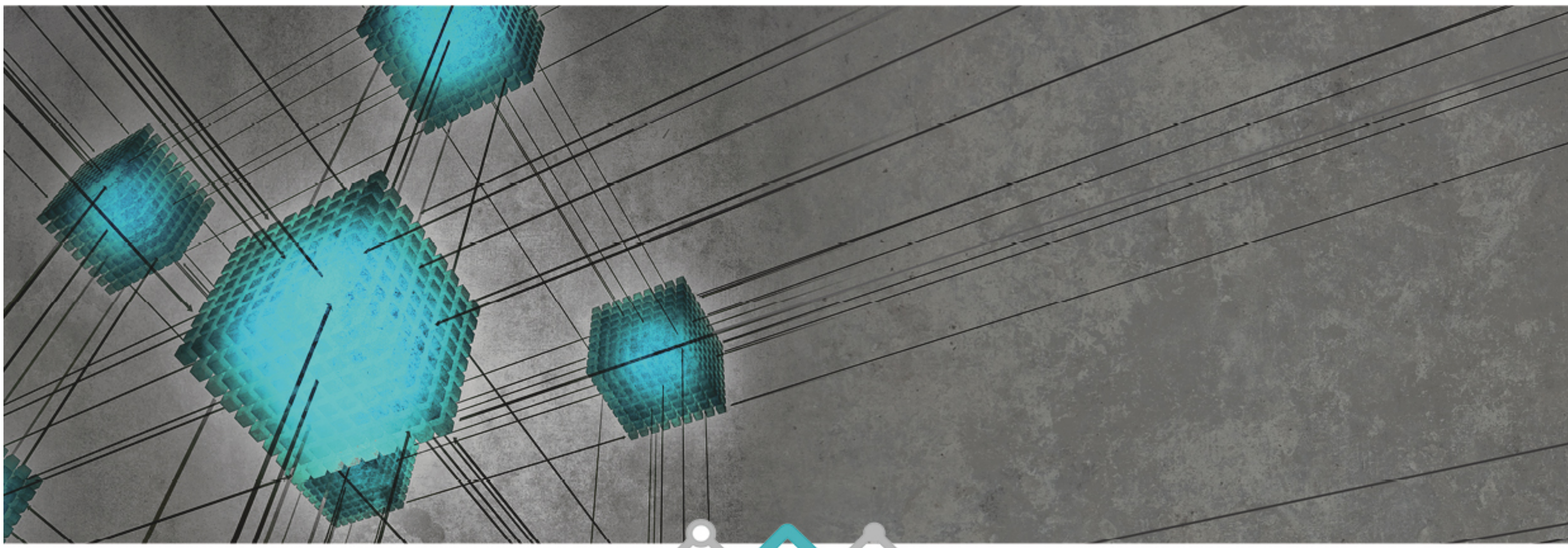


interiot

INTEROPERABILITY
OF HETEREOGENEUS
IOT PLATFORMS.

D3.2

Methods for Interoperability and Integration v.2

October 2017

INTER-IoT

INTER-IoT aim is to design, implement and test a framework that will allow interoperability among different Internet of Things (IoT) platforms.
Most current existing IoT developments are based on ”closed-loop” concepts, focusing on a specific purpose and being isolated from the rest of the world. Integration between heterogeneous elements is usually done at device or network level, and is just limited to data gathering. Our belief is that a multi-layered approach integrating different IoT devices, networks, platforms, services and applications will allow a global continuum of data, infrastructures and services that can enable different IoT scenarios. As well, reuse and integration of existing and future IoT systems will be facilitated, creating a de-facto global ecosystem of interoperable IoT platforms.
In the absence of global IoT standards, the INTER-IoT results will allow any company to design and develop new IoT devices or services, leveraging on the existing ecosystem, and bring get them to market quickly.
INTER-IoT has been financed by the Horizon 2020 initiative of the European Commission, contract 687283.

INTER-IoT

Methods for Interoperability and Integration v.2

Version: 2.3
Security: Public
October 31, 2017

The INTER-IoT project has been financed by the Horizon 2020 initiative of the European Commission, contract 687283

Disclaimer

Executive Summary

The aim of Deliverable 3.2, entitled “Methods for Interoperability and Integration v.2”, is to document the on-going work in the definition and implementation of INTER-LAYER interoperability mechanisms. The deliverable is the second version of a series of three (i.e. preceded by D3.1 and to be followed by D3.3). The deliverable provides the refined architecture description, status of components implementation, documentation and demonstrators. It reports the technical work performed in all WP3 tasks, T3.1 (Definition and Analysis of Methods for Device Layer Interoperability and Integration, M5-M30); T3.2 (Definition and Analysis of Methods for Networking Layer Interoperability and Integration, M5-M30); T3.3 (Definition and Analysis of Methods for Middleware Layer Interoperability and Integration, M5-M30); T3.4 (Definition and Analysis of Methods for Application Service Layer Interoperability and Integration, M5-M30), T3.5 (Definition and Analysis of Methods for Data and Semantics Layer Interoperability and Integration, M5-M30) and T3.6 (Definition and Analysis of Methods for Cross-Layer Interoperability and Integration, M13-M30).

INTER-LAYER is an instantiation (reference implementation) of the INTER-IoT Reference Architecture, which is a result of work carried out in T4.1 (Design of a Reference Meta-Architecture for Interoperable IoT Platforms, M7 – M24) and T4.2 (Design of a Reference Meta-Data Model for Interoperable IoT Platforms, M7 – M24). The RA is defined in the forthcoming deliverable D4.2 (Final Reference IoT Platform Meta-Architecture and Meta Data Model). On the other hand, INTER-LAYER exposes a set of APIs that are a foundation of INTER FW and INTER API, described in deliverables D4.3 (Interoperable IoT Framework Model and Engine v1) and D4.5 (Interoperable IoT Framework API and Tools v1). The relation between the two work packages (WP3, WP4) is provided in sections 3.2 and 4.8.

As already reported in D3.1, the developments have been based on the layered architecture description provided in the Description of the Action, requirements described in D2.3 and subsequent updates (INTER-IoT Requirements and Business Analysis, M9). The work has been further based on use cases and scenarios described in D2.4 (Use cases and scenarios, M12) in order to be in line with the proposed pilots. Finally, through interaction with WP4 tasks related to implementation of INTER FW and INTER API, extensibility and provision of APIs has been defined and implemented.

D3.1 focused on initial collection of the interoperability mechanisms, building blocks and interfaces of the layers: Device to Device layer with the virtual gateway; Middleware layer with the MW2MW component and the Data and Semantics layer with the Inter Platform Semantic Mediator (IPSM). In this deliverable, these layers have reached a maturity level to be applied and evaluated in INTER-IoT pilots. In D3.2, focus has been expanded to the Network to Network layer solution and the Application and Services layer.

With all INTER-Layer components defined, work has stated in the definition of Cross-Layer components, which heavily depend on both architectural choices made for INTER-Layer, but also on common needs identified therein: security, virtualisation, clusterisation and cross-component interactions.

List of Authors

Organisation	Authors	Main contributions
UPVLC	Eneko Olivares, Jara Suárez de Puga, Andreu Belsa Pellicer, Carlos Enrique Palau Salvador	D2D, N2N, INTERMW, AS2AS and Cross Layer sections. Internal review.
UNICAL	Raffaele Gravina	INTERMW, Cross Layer sections. Internal review.
PRODEVELOP	Miguel Ángel Llorente Carmona, Miguel Montesinos	INTERMW and Cross Layer sections. Relation to INTER FW and RA.
TU/e	Tim Van der Lee	N2N sections.
VPF	Pablo Giménez Salazar	Requirements and INTERMW sections.
RINICOM	Eric Carlson	N2N and Cross Layer section.
XLAB	Matevž Markovič, Flavio Fuart, Manja Gorenc Novak	Overall coordination. Introduction, progress, INTERMW, and conclusion sections.
SRIPAS	Wiesiek Pawłowski, Paweł Szmeja, Katarzyna Wasielewska-Michniewska	INTERMW and DS2DS sections.
ABC	Alessandro Bassi	Deliverable format and template.
NEWAYS	Johan Schabbink, Dennis Engbers	D2D sections. Internal review.
SABIEN	Gema Ibáñez, Vicente Traver	Ethics section.

Change control datasheet

Version	Changes	Pages
0.0	Formatting, Inter-IoT template	
0.1	Table of content draft and basic assignments	5
0.2	SotA chapters defined	8
0.3	SotA chapters first draft	30
1.0	Inter-Layer solutions, draft	80
1.1	Inter-Layer solutions, final draft	98
1.2	Relation with INTER FW, Reference Architecture	122
1.3	Ethics	108
1.4	Intro, conclusion	111
2.0	Ready for internal review	138
2.1	Reviewed by internal reviewers	138
2.2	Addressed review comments	139
2.3	Final review	137

Contents

List of Figures

List of Tables

Acronyms

AAL	Active-Assisted Living
AC	Access Control
ACP	Access Control Policy
ACR	Access Control Rule
ADC	Analog to Digital Converter
A.N.	Access Network
API	Application Programming Interface
ARM	API Request Manager
AS2AS	Application & Services Interoperability
BSN	Body Sensor Network
CEP	Complex Event Processing
CHE	Content History Entrepot
CLI	Command-line Interface
COAP	Constrained Application Protocol
COTS	Commercial off-the-shelf
D#.#	Deliverable number #.# (D2.1 deliverable 1 of work package 2)
DAC	Digital to Analog Converter
DMS	Data Management Service
DoA	Description of Action of the Project
DoS	Denial of Service
DS2DS	Data & Semantics Interoperability
D2D	Device Interoperability
EC	European Commission
EU	European Union
FRED	Front-end for Node-RED
GA	Grant Agreement
geoSPARQL	OGC Standard for Geospatial Semantic Data
GUI	Graphical User Interface

HLS	High-Level Synthesis
HTML	Hyper Text Markup Language
H2020	Horizon 2020 Programme for Research and Innovation
IDM	Identity Management
INTER-FW	INTER-IoT Interoperable IoT Framework
ILI	Interoperability Layer Interfaces
INTER-IoT	Interoperability of Heterogeneous IoT Platform
INTER-LAYER	INTER-IoT Layer integration tools
INTERMW	INTER-IoT Middleware
IoT	Internet of Things
IoT-EPI	IoT-European Platforms Initiative
IPR	Intellectual Property Rights
JSON-LD	JavaScript Object Notation used for serialization of Linked Data
jSLP	Java Implementation of SLP
JGroup	Toolkit for Reliable Messaging
KVM	Kernel-based Virtual Machine
LIIs	Layer Interoperability Infrastructures
M#	#th month of the project (M1=January 2016)
MITM	Man-In-The-Middle
ML	Model Language
MPLS	Multiprotocol Label Switching
MVP	Minimum Viable Product
MW2MW	Middleware Interoperability
NBI	Northbound Interface
NETCONF	Network Configuration Protocol
N2N	Network Interoperability
ODL	OpenDayLight
OF	OpenFlow
OF-CONFIG	OpenFlow Configuration
OM2M	oneM2M
ONOS	Open Network Operating System
OS	Operating System
OSGi	Open Services Gateway initiative
OVSDB	Open vSwitch Database Protocol
OWL	Web Ontology Language
PC	Project Coordinator
PCC	Project Coordination Committee

PCS	Port Call Service
PEP	Policy Enforcement Point
PIC	Project Implementation Committee
PPI	Platform Provider Interface
RA	Reference Architecture
RBAC	Role-based Access Control
RDF	Resource description Framework
REST	Representational State Transfer
RSPAN	Remote SPAN
Ryu	Component-based SDN framework controller
QoS	Quality of Service
SaaS	Software as a Service
SAREF	Smart Appliances REFerence
SBI	Southbound Interface
SBT	Simple Build Tool
SDK	Software Development Kit
SDN	Software Defined Networks
SLP	Service Location Protocol
SDR	Software defined Radio
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOSA	Sensor, Observable, Sample and Actuator
SPAN	Switched Port Analyzer
SPARQL	Simple Protocol and RDF Query Language
SotA	State of the Art
STH	Short Time Historic
STPM	Scientific and Technical Project Manager
SSL	Secure Sockets Layer
SSN	Semantic Sensor Network
TL	Task Leader
TLS	Transport Layer Security
ToS	Type of Service
UDP	User Datagram Protocol
UI	User Interface
URI	Uniform Resource Identifier
VUAIs	Virtualized Unified Access Interfaces
XACML	eXtensible Access Control Markup Language

XML	eXstensible Markup Language
WP	Work Package
WPL	Workpackage Leader
W3C	World Wide Web Consortium

1 Introduction

This deliverable is an evolution of D3.1, *Methods for Interoperability and Integration v.1*, submitted in December 2016. This report (D3.2) assumes that the reader is familiar with D3.1 in order to follow the progress of design and implementation approaches provided herein. Of particular importance is the introductory section of D3.1, where an overview of INTER-LAYER is provided, supplemented with definitions and terminology used throughout both reports.

State of the Art sections in both deliverables can be read independently, as they are self-standing overviews of existing approaches, standards, systems and software products. The INTER-LAYER Design section grabs the attention of the reader interested in the relation between the INTER-IoT Reference Architecture and INTER-LAYER Components. Of similar nature is the section "INTER-LAYER relation with INTER-Framework, which explains the usage of INTER-LAYER components in INTER-FW. The core of this report are the INTER-LAYER components sections that provide, for each INTER-IoT layer: an updated architectural overview, status of the developments, API and extensibility considerations, reference to code and documentation, and a description of deployed demonstrators. An exception is the Cross-Layer component, which is currently in the design phase. The last section considers the Ethical implications of INTER-LAYER developments. Proposed reading paths of the D3.1 and D3.2 bundle is provided in figure 1.

The main inputs for WP3, and subsequently for D3.1/D3.2, are the requirements defined in D2.3 (INTER-IoT Requirements and Business Analysis) and Reference Architecture defined in D4.2 (Final Reference IoT Platform Meta-Architecture and Meta Data Model). The exposed API and extensibility mechanisms of INTER-LAYER are integrated in INER-FW and INTER API, as described in D4.3 (Interoperable IoT Framework Model and Engine v1) and D4.5 (Interoperable IoT Framework API and Tools v1).

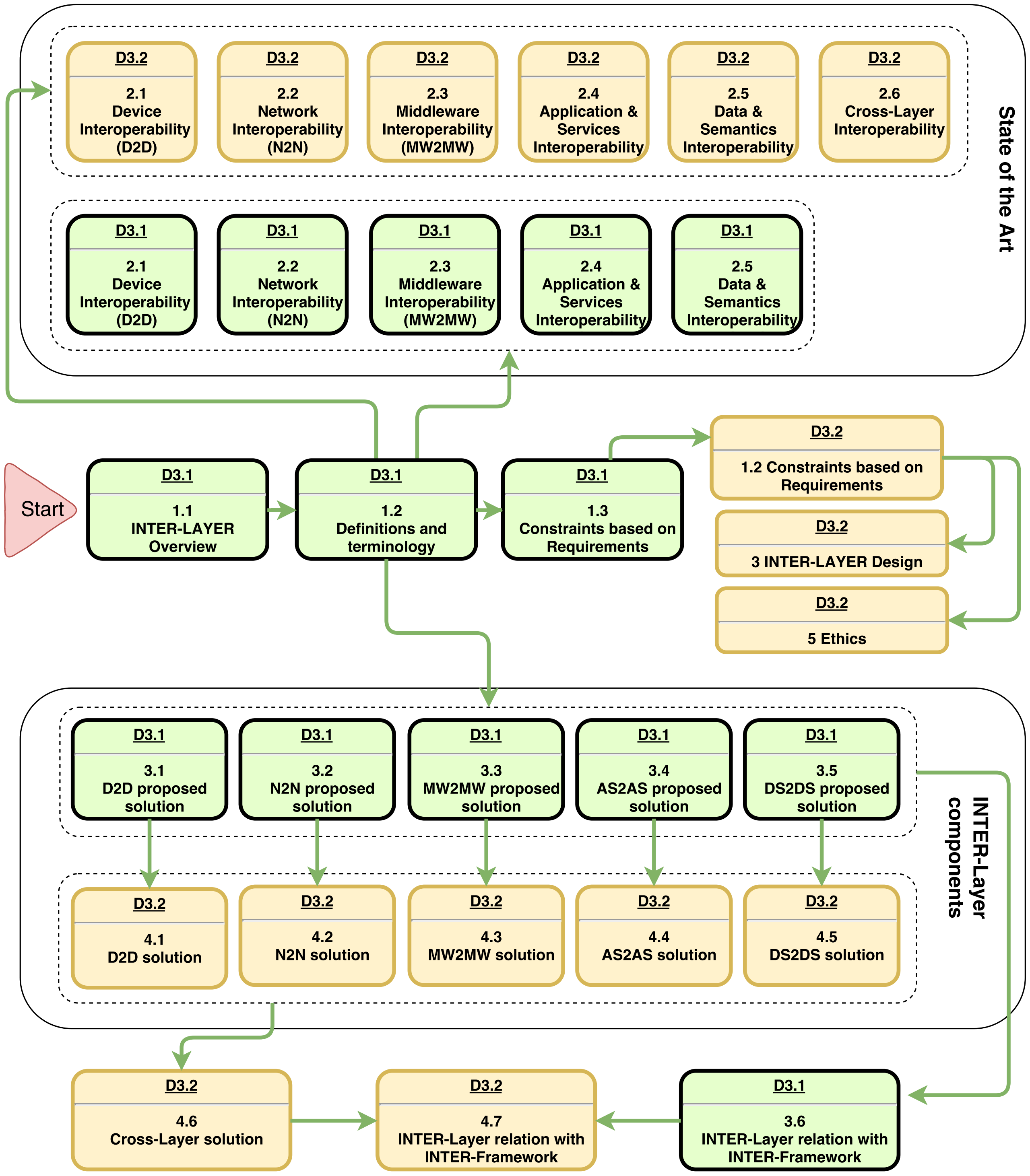


Figure 1: Proposed reading paths of deliverables D3.1 and D3.2

1.1 Progress since D3.1

Since the reporting in deliverable D3.1, a significant progress has been made in further elaboration of the proposed architecture and implementation (coding) of solutions at Device, Network, Middleware, Application and Semantic interoperability layers.

In order to support the implementation phase, several new technologies have been explored and presented in section 2. At the D2D level (Section 2.1) five new gateway implementations have been analysed since the elaboration of Deliverable 3.1, with focus on interoperability mechanisms they provide. At the N2N interoperability layer (Section 2.2) SDN and QoS concepts have been further explored and elaborated in detail. The most important contribution for this layer is the description of Ryu and the rationale for the shift from OpenDayLight to Ryu as the base controller in INTER-IoT. At the Middleware interoperability level (Section 2.3) three middleware platforms related to pilot implementations are presented. In order to design a suitable data representation for interoperability at middleware level, the VITAL-OS platform has been analysed because of it's use of JSON-LD and a section devoted to semantic data representation and processing. Section 2.4 further elaborates Node-RED, a key technology used in INTER-IoT for service interoperability and orchestration at application layer. It also describes Docker, used for virtualization and orchestration of interoperability services, as well ass Swagger, a well established standard for robust REST API definitions. The new version of the SOSA/SSN ontology is described in the Semantics interoperability section (2.5).

A new SoTA section for Cross-Layer interoperability has been added (2.6). Design and development of Cross-Layer components started after the most important features of Inter Layer have been defined. Security in IoT is elaborated, identifying and proposing security implementation for INTER-IoT. Furthermore, communication between layers and approaches for clustering and virtualization are also part of a common, cross-layer approach.

For each Inter-Layer component, progress since D3.1 is provided in the following structure: refined architecture, a detailed description of implemented components, typical examples of scenarios (use-cases), API and extensibility, code/documentation and description of demonstrators. Depending on the maturity of each component, an appropriate level of detail has been provided. On the other hand, Cross-layer has not yet reached the development phase, but the architecture and usage scenarios are provided.

1.2 Constraints based on Requirements

Requirements are the basis for developing any system or application in order to determine the main features that should be implemented.

During the first stage of the project, deep analysis of the needs of the system was carried out, particularly those of interoperability at different levels. These requirements have been collected from end-users of different products of INTER-IoT and from the expertise of consortium partners.

Since the delivery of D3.1 work has been focused on the development of many of the modules that are part of INTER-Layer. During this process, new requirements have arisen that were not taken into account in the first stage, for example the inclusion of new technologies and platforms.

For this reason a process of revision of existing requirements has been carried out and then the new ones have been added. As with the existing requirements, MosCow methodology has been used.

The definition and revision of requirements is a continuous process that must be carried out throughout the project development.

For the first stage of the project, when D3.1 was submitted, only requirements with greater priority (Must) were considered. In this deliverable, requirements have been reviewed in order to consider some of them with lower priority. These changes are reflected in the development of the different components (section 4) and in the security of the whole system (section 4.7.1).

2 Update to the State of the Art

2.1 Device Interoperability (D2D)

The D2D interoperability in this section is about:

- the ability to share information and services,
- the ability of two or more devices, systems or its components to exchange and use information,
- the ability of devices to provide and receive services from other devices.

In IoT this is typically achieved through a gateway. An IoT Gateway has the ability to allow different devices using the same or different access networks to communicate to other devices and through the northbound API to the middleware or other applications. These devices can be sensors, actuators or prosumers (sensor and actuator) and the gateway can provide other services such as device discovery, QoS, security, cache storage, and so on.

2.1.1 Current D2D gateway setup and classifications

In D3.1 ”Methods for Interoperability and Integration” initial device classifications were proposed, this has been revisited in order to update the implementation and the device definitions that the D2D Interoperability Gateway will focus on. The original class designations described in D3.1 are given in Table 1.

Name	Data Size (e.g RAM)	Code Size (e.g. Flash)
Class 0, C0	$<< 10KB$	$<< 100KB$
Class 1, C1	10KB	100KB
Class 2, C2	50KB	250KB

Table 1: Classification of constrained devices[1]

- *Class 0* devices are very constrained sensor-like motes. As they are severely constrained both in memory and processing capabilities, they most likely do not have the resources required to communicate directly with the Internet in a secure manner. They however can participate in

[1]Source: http://www.rfc-base.org/txt/rfc-7228.txt

Internet communications with the help of larger devices acting as proxies, gateways, or servers. They are most likely preconfigured with a very small dataset.

- *Class 1* devices are quite constrained in code space and processing capabilities. They cannot easily talk to other Internet nodes employing a full protocol stack such as HTTP, Transport Layer Security (TLS), and related security protocols, as well as XML-based data representations. However, they are capable of using a protocol stack specifically designed for constrained nodes (such as the Constrained Application Protocol (CoAP) over UDP) and to participate in meaningful conversations without the help of a gateway node. In particular, they can provide support for the security functions required on a large network. Therefore, they can be integrated as fully developed peers into an IP network, but they need to be sparing with state memory, code space, and in many cases also power expenditure for protocol and application usage.
- *Class 2* devices are less constrained and fundamentally capable of supporting most of the same protocol stacks as used on notebooks or servers. However, even these devices can benefit from lightweight and energy-efficient protocols and from consuming less bandwidth. Furthermore, using fewer resources for networking leaves more resources available to applications. Thus, using the protocol stacks defined for more constrained devices on Class 2 devices might reduce development costs and increase the interoperability.

2.1.2 Gateway implementations

New gateway implementations have been considered since the last time the state of the art of D2D interoperability was made. The new IoT Device Gateway implementations revisited have been:

Proprietary

This implementations were tested only for the features offered by the trial/free version and also the documentation of the paid features were checked.

- *Ubiworx*: Intel software focused in IoT gateways solutions easily configured to collect and analyze locally the measurement from the connected sensors. It provides programmable rules and events and can communicate with private/public monitoring and reporting system as SCADA or ERP.
- *FogHorn*: software at the Edge, provides high performance processing, analytics and application closer to physical systems, focused in fog computing.
- *Bosch ProSyst*: also OSGi based gateway but without focus in edge processing. Provides APIs and out-of-the box support for most protocols.

Open Source

Apart from the already studied Eclipse Kura and OM2M Gateway implementations it was also considered **IoTivity** and **Agile** (one of the IoT-EPI H2020 projects). Not only for testing features but also to understand the D2D interoperability patterns that these gateways implemented in their source code. From one side, IoTivity provides an open source reference implementation of the OCF standard specification, as a complete software but with a gateway implementation module. The software provides connectivity with protocols as; Wi-Fi Direct, CoaP, Bluetooth and BLE, ANT+, Z-Wave and Zigbee, plus other features over this access protocols as discovery, data transmission device and data management, etc. From the other side, Agile IoT solution offers a modular hardware and software gateway with support of protocols for interoperability as WiFi, BLE, ZigBee, ZWave, LoRa, RF and more modules, data and device management, IoT apps at the top and external Cloud communication.

After testing each new gateway implementation we can assure that INTER-IoT D2D Gateway implementation is still relevant and covers the following main features that are lacking in the analyzed gateways: *Flexibility and modularity* (not only supporting different devices and protocols, but also multiple open source middleware platforms) and *Lightweight Physical implementation* (since all the difficult processing tasks is shifted to the virtual gateway in the fog).

2.2 Network Interoperability (N2N)

We understand N2N interoperability as the competence of interconnect heterogeneous elements conforming a single network, or the interconnection of already created networks with different nature, to make it easier to include new elements in the network, improving the scalability, dynamically adapting the network configuration depending on the requirements and managing different QoS parameter for that purpose.

To make it easier to understand this section, we provide a list of changes and additions from the previous deliverable D3.1 :

- Included modifications in SDR technologies used, delta from the previous deliverable.
- Included deeper information about RYU and justification of the shift from ODL to RYU.
- Included extra information about QoS in SDN and the protocols used for implement it.

2.2.1 SDN technologies

In the previous deliverable we define what are the Software Defined Networks (SDN) and how these can be used as an interoperability approximation for the Internet of Things deployments. Additionally, we went across the different technologies available in the market to implement SDN paradigm. Among these technologies, we selected the most suitable ones to implement our interoperability solution. As this deliverable is an increment form the previous one, we will define in deeper detail the technologies we implement and which changes have been done from the previous to the current development state.

2.2.1.1 Open vSwitch as a virtual switch

Open vSwitch is a multilayer virtual switch designed to enable massive network automation through programmatic extension, supporting standard management interfaces and protocols, additionally open to a programmatic extension. Moreover, is thought to run in several VM environments (Xen, KVM, VirtualBox, Proxmox VE, etc.), exposing the interfaces to the virtual networking layer but, also having the possibility to support distribution across multiple physical servers. Among its feature, the following are relevant to our purposes:

- Security: VLAN isolation and traffic filtering among others.
- Netflow, sFlow, SPAN, RSPAN for monitoring, mirroring and increased visibility.

- QoS: traffic queuing and shaping.
- Southbound interface protocols as OpenFlow and OVSDB for automated control.
- Integration with virtual management cloud systems as OpenStack, OpenQRM, OpenNebula or oVirt.

Open vSwitch provides a more complex design than simple “bridges”, being these the basic components to be used but, meanwhile bridges are only executed in host kernel space, our virtual switch, uses both kernel space and user space, to create more complex rules of packet processing. Main components that compose the virtual switch are:

- **ovs-vswitchd**: daemon that implements the switch, together with a compilation of the Linux kernel module for flow-based switching.
- **ovsdb-server**: lightweight database server to store and obtain switch configuration.
- **ovs-dpctl**: tool for configuring the switch kernel module.
- **ovs-brcompatd**: daemon that allows ovs-vswitchd act as a substitute of Linux bridge.
- **ovs-vsctl**: command for queuing and updating the configuration of daemons.
- **ovs-appctl**: utility that sends commands to the switch daemons that are running
- **Others**: scripts and specs.

Moreover, apart from the main components of the switch, there are some extra tools or extension for implementation of specific features, such as:

- **ovs-ofctl**: utility that implements the OpenFlow protocol to communicate with the controller.
- **ovs-pki**: for creating and managing the public-key infrastructure of OpenFlow switches.
- **ovs-testcontroller**: in case you don’t have an OF controller, this implements a very simple one that may be useful for testing.
- **tcpdump-patch**: enables to parse OpenFlow messages.

We can observe an example of this architecture in Figure 2 and how each component relates with the other between kernel and user spaces.

The two main protocols of external management implemented in the switch are OpenFlow and OVSDB. The former protocol was slightly introduced in the first version of this deliverable, some other aspects as QoS will be contemplated in future sections, the latter protocol is a component to manage OVS implementation and state, allowing the request and modification of the switch configuration. This will be also explained with more detail in further subsections.

2.2.1.2 Ryu as a base controller

As indicated in the previous deliverable D3.1, RYU is a component-based SDN framework controller. It is simple, modular and highly designed to increase the agility of the network through its easy management and versatility. Ryu provides a principal component (Ryu-manager) that is the heart

[2]Source: `https://upload.wikimedia.org`

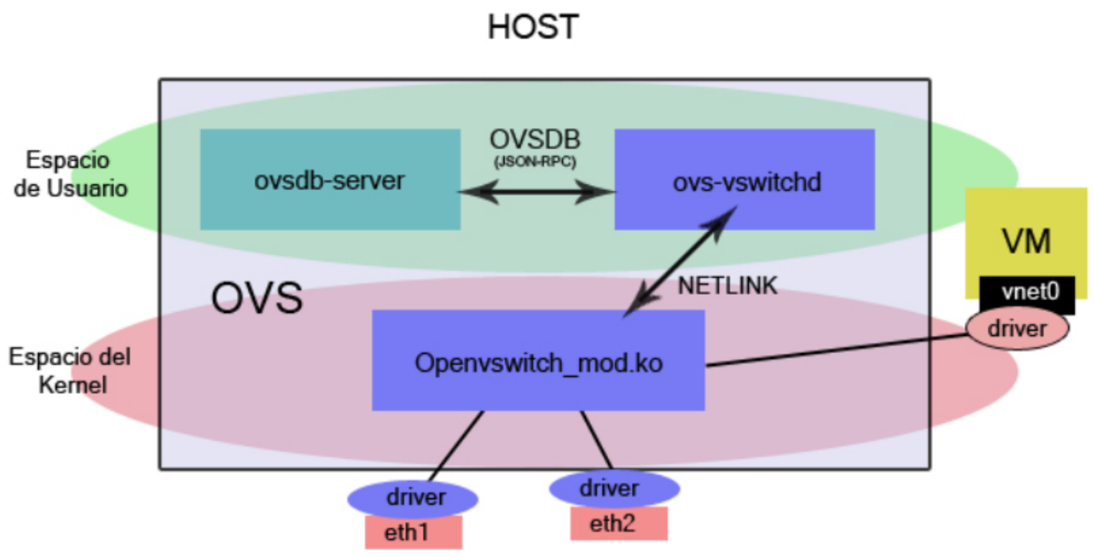


Figure 2: Open vSwitch internal components architecture[2]

of the controller and it is in charge of providing the environment where the different modules and applications will run, and the communication between these modules. Additionally, Ryu defines different APIs to access the software components make it easier for future developers to create control applications.

With all these advantages in mind and taking into account the possibility of customizing the controller attending the needs of INTER-IoT, a shift from OpenDayLight to Ryu took place after the research and delivery of the previous version of this document. Moreover, the simplicity and the lightweight of the controller are valuable characteristics at the time to take the decision. Additionally, the compatibility with systems such as OpenStack (a technology that will be explained in following sections), was the definitive reason to perform this switch in the technological choice. However, OpenDayLight has not being abandoned. This technology has been studied and analysed in order to obtain information and requirements for future implementations or improvements of the N2N solution.

