SEMIoTICS

Deliverable D3.5
Field-level middleware & networking toolbox (first draft)

Deliverable release date	**31/12/2018**
Authors	1. Prodromos Vasileios Mekikis, Kostas Ramantas (IQU) 2. Luis Sanabria-Ruso, Jordi Serra, David Pubill, Angelos Antonopoulos, Christos Verikoukis (CTTC) 3. Kostas Fysarakis, Jason Somarakis (STS) 4. Ermin Sakic, Darko Anicic (SAG)
Responsible person	Prodromos Vasileios Mekikis (IQU)
Reviewed by	Ermin Sakic (SAG), Eftychia Lakka (FORTH), Nikolaos Petroulakis (FORTH)
Approved by	PTC and PCC Members
Status of the Document	Final Version
Version	1.0
Dissemination level	Confidential

Contents

Acronyms Table

Acronym	Definition
CPU	Central Processing Unit
IoT	Internet of Things
IIoT	Industrial Internet of Things
KVM	Kernel-based Virtual Machine
LXD	Linux Containers
JSON	JavaScript Object Notation
LWM2M	Lightweight Machine-to-M
M2M	Machine-to-Machine
MQTT	Message Queuing Telemetry Transport
NETCONF	Network Configuration Protocol
NFV	Network Functions Virtualization
NFVO	NFV Orchestrator
OFCONF	OpenFlow Configuration
OVSDB	Open vSwitch Database Management Protocol
POP	Point of Presence
QoS	Quality of Service
SDN	Software-Defined Networking
SARA	Socially Assistive Robotic Solution for Mild Cognitive Impairment or mild Alzheimer's disease
SEMIoTICS	Smart End-to-end Massive IoT Interoperability, Connectivity and Security
SFC	Service Function Chaining
SPDI	Security, Privacy, Dependability, and Interoperability
SSD	Solid State Disk
TD	Thing Description
UC	Use Case
VIM	Virtualized Infrastructure Manager
VLAN	Virtual Local Area Network
VM	Virtual Machine
VNF	Virtual Network Function
vSwitch	Virtual Switch
VTN	Virtual Tenant Networks
WoT	Web of Things

1 INTRODUCTION

SEMIoTICS aims to enhance the connectivity, latency, and bandwidth in industrial environments, while reducing the cost of their Information and Communications Technology (ICT) systems through a set of technologies under the umbrella of virtualization. To be more specific, network function virtualization (NFV) is a technique that can significantly benefit industries by optimizing their network services. It allows a software-defined implementation of networks as it decouples several network functions from previously required network devices, such as firewalls, and runs them as software, i.e., Virtual Network Functions (VNFs), at a data center. In this way, the NFV infrastructure (NFVI) does not only drop the deployment cost, as less equipment and installation personnel are needed, but it also reduces the service creation time from hours to minutes resulting in an extensively more efficient procedure.

To automate even further the networking procedures in the Industrial Internet of Things (IIoT), software-defined networking (SDN) can be employed, which is a complementary approach to NFV that separates the control and forwarding planes to offer a centralized view of the network. Moreover, for the handling of the physical and virtual resources that support the network virtualization, an NFV management and orchestration (MANO) is responsible for the lifecycle management of the VNFs and it focuses on all virtualization-specific management tasks necessary in the NFV framework. To that end, a service chain of connected VNFs, i.e., a service function chain (SFC), can be created to automatically run a requested application based on the current traffic demand. This capability can be employed by industries to set up sets of connected VNFs that allow the use of a single network connection for many services that have different characteristics.

Although the set of aforementioned technologies can substantially improve the efficiency of the network layer in IIoT, there is still the obstacle of the proximity to the cloud. Since ultra-reliable low-latency communications (uRLLC) are paramount for industrial environments, the network congestion might hinder the connection with the cloud. Therefore, Multi-access Edge Computing (MEC) has been proposed to address this issue by establishing a cloud-based ICT service environment at the network edge. Thus, real-time, high-bandwidth, low-latency access to radio network information becomes reality and improves application performance by achieving related task processing closer to the user.

During the last years, various NFV/SDN implementations have been demonstrated to prove the efficacy of the aforementioned technologies. In this deliverable, we investigate the introduction and adaption of SDN/NFV and semantic bootstrapping and interoperability technologies in industrial environments. Furthermore, we employ the well-defined SEMIoTICS architecture[1] to build an experimental platform that consists of open-source software and extends the capabilities of current industrial KPIs. To that end, the contribution of this deliverable is the following:

i) In Section 2 we contribute a preliminary design for the SEMIoTICS field-level middleware and define the SEMIoTICS development and release procedure.
ii) In Section 3 we discuss how concepts like NFV and SDN can be leveraged in the Industrial IoT domain.
iii) In Section 4, we study standardized semantic models for IIoT applications and SPDI pattern-driven mechanisms that guarantee network-level semantic bootstrapping interoperability.
iv) Finally, in Section 5 we present the design of the SEMIoTICS testbed and contribute some initial performance evaluation results, that provide useful insights for the deployment of IIoT applications on top of virtualized, programmable infrastructure.

[1] SEMIoTICS webpage: https://www.semiotics-project.eu/

2 MIDDLEWARE DESIGN AND IMPLEMENTATION

The sheer number of smart objects that are expected to connect to the Internet by will increase network traffic dramatically and introduce more diversity of network traffic. A series of innovations across the IoT landscape have converged to make IoT products, platforms and devices technically and economically feasible. Specifically, Integrating IoT and SDN will increase network efficiency as it will make it possible for a network to respond to changes or events detected at the IoT application layer through network reconfiguration. Moreover, NFV architectures allow monitoring, caching, security and data analytics functions to be virtualized and placed in a local and remote clouds, or event directly at IoT smart objects and Field level IoT gateways. Finally, intelligent data analytics running locally at the Field layer are needed to implement autonomic behaviour, but considering IoT smart objects' limited resources, specialized lightweight algorithms are required. The aforementioned complexities must be abstracted from the IIoT applications and field-level devices, simplifying the development and deployment of applications. Hence, SEMIoTICS has proposed the development of a Field-Level Middleware that will integrate the application modules and networking APIs implemented in T3.1-3.4 and provide ontology-driven access to data, ensuring interoperability.

2.1 Middleware Architecture and APIs

This section will focus on the design of a unified middleware layer between the IoT applications and the communication network, to abstract the underlying protocol implementations and SDN APIs. It must be noted that the Middleware is not a separate component of the SEMIoTICS architecture, which is detailed in D2.4, but rather the collection of frameworks implemented within T3.1-3.4. These will be deployed and evaluated in a testbed environment in the framework of T3.5. The Middleware ensures that functionalities such as establishing connectivity to a service, negotiating transport protocols and networking paths will be a totally transparent process for IoT applications. The Field Level Middleware acts as a message proxy, interconnecting the Backend services and Apps with the Field Level devices. Furthermore, it forwards the application requirements in terms of delay, minimum throughput, packet error rate tolerances, etc., to the Pattern Orchestrator via appropriate NBIs. These requirements are then translated to application-specific VTN slices, deployed via the SDN controller. Finally, the field-level middleware communicates with the MANO framework to control the placement of VNFs based on application requirements and trigger the setup of SFCs.

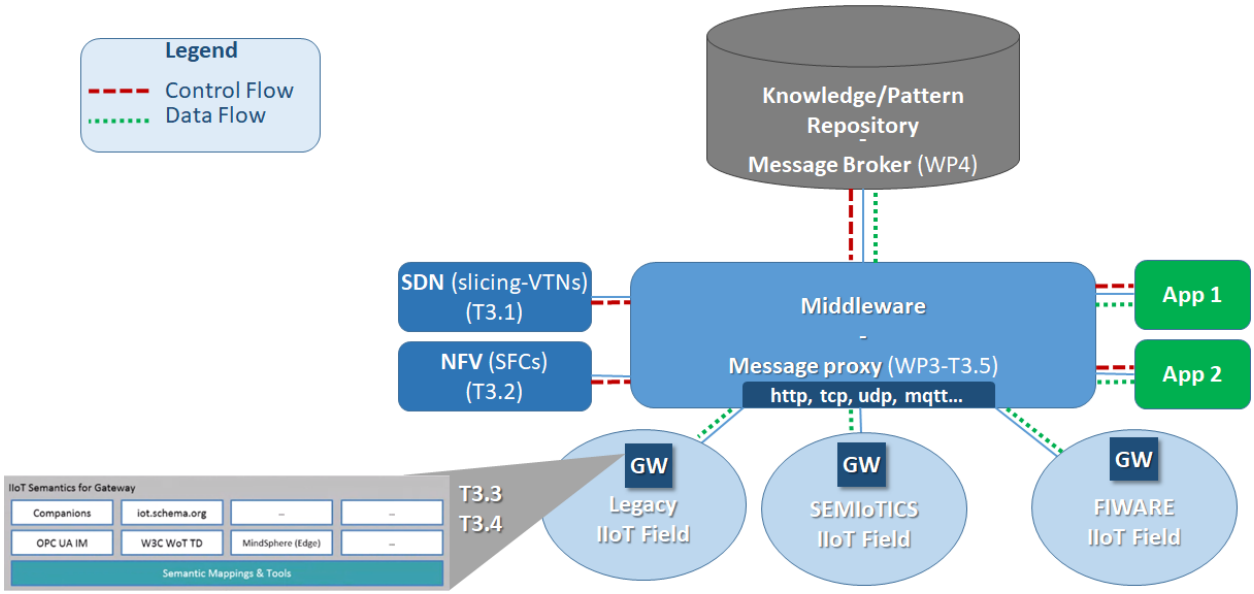

FIGURE 1: FIELD LEVEL MIDDLEWARE DESIGN

The Semantic Mapping of messages transmitted from the Field Level devices is performed at the IIoT gateway, employing the WoT data model. These messages are then delivered to a Message Broker service, which is hosted at the Backend Cloud. A Message Broker is generally the main component of IoT platforms. It works as a handler and aggregator of context data and as an interface between Applications and Field Level devices. A Context API is generally provided, e.g., via HTTP calls (GET, POST, PUT, DELETE) based on an IIoT Information Model. Some common interactions with the Message Broker include:

- Context queries, e.g., for sensor data stored at the local database
- Context updates, e.g., to update the local database with sensor values
- Context subscriptions, to receive updates when a certain device status (or a certain topic) is updated

The Field Level Middleware supports multiple south-bound device protocols (e.g., HTTP, MQTT, etc.), forwarding messages from the IIoT gateway to the Message Broker. Moreover, the field-level middleware will offer federation and interoperability with other IoT platforms, most notably FIWARE. In FIWARE, the Orion Context Broker fulfils the pub/sub Message Broker functionality and must be federated with SEMIoTICS. FIWARE leverages the NGSIv2 Data Model and API, which relies on JSON representation to make data from multiple providers accessible for data consumers. The interaction with both data providers and data consumers is taking place via the FIWARE NGSI 10 context data API. SEMIoTICS must leverage the API for context queries, context subscription and context updates to interact with the respective context elements (i.e., sensors and actuators) in a FIWARE domain. On the contrary, for FIWARE to access context elements in other domains (in this case SEMIoTICS) a specialized FIWARE entity, namely the ***Context Provider,*** must be involved. The latter can be registered via its URL as the source of context information for specific entities and attributes included in that registration, using the ORION NGSIv1 and NGSIv2 APIs. In the case of NGSIv2 Data Model, which uses JSON representation, this is provided by the field **provider**. If FIWARE Orion fails to find a context element locally (i.e. in its internal database) for a query or update operation but a Context Provider is registered for that context element, then it will forward the query or update request to the respective Provider. In this case, Orion acts as proxy, while the client that issues the request, the process is transparent. SEMIoTICS must implement the respective NGSI10 API (at least partially) to support query/update operations from FIWARE to a context element in the SEMIoTICS domain.

