQTT Broker service on the MEC Platform*

Like the case of the OPC UA Server, the MQTT broker register it service using the following JSON list that contains all the endpoints to reach that service.

```
"serInstanceId": "Mec-MQTT-Broker-1",
    "serName": "Mec-MQTT-Broker-Service",
    "serCategory": {
    "href": "/example/catalogue1",
    "id": "id12345",
    "name": "RNI",
    "version": "version1"
    },
    "version": "ServiceVersion1",
    "state": "ACTIVE",
    "transportInfo": {
    "id": "MqttBrokerId01",
    "name": "MQTT BROKER",
    "description": "MQTT BROKER",
    "type": "MB_TOPIC_BASED",
    "protocol": "MQTT",
    "version": "2.0",
    "endpoint": {
        "alternative":{
            "mqtt-topics":{
                "host" : "mqtt-broker",
                "port" : "1883",
                "topics" : []
            }
        }
    },
    "security": {
        "oAuth2Info": {
        "grantTypes": [
            "OAUTH2_CLIENT_CREDENTIALS"
        ],
        "tokenEndpoint": "/mecSerMgmtApi/security/TokenEndPoint"
        }
    },
    "implSpecificInfo": {}
    },
    "serializer": "JSON",
    "scopeOfLocality": "MEC_SYSTEM",
    "consumedLocalOnly": False,
    "isLocal": True
}
```
Table 10: MQTT Broker Registration on the MEC Platform

8.4 **The Client**

This software entity provides a frontend and a backend part. I mainly worked on the OPC UA backend implementation.

There are not two different clients for the MQTT and the OPC UA protocols. There is a hybrid client that contains which can handle the two transport protocols. Its role is to search inside the list of services that are registered the MEC platform the name of the sensors that it is interested in. Once found it, can recover all the endpoints stored inside the "*alternative*" filed and use them to get access to that sensor no matter which protocol is required, because the client is capable of both.

At the end of the day, the final user does not know anything about the sensor that it is interested in. The only information that the user must be aware of, is the name of that sensor.

The MEC Platform respond sending the endpoints to reach it. This means that the client does not know in advance if the sensor it is looking for is running OPC UA or MQTT.

In Table 11, is reported the part of the code where the client understands which protocol to use to get the data from sensors. This is done through the function "*get_sensor_data*" which requires in input the name of the wanted variable "*sensorName*". The function search inside the list of services registered if that variable is associated to the OPCUA or MQTT protocol. This can be done through the filed "*protocol*" of the "*transportInfo*" section as shown in Tables 9 and 10.

```
@app.route("/get_sensor_data/<sensorName>")
def get_sensor_data(sensorName):
    srvs = get_all_services()
    for s in srvs:
        if s['transportInfo']['protocol'] == 'MQTT':
            if sensorName in s['transportInfo']["endpoint"]['alternative']['mqtt-topics']['topics']:
                #TODO START MQTT SENSING
                address = s['transportInfo']["endpoint"]['alternative']['mqtt-topics']['host']
                port = s['transportInfo']["endpoint"]['alternative']['mqtt-topics']['port']
                topic = sensorName
                mqtt_get_data(address, port, topic)
                print
        if s['transportInfo']['protocol'] == 'OPCUA':
            for opcua_server in s['transportInfo']["endpoint"]['alternative']['opcua']:
                for object in opcua_server['objects']:
                    if object['object_name'] == sensorName.split("/")[0] and sensorName.split("/")[1] in object['variables']:
                        #TODO START OPCUA SENSING
                        opcua_get_data(opcua_server['host'], opcua_server['port'], opcua_server['uri'], sensorName.split("/")[0], sensorName.split("/")[1])
                        print
    return
```
Table 11: Client code for the selection of the protocol to use

In Table 12 are reported the differences between the registration of the two services.

OPC UA	MQTT
`"endpoint": {`	`"endpoint": {`
`    "alternative": {`	`        "alternative":{`
`        "opcua": [`	`            "mqtt-topics":{`
`            {`	`                "host" : "mqtt-broker",`
`                "port": "4840",`	`                "port" : "1883",`
`                "uri": "http://examples.freeopcua.github.io",`	`                "topics" : []`
`                "objects": [`	`            }`
`                    {`	`        }`
`                        "object_name": "Device1",`	`    },`
`                        "variables": [`	
`                            "Light",`	
`                            "Water",`	
`                            "Gas"`	
`                        ]`	
`                    },`	
`                    {`	
`                        "object_name": "Device2",`	
`                        "variables": [`	
`                            "Temperature"`	
`                        ]`	
`                    }`	
`                ],`	

Table 12: Definition of sensors in the MEC Platform OPC UA vs MQTT

Once the client has understood which protocol to use, calls the correspondent function:

- "***mqtt_get_data(****address, port, topic****)***" → For the MQTT case.
- "***opcua_get_data(****host,port,uri,objectName,sensorName****)***" → For OPC UA case.

All the input arguments of these two functions are those parameters registered on the MEC Platform, related to that "*sensorName*".

8.5 **Deployment of the IIoTaaS environment**

For the management of the Hem Charts is used *ChartMuseum* [60]. It is an open-source *Helm Chart repository* server written in Go (Golang), with support for cloud storage backends. The local repository that can be accessed through:

```
root@k8s-contr:~# curl http://localhost:18082/index.yaml
apiVersion: v1
entries:
 mec-iot-client:
 - apiVersion: v1
   appVersion: 1.16.0
   created: "2021-07-29T14:38:48.556006654Z"
   description: A Helm chart for Kubernetes
   digest: b70670647fae708621b12be356034a81dbb1e2b93e942e34d64fb2dad09a6875
   name: mec-iot-client
   type: application
   urls:
   - charts/mec-iot-client-0.1.0.tgz
   version: 0.1.0
 mec-opcua-server:
 - apiVersion: v1
   appVersion: 1.16.0
   created: "2021-07-29T14:39:09.20855309Z"
   description: A Helm chart for Kubernetes
   digest: 3cfc440b8d5c5a27ea94c2ae3eb1041878cc16d371cdf3948be570a00078c9eb
   name: mec-opcua-server
   type: application
   urls:
   - charts/mec-opcua-server-0.1.0.tgz
   version: 0.1.0
 mec-platform:
 - apiVersion: v1
   appVersion: 1.16.0
   created: "2021-07-29T14:39:20.37284848Z"
   description: A Helm chart for Kubernetes
   digest: 1b02ca98ec700e7c3420ff7247092a620005534e8546b04baae034a2ddc0720d
   name: mec-platform
   type: application
   urls:
   - charts/mec-platform-0.1.0.tgz
   version: 0.1.0
 mqtt-pub-docker:
 - apiVersion: v1
   appVersion: 1.16.0
   created: "2021-07-29T14:39:33.369192344Z"
   description: A Helm chart for Kubernetes
   digest: 3e984a60737ef7a8b689d0ea505e656357089b5267d96bb03c3ed54f5ea6f58f
   name: mqtt-pub-docker
   type: application
   urls:
   - charts/mqtt-pub-docker-0.1.0.tgz
   version: 0.1.0
generated: "2021-07-29T14:46:37Z"
serverInfo: {}
```

The output of the command reports that the 4 charts (highlighted in yellow) described in section 8.2.2 are uploaded.

After that, all the NS and KNF packages can be uploaded on OSM Platform (see Appendix C).

Then the 4 NSs can be launched through the commands:

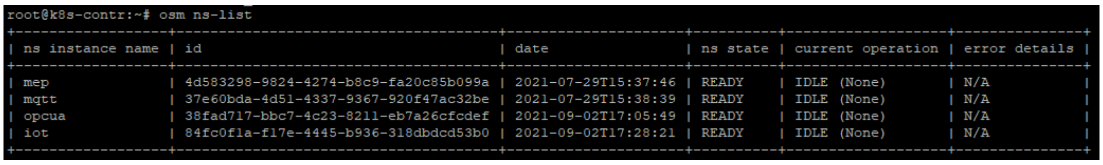
```
osm ns-create --wait --ns_name mep --nsd_name mec-platform_ns --vim_account openstack3
```

```
osm ns-create --wait --ns_name mqtt --nsd_name mec-mqtt-pub_ns --vim_account openstack3
```

```
osm ns-create --wait --ns_name opcua --nsd_name mec-opcua-server_ns --vim_account openstack3
```

```
osm ns-create --wait --ns_name iot --nsd_name mec-iot-client_ns --vim_account openstack3
```

To see that all the services are correctly launched:



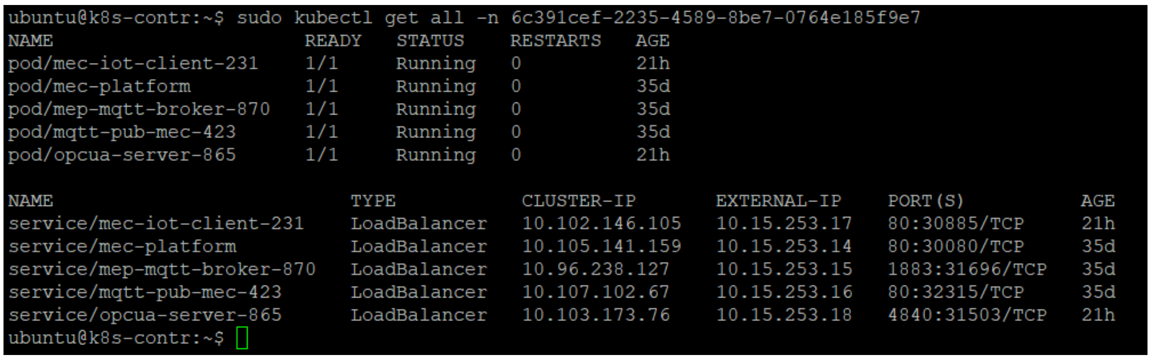
```
root@k8s-contr:~# osm ns-list
+------------------+--------------------------------------+---------------------+----------+-------------------+---------------+
| ns instance name | id                                   | date                | ns state | current operation | error details |
+------------------+--------------------------------------+---------------------+----------+-------------------+---------------+
| mep              | 4d583298-9824-4274-b8c9-fa20c85b099a | 2021-07-29T15:37:46 | READY    | IDLE (None)       | N/A           |
| mqtt             | 37e60bda-4d51-4337-9367-920f47ac32be | 2021-07-29T15:38:39 | READY    | IDLE (None)       | N/A           |
| opcua            | 38fad717-bbc7-4c23-8211-eb7a26cfcdef | 2021-09-02T17:05:49 | READY    | IDLE (None)       | N/A           |
| iot              | 84fc0f1a-f17e-4445-b936-318dbdcd53b0 | 2021-09-02T17:28:21 | READY    | IDLE (None)       | N/A           |
+------------------+--------------------------------------+---------------------+----------+-------------------+---------------+
```

Figure 8.4: OSM NS launched

In this way all the entities are deployed as K8s applications.