2.2.2 QoS in SDN

The booming sector of the Internet of Things is from the standpoint of networking always more demanding. Sometimes thousand of IoT devices can be inter-connected, with several applications or services running on these devices. These applications and services generate their own data traffic carried through the Internet. The network aspect of these different applications has to be considered in order to successfully deliver over a dense network. Some applications may require a different treatment. As an example, some applications such as video streaming require a certain bandwidth for its flows, while other applications such as the alarm system may be more delay-sensitive. Addressing these requirements needs Quality of Service (QoS) mechanisms implemented over the network.

OpenFlow supports obtainment of a per-flow QoS control in a scalable, flexible and fine-granular way. In this section, we review the QoS capabilities of the OpenFlow protocol and we do such by looking at its different versions and those of (well-know) open-source SDN control platforms.

2.2.2.1 QoS in OpenFlow

Each new OpenFlow (OF) specification introduced new features and changes related to QoS, as listed below.

- **OF1.0**: In this version, an OF switch can have one or more queues for its ports. It is also possible to read/write headers for VLAN priority and IP ToS.[3]
- **OF1.1**: This version improves the matching and tagging of VLAN and MPLS labels and traffic classes.[4]
- **OF1.2**: Supports querying all queues of a switch. Introduction of the OF-CONFIG protocol[5] to reconfigure queues within the switch. Max-rate property can be set to the queue. Flows can also be mapped to queues attached to ports.[6]
- **OF1.3**: This version introduced meters. A meter entry consists of “Meter Identifier”, “Meter Bands”, and “Counters”. A Meter Band, in turn, consists of a “Band Type” (e.g. drop or remark DSCP etc.), “Rate” (e.g. kb/s burst), “Counters”, and optional “Type-specific arguments”, such as drop and DSCP remark. Counters may be maintained per-queue, per-meter, and per-meter band etc. They help controller collect statistics about the network. There may be one or more meter bands per meter table entry. Meters can be combined with the optional set_queue action, which associates a packet to a per-port queue in order to implement complex QoS frameworks such as DiffServ. These meters complement the queue framework already in place in OpenFlow by allowing for the rate-monitoring of traffic prior to output. More specifically, with meters, we can monitor the ingress rate of traffic as defined by a flow rule. Packets can be directed to a specific meter using the optional meter(meter_ id) instruction, where the meter can then perform some operations based on the rate it receives packets.[7]
- **OF1.4**: In this version, a controller can monitor another controller, or more generally, modifications done in the flow table of predefined switches.[8]
- **OF1.5**: Multiple meters can be used.[9]

2.2.2.2 QoS in SDN frameworks

OpenFlow does not provide support for configuring queues, ports, etc., which therefore requires some configuration protocol. This could be done by manual configuration with the protocols NETCONF, OF-CONFIG ONF), etc. or with OVSDB (OpenFlow vSwitch Database Management Protocol - IETF[10]). Currently, several SDN platforms offer the possibility to configure these queues and thus handle QoS.

[3] http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf
[4] http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf
[5] https://www.opennetworking.org/sdn-resources/onf-specifications/openflow-config
[6] https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf
[7] https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf
[8] https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf
[9] https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf
[10] www.ietf.org/rfc/rfc7047.txt

There are many proprietary and commercial SDN platforms, as well as collaborative and open-source projects driven by the research community. An example of them is discussed below.

Open Network Operating System (ONOS)[11] It has limited QoS support at the moment. It supports metering mechanisms but is rarely implemented.

OpenDayLight (ODL)[12] ODL consists of many sub-projects, such as southbound protocol plug-ins (NetCONF, SNMP...) and applications complementing each other to offer a complete controller. OVSDB southbound plugin has been introduced in ODL-Lithium release, allowing the management of queues in switches. A Reservation module also allows resource reservation for QoS.

Floodlight [13] QoS module implemented for Floodlight uses the DSCP values, applying specified rules in switches. The QueuePusher extension (2014) introduces an API for OVSDB protocol to configure queues.

Ryu [14] Provides a Rest API with the OVSDB protocol to configure queues and meters allowing external applications to manage Open vSwitches.

As we observe, QoS is barely implemented in the SDN controllers. They offer the mechanisms and protocols to manage the QoS but usually an external application or algorithm is required to manage dynamically the QoS parameters of the network attending the status of itself in each moment. To better understand how the protocols of QoS configuration work, we will introduce them in the next subsections.

2.2.2.3 OF-CONFIG in Qos-SDN

The called OpenFlow configuration and Management Protocol is a protocol that defines mechanisms to access and modify the configuration data on an OpenFlow physical switch directly from the controller. Despite OpenFlow determines how packets are forwarded between individual sources and destinations, it does not provide the configuration and management functions that are needed to allocate ports or assign IP addresses. For that purpose, the OF-CONFIG was created and implemented in the majority of controllers that manage an SDN network, providing an overall view of this one and the possibility of set policies and manage the traffic across nodes. The implementation of OF-CONFIG in a switch requires modifying the switch's internal configuration database and implement the Netconf Protocol (RFC 6241) to communicate between configuration points and switches, using XML encoding for configuration data and protocol messages.

2.2.2.4 OVSDB in QoS-SDN

As OpenVSwitch is a virtual switch that enables its programming and management, its state is stored in a database server and the switch daemon is used. The named Open vSwitch Database management protocol is the one used to control the cluster database and determine the configuration of the virtual switch including its ports, bridges, interfaces and other important switch information. The

[11] http://onosproject.org/
[12] https://www.opendaylight.org/
[13] http://www.projectfloodlight.org/floodlight/
[14] https://osrg.github.io/ryu/

OVSDB Protocol uses the JavaScript Object Notation (RFC 4627) for its schema and wire protocol format and JSON-RPC 1.0 for its wire protocol.

The differences between OF-CONFIG and OVSDB protocols are several. Mainly the OVSDB is focused in configuration of virtual switches implemented with Open vSwitch while OF-CONFIG is focused in configuration of physical switches. However, more and more vendors, including physical switches manufacturers implement OVSDB within their products. Additionally, these protocols own many other differences; encoding, features, commands, etc. depending of the element they configure.

2.2.3 Software Defined Radio

Manufacturers continue to make developments building more capable, advanced and power efficient SDR solutions. Continued work in this area will lead to more capable systems and bring prices down increasing the likelihood of financially viable solutions being made available to consumers in the IoT space.

2.2.3.1 Hardware

Recently Xilinx have produced a Zynq Ultrascale+ RFSoC with extremely fast ADC's and DAC's [15]. They have a number of variations which include 8x8 and 16x16 channel versions and SD-FEC [16]. Analog devices have produced a newer agile RF transceiver, the AD9377, which has an observation path [17]. Lime micro-systems have the very capable LMS8001+ underdevelopment as well [18].

Continued review of the available hardware and the new features available will be ongoing throughout the project to ensure that the consortium is aware of the latest available hardware to further this are of research and development.

2.2.3.2 Software

In addition to GNU radio open source software, we have reviewed Xilinx's Vivado HLS development methodology and their SDSoC Development Environment. They have continued to improve it to ease development cycles.

The methodology focuses on the use of parallel development flows for the valuable differentiated logic and the shell used to integrate the differentiated logic with the rest of the ecosystem. Additionally, use of a C-based IP development flows is covered for the differentiated logic to provide simulations that are orders of magnitude faster than RTL simulations, as well as accurately timed and optimized RTL. Pre-verified, block, and component-level IP can also be used for shell development. Use of scripts to automate the flow from accurate design validation through to programmed FPGA. Users have reported 4X speed up in development time for designs a 10X speed up in the development time for derivative designs and 0.7X to 1.2X the Quality of Results (QoR).

[15] goo.gl/QZRWm7

[16] https://www.xilinx.com/products/silicon-devices/soc/rfsoc.html

[17] http://www.analog.com/en/products/rf-microwave/integrated-transceivers-transmitters-receivers/wideband-transceivers-ic/ad9375.html

[18] http://www.limemicro.com/products/

2.3 Middleware Interoperability (MW2MW)

Elaboration of INTER-IoT pilots resulted in implementation of INTERMW bridges for UniversAAL, BodyCloud and WSO2. VITAL-OS was researched and we include it in the updated SotA because of it's use of JSON-LD in communication.

A common INTER-IoT ontology has been defined at the project level. A concrete instantiation of the ontology is the implementation of a message structure used throughout INTERMW, and serialized as JSON-LD. This resulted in adaptation of the following concepts and technologies, further described in the Semantics section.

2.3.1 VITAL-OS

The main objective of the EU FP7 project VITAL[19] was to design and develop/prototype a federated middleware solution able to integrate and orchestrate heterogeneous IoT platforms/services/data streams operating/available in the smart city environment. The VITAL-OS platform offers filtering and processing mechanisms, as well as a range of tools directed towards both data providers, and end-user application developers.

Moreover, it provides a set of data models and interfaces supporting collection and annotation of data from heterogeneous sources with the help of a unified JSON-LD data representation. A web-based central management environment is also included.

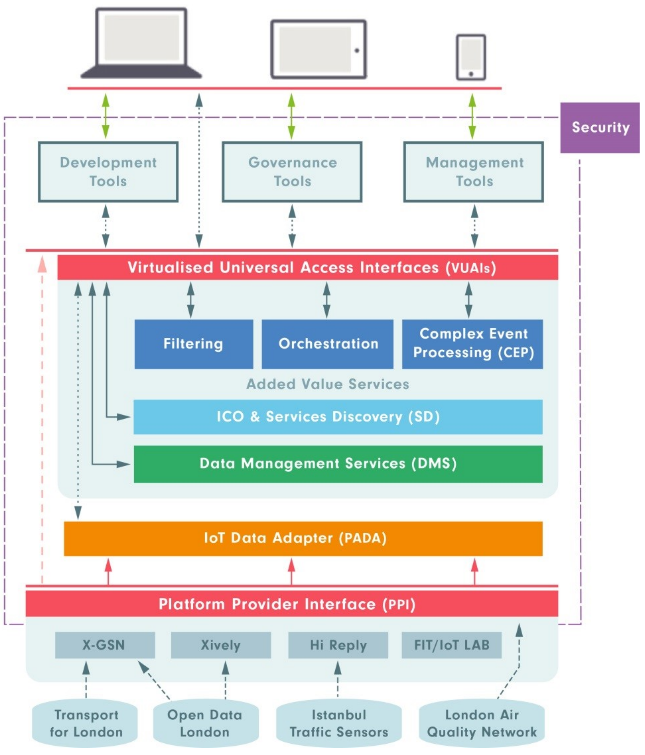


Figure 3: VITAL-OS platform[20]

[19] http://vital-iot.eu/

The overall architecture of the VITAL-OS platform is shown in Figure 3. The role of its main building blocks can be characterized as follows.

- *Platform Provider Interface* (PPI) defines a unified access to platforms' components and meta-data.
- *Data Management Service* (DMS) offers persistence and basic querying mechanisms for the semantically enriched data (and meta-data).
- *IoT Adapter* periodically fetches and pulls data to the DMS.
- *Virtualized Unified Access Interfaces* (VUAIs) provide a virtualized and platform-independent access to data and services.

The role of the remaining architectural components is clearly given by their respective names.

The VITAL-OS middleware platform was implemented using the Java programming language, as well as the JavaScript-centric Node.js ecosystem.

2.3.2 UniversAAL

UniversAAL has been described and related to the IoT-A model in the deliverable D4.1.

The EU-funded project universAAL[21] aimed to produce an open platform that provided a standardized approach to develop Active-Assisted Living (AAL) solutions. The project consolidated previous AAL efforts and projects and tried to come up with the best solution to make development open, feasible and economically viable for any developer. It has since then pivoted to an IoT-oriented support[22], given that the base technology is independent from the applications at the top (although AAL-oriented services will still find more useful resources than others will).

The core of the universAAL platform consists of a semantic matchmaking middleware. All computing devices running the middleware can communicate with each other through the middleware's peer-to-peer communications (based on jSLP[23] and jGroups[24]). For the applications connected to the middleware, communication is seamless regardless of the device they are in. All they need to do is connect to one or some of the middleware's three buses: Service, Context and User Interface.

The Service bus is a call-based bus, which works for pull communications. It is used for executing services. The Context bus is an event-based bus, which works for push communications. It is used for publishing information updates. Finally, the UI bus is a combination of both approaches and is used for creating user interfaces the user can interact with.

In all cases, the information shared through the buses is based on ontologies (using RDF and OWL semantic technologies for representation). The description of the services, their execution calls, the events and their subscriptions, everything is described ontologically. This is what enables the middleware to perform semantic matchmaking: Applications describe their abilities, and what they what to obtain, semantically, without having to specify explicit addresses, IDs, coded values, etc...

[20]Source: http://vital-iot.eu/
[21]http://universaal.sintef9013.com/index.php/en/
[22]http://www.universaal.info/
[23]http://jslp.sourceforge.net/
[24]http://www.jgroups.org/

In an example scenario of having to turn on the lights in the bathroom, an explicit service would have to look up a registry or call all other services that handle lights, probably check some metadata to determine their location, get their IDs, and then call them one by one. With UniversAAL's semantic matchmaking, it would only need to describe what it needs to once. A service call that would read like "turn on everything that is a light and is in a room called Bathroom".

A particular type of application are the Exporters. Each of these Exporters connects to a particular sensor network technology and translates the interaction with the hardware into universAAL semantic services and events. The universAAL platform already provides Exporters for technologies such as ZigBee, KNX or Eclipse Smarthome.

In the context of INTERMW, the UniversAAL bridge has been developed to support the INTER-Health Pilot and also to support any kind of IoT Health or AHA (Active and Healthy Aging) environments that require interoperability (it will be used as well in the H2020 IoT-LSP ACTIVAGE project).

2.3.3 WSO2

WSO2 has been described and related to the IoT-A model in the deliverable D4.1.

WSO2 is an open source service-oriented architecture (SOA) middleware. It is designed with independent components, so it can be adapted for a lean targeted solution to enterprise applications. The entire WSO2 middleware stack works seamlessly across private, public, WSO2 managed and hybrid clouds, as well as on-premise.

WSO2 products make heavy use of Java technology and are built on top of WSO2 Carbon, the company's SOA middleware platform. Carbon makes use of Apache Axis2 and encapsulates SOA functionality such as data services, business process management, ESB routing/transformation, rules, security, throttling, caching, logging and monitoring.

In the context of INTERMW, a WSO2 bridge has been developed to support the INTER-LogP Pilot.

2.3.4 BodyCloud

The effective and efficient management of a large number of cooperative and non-cooperative BSNs is a critical task in supporting a wide range of pervasive applications for communities of users. In fact, the huge amount of data that networks of BSNs may generates, requires a scalable and flexible infrastructure for the collection, storage and processing, including the online and offline analysis of data streams. The management of networks of BSNs and their gathered data cannot be autonomously accomplished with their limited resources. BodyCloud [1,2] tackles the problem by exploiting a Cloud computing infrastructure and providing an integrated platform, namely a Cloud-enabled BSN infrastructure, that offers:

- capabilities of using heterogeneous sensors through mobile devices acting as gateways,
- scalability of processing power for different kinds of analysis,
- scalability of data stream storage,
- ubiquitous and global access to the processing and storage infrastructure,
- easy sharing of results and

- pay-as-you-go pricing for using BSN services.

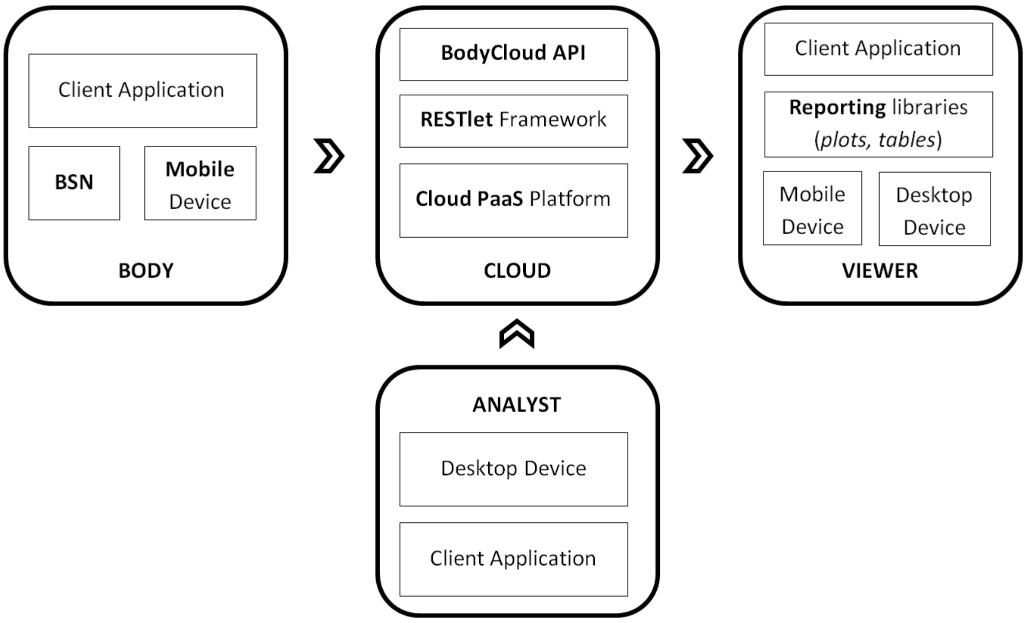


Figure 4: BodyCloud simplified architecture diagram

BodyCloud is a distributed software framework for the rapid prototyping of large-scale BSN applications. Currently based on Google App Engine, it is designed as a SaaS architecture to support the storage and management of sensor data streams and the processing and analysis of the stored data using software services hosted in the Cloud. In particular, BodyCloud endeavors to support several cross-disciplinary applications and specialized processing tasks. It enables large-scale data sharing and collaborations among users and applications in the Cloud, and delivers Cloud services via sensor-rich mobile devices. BodyCloud also offers decision support services to take further actions based on the analyzed BSN data. Its design (see Figure 4 for a simplified architecture diagram) and implementation choices allow BodyCloud to flexibly be tailored, in a very effective manner, for supporting a broad range of application domains, including m-Health, Building automation, and environmental monitoring.

2.3.5 Data representation and processing

Semantic web and graph data. The Semantic web is a standard that promotes common data formats and exchange protocols. It has two main objectives: common formats for integration and combination of data drawn from diverse sources, and a language for recording how the data relates to real world objects. Traditionally two approaches have been used to store data, and both are still very relevant today. The most used has been relational databases (for example MySQL, MS SQL), as database for all kinds of applications. In addition, hierarchical databases are widely used, such as XML, for sending information. However, there are other approaches using semantic data such us RDF (Resource Description Framework). RDF is a common acronym within the semantic web community because it forms one of the basic building blocks for forming the web of semantic data. This language is used to define a type of database called a graph database. RDF define statements in the form of subject, a predicate (property), and an object. These expressions are known as triples in

RDF terminology. The subject indicates the resource and the predicate denotes features or aspects of the resource and expresses a relationship between the subject and the object. Relational databases stores data in tables (rows and columns) while RDF represents data as a list of triples

JSON-LD. The *JavaScript Object Notation* (JSON), designed by Douglas Crockford, and defined in the IETF RFC 7159 [3], is a lightweight key-value type data-interchange format. Inspired by the JavaScript standard object literal syntax, it is currently one of the most widely used data formats.

The JSON format can also be used for serialization of *Linked Data* [4, 5]. The W3C JSON-LD [6] recommendation can be seen as effectively defining JSON serialization of RDF [7], although it allows some extensions, like *blank nodes* as predicates for example (RDF predicates have to be IRIs). Fortunately, these differences between the JSON-LD and the RDF can be treated in a uniform way – some suggested solutions can already be found in the recommendation [6] itself.

JSON-LD can be smoothly integrated into any solution which already utilizes JSON. Since any JSON-LD document is at the same time a valid JSON document, all programming tools and libraries for parsing, transforming, and storing JSON can be applied to JSON-LD data as well.

Although it was primarily designed for use in the area of Linked Data, JSON-LD is also a perfect language for formulating any kind of “self-describing” messages, which in addition to the “raw-data” can contain meta-information, further specifying their content. Together with the “compatibility” with RDF, this provides a convincing argument in favor of JSON-LD as an excellent candidate for the data format of the INTER-IoT internal messaging system.

SPARQL. The most common query language used in for querying RDF resources SPARQL. Along with RDF, OWL, and SWRL, it is one of the technologies at the core of the Semantic Web "layer cake". SPARQL is a query language to access RDF data (including insert/delete in SPARQL 1.1). It differs from SQL in that it was designed to operate over disconnected sources over a network in addition to a local database. The SPARQL protocol allows transmitting the queries and results between a client and an engine via HTTP.

Parliament [25]**.** As a database for the INTER-IoT Middleware Services (Platform and Device registry and discovery) we have decided to use Parliament, a free and open source triple store. Parliament is a high-performance triple store designed for the Semantic Web. It was originally developed for internal use in R&D programs and it was released as an open source project under the BSD license in 2009. It has a number of interesting features:

- Innovative data storage scheme that interweaves the data with a unique index. This keeps the index small and allows Parliament to keep the index up-to-date with very little extra effort. Because of the index, Parliament can answer queries efficiently by reordering query execution.
- Use of temporal index, so that it can efficiently answer queries like "find all events that occurred between times X and Y".
- Supports GeoSPARQL, the newly adopted OGC standard for geospatial semantic data. Using its geospatial index, Parliament can efficiently answer queries like "find all items located within region X".
- Parliament directly supports RDF statement reification, enabling efficient storage and lookup of provenance and other kinds of metadata.

[25] http://parliament.semwebcentral.org/

- Parliament includes a high-performance rule engine, which serves as an efficient means of inference. The rule engine applies a set of inference rules to the directed graph of data in the triple store in order to derive new facts. This enables Parliament to automatically and transparently infer additional facts and relationships in the data to enrich query results.

2.4 Application & Services Interoperability (AS2AS)

In the state of the art of the previous document, we have indicated that access and use of the services provided by IoT platforms, in most cases, is done through their APIs. We are going to describe in more detail some of these services in the Demo section (4.5.7) of this deliverable.

In the previous document was justified that the Node-RED tool is an element that allowed easy access to these services. It also has functionalities to enable interoperability, interconnection and exchange of messages between them. This tool is in a continuous evolution and has relation with other tools with different purposes. Therefore, in this state of the art, special emphasis is placed on studying the evolution of the Node-RED project and its relationship with other tools with purposes such as virtualization or API's description.

The software selected to perform the virtualisation is Docker, it is a software that can package an application and its dependencies in a virtual container that can run on any server with the Docker platform. For that reason, it offers a portable and efficient solution to run the Node-RED instances.

Finally, the software selected to work with API's description is Swagger, it is a framework of API developer tools for the OpenAPI Specification, enabling development across the entire API lifecycle, from design and documentation, to test and deployment. For that reason, it offers a interesting solution to document and design the nodes of Node-RED.

2.4.1 Node-RED

We have mentioned, among other things, the advantages of this tool, the functionalities offered, the existing nodes and the relationship with other projects. Although it is interesting to add a few new notes, to emphasize the amount of new developments that are generated daily related to this platform.

Node-RED is a project with a large number of collaborators. There are many branches and developments focused in several directions. But despite this, the main development of the Node-RED tool has a clear road map. The current state is a stable and open tool that provides a good basis to work and develop our solution, but the future functionalities that are being developed will give a more complete functionality. The current direction of Node-RED project is explained in the "Roadmap to 1.0 slides" [26].

The main points of this document are improve the following issues:

- Provide the necessary extension points to be customised
- API Stability
- Support modern development practices

[26] https://speakerd.s3.amazonaws.com/presentations/f647c01eecd94eb0ba0c0a51bbd755ab/nr-roadmap-to-onepointzero.pdf

- Collaboration issues
- Provide tools to manage a system at scale.

Node-RED growth and expansion is occurring in parallel to the development of our project. We are creating our functionalities that adapt to our specific needs. For that reason, this roadmap evolution is closely followed by us, because we must know what is going to happen. Together with this document, there are channels to follow in real time the evolution of these developments and let us to collaborate with them like the Google Group [27], Slack [28], Git repository [29] and Trello [30].

There are some projects, including Inter-IoT, that develop solutions related with the Node-RED tool and they are using different approaches.

Firstly, there are projects that are integrating this interoperability solution in their platform, as is the case of Sofia2[31]. In this case they have adapted to their needs, have developed new components and have dedicated effort to create documentation and courses to facilitate their learning. Other projects, for example, FRED takes advantage of Node-RED instances to create solutions above it.

Secondly, there are other IoT projects that are not including the complete solution but are providing the development of official nodes to facilitate the work with this platform[32]. Or in other cases, third parties have implemented nodes to access the services offered by IoT platforms[33].

For that reason, it is necessary a constant search of those trends and contents that are being developed, to know which can be adapted to the needs of the project and which must be developed by us. Not only with the purpose to achieve a solution that adapts to the needs of INTER-IoT, also, to actively collaborate in the ecosystem that is being formed around Node-RED.

2.4.2 Relation of Docker with Node-RED

The instances of the interoperability solution could be contained in a Docker container. This facilitates portability, scalability, and multi-instance creation of the Inter-IoT Node-RED instances.

The official Node-RED documentation provides many ways to run Node-RED under Docker and customize the container. Including options like:

- Storing data outside of the container
- Building the container from a source
- Building a custom image
- Updating the base container
- Linking Containers

[27] https://groups.google.com/forum/#!forum/node-red
[28] https://nodered.org/slack/
[29] https://github.com/node-red/node-red
[30] https://trello.com/b/R0O3CSrI/node-red-whiteboard
[31] https://about.sofia2.com/2017/05/10/visual-development-in-sofia2-with-raspberry-node-red-and-dashboards/
[32] https://github.com/lcarli/NodeRedIoTHu
[33] https://flows.nodered.org/

There are projects that are working with this relationship between Node-RED containers and the APIs offered by Docker. Their purpose is to create platforms in the cloud with the main features of Node-RED. A good example is the FRED project (Front-end for Node-RED).

This project provides a multi-tenant cloud hosted Node-RED system for rapid IoT integration and application development. It supports the ability to run flows for multiple users. This flows should get fair access to CPU, memory and storage resources. It provides secure access to flow editors and the flow run-time. Furthermore, the system scales with the number of users and their hosted flows.

The internal FRED components related with docker are:

- FRED-IS: they are micro services that manage the node-red processes. They are Docker container instances.
- Proxy: it is used to create, start and stop a Node-RED process, delegating the work of managing these processes to FRED-IS. Redirects communication from each user's browser to their node-red process, devices and services that are communicating with a Node-RED flow using HTTP or web sockets.
- Instance Server: they host FRED-IS services. To provide isolation from the OS and limit the memory and CPU, FRED-IS leverages the Docker container system, using the Docker API to manage processes on the host.

It is related with the solution that is being developed in Inter-IoT, because in this layer one of our objectives is to extract the best characteristics that the combination of Docker and Node-RED can offer us.

2.4.3 Relation of Swagger with Node-RED

The interaction with APIs offered by IoT services and its descriptions is a necessary approach in this layer, this is the reason to include functionalities that can provide work with Swagger.

The main goal of Swagger is to define a standard, language-agnostic interface to REST APIs which allows to discover and understand the capabilities of the services without access to source code, documentation, or through network traffic inspection. A consumer can understand and interact with the remote service with a minimal amount of implementation logic.

There are two types of existing nodes that can provide an interaction with Swagger in the interoperability solution of this layer. Their development is by the Node-RED project community, but, currently, is not fully completed and they have a list of things to do. But it is a good starting point to facilitate the tasks of development in our desired solution.

The first type is a Node-RED node able to invoke Web APIs generically based on a Swagger description[34]. It is a node that provides a generic client for Web APIs by using Swagger javascript client. All that is required for to automatically be able to invoke a given API is to have the corresponding swagger description.

This node provides support for:

- Parsing and invoking Swagger 1.0, 1.2 and 2.0 descriptions

[34] https://flows.nodered.org/node/node-red-contrib-swagger

- Content negotiation both for Request and Response content types
- Authentication via Basic HTTP Auth and API Key
- Invocation of APIs, except those with other non-supported authentication mechanisms.
- Embedded swagger descriptions server
- Feedback information about the status of the remote API

However, for example, it does not support OAuth 2.0 authentication.

The study of this node type is interesting from the point of view of the components of our interoperability solution that need to develop the access to IoT native services.

The second node type is a set of tools for generating Swagger API documentation based on the HTTP nodes deployed in a flow [35] [36].

The study of this second option is interesting from the point of view of the components that are going to implement the catalogue and discovery of services and flows.

2.5 Data & Semantics Interoperability (DS2DS)

Since publication of the Deliverable D3.1, an important update has been made to the foundational sensor and observation related ontology – W3C SSN [8, 9].

The new version of the SSN [10] introduces several important additions and overall change to the structure of the ontology. It's based on the original SSN standard, and influenced by OGC's O&M, Sensor ML, as well as various recent attempts at producing ontologies for the IoT.

The ontology has been divided into two main parts/modules – *Semantic Sensor Network* (SSN) and *Sensor, Observable, Sample, and Actuator* (SOSA) ontologies.

The SOSA ontology, which among others, offers long awaited concepts of Actuators and Actuation, constitutes the core of the new SSN. It has been designed to be lightweight and self-contained (not depending on any other modules). Its semantics is based on a description logic "profile" $\mathcal{ALI}(\mathcal{D})$, for which efficient support in modern triple stores exists.

The modular structure of the new SOSA/SSN is depicted in Figure 5. Modules are related to each other in either *horizontal* or *vertical* manner. Let us remind here, that horizontal modules can be understood as (loosely coupled) "subsets" of an ontology, which usually describe distinct aspects of an area of interest. Vertical ones, on the other hand, are built on top of each other and form hierarchies. Due to its modular structure the new SSN should be thought of as an interconnected collection of ontologies rather than one monolithic ontology. The red parts/modules in Figure 5 represent the *normative* modules, i.e., the modules that are formally part of the standard.

In addition to the main problem of the original SSN, namely its heavy-weight nature, the SOSA/SSN ontology addresses several other issues existing in the old standard. In particular, practically every concept can now be virtual. It's a very important change, substantially broadening the applicability

[35]https://flows.nodered.org/node/node-red-node-swagger-ddm
[36]https://flows.nodered.org/node/node-red-node-swagger
[37]Source: https://www.w3.org/TR/vocab-ssn/

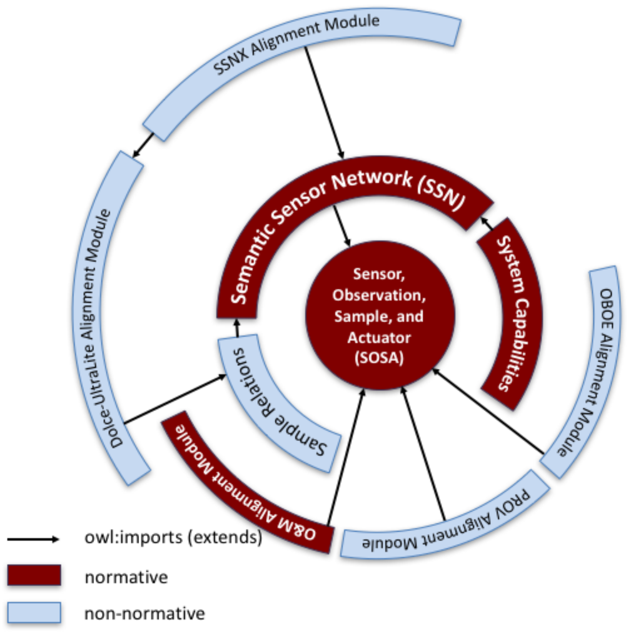


Figure 5: Modular structure of SOSA/SSN[37]

of the ontology. Also semantics of the key notions such as Observation has been clarified, and new concepts, e.g., Procedure have been introduced. Both Procedure and Observation are now modeled as dul:Event.