2.2 SEMIoTICS Implementation process

In SEMIoTICS, T3.5 is the main implementation task of WP3, which will deliver the SEMIoTICS Middleware in incremental releases. In the following sections, the software development and release processes are defined.

2.2.1 SEMIOTICS DEVELOPMENT AND RELEASE CYCLES

In the framework of T2.4 we have designed the SEMIoTICS architecture and defined the architectural components of each layer. Each architectural component is associated with a respective software module, and an owner is assigned. These software modules are implemented with an iterative process, which follows the concept of Continuous Integration (CI). This iterative development process is performed in cycles, with each cycle ending with a new software release. Each release cycle consists of the following phases, also illustrated in Figure 2, and is expected to last approximately 4 months:

1. Feature planning: The consortium agrees on the features that will be implemented in the next release. This might occur during a feature planning meeting. They compile all required mechanisms and interfaces in a high-level specification document, which also includes the test cases which will be executed during system verification. This phase requires approximately 1 month.

2. Development: With the specification document at hand, all required features are implemented by the responsible developers. Each partner is responsible for a certain number of architectural components, as defined in T2.4, and will have to implement all essential functionalities. Furthermore, appropriate testing will ensure that the developed components and feature sets perform as specified. Development requires 2 months.

3. Integration: After completion of the development phase, changes are integrated to the main SEMIoTICS codebase. Automated sanity tests are performed to rule-out regressions. The task requires 1-2 weeks.

4. System testing: The testing team deploys the new software release to the testbed and performs all the required system tests to validate that it runs as specified, and new modules and features correctly interoperate with the rest of the system. In cases of issues, they report back to the responsible developers, and depending on the required effort further development might occur to fix the issue or move the issues for resolution in upcoming releases. This phase requires 2-3 weeks.

5. System release: Eventually, the integrator generates all the release artifacts and documents and tags the current version of the software. In addition, a system release review meeting takes place to identify and discuss problems encountered during this release cycle.

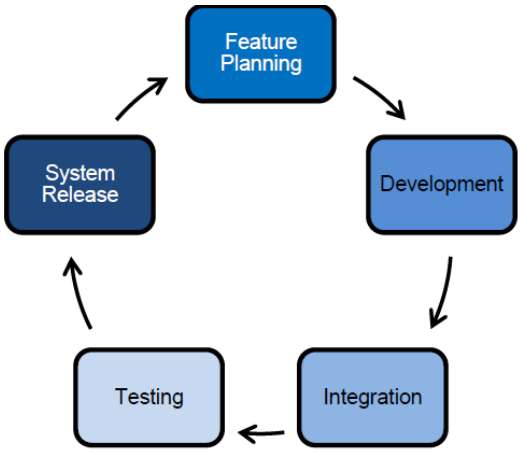

FIGURE 2: SEMIOTICS RELEASE CYCLE

Tentatively, the consortium considers the following release schedule. The development process will start on M13.

- On M17 we will have the first software release, with the basic functionality of the SEMIoTICS backend implemented
- On M23 the second software release will incorporate pattern-driven smart behaviour
- On M28 the third release will deliver the SEMIoTICS end-to-end architecture implementation
- On M32 the final stable release will be delivered

2.3 SEMIoTICS development workflow

SEMIoTICS has adopted the Git Distributed Version Control System (DVCS) for source code and asset management, as well as for monitoring the development process. We rely on a hosted solution from GitLab which will host the central SEMIoTICS repo located at gitlab.com. We will refer to this repo as the *origin*, which is the standard Git terminology, and all SEMIoTICS partners will have permissions to push and pull changes. Alternatively, developers can directly pull changes from other peers to form sub-teams, e.g., to collaboratively work on a new feature which will then be pushed to the to the origin repo.

2.3.1 SEMIOTICS GIT BRANCHES

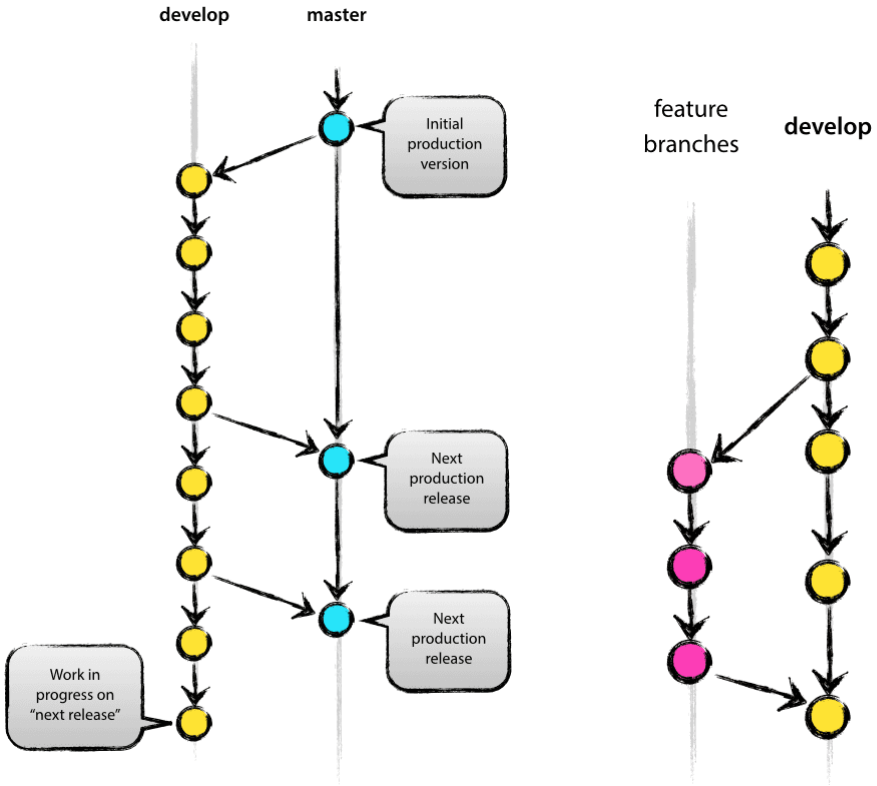

FIGURE 3: SEMIOTICS GIT REPOSITORY BRANCHES

The central SEMIoTICS repository will hold two main branches, the *master* branch, and the *develop* branch. The master is generally considered to be the main branch, that reflects the latest stable software release. The master branch integrates all delivered development changes for the next release, so it can also be considered to be the "integration branch". When the source code in the develop branch reaches a stable point and is ready to be released, all the changes should be merged back into master and then tagged with a release number.

In addition to the main branches (i.e., master and develop) *Feature* branches may be used to develop new features for the upcoming or a future release. Feature branches generally exist as long as a new feature is in development and will eventually be merged back into the develop branch, to ultimately add the new feature to an upcoming release, or even discarded in case of an experiment that led to a dead-end. Feature branches are also created in the origin repo, so multiple developers can push to the same feature branch. Multiple feature branches may exist at a time

2.3.2 CONTINUOUS INTEGRATION PIPELINE

A CI/CD pipeline is also part of GitLab, in the form of a web application with an API that stores its state in a database. It manages the project builds and provides a Graphical User Interface (GUI) which gives an easy to understand overview of the project development process. Most importantly, the CI pipeline is closely integrated with the core features of GitLab. The Gitlab CI pipeline will be part of the SEMIoTICS testing framework and will include all required unit tests and integration tests. Tests can be authored by the respective developers, or a separate testing team. Only if tests pass, then new code is committed to the source code repository. Furthermore, the system performs nightly builds and in case of build failure notifies the responsible developers to fix the issue. The SEMIoTICS Continuous Integration processes will also include the following, which may be accomplished via the GitLab system, or additional tools:

- A ticketing system to assign tasks and feature requests to partners
- A task planning system to assign features to future releases
- Team collaboration tools (e.g., Messaging, File sharing, etc.)

3 SOFTWARE-DEFINED INTEGRATION OF IOT/IIOT DEVICES

This section describes the reference points between the NFV building blocks as well as the interfaces that the NFV exposes to interact with the Middleware, and indirectly with the underlying IoT/IIoT devices. Herein, the ETSI NFV architectural framework (ETSI, 2014a) is considered as a reference. Furthermore, it details the components of the SEMIoTICS SDN controller, which will be responsible for the network integration and orchestration.

3.1 NFV MANO framework

In legacy networks Network Functions (NF), or Physical Network Functions (PNFs) are strictly related to the hardware they operate on. That is, switching, routing, firewalls and other kind of NF are provided by specialized hardware that contains the appropriate compute, storage and network capabilities each NF uses. Network Function Virtualisation (NFV) decouples NF from hardware, realizing one or many NF as software on top of commercial-off-the-shelf (COTS) devices with sufficient compute, storage and network resources. The move towards NFV promises to provide the dynamicity required to satisfy heterogenous application requirements, but also to take the most advantage out of the infrastructure by satisfying each application's constraint on top of a single, shared hardware infrastructure.

3.1.1 INTRODUCTION

The introduction of Virtual Network Functions (VNF) is strongly dependent on Software Defined Network (SDN) technologies, which in a similar manner have also achieved the decoupling of functionality from dedicated hardware by ways of separating the control and data planes. SDN is a necessary tool in NFV, mainly for realizing the interconnection of several VNF via virtual network overlays on top of a physical infrastructure. By leveraging SDN and NFV it is possible to interconnect blocks of functionality, i.e. VNFs or PNF, into chains tailored to provide a given Network Service (NS)[2], e.g. enforce security while accessing a Data Base (DB), placing embedded intelligence closer to the sensor/actuator, among others. Such NS are the result of VNF Forwarding Graphs (VNF-FG), that when coupled with Virtual Tenant Networks (VTN) allow NFV to support many NS to applications with heterogeneous requirements, effectively reducing OPEX/CAPEX relative to legacy networks.

The creation, instantiation, updating, and termination of NS is a new concept in networking, requiring the definition of new reference points (interfaces), functionality and entities. Moreover, the management of existing physical resources for virtualization, assignment of virtual resources to VNFs, lifecycle management of each VNF, and the realization of NS across a distributed set of physical resources impose new challenges to traditional networking. Efforts towards standardization in this regard have yielded ETSI's NFV Infrastructure (NFVI), which include the Virtualized Infrastructure Manager (VIM) and the NFV Orchestrator in the so-called Management and Orchestration (MANO) Framework.

The aforementioned components of the NFVI are to be described in this section, as well as the interaction among them to orchestrate NS and the role they play within the SEMIoTICS framework.

3.1.2 VIRTUALIZED INFRASTRUCTURE MANAGER

NFVI defines two Administrative Domains (ETSI, 2014b) namely the Infrastructure and Tenant domains. The former contemplates the physical infrastructure upon which virtualization is performed, and therefore application agnostic; while the latter makes use of virtualized resources to spawn VNFs and create NS. Unlike resource allocation in other virtualized environments, in NFVI requests simultaneously ask for compute, storage and network resources. Moreover, NS could be composed of VNFs with hardware affinity/anti-affinity or require specific latency/bandwidth constrains in virtual links connecting VNFs. Such demands occur dynamically, allocating or freeing resources that could then be used for other NS, e.g. scaling up VNF's compute.