To see that everything is correctly up and running, is necessary to see all the pods running in the K8s namespace.

```
kubectl get all -n 6c391cef-2235-4589-8be7-0764e185f9e7
```



```
ubuntu@k8s-contr:~$ sudo kubectl get all -n 6c391cef-2235-4589-8be7-0764e185f9e7
NAME                         READY   STATUS    RESTARTS   AGE
pod/mec-iot-client-231       1/1     Running   0          21h
pod/mec-platform             1/1     Running   0          35d
pod/mep-mqtt-broker-870      1/1     Running   0          35d
pod/mqtt-pub-mec-423         1/1     Running   0          35d
pod/opcua-server-865         1/1     Running   0          21h

NAME                             TYPE           CLUSTER-IP       EXTERNAL-IP    PORT(S)          AGE
service/mec-iot-client-231       LoadBalancer   10.102.146.105   10.15.253.17   80:30885/TCP     21h
service/mec-platform             LoadBalancer   10.105.141.159   10.15.253.14   80:30080/TCP     35d
service/mep-mqtt-broker-870      LoadBalancer   10.96.238.127    10.15.253.15   1883:31696/TCP   35d
service/mqtt-pub-mec-423         LoadBalancer   10.107.102.67    10.15.253.16   80:32315/TCP     35d
service/opcua-server-865         LoadBalancer   10.103.173.76    10.15.253.18   4840:31503/TCP   21h
ubuntu@k8s-contr:~$
```

Figure 8.5: PODs inside the namespace

It is possible to get access to the MEC Platform service, by using the external ip address of the MEC platform (10.15.253.14 as shown in Figure 8.5).

To see all the services that are registered in the *service registry* of the MEC Framework by the OPC UA Server and the Broker MQTT:

```
curl 10.15.253.14/mec_service_mgmt/v1/services | python3 -m json.tool
```

```
ubuntu@k8s-contr:~$ curl 10.15.253.14/mec_service_mgmt/v1/services | python3 -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  2449  100  2449    0     0   265k      0 --:--:-- --:--:-- --:--:--  265k
[
    {
        "serInstanceId": "ServiceInstance123",
        "serName": "ExampleService",
        "serCategory": {
            "href": "/example/catalogue1",
            "id": "id12345",
            "name": "RNI",
            "version": "version1"
        },
        "version": "ServiceVersion1",
        "state": "ACTIVE",
        "transportInfo": {
            "id": "TransId12345",
            "name": "REST",
            "description": "REST API",
            "type": "REST_HTTP",
            "protocol": "HTTP",
            "version": "2.0",
            "endpoint": {},
            "security": {
                "oAuth2Info": {
                    "grantTypes": [
                        "OAUTH2_CLIENT_CREDENTIALS"
                    ],
                    "tokenEndpoint": "/mecSerMgmtApi/security/TokenEndPoint"
                }
            },
            "implSpecificInfo": {}
        },
        "serializer": "JSON",
        "scopeOfLocality": "MEC_SYSTEM",
        "consumedLocalOnly": false,
        "isLocal": true
    },
    {
        "serInstanceId": "Mec-MQTT-Broker-1",
        "serName": "Mec-MQTT-Broker-Service",
        "serCategory": {
            "href": "/example/catalogue1",
            "id": "id12345",
            "name": "RNI",
            "version": "version1"
        },
        "version": "ServiceVersion1",
        "state": "ACTIVE",
        "transportInfo": {
            "id": "MqttBrokerId01",
            "name": "MQTT BROKER",
            "description": "MQTT BROKER",
            "type": "MB_TOPIC_BASED",
            "protocol": "MQTT",
            "version": "2.0",
            "endpoint": {
                "alternative": {
                    "mqtt-topics": {
                        "host": "mep-mqtt-broker-870.6c391cef-2235-4589-8be7-0764e185f9e7.mec.host",
                        "port": "1883",
                        "topics": [
                            "dev423/temperature",
                            "dev423/air_quality"
                        ]
                    }
                }
            },
            "security": {
                "oAuth2Info": {
                    "grantTypes": [
                        "OAUTH2_CLIENT_CREDENTIALS"
                    ],
                    "tokenEndpoint": "/mecSerMgmtApi/security/TokenEndPoint"
                }
            },
            "implSpecificInfo": {}
        },
        "serializer": "JSON",
        "scopeOfLocality": "MEC_SYSTEM",
        "consumedLocalOnly": false,
        "isLocal": true
    },
    {
        "serInstanceId": "OPC_UA_SERVER",
        "serName": "OPC_UA_SERVER-service",
        "serCategory": {
            "href": "/example/catalogue1",
            "id": "id12345",
            "name": "RNI",
            "version": "version1"
        },
        "version": "ServiceVersion1",
        "state": "ACTIVE",
        "transportInfo": {
            "id": "OPC_UA_SERVER---1",
            "name": "OPC_UA_SERVER",
            "description": "OPC_UA_SERVER",
            "type": "MB_TOPIC_BASED",
            "protocol": "OPCUA",
            "version": "2.0",
            "endpoint": {
                "alternative": {
                    "opcua": [
                        {
                            "port": "4840",
                            "uri": "http://examples.freeopcua.github.io",
                            "objects": [
                                {
                                    "object_name": "Device1",
                                    "variables": [
                                        "Light",
                                        "Water",
                                        "Gas"
                                    ]
                                },
                                {
                                    "object_name": "Device2",
                                    "variables": [
                                        "Temperature"
                                    ]
                                }
                            ],
                            "host": "opcua-server-865.6c391cef-2235-4589-8be7-0764e185f9e7.mec.host"
                        }
                    ]
                },
                "security": {
                    "oAuth2Info": {
                        "grantTypes": [
                            "OAUTH2_CLIENT_CREDENTIALS"
                        ],
                        "tokenEndpoint": "/mecSerMgmtApi/security/TokenEndPoint"
                    }
                },
                "implSpecificInfo": {}
            },
            "serializer": "JSON",
            "scopeOfLocality": "NFVI_NODE",
            "consumedLocalOnly": false,
            "isLocal": true
        }
    }
]
```

Table 13: Services registered inside the MEC Platform

From Table 13 is possible to see that both the MQTT and OPC UA services are correctly registered following exactly the JSON format definition of the two services as reported in Table 12.

Now all the services are correctly registered over the MEC Platform following MEC 011 standard. The client can get access to it to understand what are the services registered and to the get the endpoints to reach them.

The front-end part of this implementation provides a simple GUI (reported in Figure 8.6) composed by several buttons which shows on screen all the variables that are registered on the MEC Platform. The user, just by selecting one of these buttons, can see their corresponding value on that moment. This means that the user is totally unaware of which protocols is used to see that value.

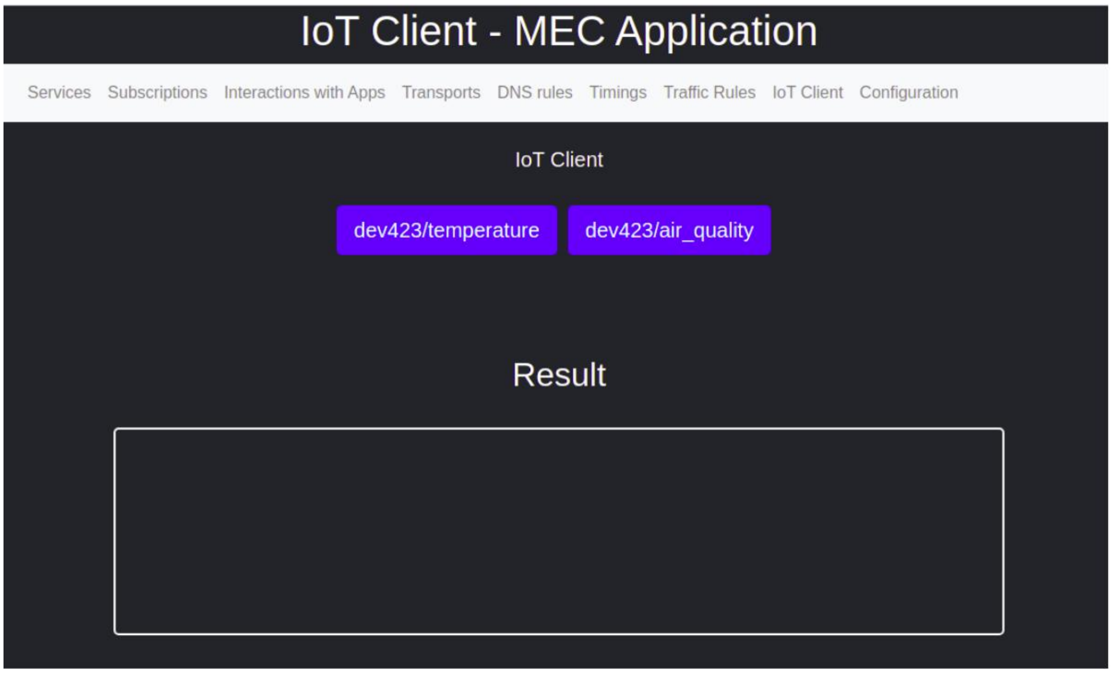


Figure 8.6: GUI IoT Client

8.6 **Measurements**

For the evaluation of the performance of the solution proposed in this thesis, were evaluated the deployment time of all the entities involved:

- *client-mec-docker*
- *docker-img-mec*
- *mqtt-pub-docker*
- *server-mec-docker*

During the deployment of the service, all the Helm Charts described in section 8.2.2 contain all the steps to be followed. As already explained, all the software entities require the correspondent Docker images that need to be pulled from the Docker Hub during the launch of the KNFs. This means that during the time required for the setup of the service is included the time required to download and extract the file of the image.