The alignment between SSN and DUL has also been improved and made fully optional at the same time – in other words SOSA/SSN no longer *depends* on the DUL ontology.

Conceptually, SOSA/SSN ontology has been divided into: System, SystemProperty, Feature, Result, Procedure, Observation/Actuation/Sampling, and Deployment "components". This division, from an actuation perspective, is depicted in Figure 6 and Figure 7, for SOSA and the entire SOSA/SSN respectively.

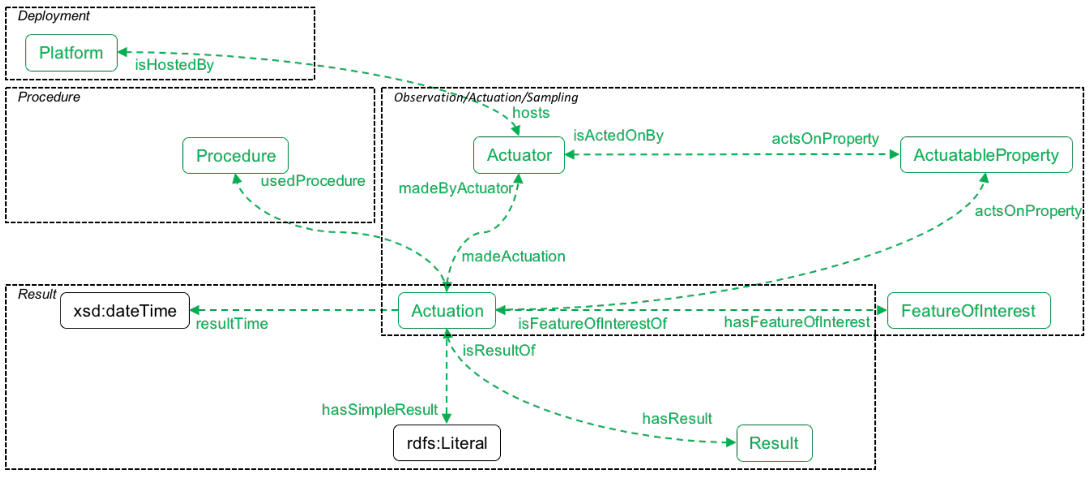


Figure 6: SOSA conceptual structure for actuation[38]

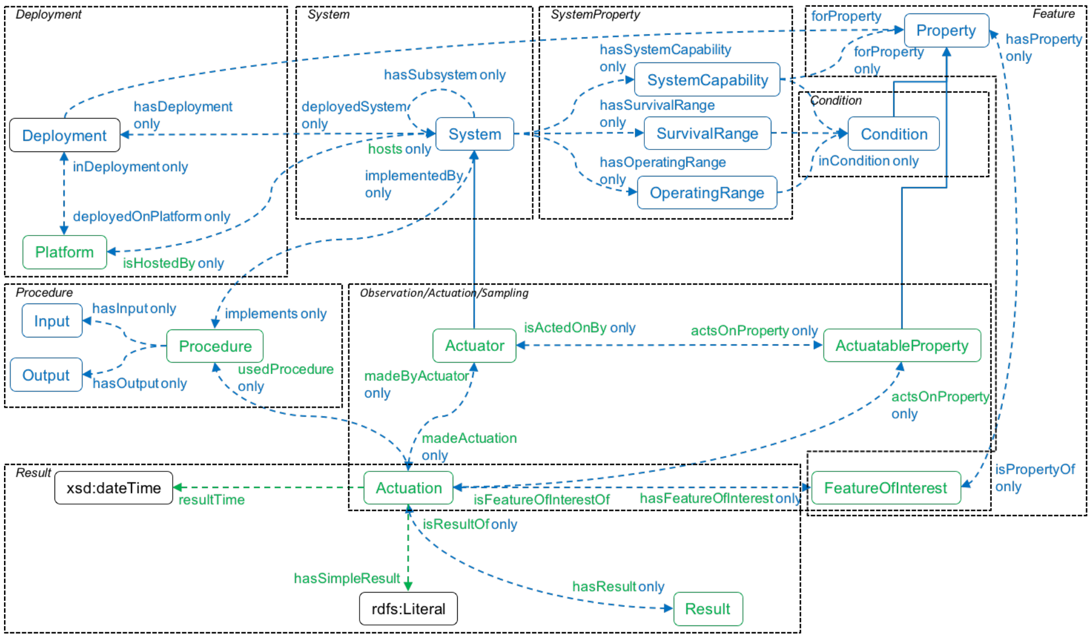


Figure 7: SOSA/SSN conceptual structure[39]

From Figure 6 it can be observed that the structure to represent actuation result has been simplified in comparison to original SSN i.e.

The SOSA/SSN ontology is an important step towards a truly foundational ontology for the IoT. Therefore, it shall replace the original SSN as one of the core modules of the INTER-IoT *central ontology* GOIoTP.

2.6 Cross-Layer Interoperability

INTER-IoT interoperability solution has been divided in 5 layers: Device to Device, Network to Network, Middleware to Middleware, Application and Services to Application and Services and Data and Semantics to Data and Semantics. In all these layers there are common features that are developed as a Cross-Layer solution. These common features are **Security**, **Layer interactions** and **Virtualization and Clusterization**.

In this state of the art section we cover security in IoT platforms and tools, as well as Virtualization and Clusterisation tools that have been reviewed to be used as part of our solution. Layer interactions is not covered in the state of the art section since it is a specific feature of INTER-IoT interoperability layered solution.

2.6.1 Security in IoT

Security is one of the aspects that are most relevant in IoT. There are many trends to deal with security, that implies a continuous research of information and test new technologies. Our purpose

[38]Source: https://www.w3.org/TR/vocab-ssn/
[39]Source: https://www.w3.org/TR/vocab-ssn/

is designing and architecture and developing an IoT interoperability solution that also addresses the security issues that are appearing.

Nowadays, the main security challenges in IoT are communications security and integrity, access control, data encryption, authorization, authentication and identity management.

As a starting point to achieve this purpose in this section we have studied and classified how current platforms work with these challenges, classifying this information into a summary table. We have also studied tools that can help us implement security in Inter-Layer. In section 4.7 we will go deeper into the security aspects of each layer of our solution, because all layers have both common and specific needs.

2.6.1.1 Security in IoT Platforms

It is very relevant to revisit and summarize the different security mechanisms that the most relevant IoT platforms implement nowadays in order to design our security solution. For that reason, in tables 2 to 8 the following security specifications of the selected IoT Platforms have been analyzed: **Communications Security**, **Access-Control Policy**, **Implementation of Authorization Management** and **Additional mechanisms**.

FIWARE

Security Mechanism	Platform Specification
Communications Security	SSL/TLS protocol with a RSA standard cipher requiring a security certificate X.509.
Access-Control Policy	Access Control module (AC) is a control access policy server implemented by Keypass, a multi-tenant XACML server with PAP (Policy Administration Point) and PDP (Policy Detention Point) capabilities and provides a flexible roles structure. These roles, policies and credentials are introduced into the IDM by the application owners.
Implementation of Authorization Management	The "core of the security" is implemented by two components (IDM and PDP). • Identity Management (IDM): Authentication implemented based on OpenStack Keystone APIs plus a SCIM APIs extension. • PDP manages authorization policies in XACML format and enforces decisions based on them when requested by PEPs to authorize or deny access requests to services.
Additional Mechanisms	FIWARE has an additional module, PEP (Policy Enforcement Point), which is a Token-driven component that runs the identity, access and authorization process after a service request using the other components.

Table 2: Security mechanism implemented in FIWARE

UNIVERSAAL

Security Mechanism	Platform Specification
Communications Security	Between internal components and services, RSA over communication bus. Externally, SSL/TLS protocol with certificate-based server authentication.
Access-Control Policy	• Management of permissions of the bus members is implemented in Java in UniversAAL own way. • Additional access-control policies can be made by the application owner with a UI.
Implementation of Authorization Management	• The identity management is carried out by username/password for PCs or a PIN for smartphones. • The authentication service validates the entered credentials against the user credentials of the security user's subprofile stored in the local profile service named Context History Entrepot (CHE). • The right authentication results in the creation of a session. • The authenticated user session enables application services to make authorization decisions based on access-control policies.
Additional Mechanisms	Application services for security provide value added security to the universAAL platform such as document encryption and consent management. As is a sensitive-data oriented platform, the storage information is further encrypted according to IHE DEN standard.

Table 3: Security mechanism implemented in UNIVERSAAL

OM2M

Security Mechanism	Platform Specification
Communications Security	SSL/TLS protocol can be enabled requiring a security certificate X.509 which isn't confined to a single cipher algorithm. For COAP communication, OM2M uses californium which does not implement DTLS, it is possible to change the binding to scandium (secure californium) to implement DTSL.
Access-Control Policy	Different resources allow access control handling • Access Control Policy (ACP) • Access Control Rule (ACR) • RBAC with assigned privilege for CREATE, RETRIEVE, UPDATE, DELETE, DISCOVERY, and NOTIFY
Implementation of Authorization Management	• Mutual Authentication with each originator and receiver. • Entities use mutual authentication, both send authentication tokens derived from master credentials, both send authentication challenge which may or may not be random dependent on security parameters. • Mutual authentication is applied to symmetric and asymmetric key based schemes. • Authorization Basic Token • Authorization controls access to resources and services hosted by all CSEs and AEs (types of node) • ACP resources include privileges and selfPrivileges that include ACRs • SelfPrivileges relate to privileges to change the ACP • Privileges relate to resources and services linked with the ACP
Additional Mechanisms	• Identity protection • AE Impersonation • Prevention

Table 4: Security mechanism implemented in OM2M

OPENIOT

Security Mechanism	Platform Specification
Communications Security	SSL/TLS protocol.
Access-Control Policy	RBAC: Roles are containers which contain permissions and are implemented with Apache Shiro's wildcard. A successful access grants the client a single session token with manageable expiration time.
Implementation of Authorization Management	For authentication and authorization in OpenIoT an OAuth2.0 enabled CAS server is used in JBoss. Open IoT uses Apache Shiro for authentication and authorization. buji-pac4j adds support for OAuth2.0 (using pac4j library) authentication to Apache Shiro. Each client has to be registered in CAS in order to be recognized for OAuth2.0. Client modules provide authentication and access control utilities through a web interface through API REST interaction.
Additional Mechanisms	Local Management Console to set-up the configuration implemented with JBoss.

Table 5: Security mechanism implemented in OPENIOT

SOFIA2

Security Mechanism	Platform Specification
Communications Security	SSL/TLS protocol with a RSA standard cipher requiring a security certificate X.509
Access-Control Policy	RBAC: • Administrator • Collaborator • User
Implementation of Authorization Management	User identity: own plugin-sofia-user either encrypted or not encrypted. Console security: own plugin-console-security based on Spring Security Back-end authorization: own plugin-sib-security based on a mechanism of Token-Session Key.
Additional Mechanisms	Ontology Schemas and JSON validation

Table 6: Security mechanism implemented in SOFIA2

AWS IOT

Security Mechanism	Platform Specification
Communications Security	TLS encrypting (AES) the connection between the device and the broker, requiring a security certificate X.509
Access-Control Policy	RBAC (IAM Roles)
Implementation of Authorization Management	• AWS IoT Policy (JSON Documents) • Cognito identities • IAM Policy
Additional Mechanisms	AWS Signature Version 4 with AWS IoT

Table 7: Security mechanism implemented in AWS IOT

AZURE

Security Mechanism	Platform Specification
Communications Security	TLS with X.509 initially. TLS/PSK and TLS/RPK on roadmap for compute-constrained devices and bandwidth limited or expensive metered links.
Access-Control Policy	Cloud-implemented manageable RBAC (IAM Roles), enabling policy-based access control (through Azure Portal). Basic control access is defined by four kind of permissions: • RegistryRead • RegistryReadWrite • ServiceConnect • DeviceConnect
Implementation of Authorization Management	• Single Identity Key for every device which is used to establish a session token. • Key Management implemented on-cloud with DocumentDB or SQL. • Channel-level authentication and authorization against the gateway. • Validation of signatures against identity registry and blacklist. • Azure IoT Hub grants access to endpoints by verifying a token against the shared access policies and identity registry security credentials. • Azure Cloud Mechanisms including Azure Active Directory and Key Vault
Additional Mechanisms	All messages are tagged with originator on service side

Table 8: Security mechanism implemented in AZURE

2.6.1.2 Security Tools

In this section, we will introduce the security tools being considered for inclusion in Inter-IoT.

Credential store:

A credential store is a library of security data. A credential can hold public key certificates, username and password combinations, or tickets. Credentials are utilized at the time of authentication, when subjects are populated with principals, and during authorization, when identifying the actions the subjects are able to perform.

We have identified the following interesting tools for Inter-IOT:

- Cryptex[40] is a secure secret storage and cryptographic key retrieval tool for Node.js.
- Auth0 with AngularJS[41] helps to implement client-side and server-side (API) authentication to add support for username/password authentication to web, API and native mobile apps.
- Passport-auth0[42] is the authentication strategy for Passport.js

Access to the backend

Oauth2: OAuth 2.0 is an authorization protocol that allows third parties (clients) to access content owned by a user (hosted in trusted applications, resource servers) without the client having to handle or know the user credentials. That is, third-party applications can access user-owned content, but these applications do not know the authentication credentials. It focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices.

In an OAuth 2.0 scenario there are three clearly identified parts:

- The resource owner is an entity capable of giving access to protected resources kept in a resource server. The resource owner receives authorization requests and grants authorization to approved clients if their identity can be authenticated.
- The client (application) is the entity requesting authorization from the resource owner (user). If authorization is granted, the client also requests an access token from the authorization server and uses this token to procure the protected resource.
- The authorization and resource servers control access to and host the protected user information. In many cases the authentication server is the same as the Resource Server. In case they are separated, the authentication server is responsible for generating access tokens and validating users and credentials.

Proposed Backends

OAuth2orize[43] is an authorization server toolkit for Node.js. It provides a suite of middleware that, combined with Passport authentication strategies and application-specific route handlers, can be used to assemble a server that implements the OAuth 2.0 protocol.

[40] https://github.com/TechnologyAdvice/Cryptex
[41] https://github.com/auth0/auth0-angular
[42] https://github.com/auth0/passport-auth0
[43] https://github.com/jaredhanson/oauth2orize

node-oauth2-server[44] Complete, compliant and well tested module for implementing an OAuth2 Server/Provider with express in node.js

Additionally, OAuth provides a list of libraries and services that support OAuth 2.0.[45]

WSO2[46] The state of the art of this security backend has been covered in Deliverable 4.1 since this is the security backend that is being deployed in Inter-IoT.

Oauth Client

Passport:[47] Passport is authentication middleware for Node.js. It is designed to serve a singular purpose which is to authenticate requests. It includes more than 300 authentication strategies and single sign-on with OpenID and OAuth. It handles success and failure, supports persistent sessions and has dynamic scope and permissions.

Everyauth[48] Is a modular tool covering Facebook and Twitter OAuth logins and basic login/password support. It is has configurable authorization strategy with an easy-to-read, easy-to-write approach. It is idiomatic and the syntax for configuring and extending authorization strategies are chainable.

OAuth2 Client[49] OAuth2 Client is a library to help you handle OAuth2 access and request tokens. It's for browser-only use (it came about because it was used for a single-page application), so it only includes the OAuth2 Implicit Grant flow.

2.6.2 Virtualization and Clusterization of Layers

Inter-Layer solutions need to take advantage of virtualization tools to improve in scalability, portability, isolation and flexibility. In addition, one of the main advantages of clusterisation is to simplify the deployment and management of large pools of servers, providing a unique point of configuration becoming these cluster of servers as easy to manage as a single workstation. The following tools offer virtualization, clusterisation or both methodologies:

OpenStack [50]

Open Stack is an open source software for creating private and public clouds managed by OpenStack Foundation. This open-source software platform has a modular architecture that permits to provide a set of core services. The Datacenter controls large pools of compute, storage, and networking resources, managed through a Dashboard that provides a web based user interface allowing users to manage and monitor cloud resources. Moreover, OpenStack differentiates between various types of cloud users, namely administrator, operator and user distinguishing the level of security access that each role has. This solution is interesting to our project as an environment for deploying the different layer solutions, concretely, in the network solution there is a specific plugin in OpenStack to connect with Ryu and so we can deploy our software defined network within OpenStack.

LXC [51]

[44] https://github.com/oauthjs/node-oauth2-server
[45] https://oauth.net/code/
[46] https://wso2.com/identity-and-access-management
[47] http://passportjs.org/
[48] https://github.com/bnoguchi/everyauth
[49] https://github.com/zalando/oauth2-client-js
[50] https://www.openstack.org/
[51] https://linuxcontainers.org/lxc/introduction/

In the Operating System (OS) level, LXC is a virtualization technology that allows multiple instances of isolated operating systems. It permits to control simultaneously multiple virtual units (containers), allowing the isolation of application and operating systems. In addition, because LXC manages real-time resource allocation, it provides near-native performance. Through Kernel Control Groups KXG is capable to manage network interfaces and applying resources inside containers.

LXD [52]

It offers a user experience similar to virtual machines but using Linux containers instead. It's image based with pre-made images available for a wide number of Linux distributions and is built around a REST API.

It's basically an alternative to LXC's tools and distribution template system with the added features that come from being controllable over the network.

This solution is interesting because provides an OpenStack plugin that integrates system containers into a regular OpenStack deployment.

Docker [53]

Docker allows to package an application with all of its dependencies into a standardized unit for software development. Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries; to sum up, anything that could be installed on a server. This guarantees that it will always run the same, regardless of the environment it is running in.

The following list shows some interesting tools that work with Docker:

- Dockerode[54]: Node.js module for Docker's Remote API.
- Docker-maven-plugin[55]: Maven plugin for running and creating Docker images.
- Docker plugin for Jenkins[56]: Docker plugin allows to use a docker host to dynamically provision build agents, run a single build, then tear-down agent.
- Docker Swarm[57]: Docker-native clustering system. It turns a pool of Docker hosts into a single, virtual host.
- Portainer[58]: A lightweight management UI for managing a Docker host or a Docker Swarm cluster.
- Rancher[59]: An open source project that provides a complete platform for operating Docker in production.
- Portus[60]: Authorization service and frontend for Docker registry.

[52] https://linuxcontainers.org/lxd/introduction/
[53] https://www.docker.com/
[54] https://github.com/apocas/dockerode
[55] https://github.com/fabric8io/docker-maven-plugin
[56] https://github.com/jenkinsci/docker-plugin/
[57] https://github.com/docker/swarm
[58] https://github.com/portainer/portainer
[59] https://github.com/rancher/rancher
[60] https://github.com/SUSE/Portus

- Docker Registry[61]: Docker toolset to pack, ship, store, and deliver content.

In this state of the art we have presented a list of different virtualization solutions that fit our needs, after studying all the advantages of those solutions, we have decided to use Docker. In section 4.7.3 will be explained in detail the deployment of our docker based solution, focusing in Docker Swarm and Docker Portainer. In N2N an approach using OpenStack is being considered, since it offers elements and functionalities that can be very beneficial in our network solution.

[61] https://github.com/docker/distribution

3 INTER-LAYER Design

3.1 Development and Demonstration Environments Setup

To enable an homogeneous controlled environment for the development of the different layers, offering the interfaces to other parallel developments such as INTER-FW or INTER-API, a development cloud environment has been set up. This environment is based in the Microsoft Azure Cloud and comprises a set of 7 commodity servers to decouple the different software modules development and, at the same time, making the latest features available to be tested. These servers are all accessed through a unique stepping-stone server. The access to this environment is securized through Microsoft Azure standard security mechanisms.

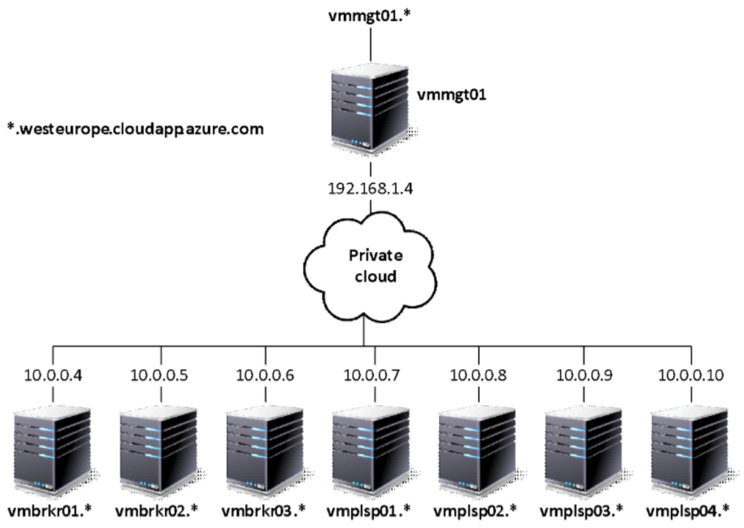


Figure 8: Development and demonstration environment setup

Apart for the development, this environment is also used in the different demonstrations performed during the project execution, thanks to the possibility of easily exposing ports and the use of dynamic DNS to link the servers to a single URL. One example of these demonstrations can be currently found in the address: http://vmbrk03.westeurope.cloudapp.azure.com/interiot_wfk/#.

These servers are available from 07.00 until 19.00 everyday. Since the development they are meant for development and punctual demonstrations, they are offline during the evening and night, while having the possibility to extend the online time up to full-time under request.

3.2 INTER-IoT RA Instantiation

In Deliverable D4.1, an Initial Reference IoT Platform Meta-Architecture and Meta-Data Model has been produced. In this deliverable, INTER-IoT has defined a Reference Model (RM) or “meta-model” for IoT Platforms Interoperability and a Reference Architecture (RA) (through an Architectural Reference Model-ARM) defined based on the RM as well. This work is currently under progress towards a final version of the deliverable (D4.2). This document has a strong basis on the works done in IoT-A EU Project and a deep analysis of 16 heterogeneous IoT platforms carried out in the INTER-IoT project. This RA has been used for the design of the INTER-LAYER architecture, and both, the INTER-IoT RA and the INTER-LAYER architecture (specifically the API design of the different interoperability layers) has been the foundation for the design of the INTER-FW architecture.

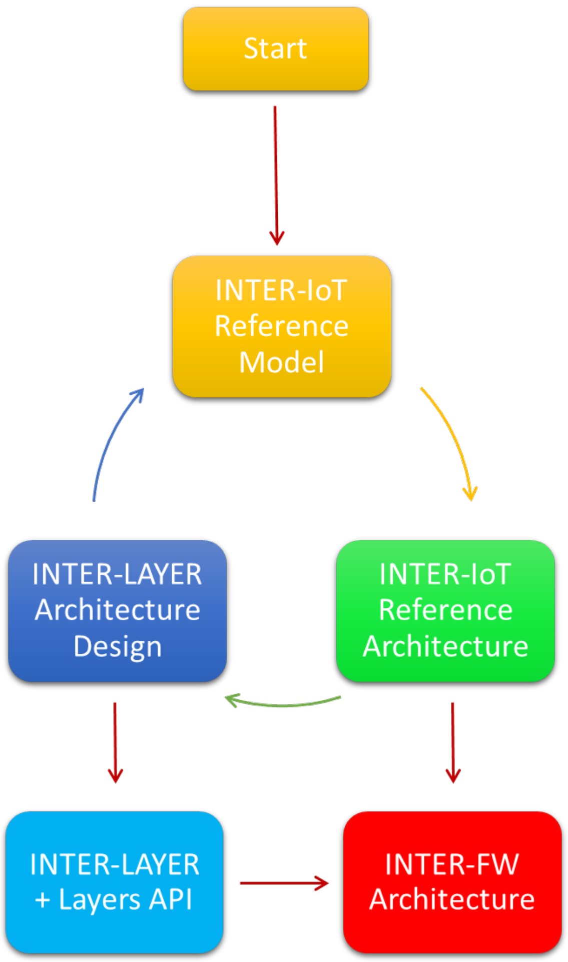

Start
INTER-IoT Reference Model
INTER-LAYER Architecture Design
INTER-IoT Reference Architecture
INTER-LAYER + Layers API
INTER-FW Architecture

Figure 9: Process followed for the generation of the INTER-LAYER architecture

According to OASIS definition:

> A reference architecture models the abstract architectural elements in the domain of interest independent of the technologies, protocols, and products that are used to implement a specific solution for the domain.

According to the same organism:

> A reference architecture is not a concrete architecture; i.e., depending on the requirements being addressed by the reference architecture, it generally will not completely specify all the technologies, components and their relationships in sufficient detail to enable direct implementation.

The INTER-IoT RA was designed for the interoperability of IoT Platforms. This was one of the objectives of the INTER-IoT project (see Objective 2). The explanation of OASIS about a reference architecture and a concrete architecture is exactly what we have done for the INTER-FW: to create an instantiation of the INTER-IoT RA to the specific needs of the INTER-FW.

3.2.1 INTER-IoT RA instantiation for INTER-LAYER

The starting point for this instantiation has been the Functional View for the INTER-IoT Reference Architecture that is depicted in the following figure:

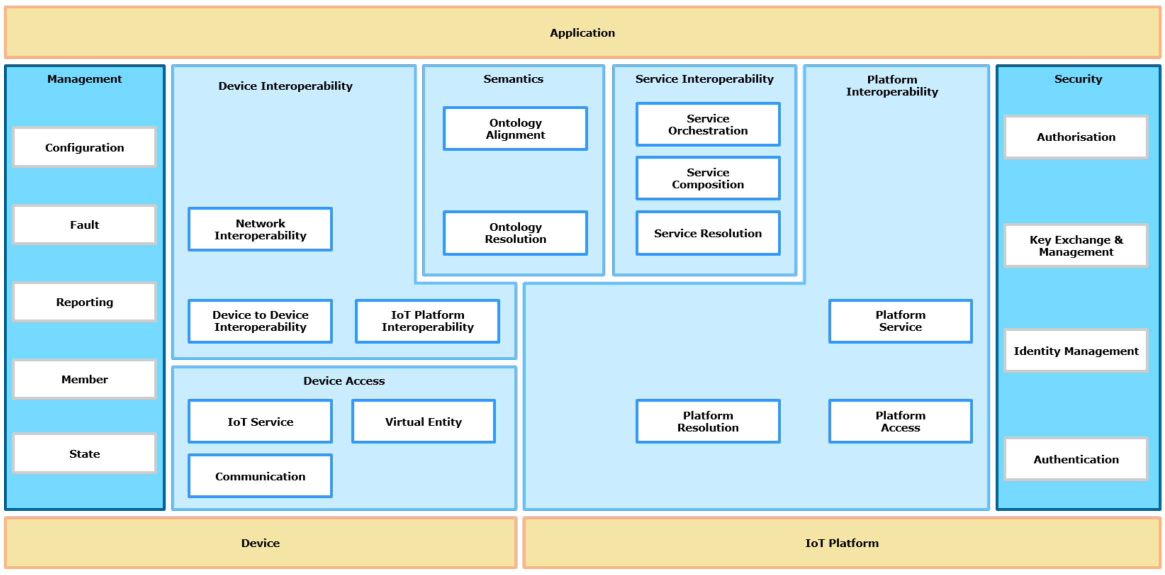


Figure 10: Functional-decomposition viewpoint of the INTER-IoT Reference Architecture[62]

From these Functional Groups (FGs – the big blocks) and Functional Components (FCs – the inner components of each FG), some of them have been identified as relevant for the INTER-LAYER according to its defined requirements. The FGs that are involved in the instantiation of the INTER-LAYER architecture can be seen in the following picture:

[62]All INTER-IoT Reference Architecture images provide from D4.1 Initial Reference IoT Platform Meta-Architecture and Meta Data Model, section 4 INTER-IoT Reference Architecture.

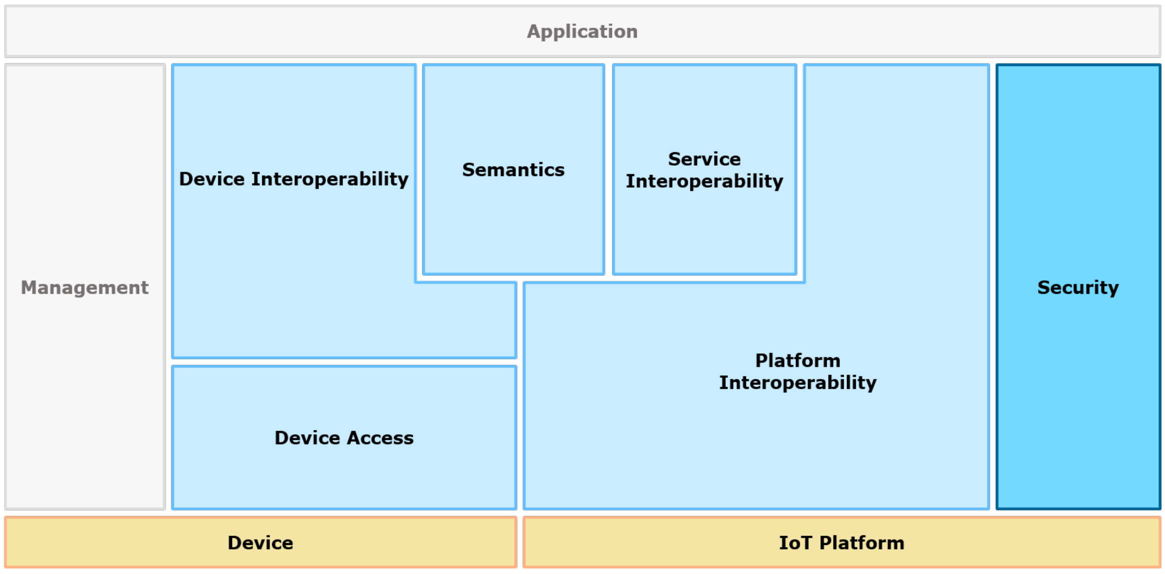


Figure 11: Functional Groups of the INTER-IoT Reference Architecture involved in the INTER-LAYER

The **Application FG** is considered out of the scope, under the premise that the Application will access INTER-FW and not directly the INTER-LAYER. The Management FG is mainly responsibility for the INTER-FW as the manager of all the features provided by the INTER-LAYER, but a small functional set regarding Configuration has been identified as relevant for the gateway, thus being instantiated in it.

The role of the rest of FGs in INTER-LAYER is described below: The **Security FG** is a key group of the INTER-LAYER architecture, which is responsible for ensuring all the security aspects involved in the interoperability of IoT Platforms. The security in our realm has two faces:

- Management of the security aspects related to the connection with underlying IoT Platforms. This implies to accomplish with the different security features that the platforms require. INTER-LAYER will need to tackle the user authentication for connecting to a platform, the authorisation management (e.g. use of authentication tokens) and the encryption of some communications.
- Management of the internal security of INTER-IoT. The connection to INTER-IoT must be secured, with appropriate authentication capabilities, and authorisation management.

The **Device Access FG** plays a relevant role, being instantiated with the components that provide the access to devices, implemented through the gateway and the virtual gateway.

The **Device Interoperability FG** plays a relevant role, being instantiated with the components that provide the D2D / N2N interoperability layers of INTER-LAYER, implemented through the virtual part of the gateway and the software defined network capabilities of the N2N layer.

The **Platform Interoperability FG** will be used for providing access to the different IoT Platforms and performing platform interactions.