[2] NS could also be composed of a single VNF.

A Virtualized Infrastructure Manager (VIM) lies in the Infrastructure Domain. It takes care of abstracting the physical resources of the NFVI and making them available as virtual resources for VNFs. This is achieved through the reference point ***Nf-Vi***, which interconnects the VIM and NFVI (see Figure 4). It allows the VIM to acknowledge the physical infrastructure (compute, storage) as well as enabling communication with network controllers (SDN Controllers) to provide virtual network resources to NS. Even-though VIMs could well control all resources of the NFVI (compute, storage and network), they could also be specialized in handling only a certain type of NFVI resource (e.g. compute-only, storage-only, network-only) (ETSI, 2014b).

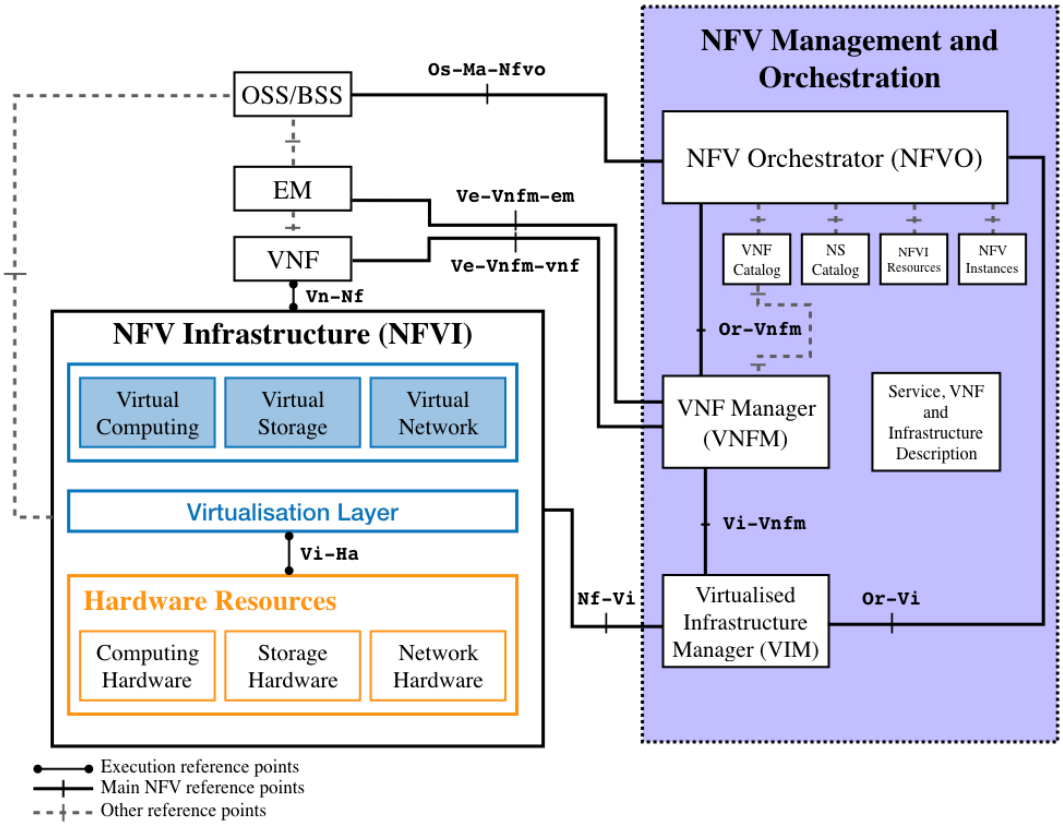

FIGURE 4 NFV REFERENCE ARCHITECTURAL FRAMEWORK

Beyond the already-mentioned functions carried on by the VIM are the following:

- Orchestrate requests made to the NFVI from higher layers (NFVO), e.g. allocation/update/release/reclamation of resources.
- Keep an inventory of allocated virtual resources to physical resources.
- Ensure network/traffic control by maintaining virtual network assets, e.g. virtual links, networks, subnets, ports.
- Management of VNF-FG by guaranteeing their compute, storage and network requirements.
- Management and reporting of virtualized resources utilization, capacity, and density (e.g. virtualized to physical resources ratio).
- Management of software resources (such as hypervisors and images), as well as discovery of capabilities of such resources.

As detailed in (ETSI, 2014b) other relevant VIM responsibilities within the NFVI network are:

- Provide “Network as a Service” northbound interface to the NFVO (realized via the ***Or-Vi*** reference point, see Figure 4).
- Abstract the various southbound interfaces (SBI) and network overlays mechanisms exposed by the NFVI network.

- Invoke SBI mechanisms of the underlying NFVI network.
- Establish connectivity by directly configuring forwarding instructions to network VNFs (e.g. vSwitches), or other VNFs not in the domain of an external network controller.

These compose the network controller part of the VIM. Nevertheless, and as mentioned previously, the required network abstractions mechanisms and management may as well be left to an external network controller, which feeds of NFVI information via the defined reference points (***Nf-Vi***, see Figure 4). It is reasonable to assume the VIM as key part of the NFVI. Being the only NFV component interfacing with the physical infrastructure it exposes open and comprehensible APIs to higher layers, i.e. NFVO, so functions could trigger them to get relevant information from the physical as well as the virtualized infrastructure, and trigger actions upon such information, e.g. create a NS with the necessary resources.

In the SEMIoTICS framework, the physical NFVI is able to support virtualization as realised by the VIM. This allows the NFVO to instantiate VNFs subject to the available compute and storage resources, as well as interconnect such VNFs together via an external SEMIoTICS SDN controller. The following subsections describe relevant Northbound Interfaces (NBI) or APIs usually exposed by VIMs, i.e. OpenStack, which are used by the Resource Orchestration function in the NFVO in order to create the NS satisfying the requirements of the SEMIoTICS use cases (UC).

3.1.2.1 COMPUTE

Compute services at the VIM not only are in charge of creating virtual servers (or containers) on top of physical machines, but also to provision bare metal nodes. In the case of OpenStack this is achieved by means of projects such as Ironic (OpenStack 2018a). The compute API for OpenStack is provided through the project Nova (OpenStack 2018b). It provides "*scalable, on demand, self-service access to compute resources*" through RESTful HTTP endpoints that can be triggered by any authorized entity. All content sent or received from the Compute API endpoints are in JavaScript Object Notation (JSON) format. As it is a text-based type, it allows developers to employ a wide range of tools in order to reach such APIs, easing automation.

The following is a non-exhaustive list of concepts related to the Compute service as well as the information they provide or actions they are able to execute through the corresponding API for SEMIoTICS UC (OpenStack 2018b):

- **Hosts**: physical machines that provide enough resources to spawn a Server. In SEMIoTICS, hosts conform the set of field level, network, and backend devices that together compose the NFVI. For instance, field level devices are assumed to provide enough compute resources to host VNFs realising local smart behaviour. Similarly, network level devices support VNFs for forwarding/routing/firewalling data to and from upper layers; and finally, backend/cloud servers have enough resources to host a wide variety of VNFs, e.g.: SCADA, Web applications and servers.
- **Server**: a virtual machine (VM) instance. In NFV it is often assumed that VNFs reside inside VMs or other type of virtualization container, such as LXC (Canonical, 2018). Some of the server status and actions reachable through the Compute API (OpenStack 2018c):
    - Status: ACTIVE, BUILD, DELETED, ERROR, SHUTOFF, SUSPENDED, among others.
    - Actions: Start/Stop, Reboot, Resize, Pause/Unpause, Suspend/Resume, Snapshot, Delete/Restore, Migrate/Live Migrate, among others.
        - Migration and live migration relate to moving the Server to another Host. Live Migration performs this action without powering off the Server, avoiding downtime.

  The ability to read the current status of Server and modify it, opens the way for dynamic (re)allocation of resources, specifically relevant as performance metrics from the underlying NFVI change in time. For SEMIoTICS this is of paramount importance, as it paves the way to optimize the end-to-end performance of network services in terms of e.g. latency or reliability.
- **Hypervisor**: the piece of computer software that creates and runs VMs. Hosts in each layer of the SEMIoTICS framework run a Hypervisor, which can be queried via the Compute API in order to obtain information regarding the Server, e.g. CPU, memory or other configuration.

- **Flavour**: virtual hardware configuration requested for a given Server, i.e. disk space, memory, vCPUs. Such configurations are onboarded prior to deployment, quantising the scaling factor of Servers e.g.: flavour small (1 vCPU), flavour medium (2 vCPUs), flavour big (4 vCPUs).
- **Image**: a collection of files used to create a Server, i.e. OS images. For SEMIoTICS, each UC component is assumed to run a preconfigured image tailored to its role, i.e. VNF. Such images are uploaded to the VIM for instantiation.
- **Volume**: a block storage device the Compute service could use as a permanent storage for a given Server.
- **Quotas and Limits**: upper bound on the resources a tenant could consume for the creation of Servers. SEMIoTICS employs such functionality to enforce an efficient sharing of the NFVI resources among the different UC.
- **Availability zones**: a grouping of host machines that can be used to control where a new server is created. As different SEMIoTICS UC require the placement of Servers at specific Hosts, this VIM capability allows the NFVO to instantiate VNFs at precisely the right locations in the NFVI.

3.1.2.2 NETWORKING

VIMs are responsible for building virtual network overlays connecting VNFs, but also should expose or relay such information to other components. For instance, if an external network controller is assigned the task of managing connectivity between virtual endpoints, as in the case with the SEMIoTICS SDN Controller, the VIM should expose API endpoints where the necessary network information can be retrieved or modified. Furthermore, in the presence of a NFVO, Network as a Service (NaaS) APIs are expected.

OpenStack Neutron Networking (Denton, 2018) provides the virtual networking resources commonly expected in NFVI, such as L2/L3 networking, security, resource management, QoS, virtual private networks (VPN), virtual tenant networks (VTN), among others (OpenStack, 2018d). To configure such functionality or to retrieve logging information, functions are exposed through a set of RESTful HTTP APIs in JSON format. The following shows a non-exhaustive list providing a description of the functionality exposed through the Networking API (as shown in (OpenStack, 2018d).

- **L2 Networking**
  - Networks: list, shows details for, creates, updates and deletes networks. It provides a wide range of extensions capable of configuring several aspects of L2 networking, such as: network availability zones, port security, definition of QoS policies, VLAN trunks, among others.
  - Ports: list, shows details for, creates, updates and deletes ports. Ports are associated with Servers (VMs). They expose a similar set of extensions than the “Networks” mentioned above.
- **L3 Networking**
  - Addresses: list, shows details for, updates and deletes address scopes. Deals with the reservation of IPv4 addresses for Servers (Floating IPs), port forwarding, among others.
  - Routers: when enabled, it allows the forwarding of packets across internal subnets and applying NAT, so they can reach external networks through the appropriate gateway. Routers can be realized in a distributed manner (spanning all compute nodes of the NFVI) or using Router availability zones.
  - Subnets: lists, creates, shows details for, updates, and deletes subnet or subnet pools.
- **Security**
  - Firewall as a Service (FWaaS): applies firewall rules to ingoing or outgoing traffic, creates and manages an ordered collections of firewall rules.
  - Security groups: lists, creates, shows information for, updates and deletes security groups. Such groups are used to classify types of traffic, allowing or prohibiting certain kind of network traffic through a set of predefined, but also user-defined rules.
  - VPN as a Service (VPNaaS): enables tenants to extend their private networks across the public network infrastructure. Provided functionality includes:

        - Site-to-Site VPN.
        - IPSec using several types of encryption algorithms.
        - Tunnel or transport mode encapsulation.
        - Dead Peer Detection (DPD).
- **Others**
    - QoS bandwidth limiting rules.
        - With the ability to distinguish between egress or ingress traffic.
    - QoS Minimum bandwidth rules.
    - QoS Differentiated Service Code Point (DSCP).
    - Logging resources.
    - DHCP servers.