These time intervals depend by several factors:

- The image size, which varies from an image to another:
    - IoT Client → 206 Mb
    - MEC Platform → 21 Mb
    - MQTT Publisher → 23 Mb
    - OPC UA Server→ 448Mb
- The download bandwidth available:

```
root@k8s-contr: # speedtest
Retrieving speedtest.net configuration...
Testing from University of Utah (128.110.218.53)...
Retrieving speedtest.net server list...
Selecting best server based on ping...
Hosted by XMission (Salt Lake City, UT) [1254.30 km]: 3.941 ms
Testing download speed................................................................................
Download: 1603.65 Mbit/s
Testing upload speed......................................................................................................
Upload: 1357.73 Mbit/s
```

- Bandwidth offered by the Docker server where the image is pulled.

- Hardware capabilities: depending by the number of virtual cores available and the memory technology used (HDD, SSD, M2 MVME) there can be big differences.

  For what concerns the resources at disposal for each pod deployed at the Cloud Lab, hardware characteristics are reported below:

```
root@opcua-server-984:/# lscpu
Architecture:       x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:         Little Endian
Address sizes:       40 bits physical, 48 bits virtual
CPU(s):             4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  1
Socket(s):          4
NUMA node(s):        1
Vendor ID:          GenuineIntel
CPU family:          6
Model:              61
Model name:          Intel Core Processor (Broadwell, IBRS)
Stepping:           2
CPU MHz:             2394.454
BogoMIPS:            4788.90
Virtualization:      VT-x
Hypervisor vendor:   KVM
Virtualization type: full
L1d cache:          32K
L1i cache:          32K
L2 cache:           4096K
L3 cache:           16384K
```

To test the system, the following procedure was followed: all software entities are deployed consecutively 10 times, with the only exception of the MEC Platform which is deployed only once. This because the MEC Platform is a single entity that cannot be deployed more than one time. As results, 31 pods were running simultaneously: 10 OPC UA Servers, 10 MQTT Publishers, 10 IoT Clients, 1 MEC Platform.

The deployment of all the services must follow some steps:

1) Creation of the NSs
2) Creation of the pod
3) Pull of the Docker image
4) Creation and start the container

Deployment times are reported from Figure 8.7 to 8.10.

- Blue bars refer to the times required for the creation of the NSs. All the OSM packages were previously created and already uploaded on the OMS platform.

- Red bars refer to the time required for the pull of the Docker images. It is possible to see that that time increases according to the image size, but this is true only at the first pull of each service. Once the docker image is pulled, does not need to be pulled again, and all the successive deployment can benefit. That's why the pulls of the image after the first one, take less than 1s.

- Green bars represent the time required for the creation of the container and the launch of the service.

It is important to notice that even if the time required for the pull of the Docker image changes according to the image size (always related to the 1st deployment of each service), the total amount of time for the creation of the NS remains constant in all the 4 cases (between 21 and 23 seconds). This is probably due to a software threshold imposed by the OSM software.