The **Semantics FG** will manage the different ontologies for the connected IoT Platforms, and the alignment and end-to-end translation among the available ontologies of the different IoT Platforms.

The **Service Interoperability FG** provides the necessary features to access to services from IoT Platforms, and composing and orchestrating new derived services among IOT Platforms.

Inside each Functional Group, some Functional Components have been identified as necessary for INTER-LAYER. Please note that other FCs are not discarded in INTER-IoT, but addressed by other products of INTER-IoT like INTER-FW.

3.2.2 INTER-LAYER Functional Components

The FCs within each FG that are involved in the instantiation of the INTER-LAYER architecture can be seen in figure 12:

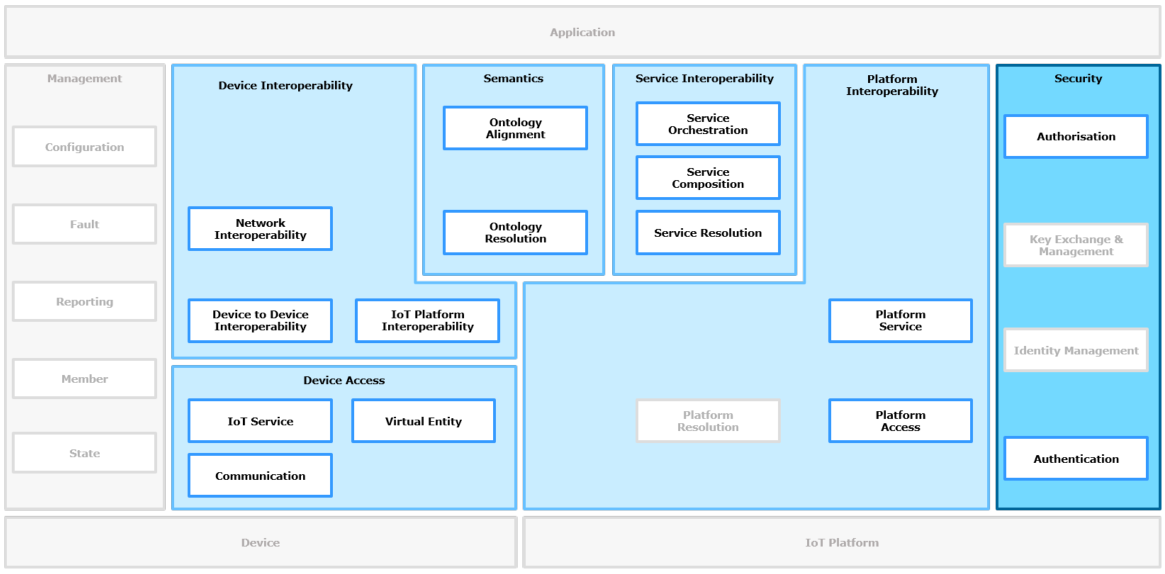


Figure 12: Functional Components of the INTER-LAYER architecture according to the INTER-IoT Reference Architecture

A more detailed description of each FC is described below. For each FC, a pre-candidate definition of architectural components is made.

3.2.2.1 Management FG

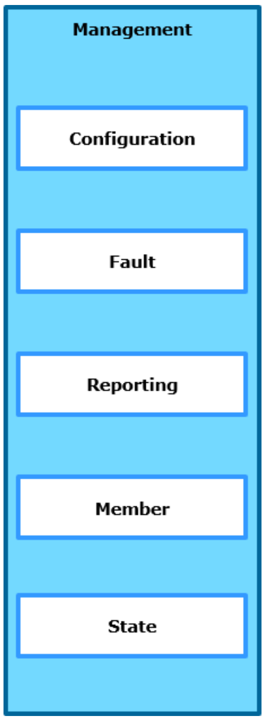


Figure 13: Functional Components of the Management FG instantiated for INTER-LAYER concrete architecture

The list of instantiated FCs of the Security Management FG is described in table 9.

Functional Component	Description
Configuration	It allows the configuration of the different devices attaches to a physical gateway and the network and transport protocols used. It is used to adapt to different situations and react to changes during the operational phase.
Architectural Components	• *D2D Gateway Configuration*: It controls the operation of the physical and the virtual planes of the gateways during the operational phase of INTER-IoT.

Table 9: List of functional components instantiated from the Reference Architecture for the Configuration FC

3.2.2.2 Security FG

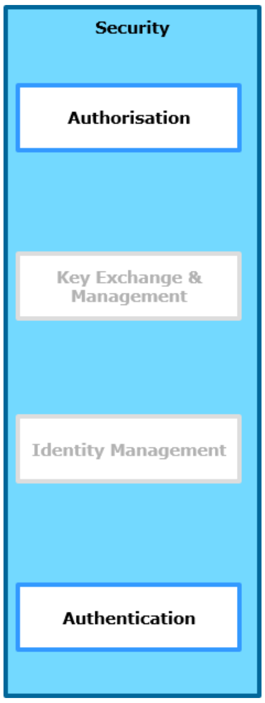


Figure 14: Functional Components of the Security FG instantiated for INTER-LAYER concrete architecture

The list of instantiated FCs of the Security FG is described in tables 10 and 11.

Functional Component	**Description**
Authorisation	Performing access control decisions to specific IoT Platforms and its resources (devices, services, etc.) under certain conditions based on access control policies. This access control decision can be called whenever access to a restricted resource is requested. For instance, a platform owner may will to give access to a subset of devices to a set of user roles, but only within a time range, or when mobile devices are at a certain location.
Architectural Components	• *D2D. Middleware controller*: It handles the connection with the upper IoT Platforms. • *MW2MW. Bridges*: It handles the connection with the different IoT Platforms at the middleware layer. • *AS2AS. Orchestrator*: It handles the connection with existing services deployed at the IoT Platforms.

Table 10: List of functional components instantiated from the Reference Architecture for the Authorisation FC

Functional Component	Description
Authentication	The user authentication for connecting to a platform. The access to the different IoT Platforms maybe user-based or anonymized depending on the decision of platform owners.
Architectural Components	• *D2D. Middleware controller*: It handles the connection with the upper IoT Platforms. • *MW2MW. Bridges*: It handles the connection with the different IoT Platforms at the middleware layer. • *AS2AS. Orchestrator*: It handles the connection with existing services deployed at the IoT Platforms.

Table 11: List of functional components instantiated from the Reference Architecture for the Authentication FC

3.2.2.3 Device Access

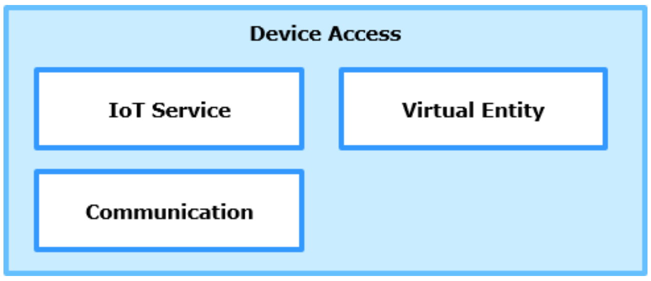


Figure 15: Functional Components of the Device Access FG instantiated for INTER-LAYER concrete architecture

The Device Access FG will be instantiated completely. It will be responsible for offering a common interface to services and virtual entities that represent and expose functionality of physical devices. The Device Access FG will be implemented by the physical gateway and the virtual gateway, operating at the D2D interoperability layer.

The list of instantiated FCs of the Device Access FG is described in tables below:

Functional Component	Description
Communication	This FC is instantiated in order to handle the whole communication protocol stack under the transport layer. This protocol stack management implies to address all the features related to the communication tasks (flow control, network access, protocol conversion, etc.). Therefore, it's responsibility of the Communication FC to manage the communication with the devices with two different aspects: • Access network. Handling the access to the different communication networks that may appear to establish the contact with the devices (WiFi, LTE, Bluetooth Low Energy, Serial, etc.). • Transport Protocol Management. Managing the necessary actions to provide end-to-end communication between devices and the gateway, specifically supporting transport protocols like MQTT, CoAP, LWM2M, Raw, etc.
Architectural Components	• *D2D. Access Network Controller*: It handles the connection with the underlying sensors through specific network protocols (WiFi, serial, LTE, BLE, etc.). • *D2D. Protocol Controller*: It handles the protocol conversion among different transport protocols (CoAP, MQTT, LWM2M, etc.) and the flow control.

Table 12: List of functional components instantiated from the Reference Architecture for the Communication FC

Functional Component	Description
Virtual Entity	This VE FC is instantiated to allow the interaction with an IoT Platform on the basis of Virtual Entities rather than IoT Services in the virtual gateway.
Architectural Components	• *D2D. Virtual Entity Dispatcher*: It handles the virtual representation of physical entities and dispatches the information between the physical gateway and the IoT Services .

Table 13: List of functional components instantiated from the Reference Architecture for the Virtual Entity FC

Functional Component	Description
IoT Service	The IoT Service FC is responsible for managing IoT Services as well as functionalities for discovery, look-up, and name resolution of IoT Services. These services expose resources of devices to the rest of the components. It may allow to gather information about a sensor in a continuous asynchronous way, after a subscription, for instance. Or it may allow to submit requests to an actuator. The IoT Service will run in the virtual plane of the gateway, decoupling the interaction with the resources of devices from their usage
Architectural Components	• *D2D. Registry*: It keeps a registry of the physical sensors connected to the gateway. • *D2D. Discovery*: It implements the discovery service of the physical sensors connected to the gateway. • *D2D. Device Manager*: It handles the connection and registry of the physical sensors connected to the gateway and the name resolution. • *D2D. Measure Storage*: It is responsible for storing the recent measurements made by the devices and offering access to these measurements from the virtual plane.

Table 14: List of functional components instantiated from the Reference Architecture for the IoT Service FG

3.2.2.4 Device Interoperability

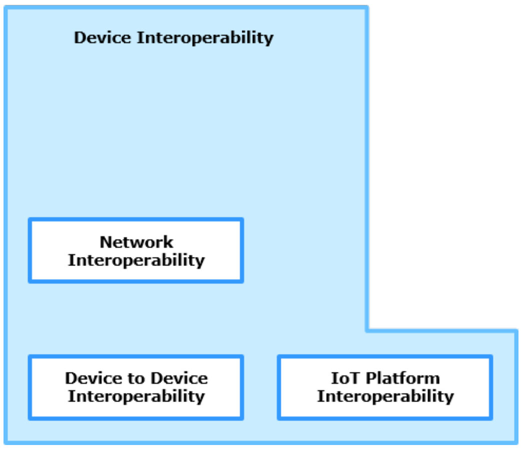

Device Interoperability
Network Interoperability
Device to Device Interoperability
IoT Platform Interoperability

Figure 16: Functional Components of the Device Interoperability FG instantiated for INTER-LAYER concrete architecture

The Device Interoperability FG will be instantiated completely. It addresses the challenges of making legacy devices and non-real IoT Platform interoperable with other IoT Platforms and systems. The Device Interoperability FG will be implemented by the virtual gateway, operating at the D2D interoperability layer and the N2N Solution.

The list of instantiated FCs of the Device Interoperability FG is the following:

Functional Component	**Description**
Device to Device Interoperability	This FC implements the needed functionalities to achieve the interoperability among devices. To enable this interoperability among devices rules are defined in the virtual plane of the gateway using a rule engine.
Architectural Components	• *D2D. Rules Engine*: It defines rules for performing interoperability among devices.

Table 15: List of functional components instantiated from the Reference Architecture for the Device to Device Interoperability FC

Functional Component	**Description**
Network Interoperability	This FC is responsible for managing the interoperability between networks or parts of the network that belong to an IoT deployment. The interoperability solution is based on software defined paradigms but mainly on two approaches: SDR for interoperability on access network and SDN/NFV for the core network
Architectural Components	• *N2N. Openflow connector*: It handles the connection with compatible switches trough OpenFlow protocol. • *N2N. Routing*: It is responsible for executing the routing algorithms. • *N2N. Host Tracking*: It handles the tracking of the different assets being managed at the N2N. • *N2N. Storage*: It stores the status, QoS, tracking and routing information being managed in the N2N. • *N2N. Discovery*: It performs the provision of a discovery service of network topology. • *N2N. Switch Manager*: It controls the different switches of connected networks.

Table 16: List of functional components instantiated from the Reference Architecture for the Network Interoperability FC

Functional Component	Description
IoT Platform Interoperability	This FC is responsible for enabling the interaction between the devices available from the Device Access FG and the Platform Interoperability FG. Please, note that the devices available from the Device Access FG are not devices tied to existing IoT Platforms. The devices connected to an IoT Platform are accessed through the interaction between the Platform Interoperability FG and the IoT Platform FG, while the devices not tied to an IoT Platform (those connected to legacy sensor systems that cannot be considered as IoT Platform), are accessed through the Device Access FG.
Architectural Components	• *D2D. Middleware Controller*: It performs the communication between the gateway and an IoT Platform in two ways: The gateway acts as a client of IoT Platforms, thus being responsible for interconnecting legacy or disparate devices into existing IoT Platforms. The gateway can also act as a kind of legacy IoT Platform from the point of view of the Platform Interoperability. This may happen when there is no IoT Platform where to attach the devices, but there is a need from an external application to access these devices interoperating their data with information from other IoT Platforms.

Table 17: List of functional components instantiated from the Reference Architecture for the IoT Platform Interoperability FC

3.2.2.5 Platform Interoperability

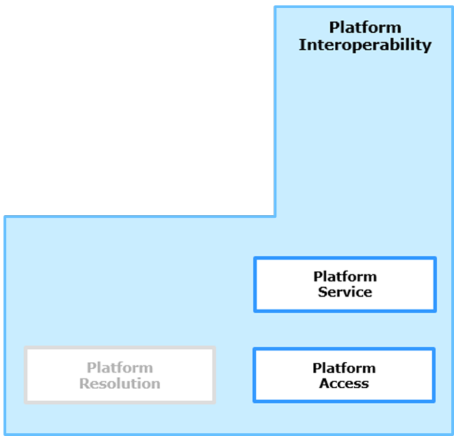


Figure 17: Functional Components of the Platform Interoperability FG instantiated for INTER-LAYER concrete architecture

The Platform Interoperability FG will be instantiated to interact with the different IoT Platforms to be interconnected. It is the responsible for accessing the IoT Platforms, not for implementing any of the features that the IoT Platforms provide. Only the Platform Access and Platform Service FCs will be instantiated. The Platform Resolution will be instantiated in the INTER-FW, as the purpose of this FC better fits into the INTER-FW objectives.

The list of instantiated FCs of the Platform Interoperability FG is the following:

Functional Component	Description
Platform Access	This FC implements the functions needed for connecting to an IoT Platform and accessing their resources (specific discovery, lookup, data query, data subscription, device registry, etc.). This includes the use of the appropriate protocols and APIs that each platform exposes.
Architectural Components	• *MW2MW. Bridges*: They are in charge of specific details of implementation for each of the IoT Platforms supported. A bridge for each platform is expected. • *MW2MW. Resource Registry*: It keeps a registry of the different resources available at each IoT Platform. • *MW2MW. Resource Discovery*: It provides the service for discovery of resources available at each IoT Platform.. • *MW2MW. Routing and Roaming*: It performs the routing and roaming of a device among different IoT Platforms.

Table 18: List of functional components instantiated from the Reference Architecture for the Platform Access FC

Functional Component	Description
Platform Service	This FC is responsible for performing device and platform interactions, like querying data from different devices and platforms in a common way, mapping sensor data flows from a source to a destination, offering subscriptions to sensor data, etc.
Architectural Components	• *MW2MW. Message Broker*: Communication channel for the message flow to and from IoT Platforms and for communicating with the IPSM. • *MW2MW. Platform Request Manager*: It control the flow of messages between the API and the bridges and also controls the service requests, acting as a mediator between the API and the Platform Resolution FC. • *MW2MW. Request Manager*: It keeps a record of the requests received from the API to provide the necessary callbacks for data streams and providing replies to specific requests.

Table 19: List of functional components instantiated from the Reference Architecture for the Platform Service FC

3.2.2.6 Service Interoperability

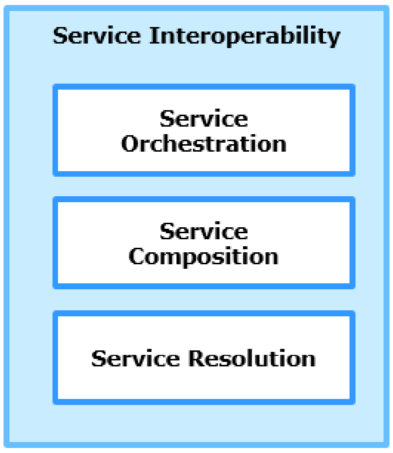


Figure 18: Functional Components of the Service Interoperability FG instantiated for INTER-LAYER concrete architecture

The Service Interoperability FG will be instantiated completely to support the Application and Service to Application and Service (AS2AS) interoperability through the definition and execution of new compound services that make use of already existing services in the underlying IoT Platforms.

The list of instantiated FCs of the Service Interoperability FG is the following:

Functional Component	Description
Service Resolution	This FC implements the functions needed for is responsible for the storage of what we call flows. A flow is a logical definition of a sequence of steps, each of which can be a service existing in an IoT Platform.
Architectural Components	• *AS2AS. Service Catalogue*: It stores a list of the registered services that are available at the connected IoT Platforms. • *AS2AS. Register Client*: It provides the registry of single services that are available at the connected IoT Platforms. • *AS2AS. Service Discovery*: It provides the discovery of the registered services that are available at the connected IoT Platforms. • *AS2AS. Flow Repository*: It stores the flows that have been defined for performing their further orchestration.

Table 20: List of functional components instantiated from the Reference Architecture for the Service Resolution FC

Functional Component	Description
Service Composition	This FC allows for the design of new compound services based on services that IoT Platforms expose.
Architectural Components	• *AS2AS. Modeller*: It allows a graphical definition of the compound services re-using already existing services in IoT Platforms.

Table 21: List of functional components instantiated from the Reference Architecture for the Service Composition FC

Functional Component	Description
Service Orchestration	This FC is responsible for the execution of the flows that are stored in the catalogue managed by the Service Resolution FC.
Architectural Components	• *AS2AS. Orchestrator*: It performs the execution of flows which are initiated by triggers (user request, IoT Platform event or alert, data received, etc.) that have been defined for each flow.

Table 22: List of functional components instantiated from the Reference Architecture for the Service Orchestration FC

3.2.2.7 Semantics

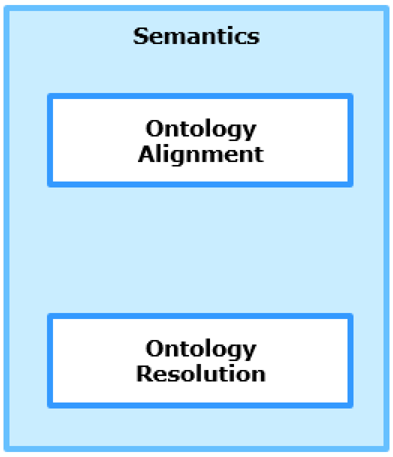


Figure 19: Functional Components of the Semantics FG instantiated for INTER-LAYER concrete architecture

The Semantics FG will be instantiated completely to addresses the challenges related to semantic interoperability of IoT Platforms. It will provide semantic translation capabilities to the rest of the interoperability layers of INTER-IoT.

The list of instantiated FCs of the Semantics FG is described below:

Functional Component	**Description**
Ontology Alignment	This FC was responsible for performing the alignment from a source data with an ontology to a target data with its own ontology. It makes the data translation between two ontologies, using the ontology definitions resolved by the Ontology Resolution FC
Architectural Components	• *AS2AS. Semantic Translation Channel*: It is responsible for, configuring a channel which all the translations of the same type go through. It calls the proper alignment applicator for the needed translation. • *AS2AS. Alignment Applicators*: It is responsible for performing the translation from input ontology A to output ontology B.

Table 23: List of functional components instantiated from the Reference Architecture for the Ontology Alignment FC

Functional Component	Description
Ontology Resolution	This FC responsible for managing the different ontologies used at the various IoT Platforms that are connected through INTER-IoT. The semantic knowledge is about being aware of the structure and meaning of the data. It stores these data descriptions and offers access to them for the Ontology Alignment FC
Architectural Components	• *AS2AS. IPSM Communication Infrastructure*: It is responsible for, given an RDF input flow, to direct the input messages to the appropriate semantic translation channel and after the translation has been performed, to direct the output messages to the RDF output flow. • *AS2AS. Channel Manager*: It is responsible for creating new semantic translation channel for any request of new translation from input ontology A to output ontology B. • *AS2AS. Alignment Repository*: It stores the registered ontologies for the connected platforms and the Global Ontology.

Table 24: List of functional components instantiated from the Reference Architecture for the Ontology Resolution FC

3.2.3 Functional Components traceability

For ensuring the right engineering process of development, we have elaborated a traceability matrix of the different Functional Components against the architectural components of INTER-LAYER.

Architectural Components	Management	Device Access			Device Interop.			Semantics		Service Interop.			Platform Interop.		Security	
	Configuration	Communication	Virtual entity	IoT Service	D2D Interoperability	IoT Plat. Interoperability	Network Interoperability	Ontology Resolution	Ontology Alignment	Service Resolution	Service Composition	Service Orchestration	Platform Access	Platform Service	Authentication	Authorisation
D2D. Gateway Configuration	✓															
D2D. Middleware controller															✓	✓
MW2MW. Bridges															✓	✓
AS2AS. Orchestrator															✓	✓
D2D. Access Network Controller		✓														
D2D. Protocol Controller		✓														
D2D. Virtual Entity Dispatcher			✓													
D2D. Registry				✓												
D2D. Discovery				✓												
D2D. Device Manager				✓												
D2D. Measure Storage				✓												
D2D. Rules Engine					✓											
N2N. Openflow connector						✓										
N2N. Routing						✓										
N2N. Host Tracking						✓										
N2N. Storage						✓										
N2N. Discovery						✓										
N2N. Switch Manager						✓										
D2D. Middleware Controller							✓									
MW2MW. Bridges													✓			
MW2MW. Resource Registry													✓			
MW2MW. Resource Discovery													✓			
MW2MW. Routing and Roaming													✓			
MW2MW. Message Broker														✓		
MW2MW. Platform Request Manager														✓		
MW2MW. Request Manager														✓		
AS2AS. Service Catalogue										✓						
AS2AS. Register Client										✓						
AS2AS. Service Discovery										✓						
AS2AS. Flow Repository										✓						
AS2AS. Modeller											✓					
AS2AS. Orchestrator												✓				
AS2AS. IPSM Communication Infrastructure								✓								
AS2AS. Channel Manager								✓								
AS2AS. Alignment Repository								✓								
AS2AS. Semantic Translation Channel									✓							
AS2AS. Alignment Applicators									✓							

Table 25: Traceability matrix of Functional Components of INTER-LAYER vs. INTER-LAYER architectural components

4 INTER-LAYER Components

4.1 Development and Demonstration Environments Setup

Including a chapter about the development environment, CI and tools (Jenkins, SonarQube, slack, etc.) and a chapter about the demo environment (including NEWAYS virtual machines etc.)

The content of this section provide a specific delta from the previous version of this deliverable (D3.1) and has been done in concordance with the solutions provided there. Also, this delta has followed the guidelines provided by architectural workpackages and requirements (D2.3 and D4.3). This section will present the improvement of the interoperability solutions presented at the beginning of WP3 and the implementation status, as well as the steps to follow in the next stages of implementation.

4.2 D2D solution

4.2.1 Refined Architecture

Due to some new insights the architecture changed slightly from the one proposed in D3.1 “Methods for Interoperability and Integration”. The new architecture of the gateway is given in figure 20.

Compared to the architecture proposed in D3.1 “Methods for Interoperability and Integration” some minor changes have been initiated. The most important change is the added connector module between the physical and virtual part of the gateway. This connector module controls the communication between the physical and virtual part of the gateway. It is worth noticing as well that during development we realized that many COTS IoT sensors and actuators have already their own dedicated protocol communication and supporting libraries to interact with them. This created a braking change in the gateway, where a Device Controller has to be created inside the Device Manager module.

Our gateway is suited to handle the current evolvement in IoT devices and sensors. For our new gateway we will define a slightly different approach which is also more aligned with the state of the art developments in IoT devices and sensors. Figure 20 shows the architecture of the gateway. In the physical architecture now there are 3 different ways to connect to IoT sensors and actuators:

1. The lowest level where a connection can take place is at the Access Network controller (A.N. controller). This is for very simple sensors or actuators that have no or very limited processing power and can be off-line for longer time periods. Sensors are commonly battery powered, actuators may have a power grid connection but usually also no or very limited processing

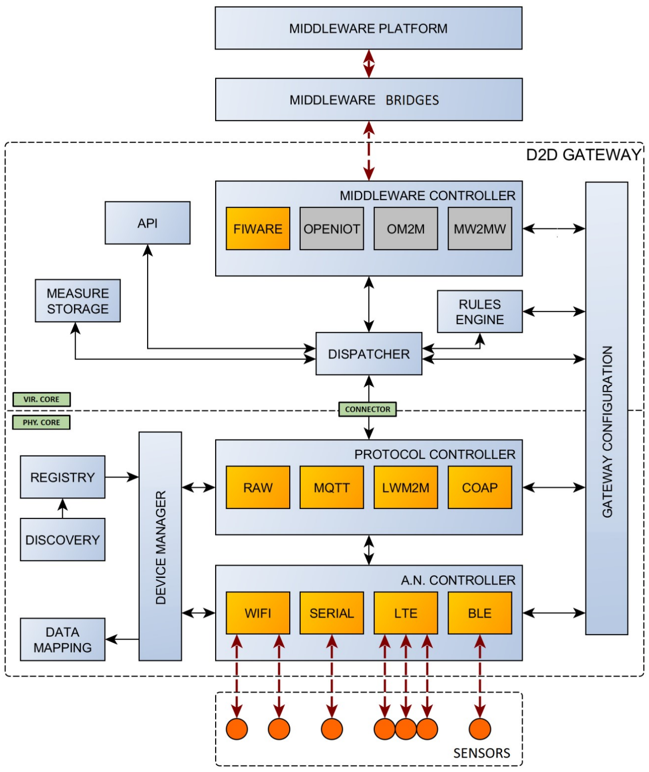


Figure 20: Gateway architecture

power. The A.N controller will do all routing and will serve as a master or access point for the sensors, afterwards the Protocol Controller will manipulate the data and create the messages to be sent to the virtual gateway.

2. At the middle level the dedicated sensors and actuators can connect, these are the COTS IoT devices. Usually these sensors and actuators have some dedicated communication protocol between the wireless sensor and some piece of electronics with a small processing core. They are capable of handling their own access controller and protocol controller and can be connected through a dedicated extension module implementing a Device Controller.

3. At the highest level we find the COTS IoT systems, they manage their own gateway and protocols and A.N. controllers. These systems can be connected via the connector directly to the Virtual Gateway of the Inter-IoT system. In this case, the COTS system software has to be modified to add a specific connection capabilities and implementation of the reference Physical-Virtual Gateway communication protocol.

4.2.2 Components

The physical and virtual gateway implementation share a common base and runtime code. Both are based in an OSGi framework wrapper (the OSGi framework has to be R4 compliant) with a customized bootstrap and initiation routines. This framework first load the third party libraries, then the core components and afterwards the extension modules. Then a routine to initiation all the modules starts, and the Physical and Virtual Core take the main thread to control the gateway. In figure 21 and figure 22 a schema and summary of the OSGi Framework, wrapper and components is shown.

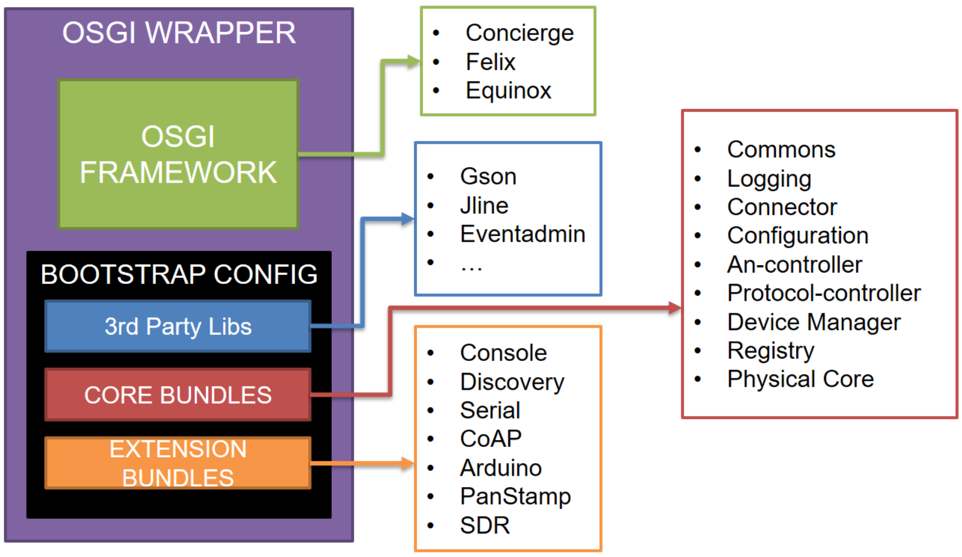

OSGI WRAPPER
OSGI FRAMEWORK
BOOTSTRAP CONFIG
3rd Party Libs
CORE BUNDLES
EXTENSION BUNDLES
• Concierge • Felix • Equinox
• Gson • Jline • Eventadmin • …
• Console • Discovery • Serial • CoAP • Arduino • PanStamp • SDR
• Commons • Logging • Connector • Configuration • An-controller • Protocol-controller • Device Manager • Registry • Physical Core

Figure 21: Physical Gateway components

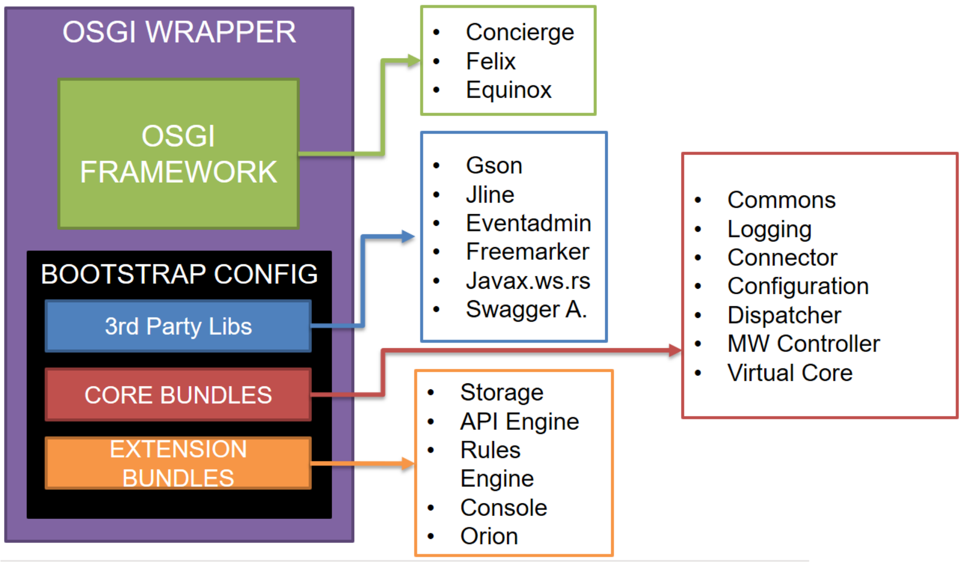

OSGI WRAPPER
OSGI FRAMEWORK
BOOTSTRAP CONFIG
3rd Party Libs
CORE BUNDLES
EXTENSION BUNDLES
• Concierge • Felix • Equinox
• Gson • Jline • Eventadmin • Freemarker • Javax.ws.rs • Swagger A.
• Storage • API Engine • Rules Engine • Console • Orion
• Commons • Logging • Connector • Configuration • Dispatcher • MW Controller • Virtual Core

Figure 22: Virtual Gateway components

4.2.3 Implementation Status

Although the development of the D2D Gateway is iterative, the following table summarizes the state of development as of writing this deliverable:

Component	GW. Type	Mod. Type	Development Phase		
			Design	Implementation	Test
Commons	Common	Core	✓	✓	✓
Logging	Common	Core	✓	✓	✓
Connector	Common	Core	✓	✓	80%
Configuration	Common	Core	✓	✓	✓
Console	Common	Extension	✓	✓	✓
Physical Core	Physical	Core	✓	80%	50%
Registry	Physical	Core	✓	✓	75%
Device Manager	Physical	Core	✓	80%	60%
Protocol Controller	Physical	Core	✓	80%	50%
A.N. Controller	Physical	Core	✓	80%	50%
Discovery	Physical	Extension	✓	20%	-
Serial	Physical	Extension	✓	80%	50%
CoAP	Physical	Extension	✓	80%	50%
Arduino	Physical	Extension	✓	75%	50%
PanStamp	Physical	Extension	✓	90%	75%
SDR	Physical	Extension	60%	-	-
Virtual Core	Virtual	Core	✓	80%	60%
Dispatcher	Virtual	Core	✓	✓	75%
MW Controller	Virtual	Core	✓	✓	✓
Storage	Virtual	Extension	✓	50%	-
API Engine	Virtual	Extension	✓	✓	✓
Rules Engine	Virtual	Extension	90%	-	-
Orion MW	Virtual	Extension	✓	75%	50%
UaaL MW	Virtual	Extension	50%	-	-

Table 26: D2D Gateway Component implementation status

4.2.4 API (and Extensibility)

In the INTER-IoT device-to-device interoperability gateway there are four different APIs:

- **Gateway CLI:** The gateway console extension provides a Command-Line Interface (CLI) to control the physical or virtual gateway instance.