SEMIoTICS falls within the particular case where the delegation of NFVI networking control is relayed to an external SEMIoTICS SDN Controller. For such cases, Neutron exposes control tools via the Modular Layer 2 (ML2) north-bound plug-in (OpenDaylight 2018). This way, external controllers could manage the network flows traversing the NFVI via southbound interfaces, such as OVSDB.

3.1.2.3 STORAGE

Block storage is common place in virtual environments. Such type of storage can be though similar to USB drives: you can attach one to a compute Server (VM), and then detach it when turning the Server off or destroying it. Particularly interesting is the fact that in a NFVI the storage and compute Hosts are separate. Despite such separation of physical hardware, VMs are exposed to users as if they were running on top of a single Node thanks to the virtual networking resources used by the VIM; allowing the NFVI to grow to massive scales, e.g. server farms.

VIMs such as OpenStack manage block storage through the Cinder project. As concisely put in (OpenStack, 2018e) "*It virtualizes the management of block storage devices and provides end users with a self-service API to request and consume those resources without requiring any knowledge of where their storage is actually deployed or on what type of device*". A non-exhaustive list of functionalities realised through the Storage API is shown below:

- Create, list, update, or delete volumes.
- Read volumes statuses:
    - Among such statuses are: creating, available, reserved, attaching, detaching, in-use, maintenance, deleting, error, backing-up, among others[1].
- Modify a volume:
    - Extend size, reset statuses, set metadata, attach/detach.
- Management of volumes: create or list volumes.
- Volume snapshots: creates point-in-time copies of the data a volume may contain.
- Volume transfer: transfer a volume from one user to another.
- Backups: full copy of a volume to an external service, as well as the restoration from such backup.
- Snapshots and Group Snapshots.
- Quotas and Limits: per tenant quotas and limits on storage resource allocation.

Compute, Networking and Storage resources are then allocated by the VIM according to requests made through the corresponding APIs. SEMIoTICS UC can be seen as NS, which in turn are the composition of a set of VNF that run within VMs with specific compute and storage resources that are connected in a predefined manner with network resources (SEMIoTICS SDN Controller) known to the VIM. Thereby, the proper allocation of computing, communication and storage resources, to run the chain of VNFs at the corresponding VMs, is fundamental to guarantee the desired performance of SEMIoTICS use cases. Namely, these performance metrics are related to latency or reliability.

All in all, SEMIoTICS UC can be considered complex NS, mostly due to their specific requirements, e.g. Host affinity/anti-affinity (e.g. smart behaviour VNFs at specific IoT gateways), specific bandwidth/delay

requirements between VNF links, firewalls at the backend/cloud, and/or others. Such specifications are collected in NS descriptors (NSD), which in turn are composed of VNF descriptors (VNFD), and VNF-FG descriptors (VNFFGD) that realize Service Function Chains (SFC) according to the specifications contained in their respective descriptors. It is then the task of the NFVO to store/maintain such descriptors and interface with the VIM to realise the NS/VNF/VNF-FG therein.

3.2 SDN based integration and orchestration

SEMIoTICS SDN Controller is responsible for orchestration of field- and network-level switching devices. We assume an OpenFlow model where SDN Controller computes the network paths used to deploy the forwarding rules for both QoS-constrained and best-effort traffic. The SDN controller does so by parsing the end-points and the service flow requirements (e.g., on bandwidth, delay, fault-tolerance/availability) from the content of pattern specification message provided by the network administrator or higher-layer orchestration element (i.e., the Pattern Orchestrator in the SEMIoTICS architecture).

Figure 5 below depicts the SDN Controller's architecture components as defined in T3.1.

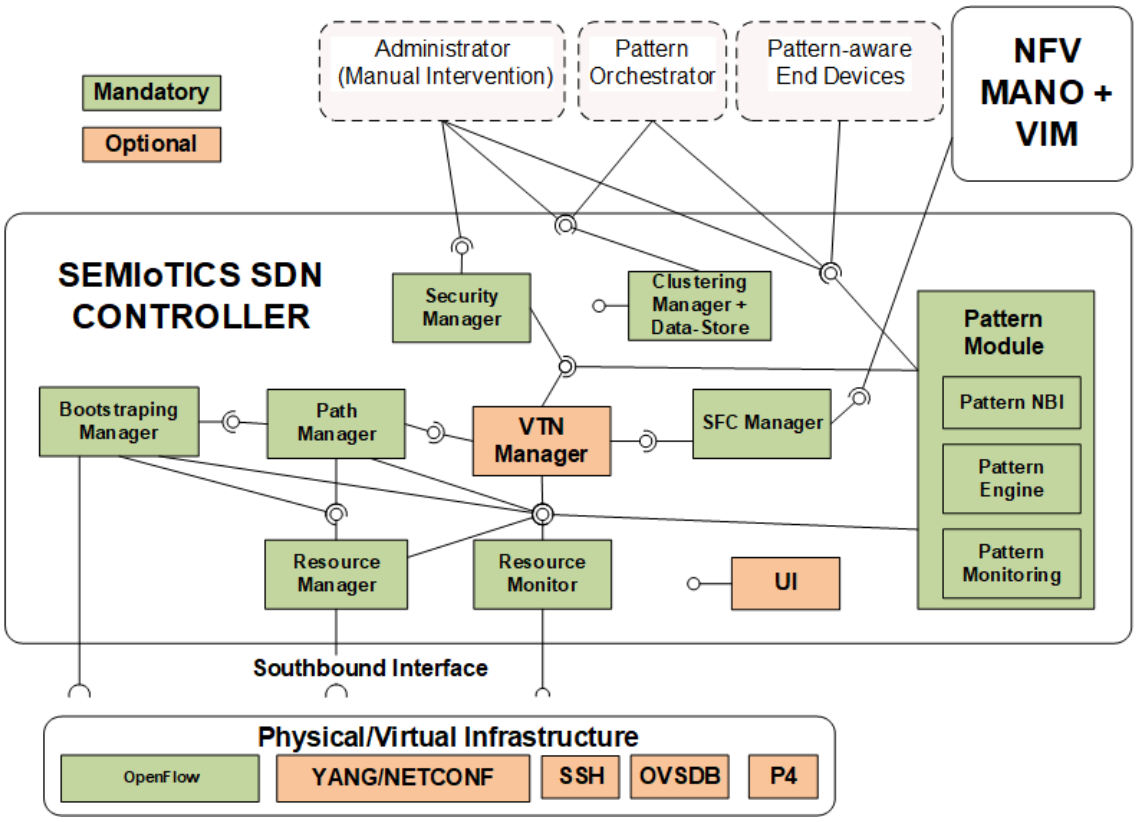

FIGURE 5: COMPONENT OVERVIEW OF THE SEMIOTICS SDN CONTROLLER DETAILED IN D3.1

The initial release of SEMIoTICS SDN Controller is expected to include the following controller components:

- **Pattern Module:** The interconnection point to the Pattern Orchestrator / System Administrator. The Pattern Module is responsible for:
  a) exposing the pattern-specification northbound interface;
  b) the corresponding pattern enforcement using a set of internal controller APIs (i.e., through interactions with other controller components);
  c) state-keeping of specified network patterns; and
  d) monitoring of embedded patterns at runtime. It is composed of respective sub-components: Pattern NBI, Pattern Engine, Pattern Registry and Pattern Monitoring.

- **Virtual Tenant Network (VTN) Manager:** Responsible for assignment of individual network services to various network tenants. It further ensures a separation of L2 traffic (i.e., ARP request broadcast propagation to ports assigned) in scope of a virtual tenant network.
- **Path Manager:** Main network path computation engine of the SDN Controller, responsible for identification of nodes and ports combined into a path that fulfills the pattern requirements (i.e., on fault-tolerance or bandwidth/delay constraints).
- **Resource Manager:** Provides Path Manager with a resource view of the network (i.e., the available topology resources, port speed, no. of queues metrics etc.). We aim to expose the metrics observable using the standardized OpenFlow 1.3 interface.
- **Security Manager:** The security component of the controller responsible for administration of tenants and assignment of applications with respective tokens used for fast authentication during runtime.
- **Service Function Chaining (SFC) Manager:** Used in enforcement of Service Function Chains given the ordering and IP addresses of nodes that are to be traversed by a tenant's traffic.
- **Registry Handler (a component of the Clustering Manager):** Used in state-keeping of other component's knowledge base, as well as for its strong consistent replication across the SDN controller instances for the purpose of fault-tolerance and high-availability. Aspects of ensuring Byzantine Fault Tolerant operation for control of highly dependable industrial networks will be investigated in scope of this module as well.
- **Management and Orchestration (MANO) extensions:** Required for any future interactions with NFV-related orchestration functions, i.e., OpenStack or Kubernetes.
- **Bootstrapping Manager:** Used in initial flow configuration of just-connected switches, so to allow for seamless interaction with IoT devices (i.e., to enable flow rules for propagation of unmatched application packets up to the controller for the purposes of ARP-based end-device discovery, MAC Learning for best-effort services or similar).

The interactions between the components of the SDN controller, developed to support the initial release cycle workflow will be described in Deliverable 2.1.

The SEMIoTICS controller will be implemented on top of the existing SDN controller OpenDaylight. The OpenDaylight controller, and especially the revised OpenDaylight controller published as a result of VirtuWind project provides numerous built-in components that will be reused in the initial implementation. For example, we plan to make heavy use of OpenDaylight's OpenFlow implementation (*OpenFlowPlugin*) to enable interaction with the OpenFlow enabled SDN switches, but also utilize and extend its data-store implementation and data models for storage of the stateful controller information.

OpenDaylight ecosystem uses the Apache Maven build tool for automated compilation and deployment of the developed source code. Maven is a software project management tool, which allows developers to define the project's build lifecycle, phases, goals, dependencies, build plug-ins and profiles, in order to provide a variety of builds for the project. The OpenDaylight project utilizes Maven as its only build tool – hence we did not want to deviate from a proven setup in our project either. Additionally, the OpenDaylight project exposes its pre-compiled components as Maven artifacts (executables) in the Maven Central Repository, which is accessible by default using the Maven dependency plugin.

The SEMIoTICS-specific controller extensions should be developed so to promote modular and extensible controller platform design. Our controller will be executed inside a Java Virtual Machine (JVM), and thus deployable on any Java–supported systems. The modular architecture of OpenDaylight adheres to OSGi specification, which aids to components' lifecycle management enabling dynamic state at runtime. For example, the components can be loaded and unloaded, updated, started or stopped without influencing other running services. In OSGi, a component is called bundle and is realized as an executable Java Archive (JAR) which contains component metadata such as bundle name, activators, version, as well as information about exported packages and required imports which are references to other component packages. We will use the open-source software **Apache Karaf** as our OSGi execution framework. The role of Apache Karaf is to launch a specified set of OSGi bundles (implementation of our components), as well as to provide a control and management interface for OSGi bundles during runtime.