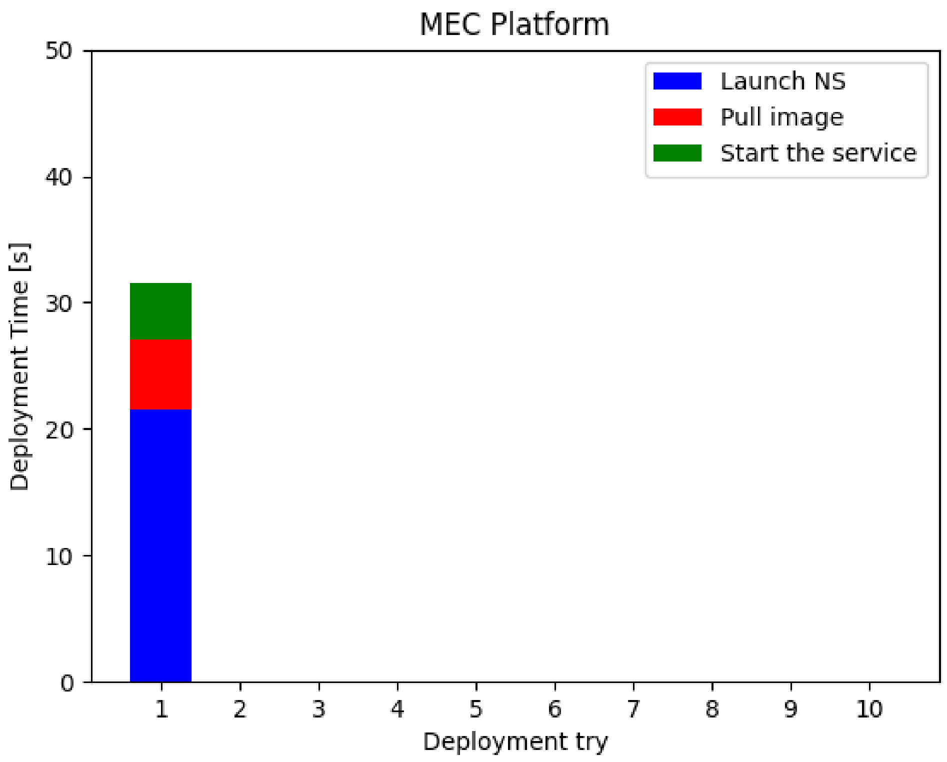


Figure 8.7: MEC Platform deployment time

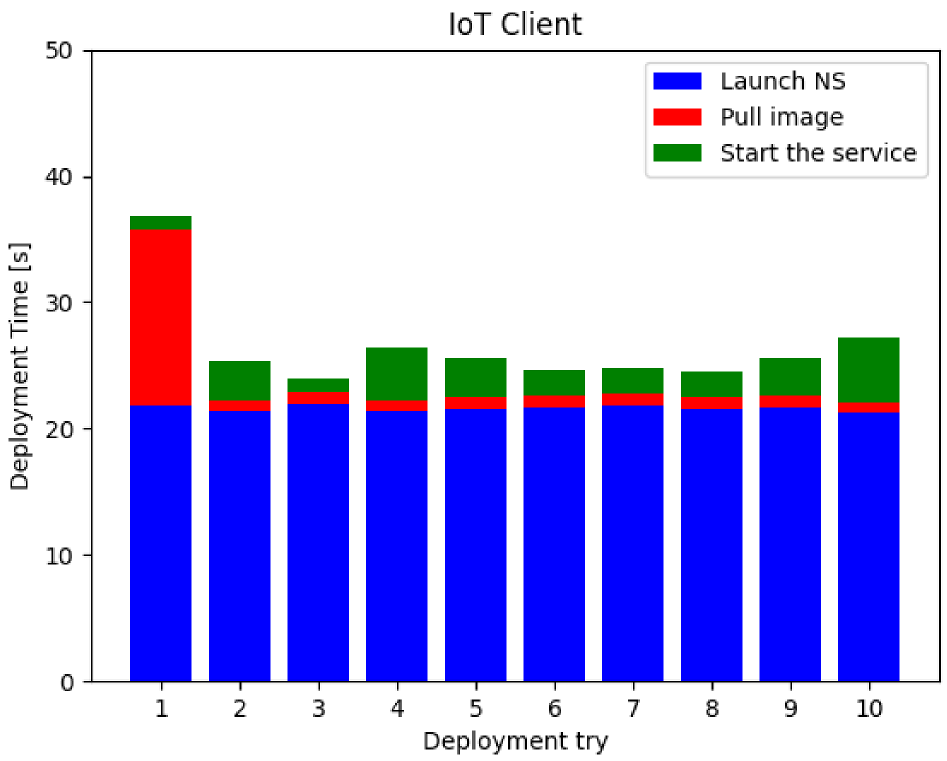


Figure 8.8: Client deployment time

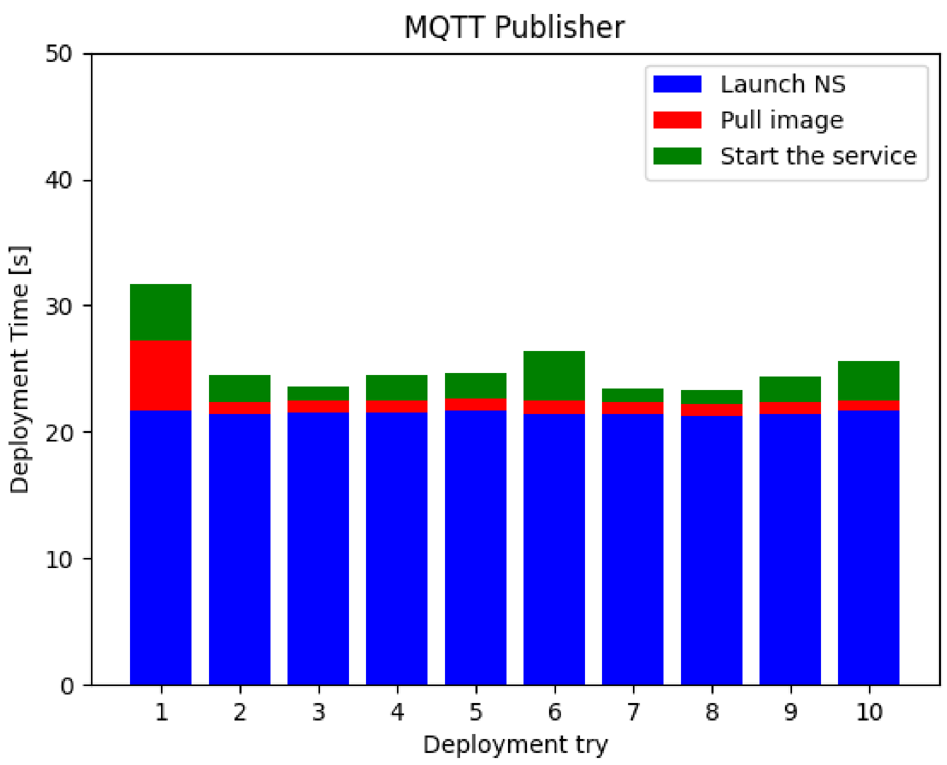


Figure 8.9: MQTT Publisher deployment time

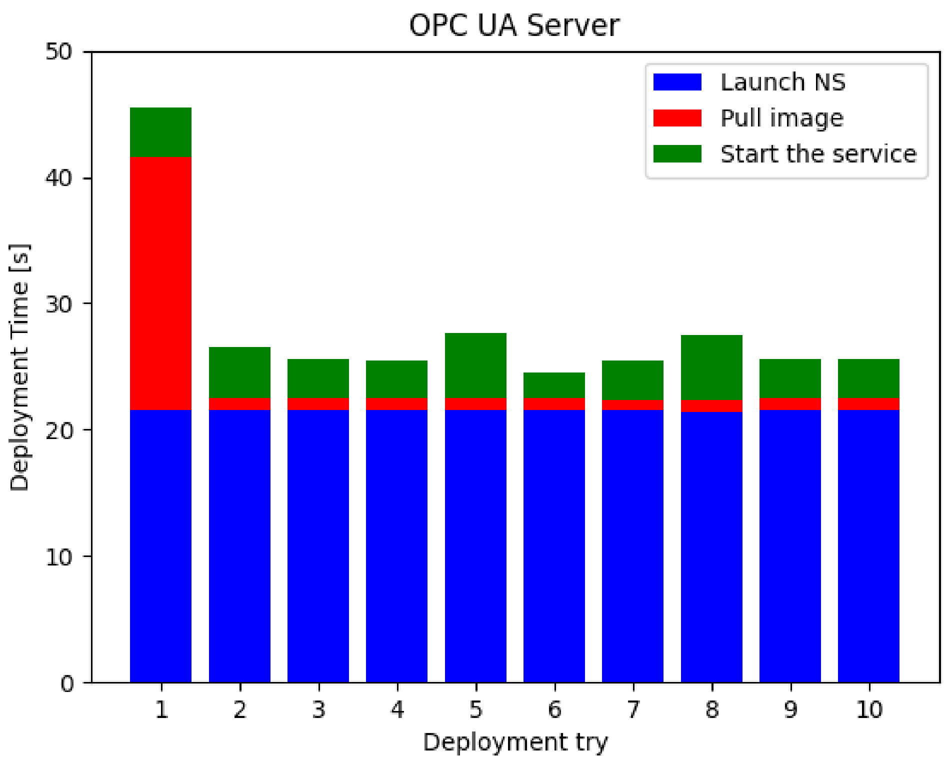


Figure 8.10: OPC UA Server deployment time

OPC UA Server	**MEC Platform**
NS creation (average): 22.585858 s Pull image (1St Pull): 18.377131 s Pull image (average): 0.986 s Service start (average): 6.4 s Total Average: 29.966 s	NS creation: 21.432464 Pull image: 5.556657 Service start (average): 4.5 s Total Average: 31.48 s

Client	**MQTT Publisher**
NS creation (average): 21.856711 Pull image (1St Pull): 12.546575 Pull image (average): 0.986 s Service start (average): 4.9 s Total Average: 27.736 s	NS creation (average): 21.586174 Pull image (1St Pull): 4.054524 Pull image (average): 0.986 s Service start (average): 3.7 s Total Average: 25.38 s

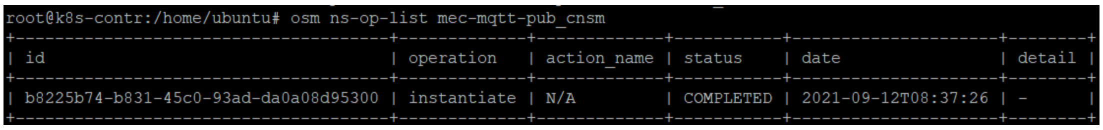
Table 14: Average deployment times

In Table 14 are reported the deployment times. All the averages are calculated excluding the first deployment.

8.6.1 *PoC Implementation*

For a real-world implementation of this scenario is required first the creation of the NS of the MEC Platform before that all the other services. This can be achieved launching the creation of the MEC Platform few seconds before the others. Once the that is done, all the other services can be created and registered over the *Service Registry* of it.

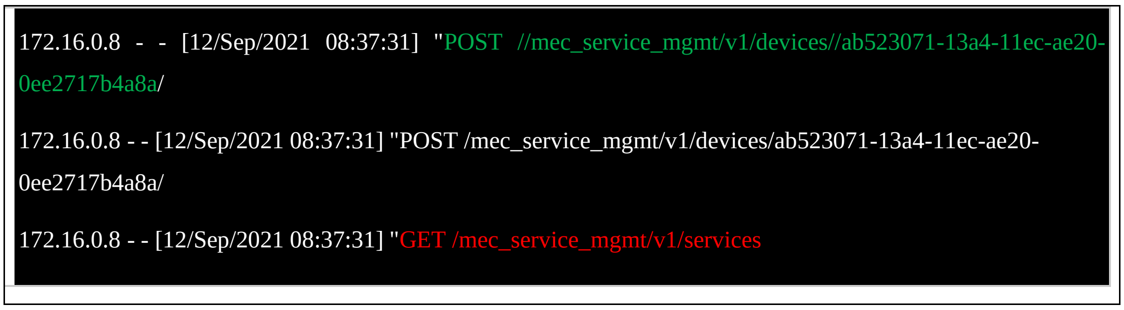
The registration procedure is not immediate but requires few seconds to be accomplished.



```
root@k8s-contr:/home/ubuntu# osm ns-op-list mec-mqtt-pub_cnsm
+--------------------------------------+-------------+-------------+-----------+---------------------+--------+
| id                                   | operation   | action_name | status    | date                | detail |
+--------------------------------------+-------------+-------------+-----------+---------------------+--------+
| b8225b74-b831-45c0-93ad-da0a08d95300 | instantiate | N/A         | COMPLETED | 2021-09-12T08:37:26 | -      |
+--------------------------------------+-------------+-------------+-----------+---------------------+--------+
```

Figure 8.11: MQTT NS creation

As shown in Figure 8.11, the creation of the MQTT NS is completed at the time **08.37.26**.



```
172.16.0.8 - - [12/Sep/2021 08:37:31] "POST //mec_service_mgmt/v1/devices//ab523071-13a4-11ec-ae20-0ee2717b4a8a/

172.16.0.8 - - [12/Sep/2021 08:37:31] "POST /mec_service_mgmt/v1/devices/ab523071-13a4-11ec-ae20-0ee2717b4a8a/

172.16.0.8 - - [12/Sep/2021 08:37:31] "GET /mec_service_mgmt/v1/services
```

Figure 8.12: MEC Platform logs

In Figure 8.12 are reported the logs of the MEC Platform related to the registration of that service. It is possible to see that the **POST** of the service inside the service registry happened 5 seconds later the creation of the service (**08:37:31**)

The IoT (once launched) can get access to the dashboard for gathering the data of all the sensors registered (**GET**), without the awareness of the type of transport protocol used or the endpoint information of that service. Everything is transparent from the Client point of view. In case of Figure 8.12 the client is already launched, and the **GET** of the service happens immediately after the **POST** of it.