- **Gateway REST API module:** REST API exposed by the virtual gateway API Engine extension module to interact with the virtual and physical gateway.
- **Physical/Virtual Communication API:** Messages exchanged between the physical and virtual through the connector module.
- **Programmatic API:** Libraries and interfaces needed to develop new extension modules for the gateway.

4.2.4.1 Gateway CLI

The gateway CLI is a basic command line interface to interact with the gateway instance that has been deployed. This API is provided by the gateway console extension module, so it can be optionally included in deployment. This module provides a gateway service to the core, so it is loaded and executed by the bootstrap runtime in a separate thread, once started it will capture the standard input and output of the system.

There is also an command interface exposed in the commons module that any other module and extension can implement and expose in order to provide more control and functionalities to new modules. Classes implementing this interface will be detected and automatically loaded in the console module.

The command list definitions for both the physical and virtual part of the gateway can be found in the following Wiki page: `https://git.inter-iot.eu/Inter-IoT/gateway/wiki/CLI+commands`.

4.2.4.2 Gateway REST API module

The gateway API engine module is loaded also as a gateway service as an extension. Once the bootstrap runtime loads the API engine module it will search for classes exposed by other modules (core and extension) that implement an interface provided by the commons module and automatically provide new endpoints.

Once the API engine is loaded and started, it will run an embedded web server in a separate thread and provide the endpoints and an automatically generated swagger file to document the REST API.

The REST API definition can be found in the following Wiki page: `https://git.inter-iot.eu/Inter-IoT/gateway/wiki/API`.

4.2.4.3 Physical/Virtual Communication API

Between the physical and virtual parts of the gateway there is a communication protocol consisting in a json message exchanged through a websocket connector. This allows to have different implementations of physical gateways (i.e. more flexibility and implementation options) if the communication fits with the reference implementation for a given version. A list of the exchanged message in the json protocol can be found in the Annex.

The specification of the communication protocol between the physical and virtual part of the gateway can be found in the following Wiki page: https://git.inter-iot.eu/Inter-IoT/gateway/wiki/Physical-Virtual+Communication+Protocol.

4.2.4.4 Programmatic API

All the gateway modules are developed as OSGi bundles, in order to extend the gateway providing new modules and extensions they only need to implement the correct interfaces depending of the type of extension (note that this is subject to change in the future):

- **Gateway Service:** Register a OSGi service that implements *CoreService.class* from the *Commons* module.
- **Device Controller:** Register a OSGi service that implements *DeviceControllerService.class* from the *Device Manager* module. Additionally add the *controller-key* property to the service registered.
- **Protocol Controller:** Register a OSGi service that implements *ProtomoduleService.class* from the *Protocol Controller* module. Additionally register the protocol instance creator class in the *ProtocontrollerService* service.
- **Access Network Controller:** Register a OSGi service that implements *ANModule.class* from the *Access Network Controller* module. Additionally register the access network instance creator class in the *ANController* service.
- **MiddleWare Controller:** Register the MiddleWare controller class implementing *MWModule.class* from the *MiddleWare Controller* module in the *MWRegistryService* service.

Additionally, any module can also register new commands for the *console* module and API endpoints as already explained in section 4.2.4.1 and 4.2.4.2.

4.2.5 Code and Documentation

The D2D Gateway software components, binaries and documentation can be found in the following sites:

- **Code and some draft documentation:** Can be found in the INTER-IoT git repository: http://git.inter-iot.eu/Inter-IoT/gateway.
- **Latest compiled artifacts:** Can be found in the INTER-IoT nexus repository: http://nexus.inter-iot.eu.
- **Latest Virtual Gateway docker image:** Can be pulled from the INTER-IoT docker repository: docker.inter-iot.eu/vgateway
- **Latest Implemented API definition:** Can be found in the INTER-IoT azure cloud machine: http://vmplsp03.westeurope.cloudapp.azure.com:8080/

Other tools are used internally for development purposes, such as Jenkins[63] for continuous integration, SonarQube[64] for code inspection and Slack[65] for communication.

4.2.6 Demo

An example demo setup was implemented to be shown in the mid-term review of this project. This setup is shown in Figure 23.

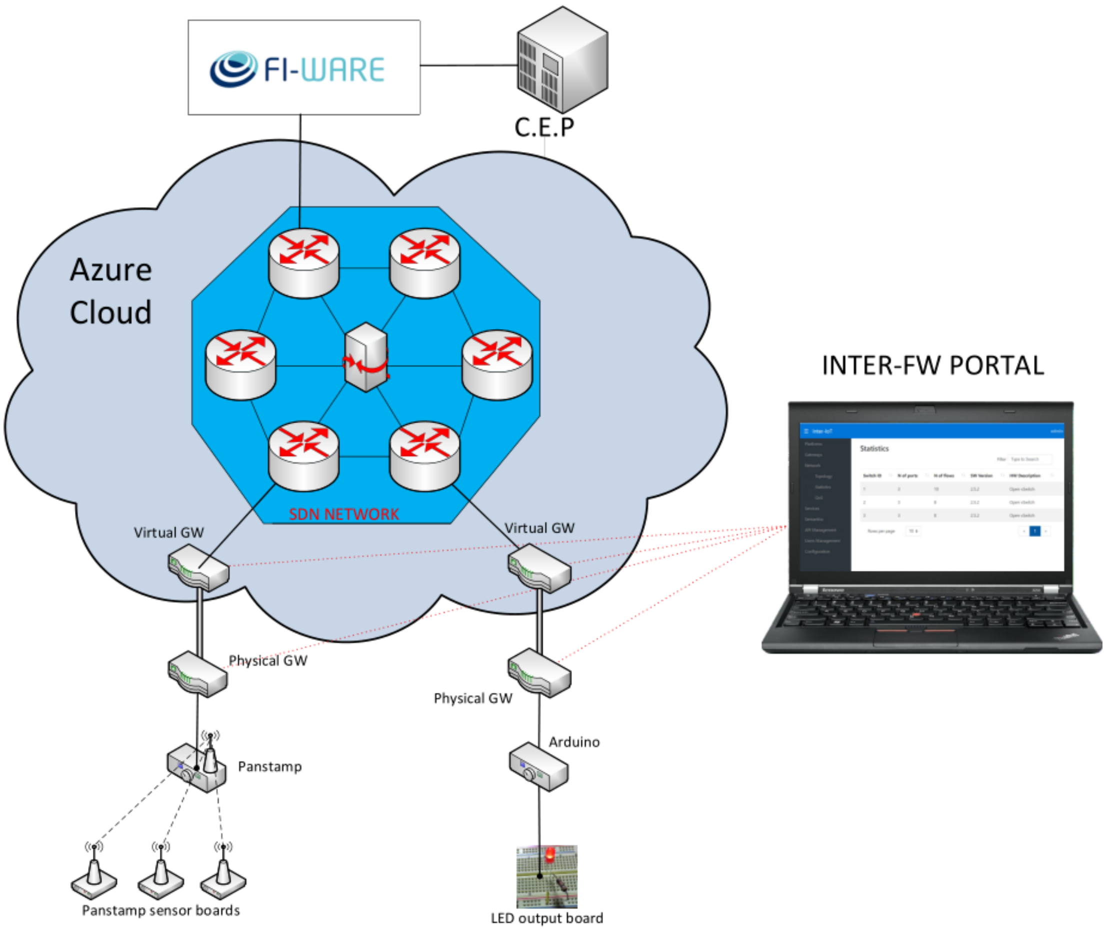


Figure 23: D2D Gateway demo

The setup consisted in a deployment of two physical gateways deployed in two Raspberry Pi units, one of them connected to PanStamp temperature and humidity sensors and the other connected to an Arduino board controlling some LEDs as actuators. Both of the physical gateways are connected to their virtual counterpart, dockerised in an Azure machine. These virtual gateway are controlled and managed through the API and Inter-FW. Finally, this virtual gateways are connected through an SDN network to the same IoT Middleware (Orion from Fiware) with a Complex Event Processor configured with simple rules.

[63] https://jenkins.inter-iot.eu/
[64] https://sonar.inter-iot.eu/
[65] https://inter-iot.slack.com/

The demo consisted in updating the actuators state based on the reading of the sensors, while demonstrating how the data flows through all the setup. This showed the versatility of the gateway and the different device controller modules to achieve D2D interoperability, while the an API exposed by the virtual gateway (without bothering the low-level physical part) and the connection through the IoT SDN network from INTER-IoT Network-to-network interoperability layer to the middleware components.

There was a second part of the demo, where a third virtual gateway was deployed but this time connected to the RINICOM Prime-IoT Gateway (the first commercial implementation of a physical gateway able to connect to the INTER-IoT virtual gateway[66]) and the actuators where controlled this time based on the readings of the new gateway, showing the possibility of interchanging the Physical Gateway if the reference communication protocol is implemented.

4.3 N2N solution

4.3.1 Refined Architecture

No additional changes have been implemented in the architecture at the moment. The creation of a SDN has been performed by means of the installation and configuration of virtual switches with Open vSwitch technologies and its connection with the Ryu controller (figure 24). The development of modules at the top of the Ryu controller, following the specified architecture, is ongoing. In the next section we will describe the progress work and the future objectives.

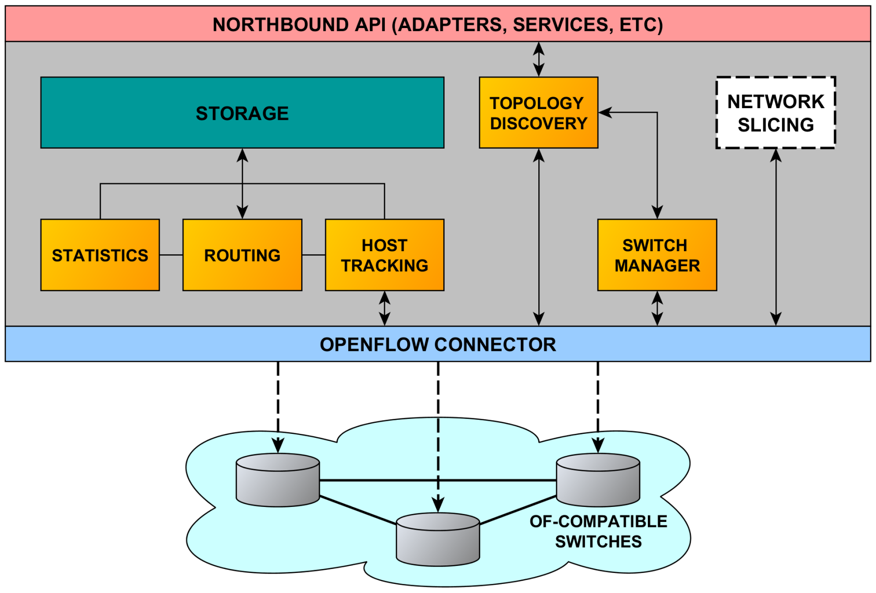


Figure 24: Network interoperability architecture

[66]http://rinicare.com/products/remote-monitoring/prime

4.3.2 Implementation Status

Our SDN solution main objective is to provide seamless integration between virtual elements within our INTER-IoT deployment. The implementation of the solution includes the creation of this software-defined network with a controller adapted for the IoT deployments. The status of this task includes:

- Install and configuration of virtual switches using Open vSwitch technology. This brings the possibility of create custom topologies (they have been created for testing purposes) with the specific characteristics of each deployment including redundancy of links or interconnection of different environments.
- Development of modules within the controller: the creation of a simple switch management application to insert rules in the switches attending the flows and the QoS parameters, the customization of topology module to show switches, links and hosts connected by the network and customization of rest API to obtain information about the switches.
- Development of a command line interface (CLI) to introduce direct commands to the controller.
- Inter-connection of virtual gateways thought this network. And connection of the virtual gateway with an IoT platform (Orion).
- Development of automatic script to run and stop the solution.
- Definition of the REST API in Swagger to be used by higher levels of INTER-IoT
- Development of a GUI embedded in INTER-FW with modules of Topology, Information and QoS to facilitate the visualization, control and configuration of the network.

Moreover, our following steps will include the improvement of both modules; information/statistics and QoS, for more automation and dynamism when it comes to acting on the network. Also, the implementation of mechanisms for network slicing and offloading are being study to be developed in the next stages of the project.

4.3.3 API (and Extensibility)

The design of an API to access the information and resources within the SDN has been also carried out as a part of the network to network solution. For this API, we took in account the possibilities that the REST interface provided by Ryu offered us and we gather the most interesting one to be specified in Swagger and to be used for other components of INTER-IoT as INTER-FW. The API is available through the WSO2[67] portal.

A description of these APIs is the following:

- **Switches**
  - /stats/switches GET
  - /stats/desc/id GET
- **Flows**
  - /stats/flow/id GET

[67] https://vmplsp02.westeurope.cloudapp.azure.com:9443/publisher/site/pages/login.jag

  - /stats/flowentry/add POST
  - /stats/flowentry/modify POST
  - /stats/flowentry/delete POST

- **Ports**
  - /stats/port/id GET
  - /stats/port/id/port GET
  - /stats/portdesc/modify POST
- **Tables**
  - /stats/table/id GET
  - /stats/tablefeatures/id GET
- **Roles**
  - /stats/role POST
- **QoS Queues**
  - /qos/queue/status/id GET
  - /qos/queue/id GET
  - /qos/queue/id POST
  - /qos/queue/id DELETE
- **QoS Rules**
  - /qos/rules/id GET
  - /qos/rules/id POST
  - /qos/rules/id DELETE
- **QoS Meters**
  - /qos/meter/id GET
  - /qos/meter/id POST
  - /qos/meter/id DELETE

In the previous list we observe the diverse resources we can request information an manipulate with respect to the network. A brief definition of each one is the following:

- Switches; the operation performed over the switches affect the configuration stored in each of them. We can obtain information about number of ports, manufacturer, software, etc.
- Flows; is the information stored within the switches following the OpenFlow protocol. These flows are stored in tables and its field are; match, statistics and actions.
- Table; a collection of flows. By default, there is always a flow installed in the first table of each switch indicating the routing of the new processed packets to the controller.

- Ports; virtual ports configured in the switch. The characteristics of these ports can be consulted and modified.
- Queues; associated to a port, define a priority treatment depending on the configuration, also, they could define the rate of the packets.
- Rules; rules to be implemented in the queues and define the aforementioned treatment.
- Meters; switch element which measures and controls the ingress rate of packets which is the rate of packets prior to the output.

Moreover, the extensibility at network layer could be achieved by two means:

- Including new elements into the network: usually new virtual switches that connects to other elements or networks and work as a dawn between the element we want to connect and our actual virtual network. These new switches could be directly created within Open vSwitch, creating new interfaces and bridges where we connect the new elements or connecting a different virtual (or physical) legacy switch provided by a different technology.
- Or including a whole new SDN with different controller: in this case we want to include another whole network with its controller and interconnect this to our network. In this case as the direct connection between several controller is something no optimum we should make use of a hypervisor that acts as another layer of network abstraction to manage or orchestrate all controllers that manage the networks. But this approach has not been yet contemplated.

4.3.4 Code and Documentation

The code can be divided into two main blocks:

- **front-end** modules
- **back-end** modules

The back-end modules comprise all modules related to SDN and acting at the network level. This includes mostly the SDN framework Ryu, but also APIs, as well as our custom INTER-IOT Ryu applications. The front-end modules are designed to enable the back-end modules to their full capacity. A graphical user interface (GUI) allows edition of the SDN network and QoS settings. This interface can be run locally, but should be used within the INTER-FW portal. In some cases, such as for device to device interoperability, the framework is not deployed. Therefore, a CLI has been designed to configure the SDN network and QoS parameters.

Internal documentation describes more precisely these modules, and a summary is presented below. Please note that this documentation is often subject to changes. The full code and documentation can be found on the GitHub repository[68].

4.3.4.1 Back-end modules

- **Ryu** Install Ryu with `pip install ryu`. Install a custom made application by using the command `python setup.py install` in the main directory of Ryu. Start any app with `ryu-manager`

[68] `https://git.inter-iot.eu/Inter-IoT/sdn`

`yourapp` assuming your app is installed, otherwise the path must be specified `ryu-manager path/to/yourapp.py`.

- **INTER-IOT Simple Switch**. INTER-IOT Simple Switch is a custom made switch app that automatically handles flows. This app is located in *ryu/ryu/app/InterIoT-simple_switch13.py* and is based on the provided example *simple_switch13.py*. The example switch supports OpenFlow1.3 and adds flows according to the first route found. Upon packet reception, if the destination is not known in the MAC address table, the network is flooded until the destination is found. Then the switch learn which ports correspond to this destination and the appropriate flow is added. INTER-IOT Simple Switch also integrates a QoS REST API, and a websocket for GUI. See the previous section for an API description.

- **GUI and QoS interface.** All corresponding apps for QoS are called upon InterIoT simple switch start (rest_qos and rest_conf_switch). A custom GUI app is called in parallel of the Inter-IoT switch in order to start the graphical interface. The file is located at */app/gui_topology/gui_topology.py*. The following apps are then called in order to ensure a good functioning of GUI *ryu.app.rest_topology, ryu.app.ws_topology and ryu.app.ofctl_rest*.

4.3.4.2 Front-end modules

- **CLI** The CLI is located in the folder */cli* and is written in python. Running the command *sh CLI.sh* in administrator mode will launch the command line interface. Python2.7 is required. This command line interface will run and understand commands described in files located in the *command* folder, and whose name starts with *cmd_*. Integrated commands are:

  - **commands**: displays all the commands
  - **exit**: exits CLI
  - **mininet**: run Mininet (for test purposes)
  - **controller**: start predefined SDN controllers (SDN controllers start scripts can be added in the folder *commands/pkg_controller*)
  - **switch**: allows to configure switches, set Openflow versions, get configuration and flow-tables of the switch
  - **qos**: allows to set rules, set queues, get rules and get queues of a QoS switch. This command also allows to set the OVSDB address, and set the controller's port and address.

  For future implementation, the QoS command should allow the user to edit and modify meters. Some security has been embedded at this level, to avoid intentional of accidental command injection.

- **GUI assets for local use** GUI assets are located in *ryu/ryu/app/gui_topology/html*. All tabs are located in the directory */tabs* and combines both HTML and javascript in order to ensure a correct web display.

- **GUI assets for INTER-FW** See INTER-FW.

4.3.5 Demo

As in the previous review there was not a specific demo for Network, we built demos to show, when needed, the characteristics of the network solutions. These demos are the following:

- Demo Device + Network + Framework: it shows the connection of virtual gateways through the network and together with a Middleware component (in this case Orion), also it shows the graphical tool developed within the FW to show topology and information from the network and QoS configuration. There graphical tool will be used also for the following demos.

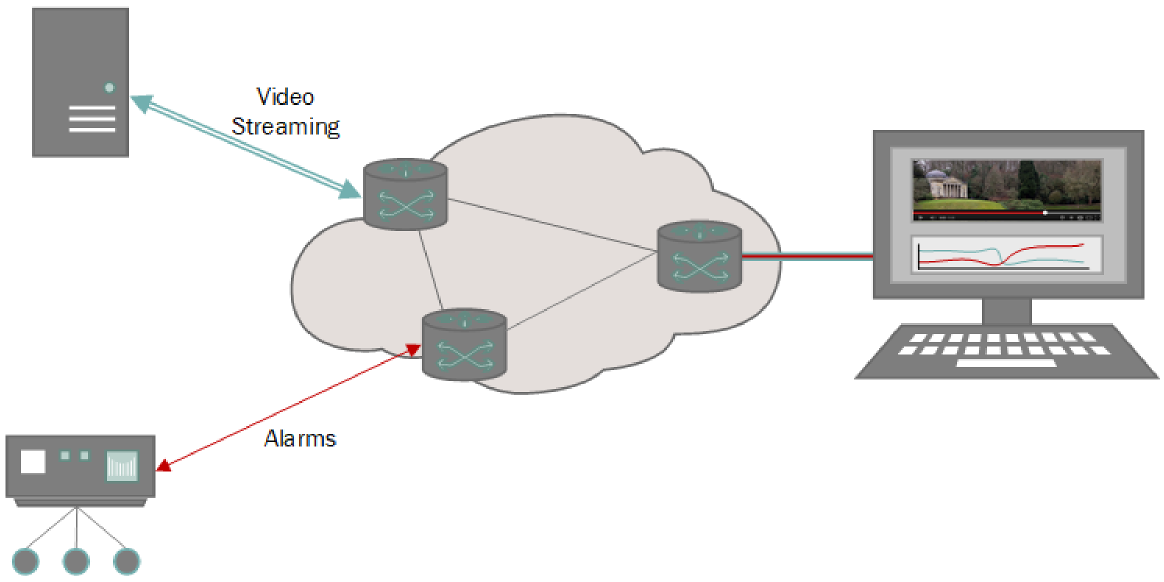


Figure 25: Example of setup for network demo

- Demo QoS-Priority: The demo is based in the set up of a video server that streams a high quality video and a gateway that send alarms when needed. An example of this architecture can be seen in Figure 25. In the first phase of this demo the server is streaming a video and the network link is configured without QoS so no priority is set. When the gateways send an alarm, also without QoS configuration, a best effort method is applied so the alarm most probably could not arrive due to the video streaming is taking the whole bandwidth. In the second phase, the QoS application is running so the priorities to the links are set, higher priority for the alarms and lower for the video. Thus, when the video is streaming and an alarm is triggered the video will drop its quality or even freeze to allow the priority alarm to arrive to its destiny taking part of the link bandwidth of the video.

4.4 MW2MW solution

4.4.1 Refined Architecture

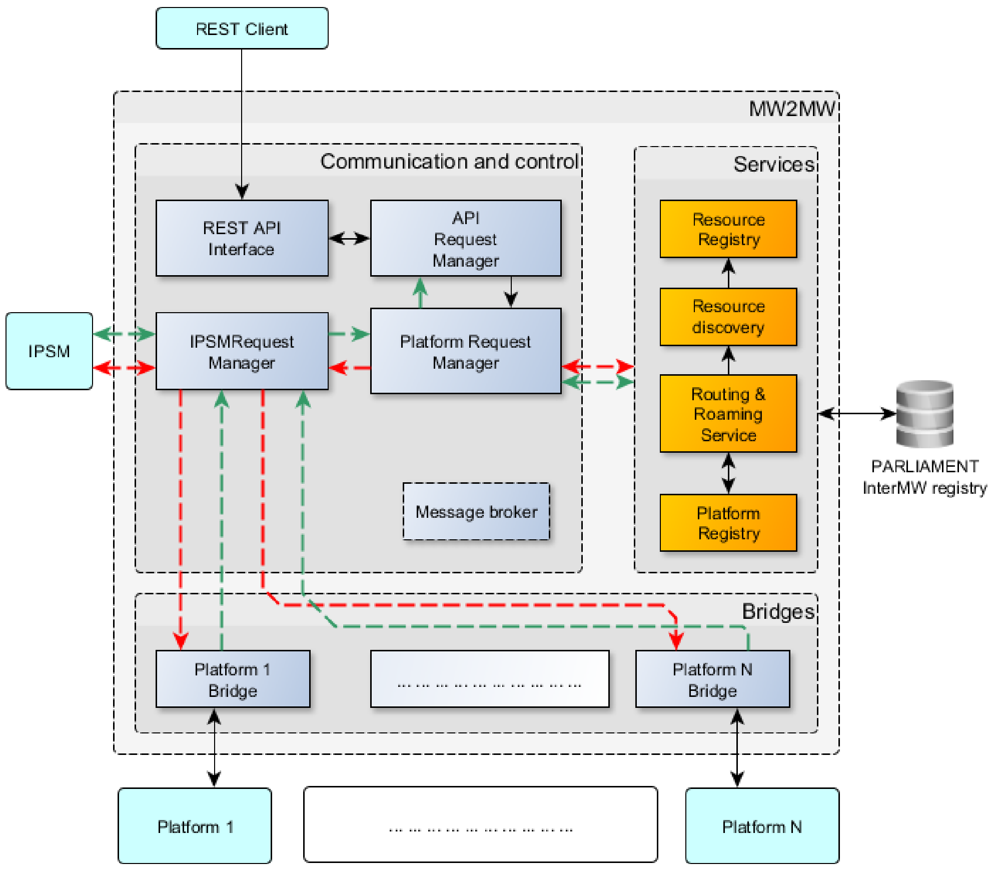


Figure 26: INTERMW architecture

The core functionality of INTERMW is facilitation of interoperability among IoT Middleware platforms, as well as the provision of a common abstraction layer to provide access to platform's features and information. Further developing these concepts, the architecture of INTERMW has evolved substantially since D3.1, as seen in picture 26. Two important design choices have been taken:

- *Common ontology.* INTERMW uses the common INTER-IoT ontology (GOIoTP) to represent all messages routed through the system. The result of work described in section 4.4.2.3 and Deliverable D4.2 has been extensively used in the implementation of INTERMW components. Thus, the InterIoT JSON-LD message structure is in the core of INTERMW.
- *Middleware abstraction.* Unifies work with IoT platforms through extensible middleware services. Common abstraction layer unifies the view on all interconnected platforms, devices and services. It does not matter what device belongs to what platform, or what service is in which platform. Implementation of a REST API interface further extends the usability of this abstraction layer by exposing this functionality through a widely used technology.

In order to implement the above mentioned design decisions, a REST API interface has been developed northbound of the API Request Manager. This REST API interface takes care of all interfacing of INTERMW with application layer components.

The logical concept of the Data Flow Manager, introduced in D3.1, has been further elaborated. The concept has been implemented through the introduction of "conversations" and consequently of data flow control. A group of messages belongs to the same conversation, if they share the same conversation identifier (ConversationID). For example, in a single conversation we would typically have first a message, that subscribes to a particular group of sensors, and then messages with sensor readings, going upstream from the sensors to the application. Subscriptions in INTERMW are also tracked by ConversationID. Technically, data flows are implemented through a message broker.

The integration with IPSM is achieved through the IPSM Request Manager component that orchestrates the communication between IPSM and INTERMW components (*Bridges*, *Platform Request Manager*).

A triplestore database provides persistence and advanced querying mechanisms for the Services subsystem. All registry-related requirements that need persistence or querying support are indirectly implemented through this triplestore. Currently, *Platform Registry*, *Resource Registry* and *Subscriptions Registry* use functionalities provided by this component. The Parliament triplestore database is used, as described in section 2.3.5.

Syntactic conversion and semantic translation of platform-specific messages have been de-coupled. Implementation of a Bridge to provide interoperability for a new platform means, for a bridge developer, implementation of a communication protocol with the platform and syntactic translation of the message between platform-specific format and INTER-IoT JSON-LD. Definition of rules for semantic translation (alignments) is still necessary, but not at the Bridge level. That part of the process if fully implemented in IPSM.

4.4.2 Components

4.4.2.1 REST Interface

INTERMW provides REST API interface that can be used by client applications. As already explained in Section 2.4.3 and D4.3 (Section 4.3 API Design), Swagger (OpenAPI) REST API definition language has been selected for definition of all REST API interfaces throughout INTER-Layer component. To facilitate the development of INTERMW and keep the definitions up-to-date, the Swagger annotation library for Java has been used to document the INTERMW REST API.

INTERMW is deployed as a webapp with the exposed REST API interface. Although in principle it can run in any web servlet container, Jetty [69] is used for development and demonstration purposes. The deployment platform will be selected in the final phase of the project.

[69] https://www.eclipse.org/jetty/

4.4.2.2 Data Flow Management

As described in the architecture, the data flow management concept has been elaborated through the definition of a "conversation" concept and of an abstract broker client interface. On a conceptual level, INTERMW components communicate through channels (queues) that connect two components. The full list of those channels is provided in table 27.

Conversations get initiated at the API Request Manager (ARM) level. This means that, for each new group of messages, ARM creates a new *Conversation Id* and returns it to the caller. The caller than uses this identifier in two scenarios: when sending requests for change of the original request in that conversation (for example, unsubscribe to an original subscribe request) or when matching responses from the middleware to an original request (for example, receiving observations - sensor readings- for a subscribe request). As consequence, components such as Bridges or Services must keep track of active conversations an match responses to a correct conversation. We could also think of conversations as permanent "sessions" or "channels" between API clients and middleware components.

The actual messaging has to be instantiated through implementation of message broker clients. Basically, any message broker that provides the basic messaging functionality can be used. At the moment of writing of this deliverable, clients for the following message brokers have been implemented: Apache ActiveMQ [70], Apache kafka [71], generic MQTT client [72], Vortex OpenSplice [73] and RabbitMQ [74].

Notes to table 27:

1. The coding convention of topic names between INTERMW and IPSM is as follows. $[mw\text{-}ipsm|ipsm\text{-}mw]$ is the direction of the flow (source to destination), which means from INTERMW (MW) to IPSM or the opposite. The second part, $[interiot\text{-}format|platform\text{-}format]$, designates the format that is posted to the topic: InterIotMsg format or platform specific. The last part is the platform type (FIWARE, universAAL, etc.).

2. These are messages sent between bridges and IPSRM. The suffix is the platform identifier. Messages flowing through these channels use platform-specific semantics. Each bridge instance is subscribed to its own topic and published data to a specific topic as well. Bridges do never share topics.