4 SEMANTIC BOOTSTRAPPING AND INTEROPERABILITY

In this section, standardized semantic models for IIoT applications and SPDI pattern-driven mechanisms that guarantee network-level semantic interoperability are detailed, that are developed in T3.3 and T3.4 respectively. These mechanisms form the basis of the semantic bootstrapping and Interoperability framework, i.e., a significant part of the SEMIoTICS field-level middleware.

4.1 Semantic bootstrapping and interoperability framework

The SEMIoTICS Deliverable D3.3 provides standardized semantic models for IIoT applications, that will form the basis of the semantic bootstrapping and interoperability framework. These models harmonize data models from existing automation systems and integrate them with standard IIoT information models. D3.3 provides semantics that aims to make field devices interoperable with new IoT devices. Second, it helps to expose capabilities of field devices in a uniform manner by an IIoT gateway. Semantics at this level is thus a key enabler for bootstrapping and easier integration of devices in an IIoT system, as well as a facilitator for creation of new applications. Current automation systems are fully integrated vertical systems. They are efficient, but inflexible. Once engineered and operational, they cannot be changed easily. For example, it is not straightforward to plug a new device into a running system and expect to be functional with respect to an already engineered system. Or it is not effortless to develop an added value service for an existing automation system. In both cases the reason is a know-how contained by experts, but not explicitly represented in machine-interpretable form.

In order to enable creation of new IIoT applications we need to explicitly represent this knowledge, thereby expressing capabilities of field devices in machine-interpretable form. The following use case describes problems found in the current vertically integrated automation systems and sketches the role of semantics in IIoT in order to amend these problems.

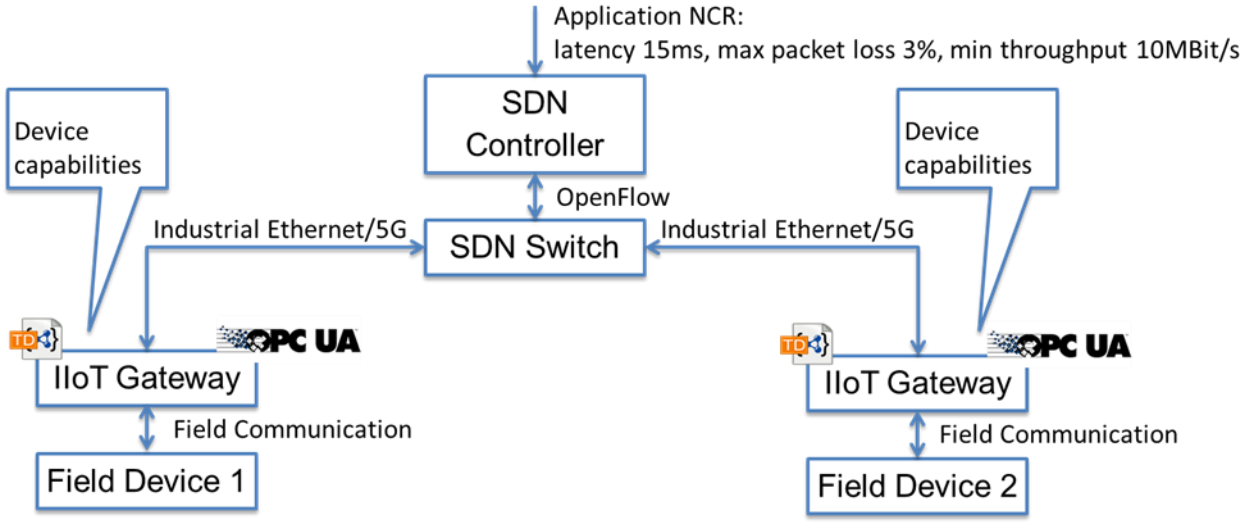

FIGURE 6: SEMANTIC-BASED ENGINEERING AND NETWORKING

Figure 6 depicts an example industrial application, which processes data from Field Device 1 and Field Device 2. In addition, the application imposes certain QoS requirements, which are here expressed as a network constraint rule (NCR). Based on this example application we will explain the role of semantics for interfacing SEMIoTICS field level devices (as the scope of Deliverable 3.3). Let us suppose that Field Device 1 and Field Device 2 are heterogeneous in terms of protocol they communicate, and data they exchange. In order to enable an application to process data from these two devices, we first need to enable a common application protocol. Second, we need to provide a common data model. Finally, we need to provide a common semantic model, which will describe interaction patterns and capabilities of device. Only then, it will be possible for an application developer to discover field devices based on their capabilities they

provide, and to put them into a semantically-correct interaction. Further, this enables the developer, as well as machines to understand the data that is produced or consumed by devices. It allows semantic validation of this data or automatically match-make the devices capabilities with the requirements of an application. All these features are useful when a new device is plugged into an existing IIoT system and needs to support an old or a new application, or a malfunctioning device needs to be replaced with the new device etc.
There are two approaches that are promising. The first is based on W3C Web of Things Thing Description[3]. The second is based on a prominent industrial standard OPC-UA[4]. In the scope of the first version of Deliverable 3.3 our focus will be on the first approach.

In general, the mission of W3C Web of Thing (WoT) is to counter the fragmentation of the IoT. That is, device from different ecosystems become interoperable under a common application layer, provided by WoTs. This should be achieved similar to Web, which has provided a unified application layer to Internet. To this end, W3C WoT standardization group has identified four building blocks.

- First, the Thing Description (TD) describes the metadata and interfaces of Things, where a Thing is an abstraction of a physical or virtual entity.
- Second, the accompanying Protocol Binding Templates[5] enable a TD to be adapted to the specific protocol usage across the different standards.
- Third, the Scripting API[6] describes a programming interface representing the WoT Interface that allows scripts run on a Thing. These scripts can be used to discover and consume other Things (via their TDs), and to expose Things characterized by their capabilities (WoT Interaction Patterns).
- Finally, Security and Privacy Considerations[7] is the fourth building block, which provides guidance for the design and deployment of a secure WoT system.

In the scope of the work in SEMIoTICS, we will focus on the first building block. But the second and the third building blocks will be used in our implementation, too.

The WoT TD can be considered as the “index.html” page for Things. It contains semantic metadata describing the Thing itself (e.g. name, location, application context, and software and hardware versions); the offered interface in the form of interaction patterns (i.e., Properties, Actions, and Events); the data model used in messages; and relations to other Things expressed through annotated Web Links [RFC8288]. In the following, we provide a short description of TD basic interaction patterns.

TD Properties expose internal state of a Thing that can be directly read or (optionally) written. Typical examples of Properties are configuration parameters, sensor readings, and set-points that control actuators through Thing-internal logic (e.g., a set-point for the temperature of a thermostat). TD Properties may also be observable. In this case they push the new state to registered subscribers, following best effort mechanisms (e.g. CoAP Observe).

TD Actions enable invocation of Thing’s functions. These functions manipulate the internal state of Thing in a way different from setting Properties. Examples are changing internal state that is hidden, i.e., not exposed as a Property; changing multiple Properties with a single Action; or changing long-running processes (i.e., time is needed to complete the process, and a Property can be used to check the process, e.g., check the state or cancel it during the execution). Actions interaction pattern can also be used to abstract RPC-like calls of existing platforms.

TD Events are raised in order to notify state changes, alarms or streams of values that are sent asynchronously to the subscriber. Unlike Properties, which can be called, TD Events are pushed to subscribers. Events may be triggered as result of conditioned state changes in a Thing. Events are different from observable Properties in that their data cannot be accessed at any time, but only when a notification is emitted by the Thing.

The TD with its presented interaction model is typically enriched with external semantic models (ontologies). TD imports additional Linked Data vocabularies in order to give semantic meaning to its constructs. For example, a TD may have a Property. In order to specify what is the type of that Property, what data it produces, in which range the data is, what is the measurement unit, what Thing’s capability this

[3] https://w3c.github.io/wot-thing-description/
[4] https://opcfoundation.org/about/opc-technologies/opc-ua/
[5] https://www.w3.org/TR/wot-binding-templates/
[6] https://www.w3.org/TR/wot-scripting-api/
[7] https://www.w3.org/TR/wot-security/

Property belongs, and so forth, we use external semantic models. A common semantic model to be used with TD is iot.schema.org.
iot.schema.org is an extension of well-known schema.org that is used to annotate Web pages. iot.schema.org provides similar concept for annotations of IoT Things. iot.schema.org features three levels for semantic annotations: Capabilities, Interactions, and Data. A Capability represents a Thing's trait. It usually consists of a set of Interactions. Interactions are semantically aligned to Interaction Patterns from W3C WoT TD. Finally, Data specifies all information about the data that a Thing provides or consumes via its Interactions.

4.2 Network Level Semantic Interoperability framework

The SEMIoTICS framework will facilitate the deployment of network services and provide seamless connectivity with all its layers and IoT applications, as is the aim of Task 3.4. To achieve that, the project will employ SPDI pattern-driven mechanisms that guarantee network level semantic interoperability for various components of SEMIoTICS. Specifically the following considerations are made:

1. Regarding the interfacing of IT & Cloud infrastructures, to support Nf-Vi, Os-Ma-Nfvo and interfaces for NS management, the NFV reference architectural framework along with the Nf-Vi, and Os-Ma-Nfvo points;
2. Regarding the IoT Platforms, to support Publish/Subscribe Context Broker, Context Producer and Context Consumer by defining and ensuring communication between them, via a different platform (i.e. FIWARE);
3. Regarding the network level of SEMIoTICS itself, to support the different needs of the 3 major use cases such as the IIoT wind park scenario.
4. Finally, regarding IoT applications, to support flows between multiple IoT applications, distributed on multiple devices (e.g. between applications of a wind turbine)

Additional considerations must also be identified and guaranteed by the pattern engine to facilitate complex interactions, of the above components such as:

- **Cross-Platform:** This covers applications or services access resources from multiple platforms though common interfaces. Further, it includes different instances of SEMIoTICS platform and/or SEMIoTICS to 3rd party IoT platforms (e.g. FIWARE, MindSphere), enabling an application deployed on one platform (e.g., an IIoT wind turbine status monitoring application aggregating information from pertinent sensors) to collect data from other platforms that process related data.

- **Cross-Layer:** This includes communication between entities that are deployed at different-non-adjacent layers of the SEMIoTICS framework, such as cloud to edge or application to network.

- **Cross-Application:** This includes communication between applications or services with applications of different domains or verticals. Such a communication means that an application could potentially gather data about environmental conditions and traffic, to propose the least polluted routes to patients with breathing issues.

- **Higher-level services:** These services, are enabled by exposed interfaces, to orchestrate existing deployments, applications, and the associated services, to provide value-added services, such as providing wind turbine failure predictions or energy demand predictions (to fine-tune energy output) from data aggregated across associated services, enabling effective predictions even for stakeholders/deployments that do not have the breadth of historical data or computational capabilities to extract this knowledge. (e.g. provide specific services to third party entities).