Creation of the service → **08.37.26**

POST of the service on the MEC Platform → **08:37:31**

GET of the service by the Client → **08:37:31**

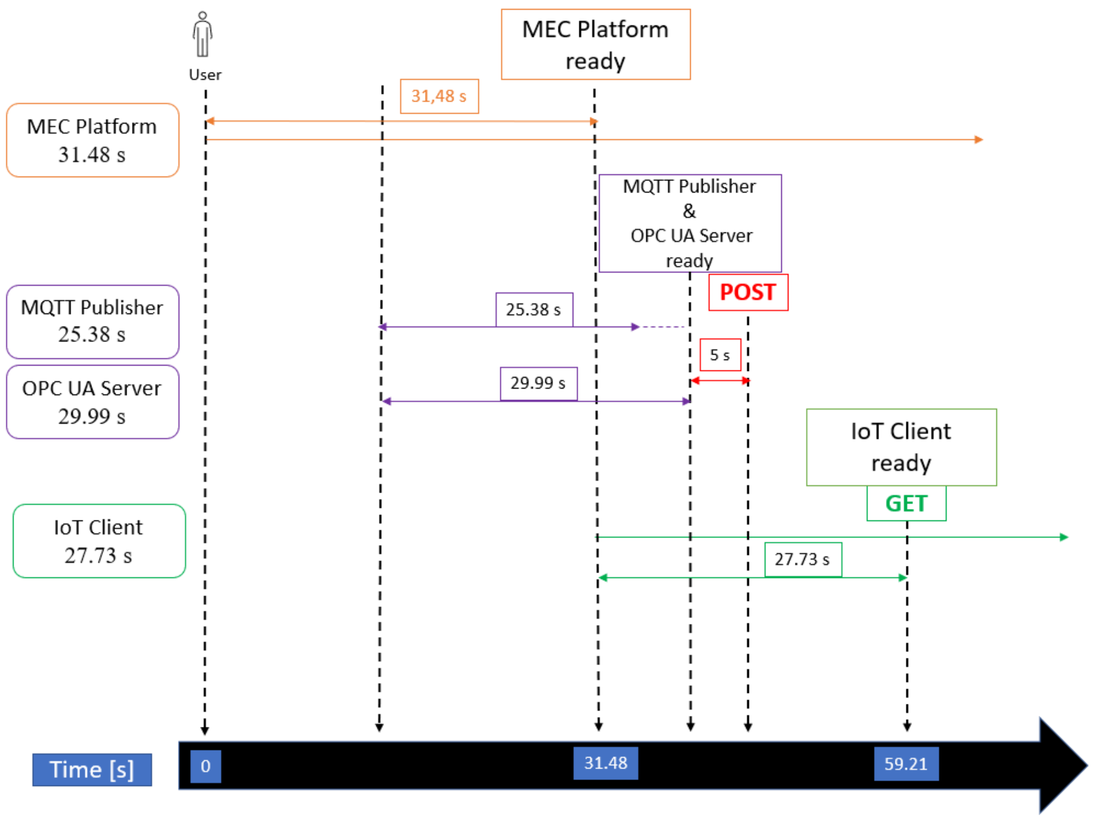


Figure 8.13: Possible deployment example

In Figure 8.13 is reported a scheme with all the deployment times in a possible real case scenario. In this case is considered that all the Docker images were already pulled (except for the MEC Platform). This means that at the start of the NSs no pulls do not need to be done, and the setup of the service can take less time.

It is possible to see that all the system is up and running (starting from scratch) in less than 60 seconds.

The deployment time can be further reduced in a system where the MEC Platform is already executed and only services and clients need to be deployed.

CONCLUSIONS

The trend in industries, cars, buildings and all the other fields where communications play an important role is by now evident. The number of IoT devices that require to be connected to the network is increasing day by day. Some solutions need to be evaluated to handle all these devices and guarantee to them connectivity.

This thesis has proposed and developed a MEC-based solution to support application management and fruition in a reference scenario of Industrial IoT as a Service. Thanks to the definition of a standardized environment for the deployment of IoT services and integrating a set of standardized APIs that can simplify the interaction between these services, even in a multi-vendor case, an Industry 4.0 scenario can be easily configured with a high degree of adaptability and flexibility.

In the purposed scenario, a multitude of IoT sensors that are running their own communication protocol (MQTT or OPC UA) can coexist without any problem. Sensing and data gathering service can be deployed on demand in a multi-vendor scenario in a very easy way using standardized APIs and in a matter of tens of seconds.

APPENDIX

All the code of used can be found at [57], including also all the bash and Python scripts used.

Here I report only Dockerfiles, Helm Charts and OSM Descriptors.

Appendix A: Dockerfiles

docker-img-mec

```
FROM alpine

RUN apk update
RUN apk add python3
RUN apk add py3-flask
RUN apk add py3-requests

COPY app.py /usr/local/bin/app.py
COPY app_site /var/www/
RUN chmod +x /usr/local/bin/app.py

EXPOSE 80

CMD /usr/local/bin/app.py
CMD /usr/local/bin/app.py
```

mqtt-pub-docker

```
FROM alpine

RUN apk update
RUN apk add python3
RUN apk add py3-flask
RUN apk add py3-requests
RUN apk add py3-paho-mqtt

ENV MY_POD_NAMESPACE=mec
ENV MEC_BASE=mec
ENV MY_POD_NAME=mec

COPY app.py /usr/local/bin/app.py
RUN chmod +x /usr/local/bin/app.py
EXPOSE 80

CMD /usr/local/bin/app.py
```

client-mec-docker

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y software-properties-common
RUN apt-get update
RUN apt install -y git
RUN apt-get install -y python3
RUN apt-get install -y python3-pip



RUN pip3 install flask
RUN pip3 install requests
RUN pip3 install paho-mqtt

RUN pip3 install opcua

ENV MY_POD_NAMESPACE=mec
ENV MEC_BASE=mec
ENV MY_POD_NAME=mec

COPY app.py /usr/local/bin/app.py
COPY app_site /var/www/

#Aggiunta di ulteriori files
COPY node.py /usr/local/bin/node.py
RUN chmod +x /usr/local/bin/node.py

COPY NodeBrowser.py /usr/local/bin/NodeBrowser.py
RUN chmod +x /usr/local/bin/NodeBrowser.py

COPY change_parameter_namespace.py /usr/local/bin/change_parameter_namespace.py
RUN chmod +x /usr/local/bin/change_parameter_namespace.py

COPY show_namespace_variable.py /usr/local/bin/show_namespace_variable.py
RUN chmod +x /usr/local/bin/show_namespace_variable.py



RUN chmod +x /usr/local/bin/app.py
EXPOSE 80

CMD /usr/local/bin/app.py
```

server-mec-docker

```
FROM lorenzobassi/opcua-base-python

#INSTALLAZIONE LIBRERIE ---------------------------
#RUN apt-get update
#RUN apt-get install -y software-properties-common
#RUN apt-get update
#RUN apt install -y git
#RUN apt-get install -y python3
#RUN apt-get install -y python3-pip
#RUN apt-get install -y cmake
#RUN apt install net-tools
#RUN apt-get install -y mysql-server
RUN pip3 install requests

ENV MY_POD_NAMESPACE=mec
ENV MEC_BASE=mec
ENV MY_POD_NAME=mec

ARG time=default_value

ENV time=${time}

COPY opc-ua-server-stuartup.py /usr/local/bin/opc-ua-server-stuartup.py
RUN chmod +x /usr/local/bin/opc-ua-server-stuartup.py



COPY run-mec-opcua-server.sh /home/run-mec-opcua-server.sh
RUN chmod +x /home/run-mec-opcua-server.sh

EXPOSE 4840
CMD /home/run-mec-opcua-server.sh $time
```

Appendix B: Helm Charts

mec-opcua_server

```
---
{{ $randNum := randNumeric 3 }}
apiVersion: v1
kind: Pod
metadata:
  name: opcua-server-{{$randNum}}
  labels:
    app: opcua-server-{{$randNum}}
spec:
  #hostNetwork: true
  #dnsPolicy: ClusterFirstWithHostNet
  containers:
    - name: opcua-server-1
      image: lorenzobassi/opcua_server_mec-cnsm
      env:
        # MEC Platform endpoint
        - name: MEC_BASE
          value: "http://mec-platform"
        - name: INFRA
          value: "k8s"
        - name: MY_POD_NAME
          value: "opcua-server-{{$randNum}}"
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
      ports:
        - containerPort: 4840
      imagePullPolicy: Always
---
apiVersion: v1
kind: Service
metadata:
  name: opcua-server-{{$randNum}}
spec:
  type: LoadBalancer
  selector:
    app: opcua-server-{{$randNum}}
  ports:
    - port: 4840
      targetPort: 4840
```

mec-iot-client

```
---
{{ $randNum := randNumeric 3 }}
apiVersion: v1
kind: Pod
metadata:
  name: mec-iot-client-{{$randNum}}
  labels:
    app: mec-iot-client-{{$randNum}}
spec:
  #hostNetwork: true
  #dnsPolicy: ClusterFirstWithHostNet
  containers:
    - name: mec-iot-client-1
      image: davideborsatti/cnsm-iot-client
      env:
        # MEC Platform endpoint
        - name: MEC_BASE
          value: "http://mec-platform"
        - name: INFRA
          value: "k8s"
        - name: MY_POD_NAME
          value: "mec-iot-client-{{$randNum}}"
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
      imagePullPolicy: Always
  #nodeSelector:
  #  project: mec
---
apiVersion: v1
kind: Service
metadata:
  name: mec-iot-client-{{$randNum}}
spec:
  type: LoadBalancer
  selector:
    app: mec-iot-client-{{$randNum}}
  ports:
    - port: 80
      targetPort: 80
```

mec-platform

```
---
apiVersion: v1
kind: Pod
metadata:
  name: mec-platform
  labels:
    app: mec-platform
spec:
  containers:
    - name: mec-platform-1
      image: davideborsatti/cnsm-mep-iot
      ports:
        - containerPort: 80
      env:
        # Application instance identifier
        - name: APP_INSTANCE_ID
          value: "997fc80a-cfc1-498a-b77f-608f09506e88"
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
      imagePullPolicy: Always
#  nodeSelector:
#    project: mec
---
apiVersion: v1
kind: Service
metadata:
  name: mec-platform
spec:
        #type: NodePort
  type: LoadBalancer
  selector:
    app: mec-platform
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30080
```

mqtt-pub-docker