[70] http://activemq.apache.org/

[71] https://kafka.apache.org/

[72] http://mqtt.org/

[73] http://www.prismtech.com/vortex/vortex-opensplice

[74] https://www.rabbitmq.com/

Topic	Description
arm_prm	API Request Manager messages to Platform request manager. ARM relates API requests with client identifiers and forwards messages to PRM. ARM communicates only with the PRM.
prm_arm	Platform request manager messages to the API Request Manager. API request manager relates message responses, through conversation identifiers, with API client identifiers and invokes client callback messages.
prm_srm	Platform request manager to Services. PRM sends all requests related to registry and discovery messages to SRM. The invocation of Services depends on message types.
srm_prm	Services response to PRM. In principle, the response contains a set of attributes from the registry. It may also contain a discovery request for platforms, which means that PRM will re-route it southbound, towards the bridges.
prm_ipsmrm	All southbound flow towards platforms passes through the IPSM Request Manager (IPSMRM), which takes care of message translation (by invocation of IPSM) and forwarding to bridges.
ipsmrm_prm	IPSM Request Manager to Platform request manager messages are already translated responses from platforms.
mw-ipsm-interiot-format-*{Platform Type}*	Messages in InterIotMsg format sent for translation (see note 1). These are messages to be sent to platforms after they have been translated by IPSM.
ipsm-mw-platform-format-*{Platform Type}*	Messages in IoT platform-specific formant returned from IPSM (see note 1). These are messages ready to be sent to platforms.
mw-ipsm-platform-format-*{Platform Type}*	Messages in IoT platform-specific format sent for translation (see note 1). These are messages sent by platforms to INTERMW.
ipsm-mw-interiot-format-*{Platform Type}*	Messages in InterIotMsg format translated by IPSM (see note 1). These are messages sent by platforms to INTERMW and already translated by IPSM.
ipsmrm_bridge_*{Platform Id}*	Message flow from IPSRM to a bridge (see note 2).
bridge_ipsmrm_*{Platform Id}*	Message flow from bridge to IPSRM (see note 2).
rest_api_*{client Id}*	Topic for client callback messages (REST). There is one queue per REST API client. Clients use pull method to get messages form the response queue.
error	Global INTERMW error topic. In case of any errors, messages get posted to this topic.

Table 27: INTERMW inter-component communication topics

4.4.2.3 Messages

The core of INTERMW are messages expressed in the common ontology. The same message structure is used by IPSM, whose role is to semantically translate messages to and from the common INTER-IoT ontology. Each message is composed of message metadata and payload.

All message routing within INTERMW is done based on information, found in the metadata message part. In the case that the payload contains essential information for routing of the message, this is information first transferred to the metadata of the message and only then used in message processing and routing within INTERMW. Within the message metadata the following information for processing and routing messages is provided: message types, destination platform(s), source platform, message id and conversation id (see also section 4.4.6 for more details).

Although implementation of the Message class allows for a message to have multiple types, in practice a message has one or two types. In these cases, RESPONSE is the second message type. This reflects the message flow mechanism: some message types are request, which expect responses from INTERMW components. The table 28 lists all message types, their usage and typical data flow scenarios.

Internally, the Message implementation used the JSON-LD format, which is divided into two RDF graphs: InterIoTMsg[75]:Metadata and InterIoTMsg:Payload. Metadata graph uses a custom INTER-MW RDF vocabulary in order to store information used by INTER-MW to route and manage messages. Metadata properties are managed by INTER-MW libraries.

Technically, metadata graph must contain a single instance of a InterIoTMsg:Meta class, called a "metadata instance". URI identifier of the instance can be any URI, but the identifiers generated by INTER-IoT are from the InterIoTMsg namespace. The instance can have any number of types, as long as it is at least of type InterIoTMsg:meta. Other types used by INTER-MW are described in table 28. The URIs of those types are prefixed by InterIoTMsg (e.g. "InterIoTMsg:SUBSCRIBE").

The fields (properties) of the metadata instance are as follows:

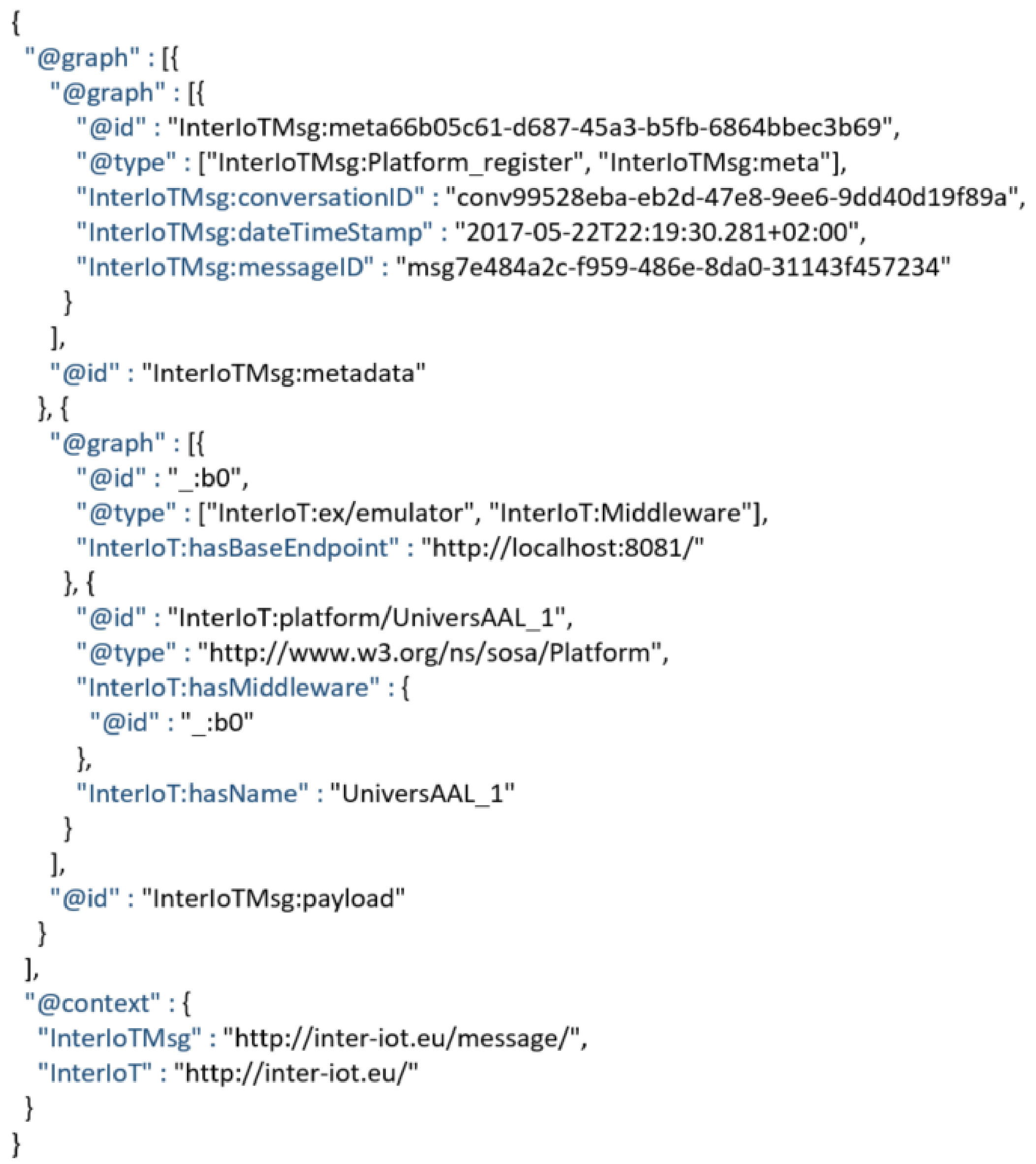
- InterIoTMsg:messageID - a string identifier of a message
- InterIoTMsg:conversationID - a string identifier of a conversation, of which this message is a part of
- InterIoTMsg:dateTimeStamp - a time information in xsd:datetimestamp format denoting time of creation of the message
- InterIoTMsg:query - Discovery service query
- InterIoTMsg:errCode, InterIoTMsg:errDesc, InterIoTMsg:errOriginalMsg, InterIoTMsg:errStackTrace - description of an error, optionally including another message that caused the original error, and a Java exception stack trace serialized in RDF
- InterIoTMsg:senderPlatformID, InterIoTMsg:receiverPlatformID - identifiers of platforms that take part in the communication
- InterIoTMsg:status - General status information, usually used to acknowledge a message (STATUS = "OK")

[75]Prefix: `http://inter-iot.eu/message/`

Message Type	Description
PLATFORM REGISTER	This request creates a new *Platform* entry in the *Registry*, crates a new *Bridge* instance for the platform and sets up the bridge.
PLATFORM UNREGISTER	The opposite from *PLATFORM REGISTER*: Platform information with corresponding devices and subscriptions data is removed from the *Registry*, clean up operations performed at the *Bridge* level and finally, the Bridge itself is destroyed.
SUBSCRIBE	Subscription request to a set of devices, defined with a SPARQL query. The list of devices and platforms they belong to is created at the *Services* - *Registry* level and this information is sent to the platforms: $PRM \Rightarrow IPSMRM \Rightarrow IPSM \Rightarrow IPSMRM \Rightarrow Bridge(s)$. A new conversation, with a corresponding identifier is created and maintained at the ARM level.
UNSUBSCRIBE	The opposite of *SUBSCRIBE*. The *Bridge* informs the IoT platform about the end of the subscription and the ARM removes the active conversation.
QUERY	Similar to *SUBSCRIBE*, but executes the query only once and returns current sensor readings.
DISCOVERY	A discovery mechanism that returns a set of resources (platforms, devices, services) that fulfill a condition defined by a SPARQL filter.
OBSERVATION	This message contains a set of observations (sensor readings) form a device. Typically, multiple messages of this type are generated as response to a *SUBSCRIBE* request, each with a single observation. In principle, INTERMW expects *OBSERVATION* messages as long as the subscription is active.
THING UPDATE	Changes the status of a thing/device.
RESPONSE	Response generated as fulfillment of a request. This type is always combined with one of the requesting types.
ERROR	The error message is sent by any INTERMW component to the global INTERMW error handler (currently implemented as a error topic). It wraps the original message that caused the error, along with additional information useful for debugging and error handling.

Table 28: INTERMW Message types

The payload graph, in principle, can contain any RDF code. INTER-MW requires the payload to use the common ontology, and utilizes IPSM to translate to and from it. Bridges use the syntactic translation components to transform messages from platforms into RDF, and put them in the payload.



```
{
 "@graph" : [{
   "@graph" : [{
     "@id" : "InterIoTMsg:meta66b05c61-d687-45a3-b5fb-6864bbec3b69",
     "@type" : ["InterIoTMsg:Platform_register", "InterIoTMsg:meta"],
     "InterIoTMsg:conversationID" : "conv99528eba-eb2d-47e8-9ee6-9dd40d19f89a",
     "InterIoTMsg:dateTimeStamp" : "2017-05-22T22:19:30.281+02:00",
     "InterIoTMsg:messageID" : "msg7e484a2c-f959-486e-8da0-31143f457234"
    }
   ],
   "@id" : "InterIoTMsg:metadata"
  }, {
   "@graph" : [{
     "@id" : "_:b0",
     "@type" : ["InterIoT:ex/emulator", "InterIoT:Middleware"],
     "InterIoT:hasBaseEndpoint" : "http://localhost:8081/"
    }, {
     "@id" : "InterIoT:platform/UniversAAL_1",
     "@type" : "http://www.w3.org/ns/sosa/Platform",
     "InterIoT:hasMiddleware" : {
       "@id" : "_:b0"
     },
     "InterIoT:hasName" : "UniversAAL_1"
    }
   ],
   "@id" : "InterIoTMsg:payload"
  }
 ],
 "@context" : {
   "InterIoTMsg" : "http://inter-iot.eu/message/",
   "InterIoT" : "http://inter-iot.eu/"
 }
}
```

Figure 27: Example JSON-LD platform registration message

4.4.2.4 IPSM Request Manager

IPSM Request Manager acts as a mediator between the Platform Request Manager, Bridges and the IPSM. It redirects messages for translation towards IPSM and receives translated messages from the IPSM. There is no direct connection between the bridges and IPSM, but instead all communication with Bridges for IoT platforms is now handled by the IPSM Request Manager. It also implements the IPSM-specific broker mechanism, which in principle can be different that the one used throughout INTERMW. The current implementation used RabbitMQ for communication among INTERMW components, while Kafka is used for communication $INTERMW \iff IPSM$.

4.4.2.5 Services

Implementation of services has been realized by connecting these to the Parliament triplestore-based database and introducing support for the SPARQL RDF query language in the QUERY and DISCOVERY message types[76]. SPARQL is custom-tailored for complex queries, and the Parliament database allows for fast (optimised) execution of those. An example of such a complex query would be returning a list of all sensors, connected to a specific platform and which are located in one location, as well as owned by one person. Wrapping such queries in QUERY and DISCOVERY messages enables the applications from the outside world and the Platform Request Manager within INTERMW to obtain lists of resources (connected platforms, things, etc.) dynamically. QUERY and DISCOVERY messages also abstract the need for INTERMW users/developers to understand the specifics of different platforms or learn discovery mechanisms they may (or may not) support.

4.4.2.6 Bridges

Bridges in INTERMW act as a middleman between INTERMW and IoT platforms. For messages that come downstream from INTERMW, a bridge processes the message and acts upon it if necessary. When a platform wants to send a message upstream, the bridge creates a message using information provided by the IoT platform and sends it upstream towards INTERMW. In addition to communication with platforms bridges also handle syntactic translation of payload between syntax used in the data model and the IoT platform format.

Semantic translation (alignments) performed by IPSM complements the syntactic conversion performed by bridges. Data models of platforms participating in communication through INTERMW thus use a two-step approach to map between format and semantics of the INTER-IoT data model. The need for platform's compliance with the internal INTER-IoT data model is in this way totally nullified, thus facilitating interoperability among heterogeneous middleware platforms, as long as someone has developed bridges for these platforms. Addition of a new platform does not require any changes in implementation of already existing bridges. This allows effective decoupling at both conceptual and implementation middleware levels.

4.4.3 Use Cases

Basic sequence diagrams have been defined in D3.1. Although further architectural and technical choices made during the INTERMW development slightly changed sequences and naming conventions defined there, the main iterations between components are still valid. The only notable difference is the implicit implantation of data flows, which means the Data Flow Manager component does not exists as a physical software components, but is rather an abstract concept implemented through conversation identifiers and the message broker.

[76] https://www.w3.org/2009/sparql/wiki/Main_Page.

4.4.4 Implementation Status

Bridges for the following platforms have been implemented: UniversAAL, FIWARE, WSO2 and BodyCloud. BodyCloud is being integrated into INTERMW to support the INTER-Health pilot. Bridge for oM2M is currently in development. Platforms UniversAAL and FIWARE were used in a functional demo, where a scale was connected to UniversAAL and sent data through INTERMW to the demo application, which pulled the data using a *pull()* REST call, and sent it back downstream towards the display, which was connected to the FIWARE platform. For the purpose of testing, we also developed bridge emulators, which enabled us to test INTERMW without connecting it to actual platforms.

Thus, at the present moment INTERMW already implements a subset of INTER-IoT requirements (D2.3), and most importantly, facilitates communication between platforms without the need to implement platform-to-platform information translation. INTERMW is also being validated through the open Call for *INTER-OM2M* and *SensiNact* projects.

INTER-OM2M will compare the development, deployment and exploitation of a demonstrator, featuring widely used application protocols (HTTP, MQTT, CoAP), over different radio (BLE, LoRA, IEEE 802.15.4) and power line communication (PLC), including security mechanisms, in the oneM2M and INTER-IoT framework.

SensiNact is a horizontal platform dedicated to IoT and in particularly used in various smart city and smart home applications. The project will provide to INTER-IoT the opportunity validate interoperability approaches with the integration of the sensiNact platform and thus access to all compatible data sets from different domains such as smart cities, smart farming, smart ski resort, smart building, smart living and well-ageing.

Important choice of using a triple store database (Parliament) for INTERMW persistence and as basis for INTERMW discovery and registry services has been made. The connection with IPSM has been implemented that we control with REST calls and send messages to and receive them from via Kafka broker client[77]. Note that Kafka is also one of Broker types supported in the abstract broker implementation, as described in D3.1.

Currently, analysis is performed to design the deployment framework in order to allow easy extensibility of the system. Then additional services have to be implemented, most notably the routing service. The last two steps to be performed is integration of results form Open Call projects and validation through the pilots.

4.4.5 API (and Extensibility)

There are several areas where INTERMW provides API and extensibility mechanisms:

- *REST API.* REST API is provided for application-layer usage. Applications can make use of the REST API in order to implement high level, application-specific functionality. This is also the interface exposed to INTER FW. REST API documentation is referenced in the documentation section (4.4.6).
- *IoT Platform Bridges.* As explained in Section 4.4.2.6, for each IoT platform to be added to INTERMW a new bridge has to be developed. In order to allow developers add new bridges

[77]See `https://kafka.apache.org/`.

with minimal effort, this mechanism will be refined and provided as part of the SDK being developed in INTER FW.

- *Services.* INTERMW has the capability to provide various MW2MW services. Currently, only registry and discovery services are provided, as they represent the minimum set of functionalities needed for MW2MW interoperability. However, more complex services may be needed in some use case scenarios, as for example Routing and Roaming Services. INTERMW provides an abstract interface that can be implemented for provision of new services. Documentation, libraries and possibly an SDK for services development will be provided in the second phase of INTERMW implementation.
- *Broker abstraction.* INTERMW defines an abstract broker interface that is being used for all INTERMW communication between components (see Section 4.4.2.2). This interface can be implemented for any message broker that supports a basic set of functionalities. As a need to extent the system to a wide range of message broker implementations has not been identified, there will be no dedicated SDK solutions for this type of extensions. Extensible documentation and examples will be provided in order to facilitate implementation.

4.4.6 Code and Documentation

INTERMW modules are implemented in the Java programming language and Apache Maven [78] is used as software project management and comprehension tool. A set of supporting tools has been provided in order to facilitate integration and management of software releases:

- *GIT Repository.* The Gogs git source repository is used for source code management. The repository is deployed at `https://git.inter-iot.eu/`.
- *Nexus Repository.* The Nexus Repository Manager has been deployed in order to organise, store and distribute INTERMW software components: `http://nexus.inter-iot.eu/`.
- *Jenkins Continuous Integration.* Jenkins is an automation server that supports support building, deploying and automating software development. The INTER-IoT instance is available at `https://jenkins.inter-iot.eu/`.

Currently INTERMW software repositories are private, but they will be released under the Apache 2.0 license after the initial validation phase. INTERMW related source repositories are:

- *INTERMW project repository.* The repository contains all INTERMW main modules described in this section. Location: `https://git.inter-iot.eu/Inter-IoT/intermw`.
- *Message.* This project implements the InterIotMsg class and related helpers. It contains the JSON-LD implementation of the message. Location: `https://git.inter-iot.eu/Inter-IoT/messaging`.
- *Syntactic translator.* A set of syntactic translators for specific IoT platform messages. Location: `https://git.inter-iot.eu/Inter-IoT/syntactic-translators`.
- *Swagger REST API definition.* The latest REST API documentation is available as part of the INTER-Layer API repository: `https://git.inter-iot.eu/Inter-IoT/layer_apis`.

[78] `https://maven.apache.org/`

- *API Management.* Latest REST API definitions and API Manager deployment scripts, as part of INTER API are available at: `https://git.inter-iot.eu/Inter-IoT/api_manager`.

4.4.7 Demo

A demo application has been implemented in order to test and demonstrate the applicability in practice. The demo set-up, shown in Figure 28, has been built based on some INTER-Health pilot elements. It connects two middleware platforms (UniversAAL and FIWARE) enables interoperability between them. The deployment integrates the following components:

- *Digital scale.* A digital, Bluetooth-enabled, scale that is paired to a smartphone and sends weighing values.
- *Smartphone.* An Android phone with an app that gets weighing values from the scale and forwards them through its universAAL app to other universAAL instances on the local network.
- *universAAL.* An IoT open platform oriented to Active-Assisted Living applications.
- *Inter-IoT Middleware-to-Middleware.* Inter-layer component that acts as a bridge between Ambient Assisted Living platforms and Hospital ICT systems. In this demo deployment we bridge UniversAAL and FIWARE.
- *IPSM.* Inter-layer component that translates IoT platform messages to the global INTER-IoT structure and vice-versa.
- *INTER FW (INTER API).* INTER FW API Request Manager, used for INTERMW management.
- *FIWARE.* The FIWARE platform provides a set of APIs that ease the development of Smart Applications in multiple vertical sectors.
- *FIWARE Orion.* Context information manager and broker for entities updates, queries, registrations and subscriptions, based on FIWARE/OMA NGSI9/10 interfaces.
- *Wirecloud GE.* A next-generation end-user centered web application mashup platform aimed at leveraging the long tail of the Internet of Services built on top of FIWARE.

While the demo is relatively simple, it encompasses most of INTERMW functions: deployment of different bridges, subscription management, message routing, using of registry/discovery services and API Management.

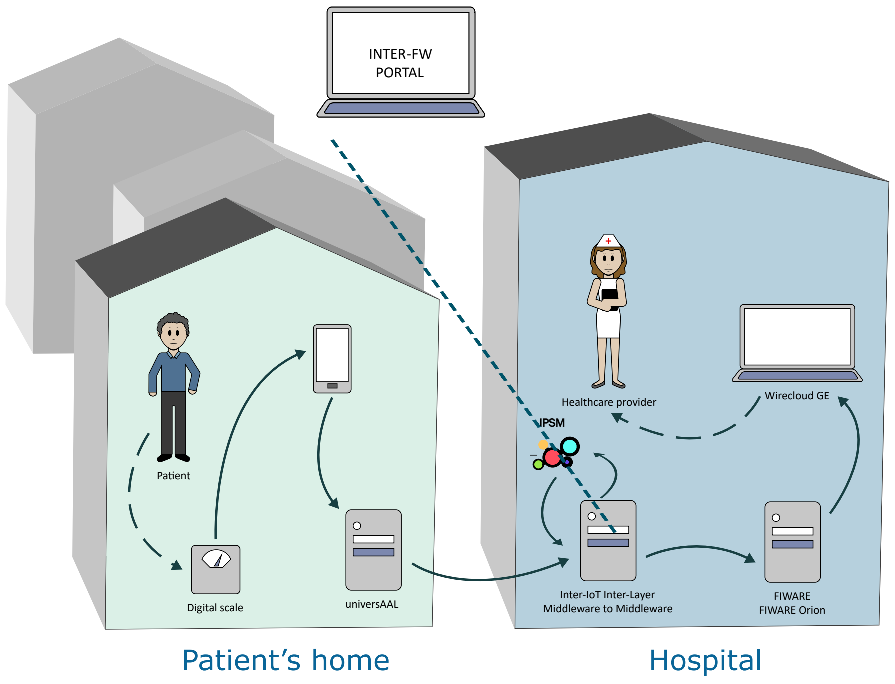


Figure 28: INTERMW demo

4.5 AS2AS solution

4.5.1 Refined Architecture

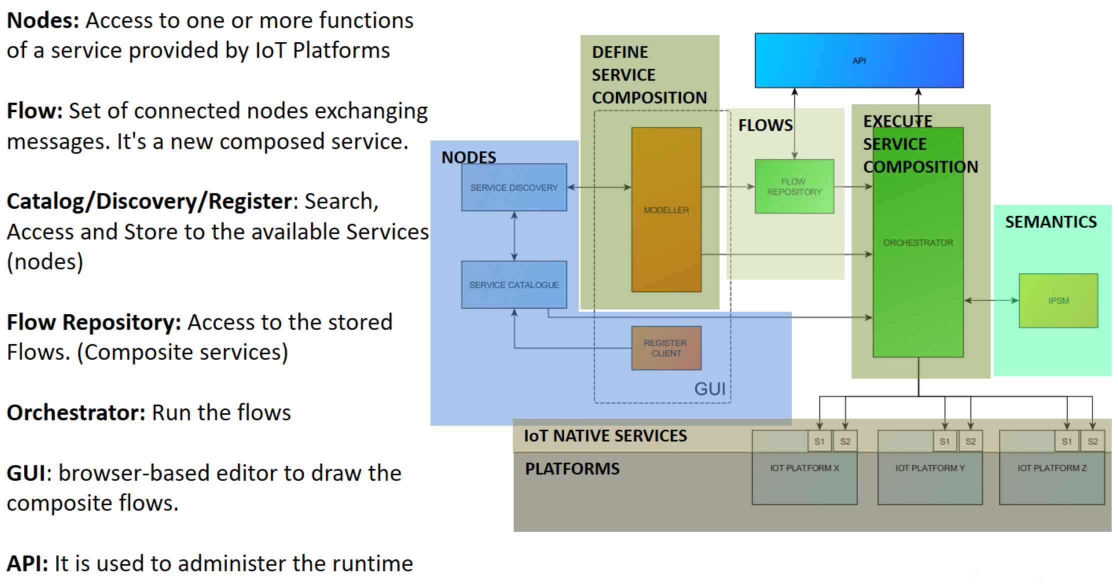
Since the publication of the Deliverable D3.1, the architecture of Application and Service to Application and Service Interoperability did not notably changed. Figure 29 shows the previous image of the architecture, highlighting and grouping the defined components and its relationship between the internal elements of the interoperability solution: the nodes and the flows.

Figure 29 emphasizes the link between these two elements and the architecture. This is the previous step to understand the following sections of this chapter, because they are the basic elements of the interoperability solution in this layer.

As a brief summary of the architecture, the Catalog, Register and Discovery components work with the nodes and its properties. The Modeller is responsible to create and modify the flows, which are composed by the interconnection of several nodes. The purpose of the Modeller is to define the service composition.

These flows are stored and loaded in the Flows Repository. These are executed by the Orchestrator, which is the one who starts the operation of the service composition. This component is the one that makes calls to the native services of the IoT platforms through the nodes. Furthermore, manages the execution of the interaction between nodes. Finally, the API interacts with the Orchestrator and the Flow Repository to work with the interoperability solution in real time.

The use of nodes and flows will be discussed in more detail in the next section 4.5.2, Components. This section details the process of creation and implementation of this elements. The section 4.5.6


Nodes: Access to one or more functions of a service provided by IoT Platforms

Flow: Set of connected nodes exchanging messages. It's a new composed service.

Catalog/Discovery/Register: Search, Access and Store to the available Services (nodes)

Flow Repository: Access to the stored Flows. (Composite services)

Orchestrator: Run the flows

GUI: browser-based editor to draw the composite flows.

API: It is used to administer the runtime

Figure 29: AS2AS architecture

indicates the code and documentation of nodes developed and the flows created. Finally, the section 4.5.7 show real examples of nodes and flows running.

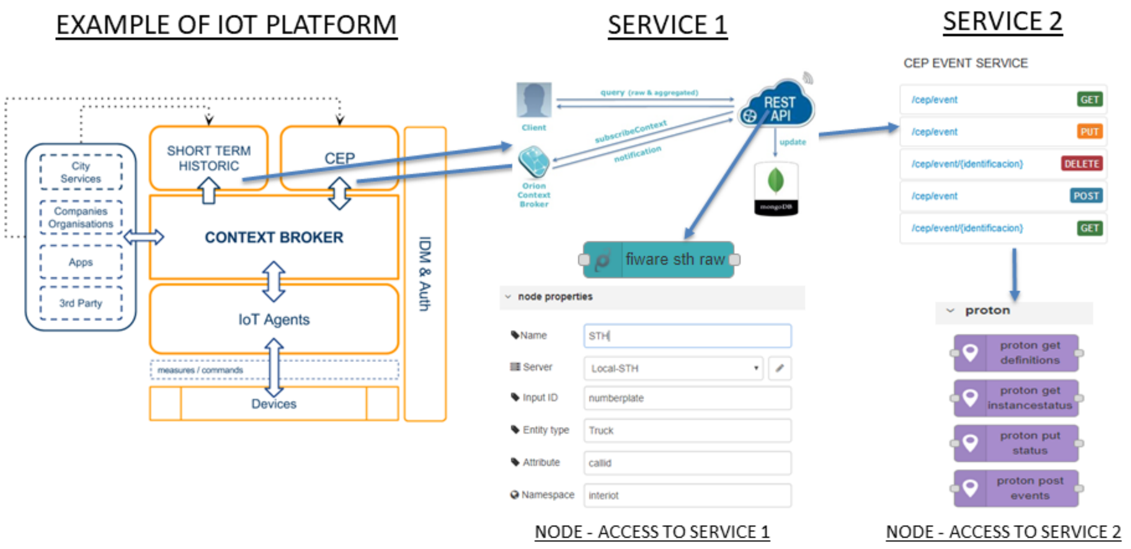
The other aspect to comment about the architecture is the access to a complete instance of the interoperability solution. The flow designed by the modeller and executed by the orchestrator, can be accessed by one or more users, at the same time and with different permissions, through the APIs. But, only one flow can be executed in each instance of the solution. Therefore to have several flows of the interoperability solution running at the same time, we need to work with different instances of the solution. These aspects will be explained in section 4.5.5 API (and Extensibility).

4.5.2 Components

One of the main requirements identified to achieve an interoperability solution in this layer is access to native services and applications offered by IoT platforms. Figure 30 explains how we do this process:

On the left side of the image there is an IoT platform and its components. In the northbound of this architecture two services are offered (in this example, a CEP and a Short Term Historic). Our interoperability solution needs to use the functions provided by these services, therefore, it is necessary to study the modes (REST API, SOAP...) available to access to the service and to implement a solution that provides access to these functionalities. In the D3.1 we defined that the node will be the element in charge of sending the requests, executing the services, managing the results and the returned messages, in a format compatible with our interoperability solution.

In the first example, all the functionalities of the service are contained in a single node, in contrast to the second example, in which four nodes have been used to access the service. If the function is implemented by a greater number of nodes, it is easier to understand the purpose of the functions, but the code updates are more complex. On the other hand, a smaller number of nodes for a service means, implies that you have to enter more parameters in the form and the functions are complicated



Name	STH
Server	Local-STH
Input ID	numberplate
Entity type	Truck
Attribute	callid
Namespace	interiot

Figure 30: AS2AS Native Services

to understand by the user. It can be observed in the form that appears in the first example, where the user has to fill several parameters to use the node.

Paying attention to technical issues, a node consists in a JavaScript file that runs in the Node-RED service, and an HTML file consisting in a description of the node. The description appears in the node panel with a category, colour, name and an icon, code to configure the node, and help text. Nodes can have at most one input, and zero or more outputs. During the initialization process, the node is loaded into the Node RED service. When the browser accesses the Node RED editor, the code for the installed nodes is loaded into the editor page. Node RED loads both HTML for the editor and JavaScript for the server from the node packages.

There are three main types of nodes. Firstly, the input nodes that generate messages for downstream nodes. Secondly, the output nodes that consume messages, for example to send data to an external service, and may generate response messages. Finally, the processing nodes: messages that process data in some way, emitting new or modified messages.

Generally nodes are designed to interact with a service, but there is a type of nodes called configuration nodes, which are responsible of share configuration information between the different nodes that are part of a service, for example, connection data to the same service. These nodes do not access to the service, they are only used to create and store this type of configuration data.

The flows are a collection of nodes wired together to exchange messages, the data contained in the flow is stored in a file in JSON format. It consists in a list of JavaScript objects that describe the nodes and their configurations, as well as the list of downstream nodes they are connected to, the wires. The wires define the connections between node input and output endpoints in a flow. The messages passed between nodes in Node-RED are, by convention, JavaScript Objects called msg. Messages are the primary data structure used in Node-RED and are, in most cases, the only data that a node has to work with when it is activated. This ensures that a Node-RED flow is conceptually clean and stateless.