To support the above, two basic properties have been ensured across the deployment that also affected the design of then networking interfaces:

- **Platform-scale independence**, allowing the integration of resources from platforms at different scale. More specifically: at the Cloud/IoT backend level, platforms can host high volumes of data from a vast number of devices; field-level deployments (e.g., fog) interact with nearby devices in the

field and maintain information in a constraint spatial scope; device level platforms (e.g. at the IoT gateway level) have direct communication with the things, managing small amounts of data. In this context, in the SEMIoTICS framework an application should be able to uniformly aggregate information for the different scale platforms (e.g. collect wind turbine status values for a specific area via cloud or minimally processed data via a platform at field).

- **Platform independence**, allowing the integration of distinct platforms that implement the same functionality, like an IIoT wind turbine status monitoring in different wind parks. The platforms may utilize different equipment and techniques to monitor the wind turbines (e.g. legacy wired sensors attached to smart gateway or newer wireless sensors); a single application at the backend should be able to interface with all instances in a uniform manner without requiring any changes.

The vision of such a heterogeneous and flexible deployment is sketched in Figure 7, while driven by the above, in the subsections below more specific requirements are investigated, focusing on particular layers and types of interactions.

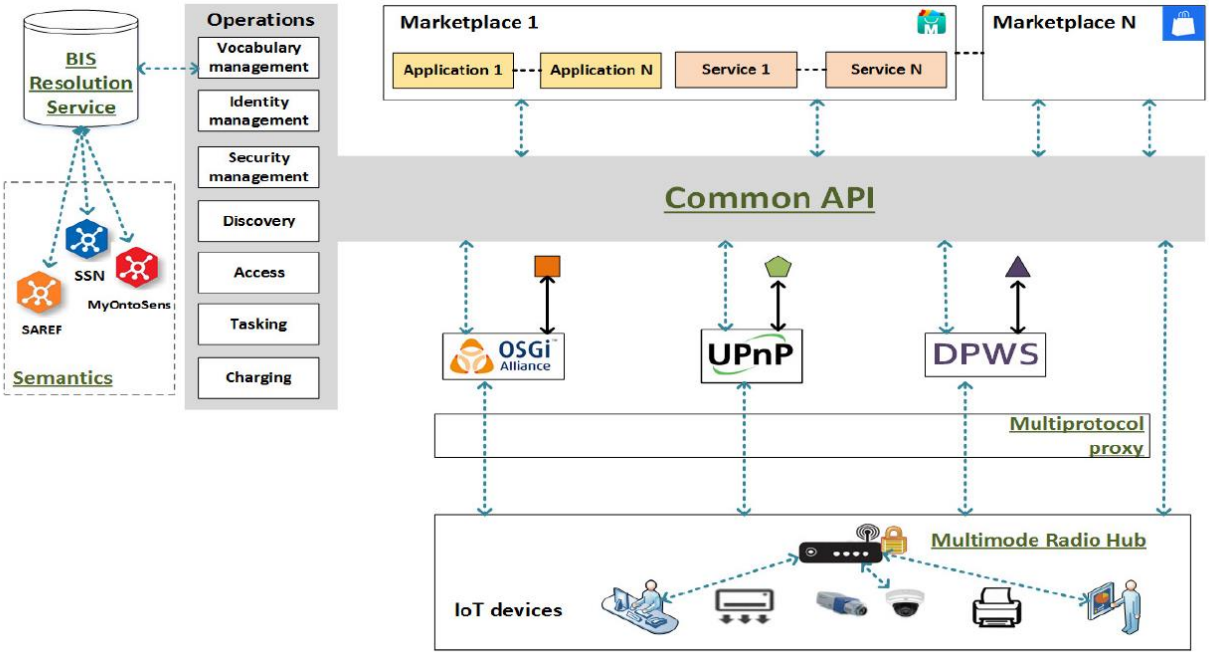

FIGURE 7: SEMIOTICS FRAMEWORK, INTEROPERABILITY ACROSS ALL 4 LEVELS

The foretold considerations that are described, in detail in D3.4, are translated to pattern language requirements and based on them the pattern language is created.

Regarding the **IT & Cloud infrastructures,** the pattern language will define and provide via the pattern engine the essential connection between the NFV Management and Orchestration components and the NFV Infrastructure. Additionally, the pattern engine should facilitate the communication with the North Bound Interface via the Os-Ma-Nfvo endpoint.

Considering the **IoT Platforms,** Network interoperability with other platforms e.g. FIWARE must be considered in the designing phase of the pattern engine as it is an essential part of SEMIoTICS. The pattern language will define and ensure via the pattern engine that SEMIoTICS can utilize certain components of FIWARE such as Orion Context Broker and provide the necessary communication bridges between them, Context Producers (e.g. a sensor) should be able to interact with Context Consumers (e.g. a context-based application) via a FIWARE based Context Broker, Orion; finally, the pattern engine should provide a proxy service in case in of failure of the said CB and be able to implement partially the respective NGSL10 API to support query/update operations from FIWARE to a context element in the SEMIoTICS domain.

With regard to **network-level interfacing with SEMIoTICS,** the pattern language will define and enforce (via the pattern engine) mechanisms that guarantee the establishment of E2E connectivity (e.g. by 5g

cellular network, Bluetooth BLE) between different types of devices (e.g. SARA hubs, sensors, backend servers), actors (e.g. human operators, applications) and interaction type (e.g. maintenance, medical staff, simple user/patient).

Additionally, the pattern engine will support various more complex interactions such as cross platform (e.g. cloud apps <-> private cloud), cross layer interactions (e.g. field devices <->backend), cross application (e.g. SDN controller <-> remote management service) or interactions with higher level services (e.g. Third-party entities). The interoperability pattern mechanisms will also need to ensure that all the devices support the required protocols (e.g. MQTT, HTTP, etc.) for bootstrapping, discovery and registration operations and that they fulfil these actions also in different layers of the framework. (e.g. cloud interfacing with the IoT sensing gateway). These will have to be achieved in a secure manner, that is enforced by security/privacy-driven patterns explored and defined in D4.1

Virtual network components' prerequisites (e.g. setup/configuration parameters, support tunneling) will also be considered and ensured by interoperability mechanisms defined via the pattern language. Additionally, physical network components' prerequisites (e.g. hardware specification) will also be examined and guaranteed that they meet the expectations by pattern-driven mechanisms. As to the **IoT applications,** the pattern language will define and guarantee via the pattern engine the communication between various IoT devices through their interfaces. Further, the interaction with edge devices will be assured. Finally, using pattern-based operations SEMIoTICS should translate high-level application QoS constraints to network-level QoS constraints.

Moreover, in the context of this task is the identification and description of key enabling technologies that further advance the interoperability of SEMIoTICS, such as network protocols and data formats that can be used for the communication from field devices to the backend cloud.

Utilized network protocols include:

- **Hypertext Transfer Protocol (HTTP) – Representational State Transfer (REST):**
  - HTTP/1.1. (i.e. the most commonly accepted version of this protocol) is the fundamental client-server protocol used for the Web.
  - REST is a distinguished architecture style used for developing of web services. With the rapid success of IoT the combination of HTTP & REST offers very easy ways to create, read, update and delete data, making it essential for SEMIoTICS.
- **Advanced Message Queuing Protocol (AMQP),** is an open standard protocol following the publish-subscribe paradigm, aimed to offer interoperability between a large diverse set of applications and systems, regardless of their internal designs.
- **Constrained Application Protocol (CoAP),** is designed by the Constrained RESTful Environments (CoRE) with recent versions using a like publish-subscribe approach, to provide HTTP REST capabilities for constrained devices with limited processing resources, such as IoT devices.
- **and Message Queuing Telemetry Transport (MQTT),** is another protocol that follows the publish-subscribe paradigm. It is especially efficient and lightweight, designed for constrained devices and non-optimal connectivity conditions, such as low bandwidth and high latency.

Employed data formats involve:

- **Extensible Markup Language (XML),** is markup language made for encoding data in a format that is both human-readable and machine-readable.
- **JavaScript Object Notation (JSON),** is a lightweight open-standard file format based on a portion of JavaScript, made to transmit data objects using human-readable text (that can be easily parsed and produced by a machine).
- **SensorThings,** is an Open Geospatial Consortium (OGC) standard that provides an open and unified framework used by IoT sensing devices, data, and applications to communicate over the Web.
- **Google Protocol Buffers** is a solution designed to serialize structured data in an automated, flexible and efficient way; it's like XML but better, in terms of size, speed and simplicity.

Finally, the network interoperability framework includes network services APIs based on existing standards, the definition of SPDI patterns for network-level semantic interoperability and their implementation, including the testing client, along with some initial examples and results.

5 SEMIOTICS TESTBED IMPLEMENTATION AND FIELD-LEVEL MIDDLEWARE VERIFICATION

5.1 SEMIoTICS SDN/NFV testbed

The main objective of SEMIoTICS project, and hence the demonstration platform, is to enable secure and dependable actuation and semi-autonomic behaviour in IoT and IIoT application scenarios. The SEMIoTICS platform used to demonstrate an end-to-end IIoT SDN/NFV architecture, complete with the local cloud, SDN networking and Field layers that demonstrate smart actuation, monitoring and analytics functionalities. A preliminary version of the SEMIoTICS testbed was demonstrated at the EUCNC 2018 exhibition (see Figure 8). The SEMIoTICS testbed will be employed to implement and verify the Field-Level middleware, which will then be leveraged by the SEMIoTICS use cases in production environments. The SEMIoTICS testbed currently includes the following hardware components and is constantly upgraded:

- One 4-core 64-bit server with 32 GB RAM acts as the Controller, and hosts all services related to Management, Orchestration and SDN control.
- Two 6-core 64-bit servers with 16 GB RAM act as the Compute Nodes, or hypervisors, that host all IIoT services and VNFs in dedicated Virtual Machines (VMs).
- Two Odroid C2 Single-Board Computers (SBCs) act as the Field layer Virtualized IoT gateways. An 802.15.4 radio module is employed to interconnect Field devices (smart sensors) with the gateway.
- Field layer smart sensors transmit temperature, humidity, and light intensity values wirelessly over 802.15.4.
- SDN access switches are employed in the Network layer, to interconnect the Compute Nodes and IIoT gateways.

FIGURE 8: IIOT TESTBED INFRASTRUCTURE, SHOWING THE CONTROLLER NODE, IOT GATEWAYS, SMART SENSOR AND ACTUATORS (SMART LIGHTS) AT THE EUCNC 2018 EXHIBITION

IIoT services **related to smart monitoring and actuation are** implemented in the form of VNFs that can be automatically deployed and orchestrated by the **cloud controller**. Currently, we have implemented and deployed one VNF for smart monitoring and one for actuation, each in a dedicated Tenant Network, that compete for resources. In what follows, the 3 individual layers of the IIoT testbed, i.e., Backend Cloud, Network and Field are presented in detail.