```
---
{{ $randNum := randNumeric 3 }}
apiVersion: v1
kind: Pod
metadata:
  name: mqtt-pub-mec-{{$randNum}}
  labels:
    app: mqtt-pub-mec-{{$randNum}}
spec:
  #hostNetwork: true
  #dnsPolicy: ClusterFirstWithHostNet
  containers:
    - name: mqtt-pub-mec-1
      image: davideborsatti/cnsm-mqtt-pub
      env:
        # MEC Platform endpoint
        - name: MEC_BASE
          value: "http://mec-platform"
        - name: INFRA
          value: "k8s"
        - name: MY_POD_NAME
          value: "mqtt-pub-mec-{{$randNum}}"
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
      imagePullPolicy: Always
      volumeMounts:
        - name: config-volume
          mountPath: /etc/mqtt

  volumes:
    - name: config-volume
      configMap:
        name: topic-configmap-{{$randNum}}
  #nodeSelector:
  #  project: mec
---
apiVersion: v1
kind: Service
metadata:
  name: mqtt-pub-mec-{{$randNum}}
spec:
  type: LoadBalancer
  selector:
    app: mqtt-pub-mec-{{$randNum}}
  ports:
    - port: 80
      targetPort: 80
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: topic-configmap-{{$randNum}}
data:
  topics.json: |
    {"topics": ["dev{{$randNum}}/temperature","dev{{$randNum}}/air_quality"]}
```

Appendix C: OSM Descriptors

mec-iot-client_vnfd

```
vnfd:
    description: KNF for deploying a MEC IoT Client
    df:
    -   id: default-df
    ext-cpd:
    -   id: mgmt-ext
        k8s-cluster-net: os
    id: mec-iot-client_knf
    k8s-cluster:
        nets:
        -   id: mgmtnet
    kdu:
    -   helm-chart: chartmuseum/mec-iot-client
        name: mec-iot-client
    mgmt-cp: mgmt-ext
    product-name: mec-iot-client_knf
    provider: Unibo
    version: '1.0'
```

mec-iot-client_nsd

```
nsd:
    nsd:
    -   description: NS consisting of a single KNF mec-mqtt-
pub_knf connected to mgmt
            network
        designer: Unibo
        df:
        -   id: default-df
            vnf-profile:
            -   id: mec-iot-client
                virtual-link-connectivity:
                -   constituent-cpd-id:
                    -   constituent-base-element-id: mec-iot-client
                        constituent-cpd-id: mgmt-ext
                    virtual-link-profile-id: mgmtnet
                vnfd-id: mec-iot-client_knf
        id: mec-iot-client_ns
        name: mec-iot-client_ns
        version: '1.0'
        virtual-link-desc:
        -   id: mgmtnet
            mgmt-network: 'true'
            vim-network-name: mgmt
        vnfd-id:
        - mec-iot-client_knf
```

mec-mqtt-pub_vnfd

```
vnfd:
    description: KNF for deploying a MEC MQTT Publisher
    df:
    -   id: default-df
    ext-cpd:
    -   id: mgmt-ext
        k8s-cluster-net: os
    id: mec-mqtt-pub_knf
    k8s-cluster:
        nets:
        -   id: mgmtnet
    kdu:
    -   helm-chart: chartmuseum/mqtt-pub-docker
        name: mec-mqtt-pub
    mgmt-cp: mgmt-ext
    product-name: mec-mqtt-pub_knf
    provider: Unibo
    version: '1.0'
```

mec-mqtt-pub_nsd

```
nsd:
    nsd:
    -   description: NS consisting of a single KNF mec-mqtt-
pub_knf connected to mgmt
            network
        designer: Unibo
        df:
        -   id: default-df
            vnf-profile:
            -   id: mec-mqtt-pub
                virtual-link-connectivity:
                -   constituent-cpd-id:
                    -   constituent-base-element-id: mec-mqtt-pub
                        constituent-cpd-id: mgmt-ext
                    virtual-link-profile-id: mgmtnet
                vnfd-id: mec-mqtt-pub_knf
        id: mec-mqtt-pub_ns
        name: mec-mqtt-pub_ns
        version: '1.0'
        virtual-link-desc:
        -   id: mgmtnet
            mgmt-network: 'true'
            vim-network-name: mgmt
        vnfd-id:
        - mec-mqtt-pub_knf
```

mec-opcua_server_vnfd

```
vnfd:
    description: KNF for deploying an OPC UA Server
    df:
    -   id: default-df
    ext-cpd:
    -   id: mgmt-ext
        k8s-cluster-net: mgmtnet
    id: mec-opcua-server_knf
    k8s-cluster:
        nets:
        -   id: mgmtnet
    kdu:
    -   helm-chart: chartmuseum/mec-opcua-server
        name: mec-opcua-server
    mgmt-cp: mgmt-ext
    product-name: mec-opcua-server_knf
    provider: Unibo
    version: '1.0'
```

mec-opcua_server_nsd

```
nsd:
    nsd:
    -   description: NS consisting of a single KNF mec-opcua-
server_knf connected to mgmt
            network
        designer: OSM
        df:
        -   id: default-df
            vnf-profile:
            -   id: mec-opcua-server
                virtual-link-connectivity:
                -   constituent-cpd-id:
                    -   constituent-base-element-id: mec-opcua-server
                        constituent-cpd-id: mgmt-ext
                    virtual-link-profile-id: mgmtnet
                vnfd-id: mec-opcua-server_knf
        id: mec-opcua-server_ns
        name: mec-opcua-server_ns
        version: '1.0'
        virtual-link-desc:
        -   id: mgmtnet
            mgmt-network: 'true'
            vim-network-name: mgmt
        vnfd-id:
        - mec-opcua-server_knf
```

mec-platform_vnfd

```
vnfd:
    description: KNF for deploying a MEC Platform
    df:
    -   id: default-df
    ext-cpd:
    -   id: mgmt-ext
        k8s-cluster-net: os
    id: mec-platform_knf
    k8s-cluster:
        nets:
        -   id: mgmtnet
    kdu:
    -   helm-chart: chartmuseum/mec-platform
        name: mec-platform
    mgmt-cp: mgmt-ext
    product-name: mec-platform_knf
    provider: Unibo
    version: '1.0'
```

mec-platform_nsd

```
    sd:
        nsd:
        -   description: NS consisting of a single KNF mec-
platform_knf connected to mgmt
                network
            designer: Unibo
            df:
            -   id: default-df
                vnf-profile:
                -   id: mec-platform
                    virtual-link-connectivity:
                    -   constituent-cpd-id:
                        -   constituent-base-element-id: mec-platform
                            constituent-cpd-id: mgmt-ext
                        virtual-link-profile-id: mgmtnet
                    vnfd-id: mec-platform_knf
            id: mec-platform_ns
            name: mec-platform_ns
            version: '1.0'
            virtual-link-desc:
            -   id: mgmtnet
                mgmt-network: 'true'
                vim-network-name: mgmt
            vnfd-id:
            - mec-platform_knf
```

Appendix D: Python Scripts

opc-ua-server-stuartup.py

```
#!/usr/bin/env python3

#!/usr/bin/env python3

import os
import json
import requests
import uuid
import socket
import sys

# GET OPCUA DATA
with open('/usr/local/bin/opcua_fields.json') as file:
   data = json.load(file)

#ipadd = socket.gethostname()
#valore_ip = socket.gethostbyname(ipadd)

#valore_porta = data["opcua"][0]["port"]
#valore_uri = data["opcua"][0]["uri"]

#valore_object_name = data["opcua"][1]["objects"][0]["object_name"]
#valore_sensor1 = data["opcua"][1]["objects"][0]["variables"][0]
#valore_sensor2 = data["opcua"][1]["objects"][0]["variables"][1]

# MEC Service endpoint
MEC_SERVICE_MGMT="mec_service_mgmt/v1"

#mec_base = 'http://{}'.format(sys.argv[1])

mec_base = os.environ['MEC_BASE']

pod_namespace = os.environ['MY_POD_NAMESPACE']
pod_name = os.environ['MY_POD_NAME']

service_data = {
    "serInstanceId": "OPC_UA_SERVER",
    "serName": "OPC_UA_SERVER-service",
    "serCategory": {
    "href": "/example/catalogue1",
    "id": "id12345",
    "name": "RNI",
    "version": "version1"
    },
        "version": "ServiceVersion1",
        "state": "ACTIVE",
        "transportInfo": {
        "id": "OPC_UA_SERVER---1",
        "name": "OPC_UA_SERVER",
        "description": "OPC_UA_SERVER",
        "type": "MB_TOPIC_BASED",
        "protocol": "OPCUA",
        "version": "2.0",
        "endpoint": {
            "alternative": {
                "opcua":[]

         },
        "security": {
            "oAuth2Info": {
            "grantTypes": [
                "OAUTH2_CLIENT_CREDENTIALS"
            ],
            "tokenEndpoint": "/mecSerMgmtApi/security/TokenEndPoint"
            }
        },
        "implSpecificInfo": {}
        },
        "serializer": "JSON",
        "scopeOfLocality": "NFVI_NODE",
        "consumedLocalOnly": False,
        "isLocal": True
    }
    }

    service_data["transportInfo"]["endpoint"]["alternative"]["opcua"] = data["opcu
a"]
    app_instance_id = uuid.uuid1()

    #hostname = "{}.{}.mec.host".format(pod_name, pod_namespace)
    hostname = "{}.{}.mec.host".format(pod_name, pod_namespace)
    service_data['transportInfo']['endpoint']['alternative']['opcua'][0]['host'] =
  hostname

    query_base = "{}/{}/applications/{}/services".format(
                mec_base,
                MEC_SERVICE_MGMT,
                app_instance_id
        )
    headers = {"content-type": "application/json"}
    r = requests.post(query_base, data=json.dumps(service_data), headers=headers)
```

server-example.py