The components interact with the nodes and flows in the way explained in D3.1 and in the Refined

Architecture section of this deliverable. Furthermore, in D3.1 (section 3.4.4 Technologies) it was established a map of AS2AS and Node-RED features establishing a connection between its elements and our architecture. We continue to work in that direction.

Once the first instances of the interoperability solution are designed, it is necessary to explain a new issue that involves the deploy of our solution, this is the place where the nodes and flows will be located to interact with the elements of the interoperability solution. Assuming that in the early stages of the solution the Node-RED environment will be found in a Docker container, inside a docker container server. We are working on the following storage distribution:

As shows in figure 31, to facilitate a solution that provides extensibility, persistence and collaborative work, a Git repository is used to storing and accessing to nodes and flows. Using a dockerised instance of the Gogs environment, which is a self-hosted Git service. It is the current solution but not final, because we are analyzing other types of repositories that better fit the needs of the catalog of services and discovery.

It can be observed that each instance of Node-RED has associated an external folder to the instance, but accessible from it. It stores all the information related to the flows and nodes, in addition to some external data that may be necessary for the solution.

The Dockerfile file is responsible for creating the running instance of Node-RED custom solution. A list of the services selected by the user of the node catalog is generated. These selected nodes are added to a clean image in docker of the interoperability solution. This is a Dockerfile repository with the custom images of Node-RED developed by Inter-IoT. The user can choose between some Node-RED images with different characteristics, versions and customized elements. Finally, the result is a docker file, which results in an instance of the interoperability solution with those services that the user has selected.

Once the instance was created if someone wants to add a new node, it would be necessary to install it from Inter-FW and restart the instance to be uploaded. The flows are in another file, the user can start to develop a new flow from scratch or take a copy of the json file from the repository to work with it. The changes made in the flow will be stored in the shared folder of the instance and can be sent to the flows repository.

In Cross-layer section, it will be explained generic concepts about how docker facilitates the scalability Inter-Layer, helping to define security, having a solution with multiple users and facilitating integration into Inter-FW.

4.5.3 Use Cases

As indicated there are no significant changes in the architecture, so the sequence diagrams designed in the D3.1 deliverable are still valid. Any changes that are made in the future, will be contemplated in the following deliverables.

4.5.4 Implementation Status

We are offering a solution based on Flow-Based Programming that defines applications as “black-box” processes, which exchange data through predefined connections with message passing. These

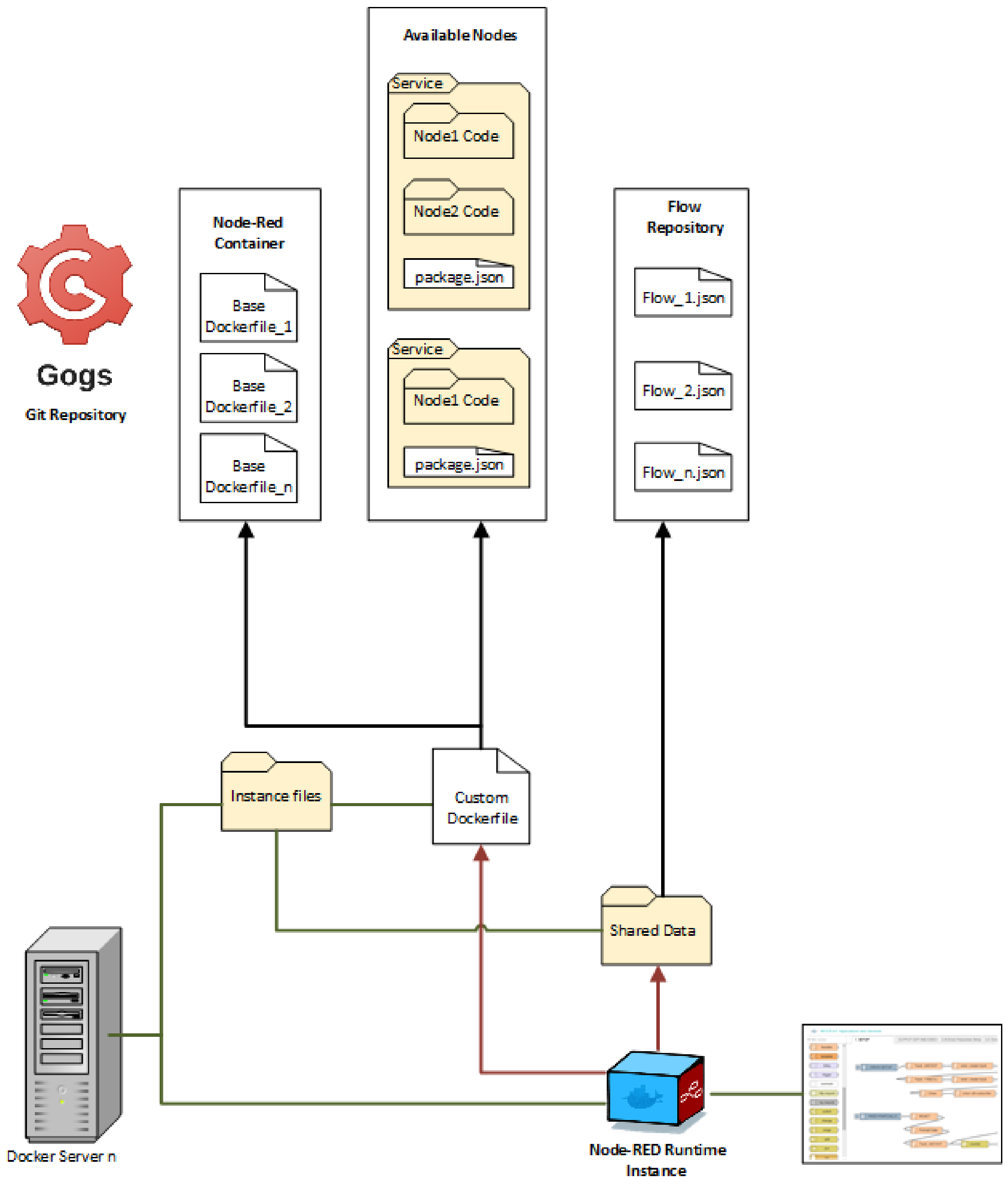


Figure 31: Persistence with Docker and GIT

can be connected to create different solutions without the need of being modified internally by an end-user.

We are adapting a tool (Node-RED) that implements this paradigm according to the INTER-IoT interoperability requirements. The solution enables a number of IoT services to be available in our development environment. Access to IoT services has been achieved by accessing its REST APIs and wrapping them through a node with a series of functionalities for the user. For those services that do not have REST API other alternatives have been looked (e.g SOAP web services).

A demo related with transport and logistics to illustrate a practical example has been developed (section 4.5.7), involving two IoT applications provided by FIWARE and a Port Community System. This applications exchange its data to display alerts with this information and create extra value adding new services over these two as a dashboard, a map, etc.

A common methodology has been established for the study of the new services, following a series of steps to carry out the implementation and documentation of the nodes, as presented in section 4.5.6. We have used this methodology to implement our own nodes.

A node (IPSM node) has been developed to work with the semantic aspects. Furthermore, we are improving the security and authorization in the access to the nodes. This two issues will be explained in the Cross Layer Section 4.7.

Regarding more specific aspects, in Figure 31 it is possible to observe how we have implemented the relationship between an instance of the interoperability solution and the access to nodes, flows and our custom Node-RED images stored in our Git repository. Currently, it is the way we are working because it allows us to take advantage of the persistence and control of versions to program and access everything that we designed in a centralized way.

All third party information is accessed directly in their repositories, although the access links are stored in our repository. We have elaborated an internal catalog of available nodes and services of IoT Platforms and we are updating it with the new nodes available. This is often accompanied by information on how to deploy the services in order to work with them.

The catalog solution has to be improved in the future; we are working on having a hybrid system based on Git for the data and store the descriptions of the services in another place, with metadata and links to the addresses of this. Different options are being considered as a custom solution designed by us, semantic databases in the style of the solution of MW2MW or Hypercat as mentioned in the previous deliverable. As indicated in the State of the Art, we are working with Swagger nodes to validate if they can be a key element to unify the description and catalog of services and to facilitate the process of create flows.

Furthermore, the current development is focused in the following steps, such as:

- introducing new platforms and services,
- standardizing the JSON messages returned by the nodes,
- improving the components,
- developing functionalities above Node-RED,
- creating new functionalities to interact with nodes and flows,
- improving the interaction with the available API and create new APIs,

- defining new scenarios and demonstrations,
- dealing with the security.

Other aspect is the interaction with Docker, to generate multiple instances. Now we are providing an environment like the one presented in Figure 32:

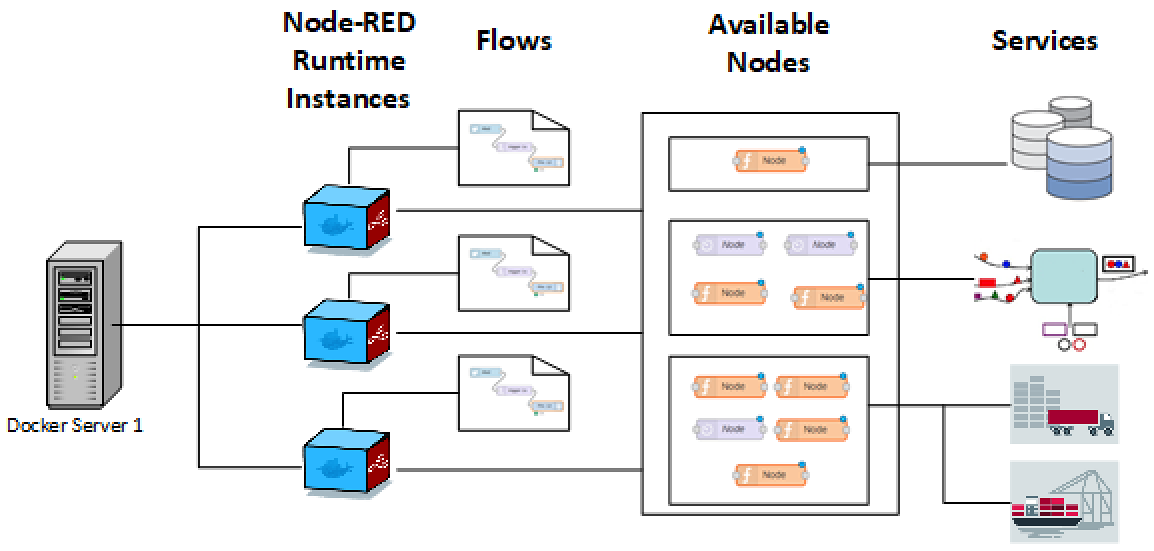


Figure 32: Current stauts with Docker

A solution in a docker container allows us to access to different instances of the interoperability solution on the same host. Each instance of the server have its own folders with the same distribution shown in Figure 31.

4.5.5 API (and Extensibility)

There are several interactions between the users and the AS2AS solution. As can be seen in Figure 29 (AS2AS architecture), users interact with the modeller to define the composition of services. The modeller is a web environment with a palette of nodes and functions to define a flow. The designed flow is stored in a JSON file.

The other graphic element to interact with the user, is the node registration client. Currently the graphical interface of this component is not designed, and the nodes are being loaded in a Git repository through a script. New features of the services catalog have been defined, then that user interface will be finally defined.

Node-RED instances provide APIs that interact with their following elements:

- Authentication: Get the active authentication scheme, exchange credentials for access token and revoke credentials. It is used for the security/authentication options.
- Setting: Get the runtime settings of Node-RED. It is used in Cross-Layer.
- Flows: Get/Set the active flow configuration.
- Flow: Add a flow to the active configuration and get/update/delete a flow configuration.

- Nodes: Get a list of the installed nodes in a instance of the solution and install a new node

These APIs are going to be consumed by Inter-FW, but are also consumed by the components of the architecture to interact internally with the Node-RED tool.

The APIs related to the Flow Repository are being defined in this moment and will be updated in the next version of this deliverable.

As explained in the previous section, the creation of instances of docker with a custom version of Node-RED allows to create multiple instances managed by multiple users. Figure 32 shows the current state with a host that had multiple instances of docker. However, in Figure 33 we can observe the proposed architecture for future deployment of AS2AS.

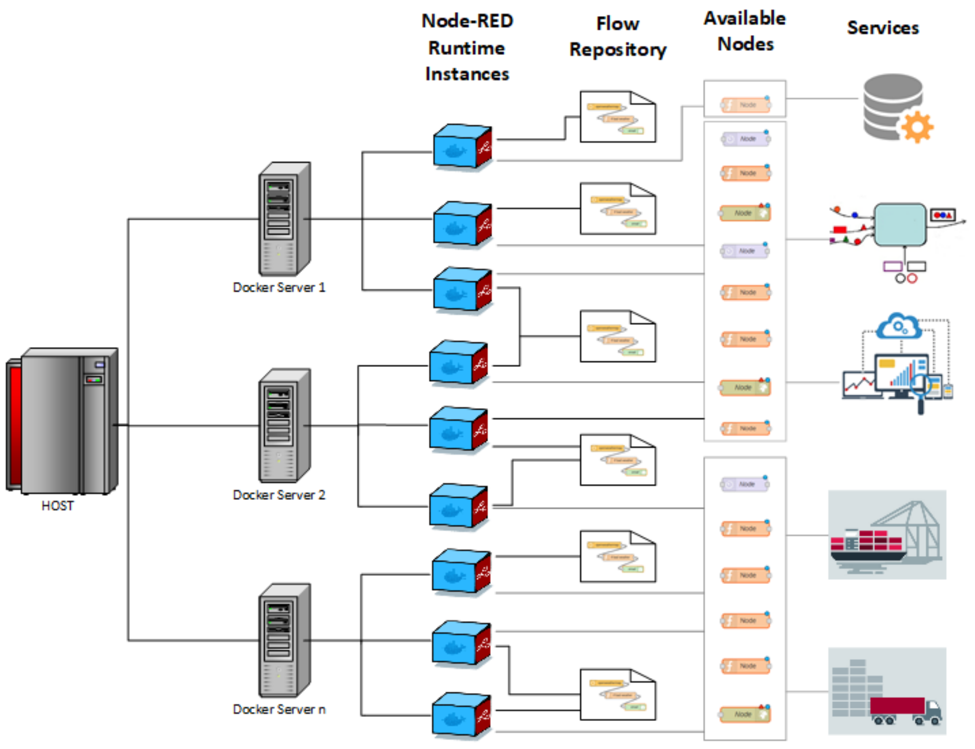


Figure 33: Extensibility with Docker

In Figure 33 there are multiple hosts with multiple instances of Node-RED, which must be treated in the same way despite being on different hosts. Some tools like Docker Swarm and Docker Portainer facilitate this management, this is described in the Cross-Layer Section 4.7. But in summary, this relationship with the APIs or options offered by Docker facilitates the extensibility of the solution in terms of scaling, portability, security and easy deployment of solutions.

Finally, another relevant issue related to the extensibility of the solution is the development of new nodes compatible with the Node-RED tool and the development of components to interact with this tool. In the following section, details are provided about these tasks.

4.5.6 Code and Documentation

The documentation of the project has been produced using shared documents or files uploaded to the Git[79] or Horde repository.

It has been used for different purposes:

- Internal methodology for the analysis of services and the creation of nodes.
- Internal document about the evolution of the development of the interoperability solution.
- Guide about how to create a basic node.
- Documentation of APIs available in Node-RED.
- Explanation of the demonstration.
- Use of docker instances of Node-RED.
- Support material like presentations or videos.

The code is available in the AS2AS repository, where there are the nodes developed by us, the services associated with these nodes, our current custom Node-RED version, the current status of the developments in the components, the demonstration, the swagger description of the APIs, the developed flows and the scripts of the dockerfiles.

In the early stages of the project development, it was necessary to make an effort to become familiar with the nodes, services and work environment. So we have defined a methodology to create nodes that implies an effort in elaboration of documentation and code. The document of how to elaborate a node was sent to all the participants in this task and was explained by videoconference, to clarify possible doubts. Significant effort was also put into knowing the Node-RED community and code in depth. The steps that follow this methodology are as follows:

1) Analysis of the service:

- Development or access to a functional instance of the IoT platform.
- Get access to the services that the platform offers.
- Deploy or access an instance of the service.
- Test the service with data.
- Analyze the functionalities offered by the service.
- Study the API that provides the service.
- Document the functionalities and APIs.
- Analyze the methods of access and execution of these functionalities.
- Analyze the messages or actions that return the execution of the functionalities of the service.

2) Node Implementation:

- Group the functionality of the service, to choose the number of nodes and how they will be the nodes that implement it.

[79] https://git.inter-iot.eu/Inter-IoT/interas

- Identify the parameters needed to access the service.
- Creation of configuration nodes. For example, they store the connection variables.
- Create the interface that collects the parameters that will consume the service. (html)
- Create the code that will execute the functionality (JavaScript).
- Define the messages that the node sends and receives.
- If it is possible attach an instance of the service in docker.

3) Test:

- Access to the node with real data.
- Test the correct operation.
- Fix bugs and catch errors.

4) Documentation

- Document about how to deploy service with real data, to have examples of operation.
- Document about the characteristics of the node

This methodology will be followed throughout the project. It is recommended to attach a Readme.txt file in the Git folders of the nodes, containing a summary of the main information of this process, to improve their usability. Now, the weight of the efforts will be transferred to other tasks. But the services analysis should be completed with more precise descriptions of the nodes (for example metadata description). This will be done when the complete functionalities of the service catalog component have been defined.

Concerning the flows, they are stored in repositories and a file with its characteristics is attached. The mechanisms (function nodes) that act as gateways to connect two services, are being described. The flow documentation part will also be taken into account how are working the connection between services and the modeling of the glow, in order to extract information to define good practices. We are working on providing small examples of how to make the most common connections.

It is also documented the modifications made over the original Node-RED code. Thus, it is possible to know how to act in case of an update of the Node-RED version. This will be fundamental in the new development phase, because the code and documentation are now focusing on changes on the components. We will proceed to restructure these sections in our repositories to adapt them to the needs that are emerging.

4.5.7 Demo

In this section we can find a practical example developed to show the interoperability in this layer, involving several IoT applications. A test scenario was defined, with a series of services that were deployed and nodes were programmed by us to access them. From our programming environment we created a flow that allowed the connection between them. The interoperability flow is accessible and executable from Inter-FW.

The interoperability solution is in a docker file, where our nodes and our custom Node-RED solution are available. To facilitate the test of the example a Docker Compose file was created that allows to configure and to execute all the services included in the test, except the PCS that is accessed from its Website.

4.5.7.1 Test Scenario

The demonstration starts with the position and data of a truck.

This information is obtained from the platforms that monitor the position of the devices that are present in the vehicles.

The designed flow, depicted in Figure 34, is responsible of composing this services.

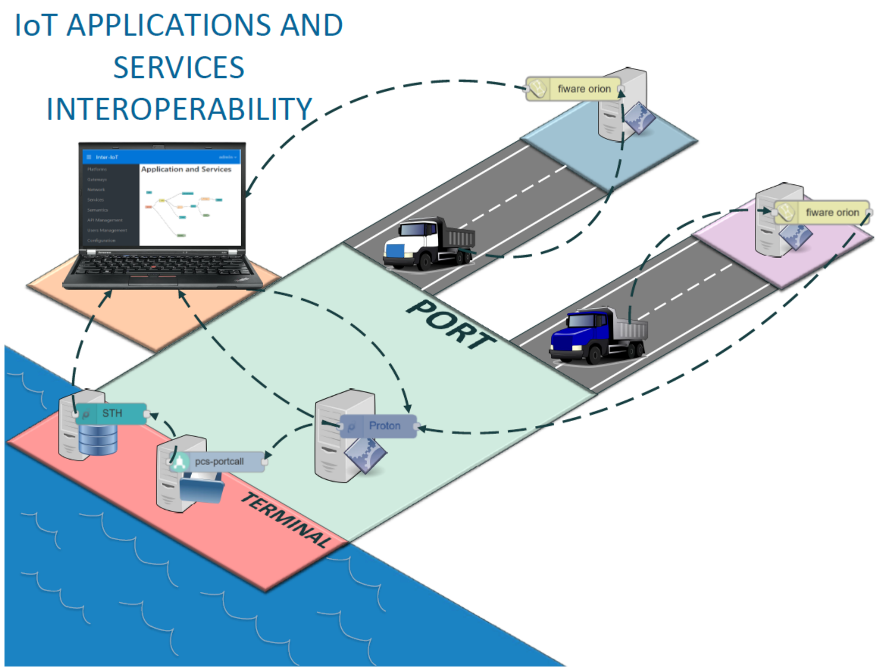


Figure 34: AS2AS Demo Scenario

Automatically when a rule is fulfilled, in the terminal of the port, they will receive information of the truck, the scale where it is directed and information of historical events of this truck. In addition, this information can be presented through maps, dashboards and panels offered by a dashboard service.

The flow is managed via the INTER-FW.

4.5.7.2 Services/Nodes Involved

Nodes provide access to native services. These nodes facilitate a simple and visual access to the functions provided by the services. This section will describe the functionality related to the demonstration and how some nodes are related to each other. To each node we are going to describe the service that is going to be accessed and the node created to access to it.

The nodes that access to the services of this demo have been developed by us, because they did not exist. However, the nodes responsible for representing the information and designing the dashboard, were already available.

Proton:

Proton Service:

Provides the following actions:

- Analyses event data in real-time.
- Generates immediate insight.
- Enables instant response to changing condition

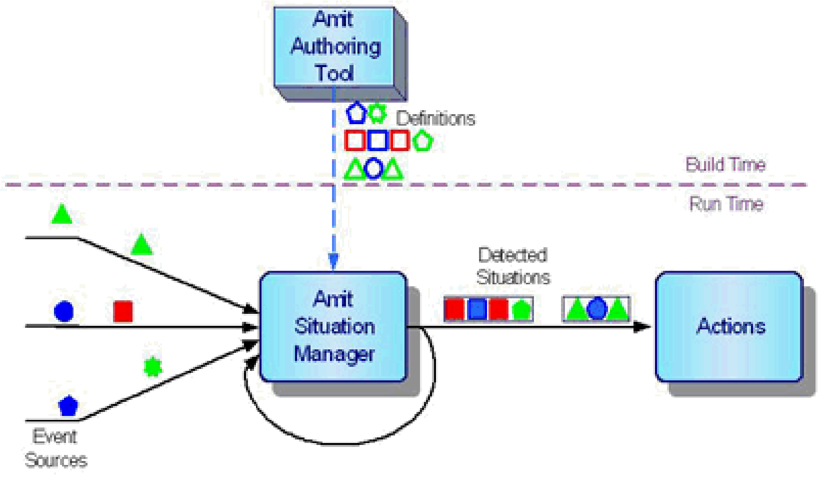


Figure 35: Proton Service[80]

In this demonstration, The CEP activates an action when a rule is detected. It is responsible of activating the execution of the flow.

Proton Node:

The Proton Node-Red node provides the following ways to interact with the proton CEP:

- Sending Events
- Managing the Definitions Repository
- Administrating Instances

PCS

PCS Service:

Port Call Service: PCS users have access to the Port Call Service inquiries that provide real-time information on both planned and current vessel port calls.

In this demonstration, with the CEP data, a Node-RED node makes the call to the PCS, obtaining the port call data from PCS (port call data retrieval service).

[80]Source: https://www.ibm.com/developerworks/ssa/local/im/ssa/identity-insight-complex-event-processing/index.html

[81]Source: http://www.valenciaportpcs.com/valenciaportpcs/

Figure 36: PCS Service[81]

PCS Node:

The PCS node-red node provides two methods to access the port call information:

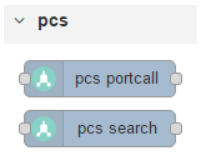


Figure 37: PCS Node

- Get the port call data from PCS from its identification number.
- Get a list of port calls from PCS in a range of dates.

STH

STH Service:

Short Time Historic: manages the historical raw and aggregated time series context information about the evolution in time of context data.

In this demonstration, the flow makes a call to the Historical Node to get information about the evolution in time of context data.

STH Node:

Figure 38: STH Node

The STH node has the following features:

- It is the first existing node integrated with FIWARE-STH.
- Gets the historic values of a node.
- STH attached to FIWARE Orion.
- Brings data visualization powers to Context-Based components.

In this demo concept, it gathers last call IDs of a truck when it is inside port area.

Dashboard nodes

This nodes provides, among others, the following functions in the demonstration:

- The quickly creation of a live data dashboard.
- Send and receive emails, with valid email credentials for an email server.
- A world map web page for plotting things on.

Figure 39: Dashboard

4.5.7.3 Service Composition/Flow

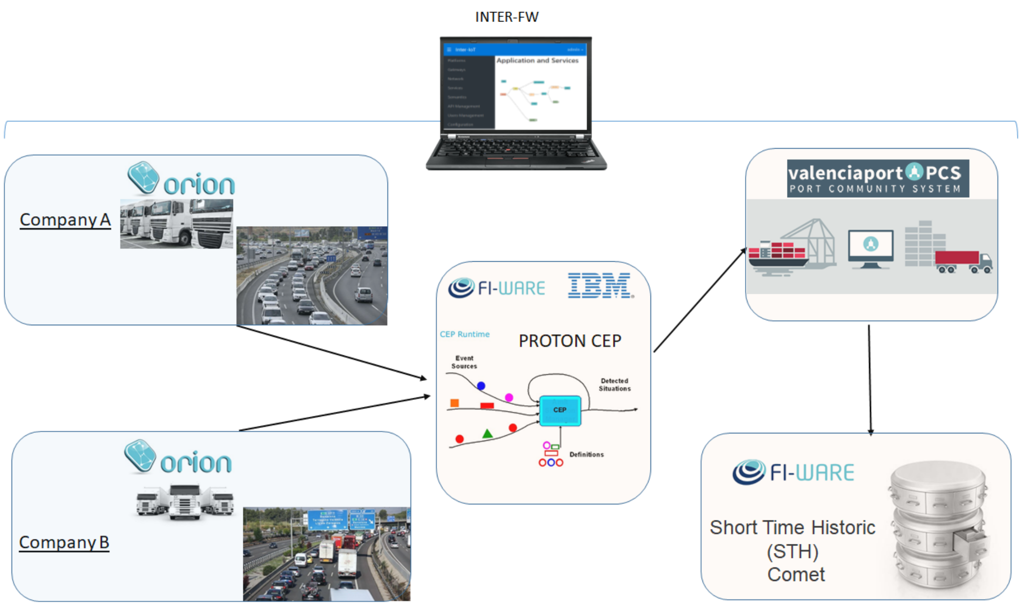


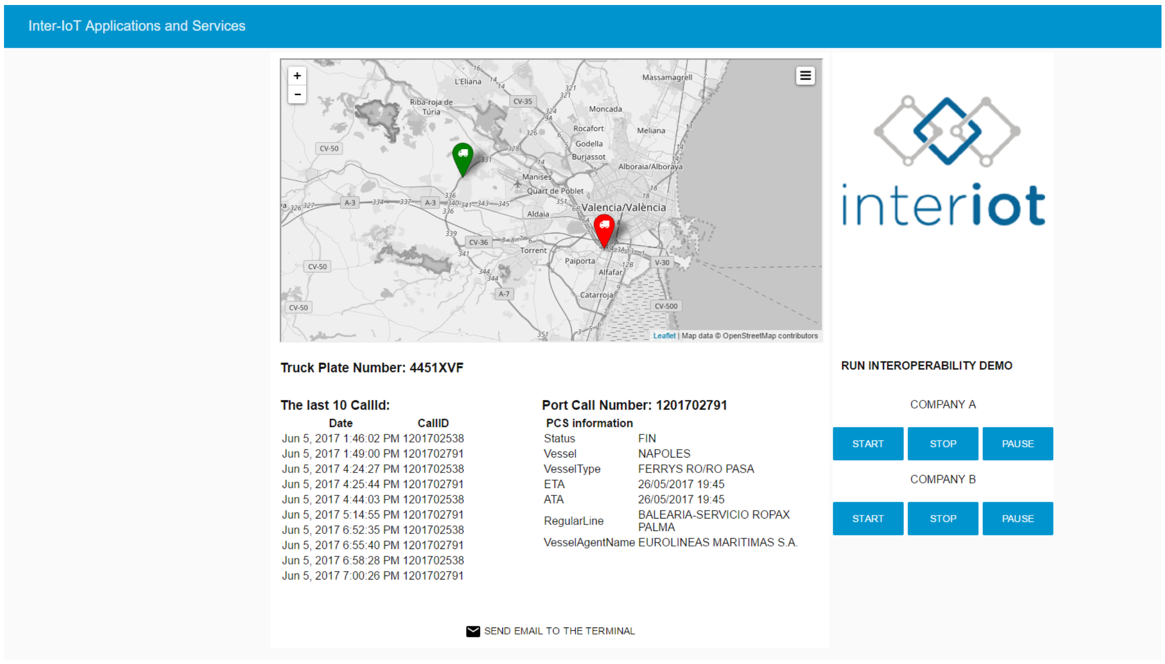
Figure 40: Service Composition

The steps required for service composition (as represented in figure 40) are the following:

- CEP FIWARE (Proton) receives information from Truck Companies.
- CEP calculates the distance of the trucks to a defined point.
- It triggers an alert when the truck is nearer than 10 kilometers. It gives the call ID (call number), truckID and position (lon, lat).

- Obtains the layover data from PCS (Layover Data Retrieval Service).
- Obtains Historical Data from the STH
- Shows data to the Port Terminal via maps/dashboards/email.

Figure 41 is the representation of the data being displayed in real-time using the dashboard service:


Inter-IoT Applications and Services
Leaflet | Map data © OpenStreetMap contributors
interiot
RUN INTEROPERABILITY DEMO
COMPANY A
START
STOP
PAUSE
COMPANY B
START
STOP
PAUSE
SEND EMAIL TO THE TERMINAL

Truck Plate Number: 4451XVF

The last 10 CallId:

Date	CallID
Jun 5, 2017 1:46:02 PM	1201702538
Jun 5, 2017 1:49:00 PM	1201702791
Jun 5, 2017 4:24:27 PM	1201702538
Jun 5, 2017 4:25:44 PM	1201702791
Jun 5, 2017 4:44:03 PM	1201702538
Jun 5, 2017 5:14:55 PM	1201702791
Jun 5, 2017 6:52:35 PM	1201702538
Jun 5, 2017 6:55:40 PM	1201702791
Jun 5, 2017 6:58:28 PM	1201702538
Jun 5, 2017 7:00:26 PM	1201702791

Port Call Number: 1201702791

PCS information

Status	FIN
Vessel	NAPOLES
VesselType	FERRYS RO/RO PASA
ETA	26/05/2017 19:45
ATA	26/05/2017 19:45
RegularLine	BALEARIA-SERVICIO ROPAX PALMA
VesselAgentName	EUROLINEAS MARITIMAS S.A.

Figure 41: Demonstration

This screenshot of the dashboard service shows a truck inside the indicated area (less than 10 km), then the truck has a red color and the information obtained from the services is shown in the dashboard (the portcall information and the last port call numbers visited) and send an email to the terminal informing that it is arriving. The other truck has a green color, because it is outside the indicated area, therefore, the information we have from it is its position.

We are working to integrate more companies, which are using other IoT platforms and to include new services to extend this demo flow.