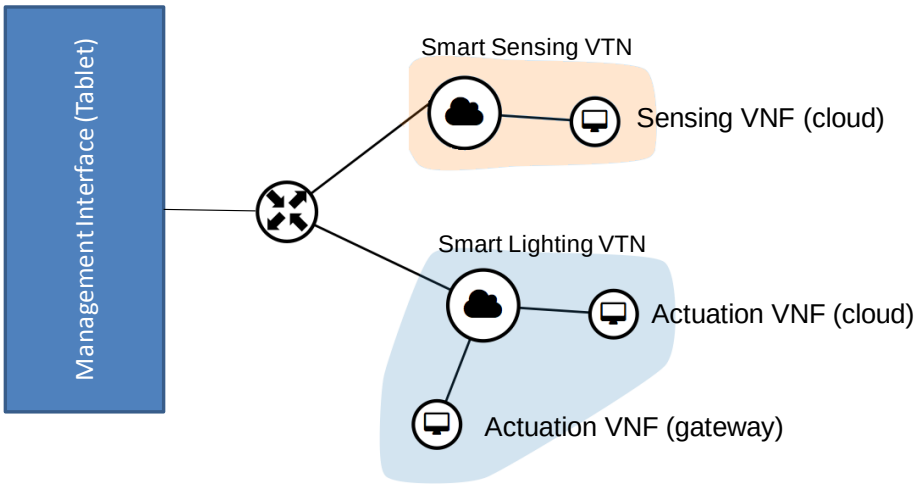

FIGURE 9: IIOT SERVICES AND TENANT NETWORKS

5.1.1 LOCAL CLOUD

The local cloud of our testbed is based on the OpenStack ecosystem, which is responsible for deploying VMs and managing their lifecycle. OpenStack is a complex software framework with multiple components that handle security and authentication, VM image storage, VM instantiation and termination, etc. In our testbed, a Controller node hosts all OpenStack services in Linux Containers. Linux Containers (LXD) is an emerging virtualization solution which allows services to run almost to the "bare metal" with minimal performance penalties, but with the requirement that they share the same kernel with the host (in this case the Controller node). The following OpenStack services are deployed in our Controller:

- **Glance** stores the VNF (or VM) images in its local filesystem
- **Keystone** acts as the identity service, keeping track of OpenStack users and their respective permissions (e.g., admin, user, etc.)
- **MySQL** stores configuration options in a master database
- **Neutron** is the OpenStack networking layer, which handles connectivity among VMs and applications. It is responsible for deploying end-to-end slices and virtual networks among VNFs that can physically reside in different physical servers
- **openstack-dashboard** implements the OpenStack Horizon GUI which allows us to manage our network and VMs with an easy to use GUI.
- **Tacker** serves as the VNF Manager, which handles the delivery of end-to-end network services. It supports the lifecycle management of network services, catalogue management and on-boarding/configuration of network services and VNFs.
- **Nova** is the OpenStack hypervisor service. OpenStack Nova employs KVM (i.e., Kernel-based Virtual Machine) technology to natively execute multiple VMs at a host operating system.
- **RabbitMQ-server** implements a fast message bus that allows individual OpenStack services to communicate and exchange information.

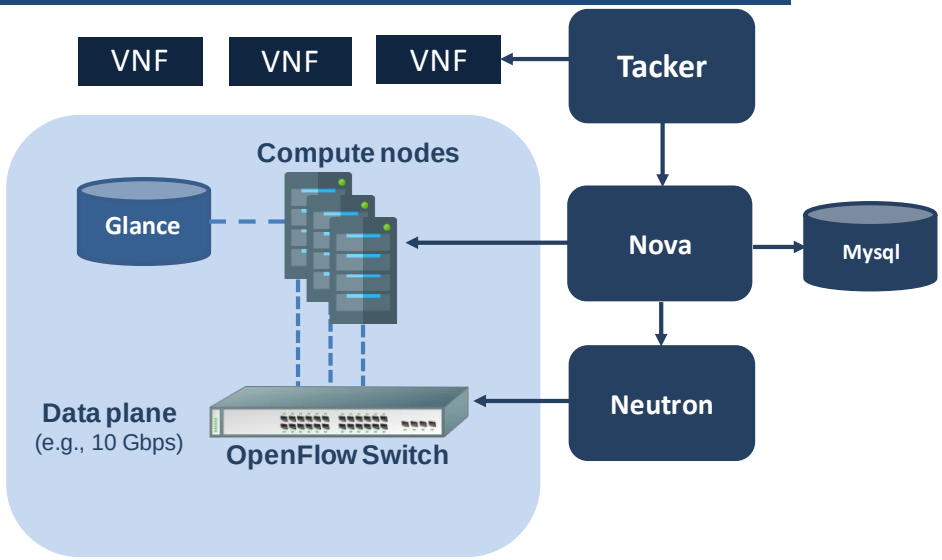

FIGURE 10: BACKEND CLOUD

All services in our NFV enabled Testbed are packaged in VNFs. VNFs are hosted in dedicated VMs that are placed in Compute Nodes (or hypervisors) by OpenStack Tacker, the VNF Manager. Compute Nodes are inter-connected by the data plane, which is implemented with SDN switches. VNF metadata are described by VNF Descriptors, or VNFDs. VNFDs define service behavioural and deployment information in a template file which is based on TOSCA standards and is written in YAML. This allows deployment and orchestration of services to be performed automatically by OpenStack Tacker, which serves as the platform VNF Manager. OpenStack Tacker implements a Resource Orchestrator which coordinates the allocation and setup of the computing, storage and network resources that are necessary for the instantiation and interconnection of VNFs. Moreover, it performs Resource Checks to ensure that the VNF requirements are met. This allows the automatic deployment and lifecycle management of services, without user interaction. Moreover, VNFs can be individually scaled, i.e., multiple instances can be deployed to meet user demand. Moreover, the VNF Manager can migrate VMs to a different hypervisor for optimization purposes. For example, to meet service KPIs a VNF may have to be moved to a hypervisor with a lower load. VNF migration is a relatively complex procedure and care should be taken not to cause downtime. Specifically, there are two modes of operation for VNF migration:

- Legacy mode involves shutting down and then restarting the VM that hosts the VNF in a different hypervisor.
- Live migration mode involves running both instances (in the old and new hypervisor) in parallel while the migration is performed, and only migrating RAM contents as a final step. This mode causes minimal service disruption.

5.1.2 FIELD LAYER

Our testbed Field layer includes a virtualized IIoT gateway that interconnects a set of sensors and actuators with the backend cloud. Our IoT gateway supports KVM virtualization, enabling us to push VNFs down to the gateway tier. This concept, also known as MEC, allows services with ultra-low latency requirements to be pushed to the edge, hence minimizing latency. The relatively modest resources available at the gateway, which is implemented with a 64-bit ARM-based Single-Board Computer, means that it must be used for a minimum number of VNFs with low processing needs.

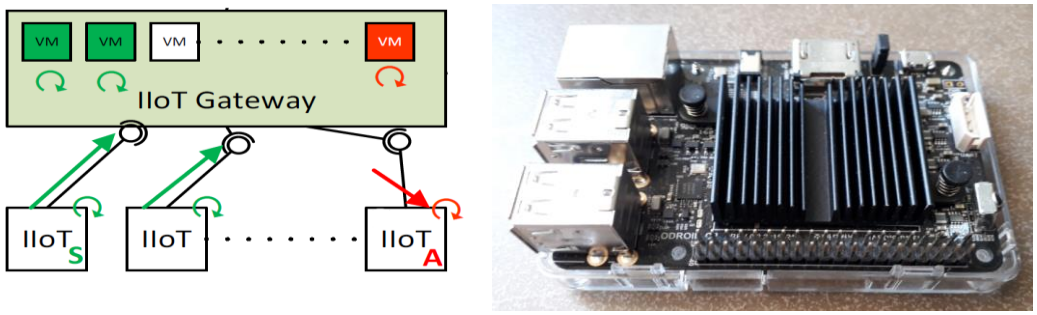

FIGURE 11: VIRTUALIZED IIOT GATEWAY

For the field-layer smart sensors, we employ custom-designed battery operated 802.15.4 and BLE devices that perform periodic measurement of CO2, Temperature and Light (Lux) values. Sensor values are encapsulated in IPv6 packets and transmitted to the IIoT gateway via MQTT. The actuators are commercial Philips Hue Smart Lights that are connected to the IIoT gateway via a Hue bridge. The Sensors and Actuators are communicating with the respective VNFs, that are hosted at the Cloud or IIoT gateway hypervisors.

5.2 Middleware implementation and verification

At this early stage of SEMIoTICS project, we are still at the design process, but nevertheless the implementation and verification of individual middleware modules has started. The slicing module, detailed in the following section, is responsible for reserving resources for critical applications (e.g., critical infrastructure monitoring) such that they are offered performance guarantees related to throughput, latency, and packet error rate. Slicing generally involves bandwidth reservation at the respective network interfaces of the application VTN.

5.2.1 SLICING FRAMEWORK IMPLEMENTATION

SEMIoTICS reference architecture includes SDN switches at its Network layer, that interconnect Field Layer IIoT gateways. SDN switches in SEMIoTICS will be implemented with Open vSwitch (OvS), a production quality, multilayer virtual switch licensed under the open source Apache 2.0 license The OvS switches are controlled by the Neutron controller via the OpenStack ML2 API which supports Open vSwitch out of the box. ML2 (Modular Layer 2) technology bundled with OpenStack supports a wide variety of Layer 2 technologies. To implement slicing, The Slicing framework leverages the Neutron controller QoS API as well as the ML2 API to communicate QoS policies to the relevant hypervisor interfaces and SDN switches that lie at the VTN data path. QoS rules are stored at the OvS database and applied to the OvS switch ports, forming the basis to implement slicing. The QoS model supported by Neutron and Open vSwitch, shown in Figure 12, includes three QoS rules, that appropriately manage the network ports' priority queues:

- DSCP marking of packets allows traffic prioritization
- Bandwidth limit prevents individual VNFs from saturating the network
- Minimum bandwidth guarantee reserves bandwidth

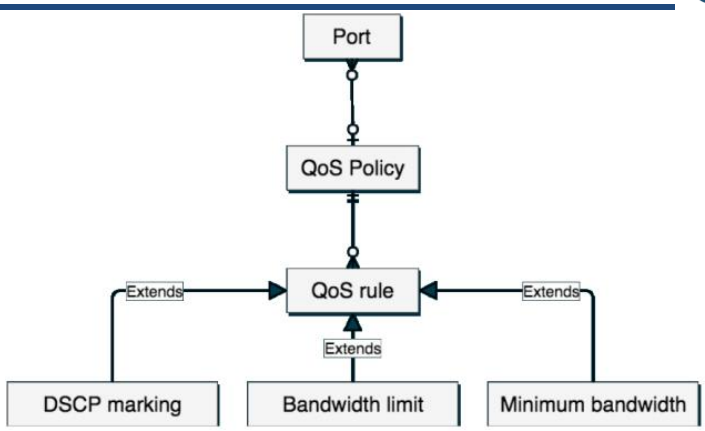

FIGURE 12: NETWORKING LAYER QOS

From the 3 QoS policies supported, bandwidth guarantee is the most critical for Industrial IoT networks that often need strict delay and throughput assurances (e.g., for infrastructure monitoring and smart actuation use cases). End-to-end slicing is implemented in our testbed by reserving bandwidth in all switch ports that lie across the path from an IIoT gateway at the Field level to the IIoT Application VM at the backend cloud. Bandwidth reservation is performed via the Neutron QoS API. However, it must be noted that the OpenStack ecosystem is not able, in its current iteration, to offer strict end-to-end guarantees to applications. Specifically, the underlying infrastructure can't guarantee that hypervisor network interfaces will never be over-subscribed when scheduling new VMs. Hence, an additional verification and Live Migration support step was implemented in our testbed. Overall, service deployment involves the following steps:

1. The VNF image file is uploaded to Glance image storage
2. A VNFD file is supplied to the VNF Manager with service metadata and requirements.
3. The VNF Manager instantiates the VNF, which is automatically placed at a Data Centre hypervisor.
4. An end-to-end slice is deployed based on service requirements, using Neutron QoS APIs.
5. A verification step checks if the hypervisor interface was over-subscribed
6. If the verification fails, select a VNF for Live Migration with the Best Fit scheduling algorithm and go to step 4.