```
import uuid
from threading import Thread
import copy
import logging
from datetime import datetime
import time
from math import sin
import sys
import random
import json
from opcua.ua import NodeId, NodeIdType



sys.path.insert(0, "..")

try:
    from IPython import embed
except ImportError:
    import code



    def embed():
        myvars = globals()
        myvars.update(locals())
        shell = code.InteractiveConsole(myvars)
        shell.interact()

from opcua import ua, uamethod, Server



class SubHandler(object):
    """
    Subscription Handler. To receive events from server for a subscription
    """


    def datachange_notification(self, node, val, data):
        print("Python: New data change event", node, val)

    def event_notification(self, event):
        print("Python: New event", event)



# method to be exposed through server

def func(parent, variant):
    ret = False
    if variant.Value % 2 == 0:
        ret = True
    return [ua.Variant(ret, ua.VariantType.Boolean)]



# method to be exposed through server
# uses a decorator to automatically convert to and from variants

@uamethod
def multiply(parent, x, y):
    print("multiply method call with parameters: ", x, y)
    return x * y



class VarUpdater(Thread):
    def __init__(self, var):
        Thread.__init__(self)
        self._stopev = False
        self.var = var

    def stop(self):
        self._stopev = True

    def run(self):
        while not self._stopev:
            v = sin(time.time() / 10)
            self.var.set_value(v)
            time.sleep(0.1)



if __name__ == "__main__":
    # optional: setup logging
    logging.basicConfig(level=logging.WARN)
    #logger = logging.getLogger("opcua.address_space")
    # logger.setLevel(logging.DEBUG)
    #logger = logging.getLogger("opcua.internal_server")
    # logger.setLevel(logging.DEBUG)
    #logger = logging.getLogger("opcua.binary_server_asyncio")
    # logger.setLevel(logging.DEBUG)
    #logger = logging.getLogger("opcua.uaprocessor")
    # logger.setLevel(logging.DEBUG)

    # now setup our server
    server = Server()
    #server.disable_clock()
    #server.set_endpoint("opc.tcp://localhost:4840/freeopcua/server/")
```

```
	hostname = str(sys.argv[1])
	port = str(sys.argv[2])

	print("hostname in input: ",hostname)
	print("port in input: ",port)

	domain = "opc.tcp://"
	ddd = ":"
	final_address = (domain+hostname+ddd+port)
	server.set_endpoint(final_address)

	server.set_server_name("FreeOpcUa Example Server")
	# set all possible endpoint policies for clients to connect through
	server.set_security_policy([
	            ua.SecurityPolicyType.NoSecurity,
	            ua.SecurityPolicyType.Basic256Sha256_SignAndEncrypt,
	            ua.SecurityPolicyType.Basic256Sha256_Sign])

	# setup our own namespace
	#uri = "http://examples.freeopcua.github.io"
	#uri = str(sys.argv[3]) ******************************
	uri = str(sys.argv[4])
	print("uri in input: ",uri)
	idx = server.register_namespace(uri)

	# create a new node type we can instantiate in our address space
	dev = server.nodes.base_object_type.add_object_type(idx, "MyDevice")
	dev.add_variable(idx, "sensor1", 1.0).set_modelling_rule(True)
	dev.add_property(idx, "device_id", "0340").set_modelling_rule(True)
	ctrl = dev.add_object(idx, "controller")
	ctrl.set_modelling_rule(True)
	ctrl.add_property(idx, "state", "Idle").set_modelling_rule(True)

	# populating our address space

	# First a folder to organise our nodes
	myfolder = server.nodes.objects.add_folder(idx, "myEmptyFolder")
	# instanciate one instance of our device
	mydevice = server.nodes.objects.add_object(idx, "Device0001", dev)
	mydevice_var       =       mydevice.get_child(["{}:controller".format(idx),
"{}:state".format(idx)])  # get proxy to our device state variable




	# create directly some objects and variables
	# READ OBJECTS AND VARIABLE FORM THE JSON FILE
```

```
       with open('opcua_fields.json') as file:
           data = json.load(file)

       dim_obj_list = len(data["opcua"][0]["objects"])



       for i in range(0, dim_obj_list):
         dim_var_list = len(data["opcua"][0]["objects"][i]["variables"])
         globals()[f"obj_{i}"]   =   data["opcua"][0]["objects"][i]["object_name"]
#cration of: obj_0, obj_1
         globals()[f"myobj_{i}"]      =      server.nodes.objects.add_object(idx,
globals()[f"obj_{i}"])                #creation of: myobj_0, myobj_1
         #print("----------------------------------------")
         #print("i:", i)
         #print(data["opcua"][0]["objects"][i]["object_name"])
         for j in range(0, dim_var_list):
                globals()[f"variable_{i}_{j}"]                                  =
data["opcua"][0]["objects"][i]["variables"][j]        #creation  of:  variable_0_0,
variable_0_1
                globals()[f"myvar_{i}_{j}"]                                     =
globals()[f"myobj_{i}"].add_variable(idx,   globals()[f"variable_{i}_{j}"],   6.7)
#creation of: myvar_0_0, myvar_0_1
                globals()[f"myvar_{i}_{j}"].set_writable()
                #print(variable_{i}_{j})
                #print(i,j)
                #print(data["opcua"][0]["objects"][i]["variables"][j])



       # create directly some objects and variables
       myobj = server.nodes.objects.add_object(idx, "MyObject")
       myvar = myobj.add_variable(idx, "MyVariable", 6.7)
       mysin = myobj.add_variable(idx, "MySin", 0, ua.VariantType.Float)
       myvar.set_writable()    # Set MyVariable to be writable by clients
       mystringvar  =  myobj.add_variable(idx,  "MyStringVariable",  "Really  nice
string")
       mystringvar.set_writable()  # Set MyVariable to be writable by clients
       myguidvar  =  myobj.add_variable(NodeId(uuid.UUID('1be5ba38-d004-46bd-aa3a-
b5b87940c698'), idx, NodeIdType.Guid),
                                      'MyStringVariableWithGUID',  'NodeId  type  is
guid')
       mydtvar = myobj.add_variable(idx, "MyDateTimeVar", datetime.utcnow())
       mydtvar.set_writable()    # Set MyVariable to be writable by clients
       myarrayvar = myobj.add_variable(idx, "myarrayvar", [6.7, 7.9])
       myarrayvar     =     myobj.add_variable(idx,     "myStronglytTypedVariable",
ua.Variant([], ua.VariantType.UInt32))
       myprop = myobj.add_property(idx, "myproperty", "I am a property")
       mymethod = myobj.add_method(idx, "mymethod", func, [ua.VariantType.Int64],
[ua.VariantType.Boolean])
       multiply_node     =     myobj.add_method(idx,     "multiply",     multiply,
[ua.VariantType.Int64, ua.VariantType.Int64], [ua.VariantType.Int64])

       # import some nodes from xml
       server.import_xml("custom_nodes.xml")

       # creating a default event object
       #  The  event  object  automatically  will  have  members  for  all  events
properties
       # you probably want to create a custom event type, see other examples
       myevgen = server.get_event_generator()
       myevgen.event.Severity = 300

       # starting!
       server.start()
       print("Available loggers are: ", logging.Logger.manager.loggerDict.keys())
       vup = VarUpdater(mysin)  # just  a stupide class update a variable
       vup.start()
       try:
           # enable following if you want to subscribe to nodes on server side
           #handler = SubHandler()
           #sub = server.create_subscription(500, handler)
           #handle = sub.subscribe_data_change(myvar)
           # trigger event, all subscribed clients wil receive it
           var  =  myarrayvar.get_value()   #  return  a  ref  to  value  in  db  server
side! not a copy!
           var = copy.copy(var)  # WARNING: we need to copy before writting again
otherwise no data change event will be generated
           var.append(9.3)
           myarrayvar.set_value(var)
           mydevice_var.set_value("Running")
           myevgen.trigger(message="This is BaseEvent")
           #server.set_attribute_value(myvar.nodeid, ua.DataValue(9.9))  # Server
side write method which is a but faster than using set_value
           server.set_attribute_value(myvar_0_0.nodeid,   ua.DataValue(9.9))     #
Server side write method which is a but faster than using set_value



           #sleep = sys.argv[3] if len(sys.argv) >= 5 else '1'
           #sleep = int(sys.argv[3])

           #if type(int(sys.argv[4])) == int:
           #     sleep = int(sys.argv[4])
           #else:
           #     sleep = 1
           #sleepstr = sys.argv[4] if len(sys.argv) >= 5 else 1
           #print(sleepstr)
           #sleep = int(sleepstr)



           #if sys.argv[4] == 'default_value':

           #if sys.argv[4] == 'http://examples.freeopcua.github.io':
           #    sleep = 1
           #else:
           #    sleep = int(sys.argv[4])

           if sys.argv[3] == 'default_value':
               sleep = 1
           else:
               sleep = int(sys.argv[3])
           print("sleep in input: ",sleep)



           while 15 == 15:
                 temp = random.randint(1,10)
                 server.set_attribute_value(myvar_0_0.nodeid, ua.DataValue(temp))
                 print("Nuovo valore myvar_0_0 : ", temp )
                 time.sleep(sleep)

           embed()
       finally:
           vup.stop()
           server.stop()
```

get_all_sensors.py

```
import json

with open('opcua_fields.json') as file:
   data = json.load(file)

dimensione_lista = len(data["opcua"][0]["objects"][0]["variables"])
i=0

while (i<dimensione_lista):
     valore_var = data["opcua"][0]["objects"][0]["variables"][i]
     print(valore_var)
     i=i+1
```

BIBLIOGRAPHY

[1] M. P. A. R. a. J. S. Fredrik Dahlqvist, «McKinsey & Company,» [Online]. Available: ***https://www.mckinsey.com/industries/private-equity-and-principal-investors/our-insights/growing-opportunities-in-the-internet-of-things.***

[2] I. Z. R. Laboratory, «Request for Comments: 3234,» [Online].

Available: ***https://tools.ietf.org/html/rfc3234.***

[3] ETSI, «ETSI,» ETSI, [Online].

Available: ***https://www.etsi.org/technologies/nfv.***

[4] Kubernetes, [Online].

Available: ***https://kubernetes.io/.***

[5] ETSI, ETSI, [Online].

Available: ***https://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/001/01.01.01_60/gs_NFV-INF001v010101p.pdf.***

[6] Wikipedia, «Wikipedia,» [Online].

Available: ***https://en.wikipedia.org/wiki/FCAPS#Accounting_management***.

[7] ETSI, «ETSI GS NFV-MAN 001 V1.1.1,» ETSI, [Online].

Available: ***https://www.etsi.org/deliver/etsi_gs/nfv-man/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf.***

[8] «Open Baton,» Open Baton, [Online].

Available: ***https://openbaton.github.io/documentation/ns-descriptor/.***

[9] O. O. 2017, «OASIS,» OASIS , 2017. [Online].

Available: ***http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/csd03/tosca-nfv-v1.0-csd03_files/image004.png.***

[10] ETIS, «ETSI,» ETSI, [Online].

Available: ***https://www.etsi.org/technologies/open-source-mano.***

[11] ETSI, «ETSI,» ETSI, [Online].

Available: ***https://osm.etsi.org/wikipub/index.php/RO_Northbound_Interface***.

[12] ETSI, «ETSI,» [Online].

Available: ***https://osm.etsi.org/wikipub/images/2/OSM(16)000020_MWC_demo_components_-_OpenMANO.pdf.***

[13] ETSI, «ETSI OSM,» 2019. [Online].

Available: ***https://osm.etsi.org/docs/vnf-onboarding-guidelines/00-introduction.html.***

[14] C. Ltd., «JAAS,» Canonical Ltd., 2021. [Online].

Available: ***https://jaas.ai/how-it-works.***

[15] C. Ltd, «Canonical Ltd,» [Online].

Available: ***https://cloud-init.io/.***

[16] C. Ltd, «Canonical Ltd,» Canonical Ltd, 2021. [Online].

Available: ***https://juju.is/docs/concepts-and-terms.***

[17] ETSI, «ETSI SOM,» [Online].

Available: ***https://osm.etsi.org/wikipub/index.php/Creating_your_VNF_Charm.***

[18] ETSI, «ETSI OSM,» ETSI, [Online].

Available: ***https://osm.etsi.org/wikipub/index.php/Creating_your_VNF_Charm.***

[19] C. Ltd, «Canonical Ltd,» Canonical Ltd, 2021. [Online].

Available: ***https://jaas.ai/how-it-works.***

[20] E. OSM, «ETSI,» ETSI OSM, [Online].

Available: ***https://osm.etsi.org/wikipub/index.php/Creating_your_VNF_Charm.***

[21] E. OSM, «ETSI,» [Online].

Available: ***http://osm-download.etsi.org/ftp/osm-6.0-six/8th-hackfest/presentations/8th%20OSM%20Hackfest%20-%20Session%207.1%20-%20Introduction%20to%20Proxy%20Charms.pdf***.

[22] M. Farina, «Helm Blog,» Cloud Native Computing Foundation graduated project., [Online].

Available: ***https://helm.sh/blog/new-location-stable-incubator-charts/.***

[23] E. OSM, 2019. [Online].

Available: ***https://www.etsi.org/deliver/etsi_gs/mec/001_099/003/02.01.01_60/gs_mec003v020101p.pdf.***

[24] E. OSM, «RGS/MEC-0003v211Arch,» [Online].

Available: ***https://www.etsi.org/deliver/etsi_gs/mec/001_099/003/02.01.01_60/gs_mec003v020101p.pdf.***

[25] «OpenStack main page,» OpenStack, [Online].

Available: ***https://www.openstack.org/software/.***

[26] G. L. (Whitestack), «ETSI OSM,» 2019. [Online].

Available: ***http://osm-download.etsi.org/ftp/osm-6.0-six/8th-hackfest/presentations/8th%20OSM%20Hackfest%20-%20Session%207.1%20-%20Introduction%20to%20Proxy%20Charms.pdf.***

[27] «OpenStack,» 2021. [Online].

Available: ***https://docs.openstack.org/install-guide/get-started-logical-architecture.html.***

[28] «OpenStack Install Guide,» [Online].

Available: ***https://docs.openstack.org//install-guide/InstallGuide.pdf.***

[29] «OpenStack Compute,» OpenStack, [Online].

Available: ***https://docs.openstack.org/nova/latest/.***

[30] «OpenStack Nova Architecture,» [Online].

Available***: https://docs.openstack.org/nova/latest/user/architecture.html.***

[31] «OpenStack project,» OpenStack, [Online].

Available: ***https://docs.openstack.org/cinder/ussuri/index.html***. [Consultato il giorno 08 2021].

[32] «OpenStack project,» OpenStack, [Online].

Available: ***https://docs.openstack.org/neutron/pike/install/common/get-started-networking.html***.

[33] «OpenStack,» [Online].

Available: ***https://wiki.openstack.org/wiki/Neutron***.

[34] «Cloudflare, Inc.,» Cloudflare., [Online].

Available: ***https://www.cloudflare.com/it-it/learning/cloud/what-is-multitenancy/.***

[35] «OpenStack project,» OpenStack , 08 2021. [Online].

Available: ***https://docs.openstack.org/heat/pike/.***

[36] «OpenStack project,» OpenStack , 08 2021. [Online].

Available: ***https://docs.openstack.org/tacker/latest/user/introduction.html***.

[37] «OpenStack project,» OpenStack , 08 2021. [Online].

Available: ***https://docs.openstack.org/tacker/latest/_images/tacker-design-etsi.png.***

[38] «OpenStack project,» 05 2018. [Online].

Available: ***https://docs.openstack.org/glance/victoria/install/get-started.html.***

[39] «OpenStack projec,» 11 2018. [Online].

Available: ***https://docs.openstack.org/openstack-ansible-os_swift/latest/configure-swift.html#overview.***

[40] «OpenStack projec,» 04 2021. [Online].

Available: ***https://docs.openstack.org/kayobe/latest/configuration/reference/ironic-python-agent.html.***

[41] M. Tesch, «LeanBI,» leanbi.ch, [Online].

Available: ***https://leanbi.ch/en/blog/iot-and-predictive-analytics-fog-and-edge-computing-for-industries-versus-cloud-19-1-2018/.***

[42] «OPC Foundation, The Industrial Interoperability Standard,» OPC Foundation, 04 2015. [Online]. Available: ***https://opcfoundation.org/news/opc-foundation-news/update-iec-62541-opc-ua-published/.***

[43] «OPC Foundation,» [Online].

Available: ***https://opcfoundation.org/wp-content/uploads/2016/05/OPC-UA-Interoperability-For-Industrie4-and-IoT-EN-v5.pdf.***

[44] «Unified Automation,» [Online].

Available: ***https://documentation.unified-automation.com/uasdkdotnet/3.0.2/html/L2UaAddressSpaceConcepts.html.***

[45] «Unified Automation,» [Online].

Available: ***https://documentation.unified-automation.com/uasdkdotnet/3.0.2/html/L2UaNodeClasses.html.***

[46] «Unified Automation GmbH,» [Online].

Available: ***https://documentation.unified-automation.com/uasdkhp/1.1.1/html/_l2_ua_discovery_connect.htm.***

[47] «Open62541 official page,» Open62541, [Online].

Available: ***https://open62541.org/.***

[48] «GitHub,» [Online].

Available***: https://github.com/FreeOpcUa/python-opcua.***

[49] «OPC UA Online Reference,» 2021 OPC Foundation, [Online].

Available: ***https://reference.opcfoundation.org/v104/Core/docs/Part6/7.1.2/.***

[50] «OPC UA Online Reference,» [Online].

Available: ***https://reference.opcfoundation.org/v104/Core/docs/Part4/5.12.1/.***

[51] «GitHub,» [Online].

Available: ***https://github.com/FreeOpcUa/opcua-client-gui***.

[52] «GitHub,» [Online].

Available: ***https://github.com/FreeOpcUa/opcua-modeler.***

[53] «OSM,» ETSI, [Online].

Available: ***https://osm.etsi.org/wikipub/index.php/OSM11_Hackfest.***

[54] «CNSM - 17th International Conference on Network and Service Management,» [Online].

Available: ***http://www.cnsm-conf.org/2021/cfd.html.***

[55] D. Borsatti, G. Davoli, W. Cerroni e C. Raffaelli, «IEEE Communications Magazine, vol 59,» 08 2021. [Online].

Available: ***https://ieeexplore.ieee.org/document/9530503***.

[56] D. Borsatti e L. Bassi, «GitHub,» 08 2021. [Online].

Available: ***https://github.com/DavideBorsatti/CNSM2021-MEC-IIoT.***

[57] L. Bassi, «GitHub,» 08 2021. [Online].

Available: ***https://github.com/lorenzobassi96/opcua_server_CNSM/blob/main/get_json_data/opcua_fields.json.***

[58] «Chart Museum,» Kubernetes, [Online].

Available: ***https://chartmuseum.com/docs/#.***

FIGURES INDEX

TABLES INDEX