5.2.2 SLICING FRAMEWORK VERIFICATION AND EXPERIMENTAL RESULTS

In this section, the IIoT testbed is evaluated in terms of its ability to guarantee bandwidth reservations in Tenant Networks with slicing, as well as the effectiveness of Live Migration in optimizing VM placement. Finally, the suitability of a virtualized IIoT gateway, which is capable of hosting VNFs, for industrial and haptic applications is also evaluated. In all our experiments, the traffic was generated with the D-ITG traffic generator which can generate TCP traffic with various profiles, e.g., Pareto, Exponential, etc., as well as write trace files. Moreover, a Smart Sensing and an Actuation VNF were deployed, each in a dedicated Tenant Network, that compete for testbed resources.

5.2.2.1 TENANT NETWORK SLICING

In this experiment, we measured the maximum throughput that could be sustained between the two VNFs, both hosted at the Backend Cloud, and a client device which was connected at the Field layer. At first, the link capacity, which is 1 Gbps, is equally shared by the two VNFs, as shown in Figure 13. At time t=11s the Neutron API is employed to setup an end-to-end Network Slice for VNF2, with a dedicated throughput of 700 Mbps. Figure 13 shows that the measured throughput of both VNFs changes instantaneously to 700 Mbps for VNF2 and 300 Mbps for VNF1. This was achieved with successful bandwidth reservation at the hypervisor network interface, as well as at the SDN switch output port where the client device is connected.

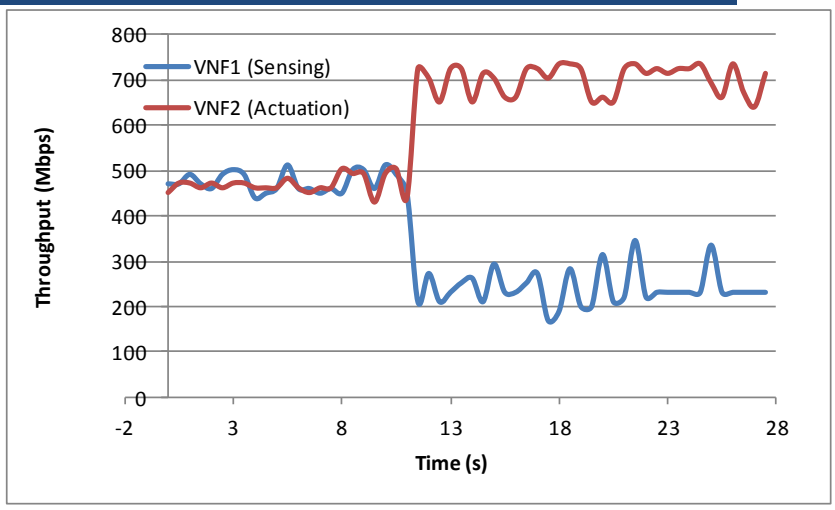

FIGURE 13: THROUGHPUT MEASUREMENT VS. TIME FOR VNF1, VNF2

5.2.2.2 VNF PACKET DELAY

In terms of resource usage, slicing is a relatively expensive solution, and hence often reserved only for the most critical services. An alternative solution to afford low latencies to delay-sensitive services is to place them directly at the IIoT gateway. This way, they bypass the Network Layer and its potential bottleneck, and can directly communicate with Field Layer devices. In the following experiment, the Round-Trip Time (RTT) of packets transmitted from the actuation VNF to the Hue bridge is measured, when it is placed at the backend cloud, or directly at the virtualized IIoT gateway. The RTT of the local cloud is also compared to the cloud service provided by the smart light vendor. In both cases background traffic with an Exponential traffic profile is also generated, with a Load that varies from 0 (no background traffic) to 0.8 (severe congestion). The measured packet delay of the actuation VNF, when hosted at the Local or Remote cloud or at the Gateway is plotted in Figure 14. We conclude that sub-millisecond latencies are achievable for services hosted directly at the IIoT Gateway, which are unaffected by network congestion. Therefore, given that uRLLC is crucial for the manufacturing process, we show that our platform can attain sub-millisecond end-to-end communication, proving the suitability of our platform for tactile internet industrial applications. This is also possible for local cloud services, as long as the link load is less than 0.5, which can be achieved with dedicated slices. However, as shown in Figure 14, even when slicing is employed, queueing delay of Exponential traffic increases noticeably when input load exceeds 50%. Hence, a dedicated slice typically uses up twice the bandwidth required on average and is therefore considered an expensive solution. Finally, Remote Cloud solutions should be avoided for delay sensitive services, as they are subject to significantly higher latencies.

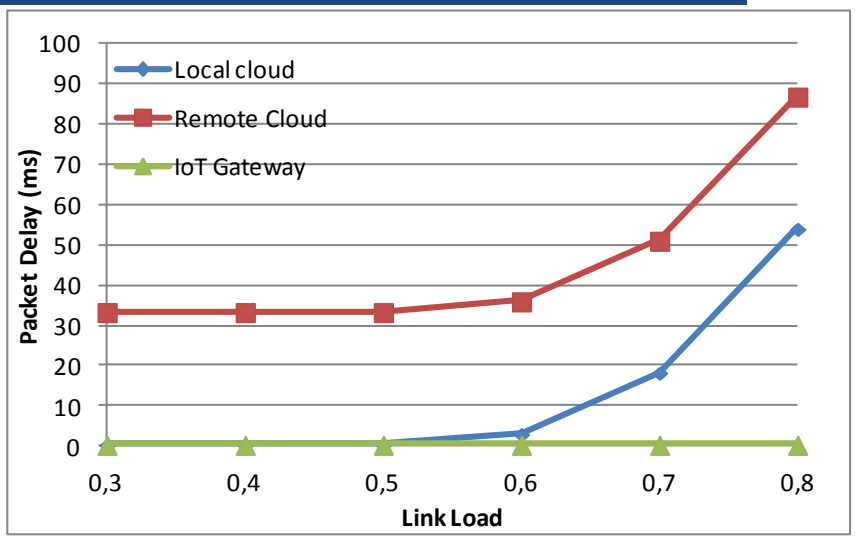

FIGURE 14: *PACKET DELAY VS. LOAD FOR DIFFERENT VNF PLACEMENT OPTIONS*

5.2.2.3 VM MIGRATION

In our last experiment, we explore whether VM migration is an efficient mechanism for the optimal placement of VNFs. Specifically, we test the service disruption caused when VMs are migrated to a different hypervisor at the backend cloud. Figure 15 shows how the throughput measurement of the two VNFs in 0.1 second intervals, when measured from a Field layer client device. The migration time was found comparable is both cases, as in our testbed it is dominated by the copying of Virtual Hard Disk of the VMs. However, in the case of Legacy migration a service disruption of around 8.5 seconds was measured, while services and TCP connections would terminate and need to be restarted. On the other hand, Live Migration caused no service disruption and was only noticeable by a small drop in the measured throughput, which dropped by 40% for a duration of less than 0.5 seconds.

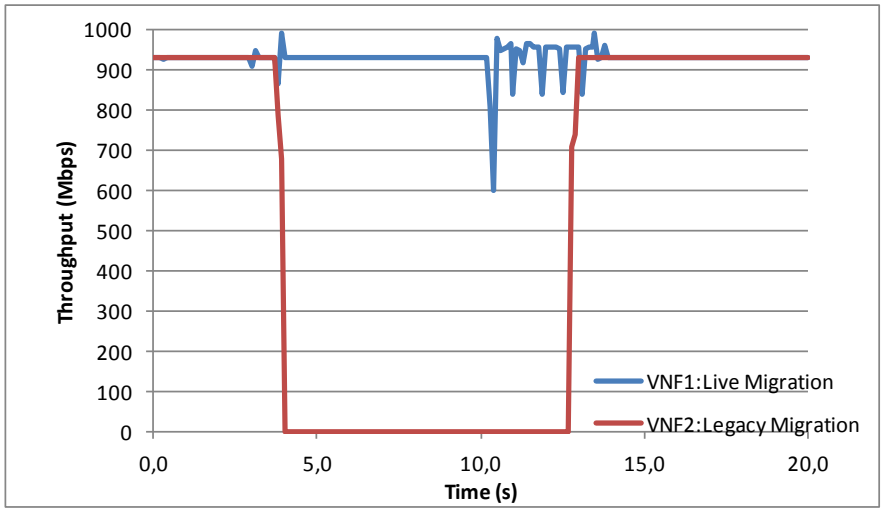

FIGURE 15: THROUGHPUT VS. TIME FOR LIVE AND LEGACY MIGRATION

6 CONCLUSIONS

This deliverable provides initial design of the Field-level middleware, for giving access to sensor data via semantically annotated interfaces over multiple messaging protocols (Ref. section 4.1, 4.2 and 5.1). In this deliverable we discussed how concepts, like NFV, SDN, semantic bootstrapping and interoperability can be leveraged by IIoT networks to increase their reliability, flexibility, and performance. Furthermore, we drafted a preliminary design for the SEMIoTICS field-level middleware and an NFV-enabled experimental platform, which will serve as a SEMIoTICS testbed. The SEMIoTICS testbed will implement an end-to-end IIoT SDN/NFV architecture, complete with the local cloud, SDN networking and Field layers that demonstrate smart actuation, monitoring and analytics functionalities. Standardized semantic models for IIoT applications and SPDI pattern-driven mechanisms that guarantee network-level semantic interoperability were detailed. These form the basis of the semantic bootstrapping and Interoperability framework, which is a significant part of the SEMIoTICS field-level middleware. Finally, we contributed experimental results regarding the deployment of IIoT applications on top of virtualized infrastructure. In one scenario we achieved sub-millisecond latencies for services hosted directly at the IIoT Gateway, which are unaffected by network congestion. Initial concepts and designs in this deliverable will serve as inputs to the second draft of “Field-level middleware & networking toolbox” (D3.6) and networking related deliverables in WP4.

7 REFERENCES

ETSI, 2014a Architectural Framework (ETSI GS NFV 002 V1.2.1). Available: https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf. [Accessed November 2018].

ETSI, 2014b Management and Orchestration (ETSI GS NFV-MAN 001), December 2014b. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf. [Accessed November 2018].

OpenStack 2018a, OpenStack Ironic Project: Bare metal provisioning. Available: https://wiki.openstack.org/wiki/Ironic

OpenStack 2018b, Compute API. Available: https://developer.openstack.org/api-guide/compute/.

Canonical, 2018, Linux Containers. Available: https://linuxcontainers.org/.

OpenStack 2018c, OpenStack Docs: Server concepts. Available: https://developer.openstack.org/api-guide/compute/server_concepts.html.

J. Denton, 2018, Learning OpenStack Networking (Neutron), Second Edition, Birmingham: Packt Publishing Ltd.

OpenStack, 2018d, OpenStack Docs: Networking API v2. Available: https://developer.openstack.org/api-ref/network/v2/.

OpenDaylight 2018, OpenStack and OpenDaylight. Available: https://wiki.opendaylight.org/view/OpenStack_and_OpenDaylight.