4.6 DS2DS solution

4.6.1 Implementation Status

The initial version of the Inter-Platform Semantic Mediator (cf. Figure 42) has already been implemented and was successfully utilized during the IoT-EPI Review Meeting in Athens, in September 2017. Presentations at the Review demonstrated IPSM not only as a standalone semantic translation component, but also as a part of the INTER-MW infrastructure (for more information, see Section 4.6.4). Prior to the Review Meeting we have also conducted a series of performance, and “vertical” scalability tests for the IPSM. Their results (as described in [11]) are very promising. They indicate that both the IPSM design/architecture and the choice of the technological stack, described in Section 3.5.2 of Deliverable D3.1, and subsequently used in the implementation process were appropriate. IPSM accepts JSON-LD messages in the same format as INTER-MW (see Section 4.4.2.3). The "metadata" graph in JSON-LD messages is ignored, and only "payload" is translated. Although,

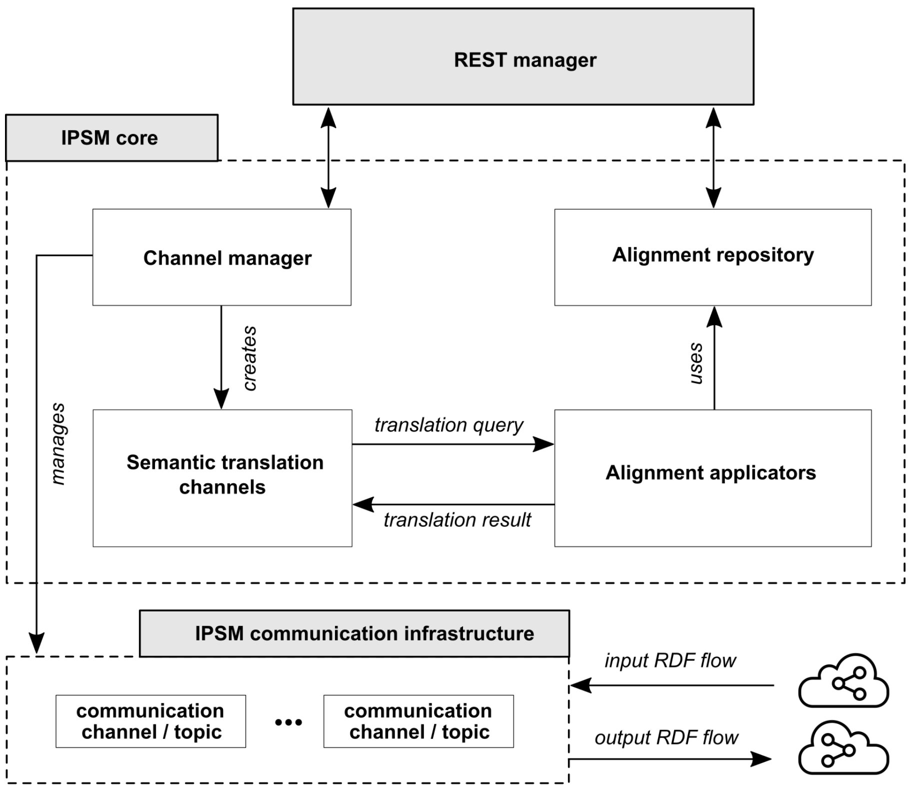


Figure 42: IPSM architecture overview

in its current state, the IPSM is already fully functional we plan to further optimize and enhance it. One of the areas of enhancement is the data format for representing alignments. Currently IPSM uses an XML-based format (cf. [12]) inspired by the Alignment API [13] and EDOAL [14]. We plan to replace it with an RDF-based *IPSM Alignment Format* (IPSM-AF), that we have introduced in [15], and which is fully compatible with the Alignment API. We also plan to enhance and simplify the way functions/transformations are defined within alignments. Currently, the IPSM offers support for functions natively provided by Apache Jena library, as well as for several additional operators offered by the IPSM implementation itself. We plan to extend this support to an even wider set of functions/operators.

Thanks to the excellent scalability properties of Akka toolkit and Apache Kafka, the "horizontal" scalability of the IPSM is also good. As a standalone component, IPSM can be deployed and concurrently utilized in many instances. In the near future we shall investigate the possibility of forming "IPSM clusters" which could be configured and operated in a uniform way.

4.6.2 API (and Extensibility)

The IPSM has a REST API exposing operations for configuration. The operations are divided into two groups – related to alignments and translation channels.

For alignments there are:

- `GET [host:port]/alignments` - lists alignments available in the IPSM instance; format is JSON which object for each alignment has fields:
  - `descId` (string): business identifier
  - `id` (integer): technical identifier
  - `date` (integer): UNIX timestamp of upload to IPSM
  - `name` (string): Name of the alignment
  - `sourceOntologyURI` (string): URI of the source ontology for alignment
  - `targetOntologyURI` (string): URI of the target ontology for alignment
  - `version` (string): alignment version
  - `creator` (string): alignment creator
  - `description` (string): alignment description
- `POST [host:port]/alignments` - uploads a new alignment as XML,
- `DELETE [host:port]/alignments/[alignmentId]` - deletes an alignment with an identifier passed as parameter,
- `GET [host:port]/alignments/[alignmentId]` - gets an alignment with identifier passed as parameter as XML.

For channels there are:

- `GET [host:port]/channels` - lists translation channels created in the IPSM instance; format is JSON which object for each channel has fields:
  - `id` (integer): identifier of the channel
  - `descId` (string, optional): business identifier of the channel
  - `source` (string): identifier representing the source of the channel, i.e. Apache Kafka topic from which IPSM reads the RDF data to be translated
  - `inpAlignmentId` (integer): identifier of the input alignment, used for translating the incoming RDF data
  - `outAlignmentId` (integer): identifier of the output alignment, used for translating the outgoing RDF data
  - `sink` (string): identifier of the sink of the channel, i.e. Apache Kafka topic to which IPSM publishes translated RDF data
- `POST [host:port]/channels` - creates a new translation channel; format is JSON object with the following fields:

  - `source` (string): identifier representing the source of the channel, i.e. Apache Kafka topic from which IPSM reads the RDF data to be translated
  - `inpAlignmentId` (integer): identifier of the input alignment, used for translating the incoming RDF data
  - `outAlignmentId` (integer): identifier of the output alignment, used for translating the outgoing RDF data
  - `sink` (string): identifier of the sink of the channel, i.e., Apache Kafka topic to which IPSM publishes translated RDF data
  - `parallelism` (integer, optional): internal parallelism of the channel, e.g., the value 5 means that the channel can consume 5 messages in parallel (preserving their time order), default: 1

- `DELETE [host:port]/channels/[channelId]` - deletes a channel with identifier passed as parameter.

The Swagger documentation of REST API is available on IPSM Azure deployment (see Section 4.6.3).

IPSM is extensible in terms of configuration – new alignments can be uploaded, and based on them new communication channels can be created.

4.6.3 Code and Documentation

The full IPSM source code is available from the INTER-IoT code repository[82]. To compile it only the Simple Build Tool (SBT)[83] is required. All necessary libraries, even Scala compiler, will be automatically downloaded when "`sbt compile`" command is issued for the first time. The only user documentation for the IPSM available at the moment is the description of its REST API. It can be accessed, for example, from the Azure deployment of the IPSM at:

`http://vmplsp01.westeurope.cloudapp.azure.com:8080/swagger/`.

The Azure machine should normally be accessible between 7am and 7pm as indicated in section 3.

4.6.4 Demo

A demo presenting how IPSM works in available on the IPSM Azure deployment at:

`http://vmplsp01.westeurope.cloudapp.azure.com:3000/`.

It is accessible from IPSM dashboard application that was developed to provide more "user friendly" interface for configuration, and possibility to the translation channels defined in related IPSM instance. Specifically, it allows to perform semantic translation of sample messages (with the possibility to modify the message inline).

The background story for the demonstration is as follows (see Figure 43):

[82] https://git.inter-iot.eu/Inter-IoT/ipsm/src/master/ipsm-core
[83] http://www.scala-sbt.org/

- There are 4 IoT artifacts/platforms that cooperate in e.g. a port environment. They have the following roles: `P1` – produces sensor observations; `P2` – analytical platform that should receive observations produced by `P1`; `P3` and `P4` – business logic platforms that consume observations published by `P2`.
- The architecture of IPSM assumes an agreement on central ontology (`CO`) specific for a deployment. In this case, central ontology is based on SOSA and geoSPARQL for geospatial data representation.
- Each platform uses a different ontology:
  - `P1: http://platform1.eu/sensors#`
    extending SSN and Basic Geo Vocabulary for geospatial data,
  - `P2: http://platform2.eu/sensors#`
    extending SAREF and Basic Geo Vocabulary for geospatial data,
  - `P3: http://platform3.eu/sensors#`
    extending SSN and Basic Geo Vocabulary for geospatial data,
  - `P4: http://platform4.eu/sensors#`
    extending SSN and geoRSS for geospatial data.

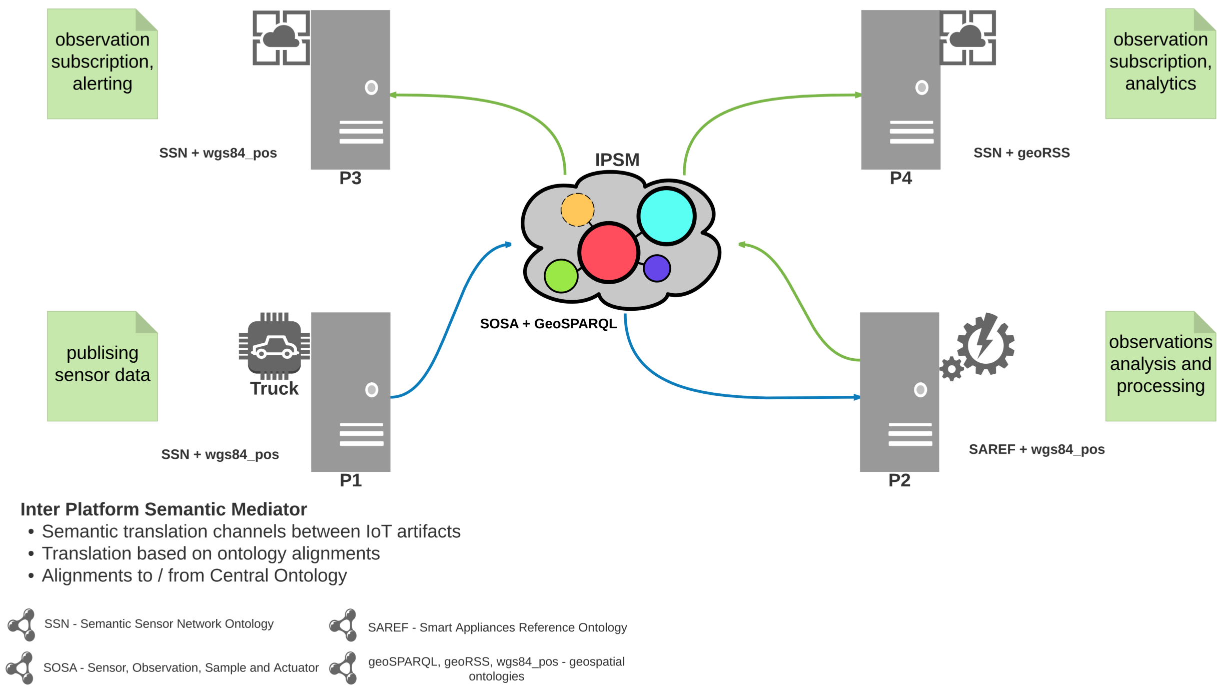
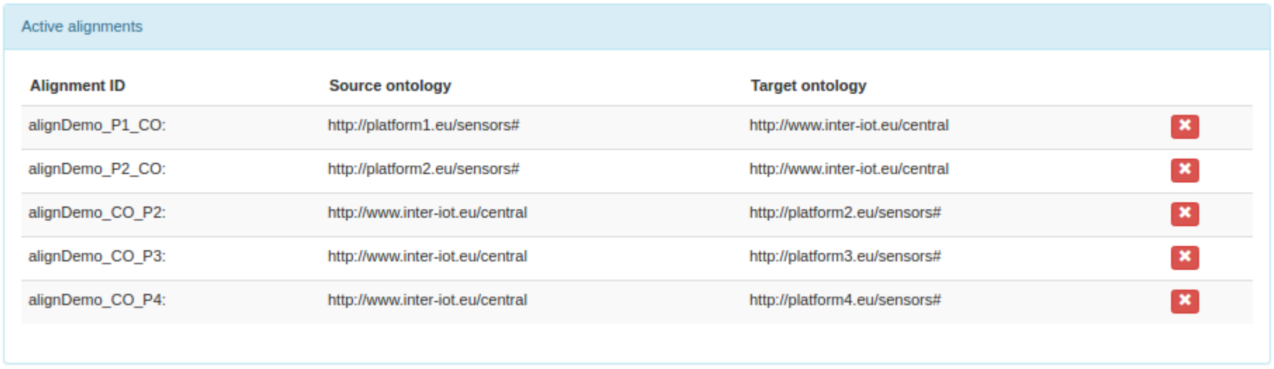

observation subscription, alerting
SSN + wgs84_pos
P3
IPSM
observation subscription, analytics
SSN + geoRSS
P4
SOSA + GeoSPARQL
publishing sensor data
Truck
SSN + wgs84_pos
P1
observations analysis and processing
SAREF + wgs84_pos
P2
Inter Platform Semantic Mediator
• Semantic translation channels between IoT artifacts
• Translation based on ontology alignments
• Alignments to / from Central Ontology
SSN - Semantic Sensor Network Ontology
SAREF - Smart Appliances Reference Ontology
SOSA - Sensor, Observation, Sample and Actuator
geoSPARQL, geoRSS, wgs84_pos - geospatial ontologies

Figure 43: IPSM demo overview

To verify alignment configuration go to Configuration → Alignments (Figure 44).

To verify channels configuration go to Configuration → Channels (Figure 45).

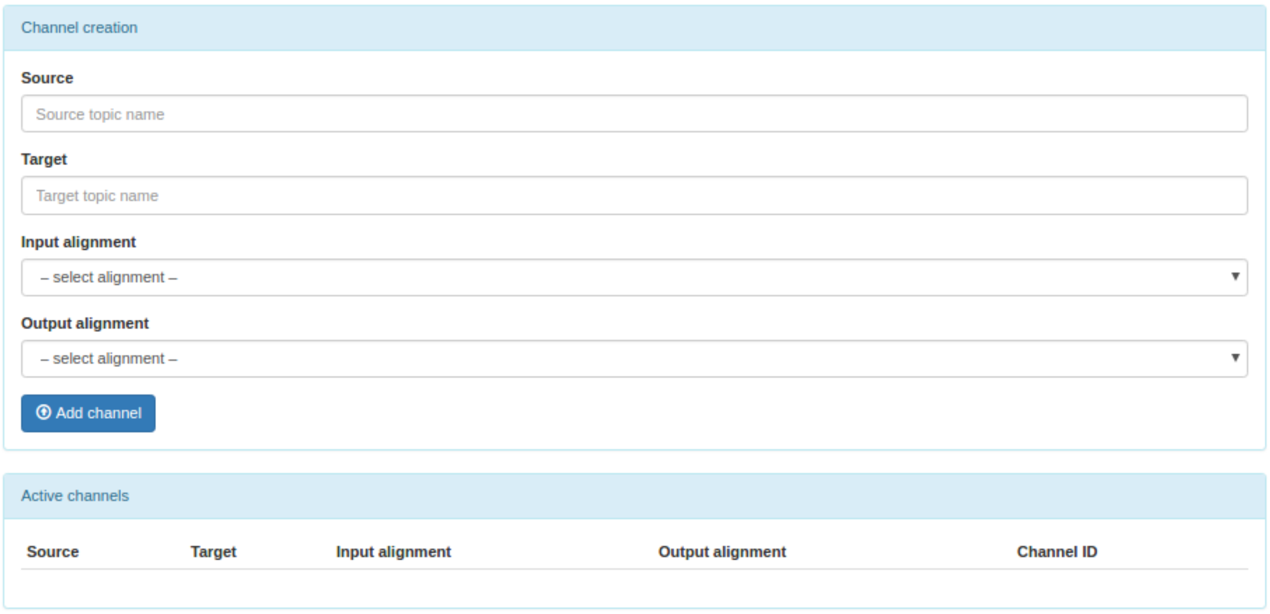
If not defined, add channels with the following definitions:



Active alignments

Alignment ID	Source ontology	Target ontology	
alignDemo_P1_CO:	http://platform1.eu/sensors#	http://www.inter-iot.eu/central	✖
alignDemo_P2_CO:	http://platform2.eu/sensors#	http://www.inter-iot.eu/central	✖
alignDemo_CO_P2:	http://www.inter-iot.eu/central	http://platform2.eu/sensors#	✖
alignDemo_CO_P3:	http://www.inter-iot.eu/central	http://platform3.eu/sensors#	✖
alignDemo_CO_P4:	http://www.inter-iot.eu/central	http://platform4.eu/sensors#	✖

Figure 44: IPSM alignments configuration



Channel creation

Source: Source topic name

Target: Target topic name

Input alignment: – select alignment –

Output alignment: – select alignment –

Add channel

Active channels

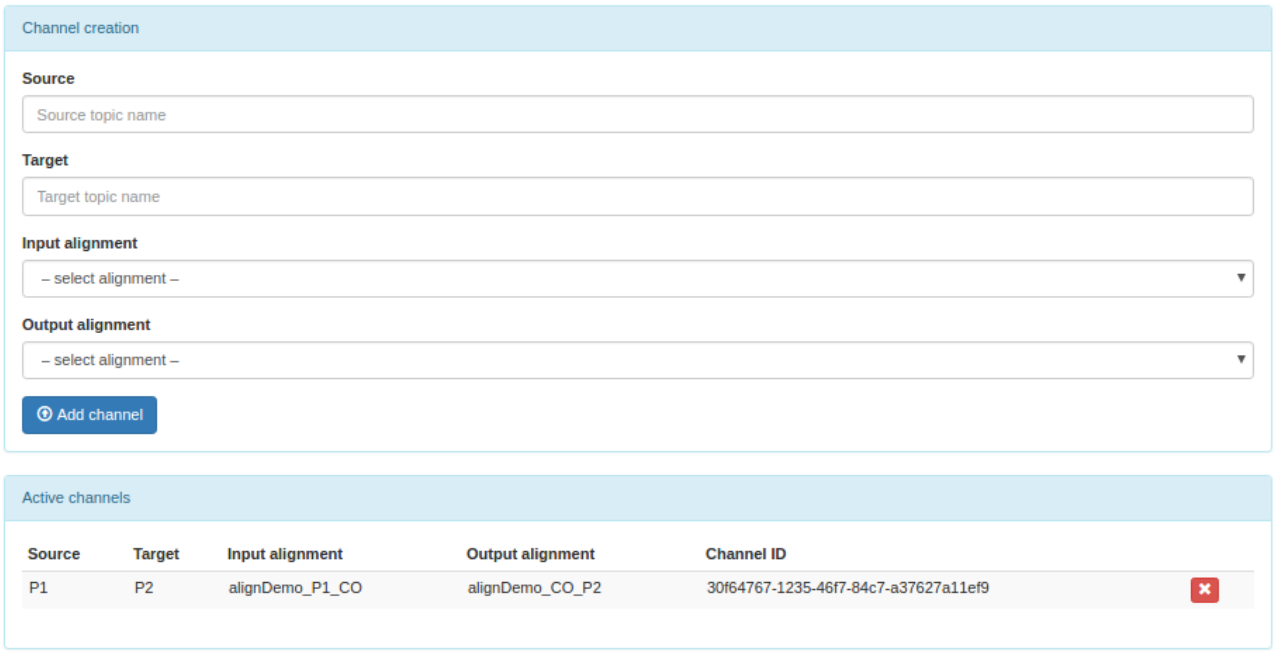
Source	Target	Input alignment	Output alignment	Channel ID

Figure 45: IPSM channels configuration

- source: `P1`, sink: `CO`,
  input alignment: `alignDemo_P1_CO`,
  output alignment: `IDENTITY`
- source: `P1`, sink: `P2`,
  input alignment: `alignDemo_P1_CO`,
  output alignment: `alignDemo_CO_P2`
- source: `P2`, sink: `CO`,
  input alignment: `alignDemo_P2_CO`,
  output alignment: `IDENTITY`
- source: `P2`, sink: `P3`,
  input alignment: `alignDemo_P2_CO`,
  output alignment: `alignDemo_CO_P3`
- source: `P2`, sink: `P4`,
  input alignment: `alignDemo_P2_CO`,
  output alignment: `alignDemo_CO_P4`

Note that `IDENTITY` alignment results in no translation applied.

After adding, newly created channel should appear on the list as in Figure 46.

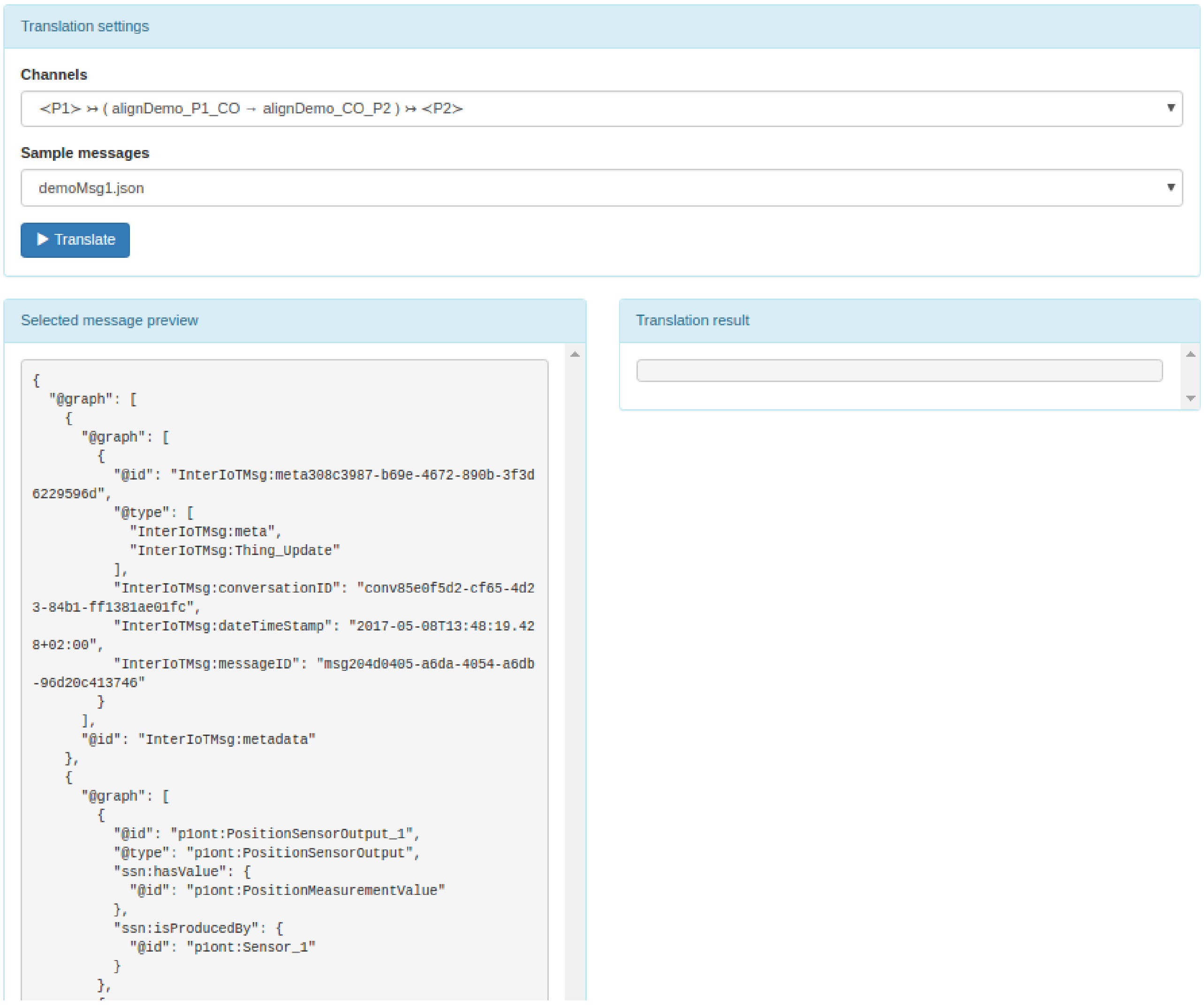


Source	Target	Input alignment	Output alignment	Channel ID
P1	P2	alignDemo_P1_CO	alignDemo_CO_P2	30f64767-1235-46f7-84c7-a37627a11ef9

Figure 46: IPSM channels configuration 2

To run translation select Translation from the main menu (Figure 47). From drop down lists select a channel and a sample message (it can be edited in place). Sample input messages that can be used include: `demoMsg12` – input message for channel with `P1` source; `demoMsg34a`, `demoMsg34b` – input messages for channels with `P2` source.

The translated message will appear in the box on the right as in Figure 48.



Translation settings

Channels

<P1> ↣ (alignDemo_P1_CO → alignDemo_CO_P2) ↣ <P2>

Sample messages

demoMsg1.json

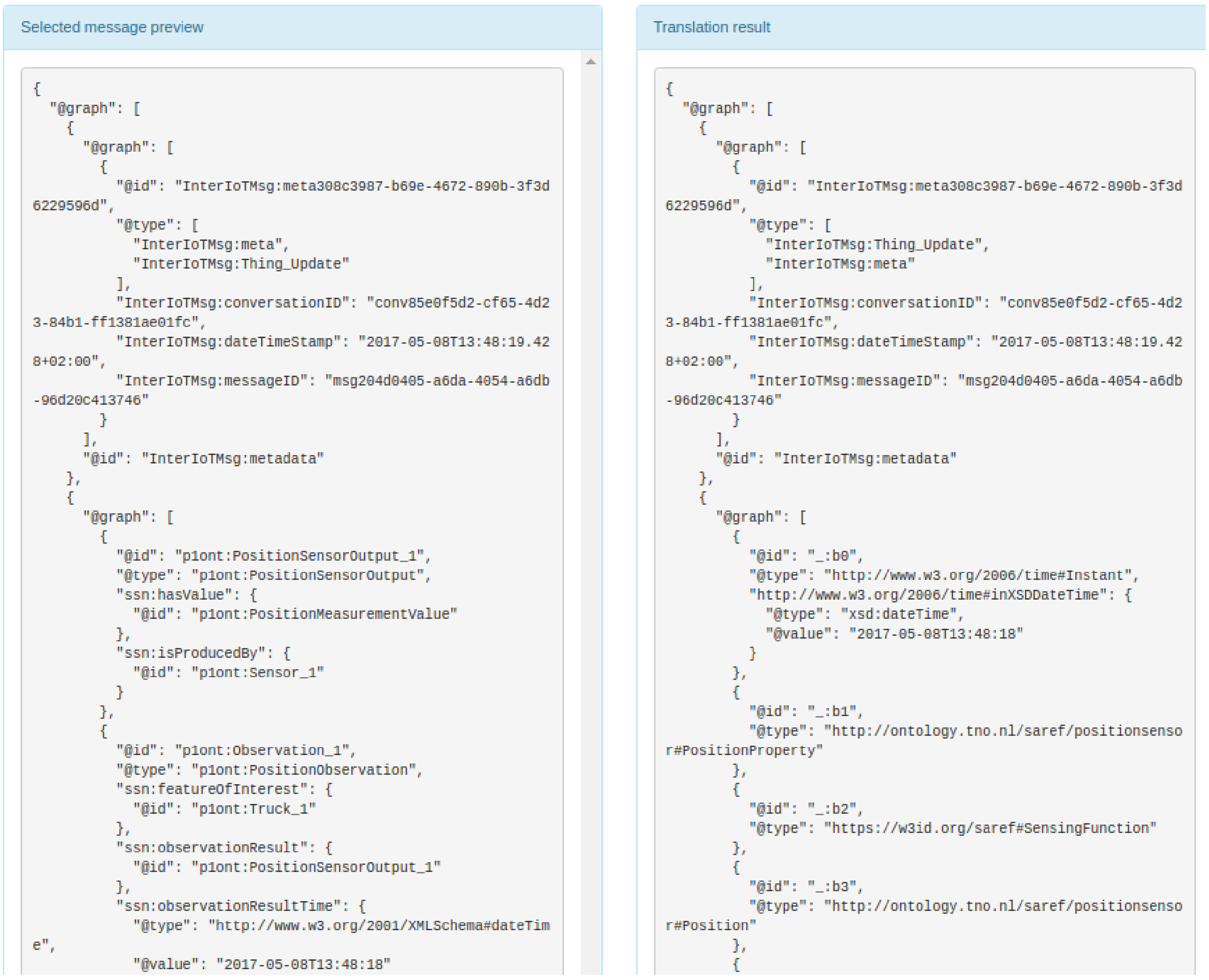
Translate

Selected message preview

```
{
  "@graph": [
    {
      "@graph": [
        {
          "@id": "InterIoTMsg:meta308c3987-b69e-4672-890b-3f3d
6229596d",
          "@type": [
            "InterIoTMsg:meta",
            "InterIoTMsg:Thing_Update"
          ],
          "InterIoTMsg:conversationID": "conv85e0f5d2-cf65-4d2
3-84b1-ff1381ae01fc",
          "InterIoTMsg:dateTimeStamp": "2017-05-08T13:48:19.42
8+02:00",
          "InterIoTMsg:messageID": "msg204d0405-a6da-4054-a6db
-96d20c413746"
        }
      ],
      "@id": "InterIoTMsg:metadata"
    },
    {
      "@graph": [
        {
          "@id": "p1ont:PositionSensorOutput_1",
          "@type": "p1ont:PositionSensorOutput",
          "ssn:hasValue": {
            "@id": "p1ont:PositionMeasurementValue"
          },
          "ssn:isProducedBy": {
            "@id": "p1ont:Sensor_1"
          }
        },
```

Translation result

Figure 47: IPSM translation



```
{
  "@graph": [
    {
      "@graph": [
        {
          "@id": "InterIoTMsg:meta308c3987-b69e-4672-890b-3f3d
6229596d",
          "@type": [
            "InterIoTMsg:meta",
            "InterIoTMsg:Thing_Update"
          ],
          "InterIoTMsg:conversationID": "conv85e0f5d2-cf65-4d2
3-84b1-ff1381ae01fc",
          "InterIoTMsg:dateTimeStamp": "2017-05-08T13:48:19.42
8+02:00",
          "InterIoTMsg:messageID": "msg204d0405-a6da-4054-a6db
-96d20c413746"
        }
      ],
      "@id": "InterIoTMsg:metadata"
    },
    {
      "@graph": [
        {
          "@id": "p1ont:PositionSensorOutput_1",
          "@type": "p1ont:PositionSensorOutput",
          "ssn:hasValue": {
            "@id": "p1ont:PositionMeasurementValue"
          },
          "ssn:isProducedBy": {
            "@id": "p1ont:Sensor_1"
          }
        },
        {
          "@id": "p1ont:Observation_1",
          "@type": "p1ont:PositionObservation",
          "ssn:featureOfInterest": {
            "@id": "p1ont:Truck_1"
          },
          "ssn:observationResult": {
            "@id": "p1ont:PositionSensorOutput_1"
          },
          "ssn:observationResultTime": {
            "@type": "http://www.w3.org/2001/XMLSchema#dateTim
e",
            "@value": "2017-05-08T13:48:18"
```

```
{
  "@graph": [
    {
      "@graph": [
        {
          "@id": "InterIoTMsg:meta308c3987-b69e-4672-890b-3f3d
6229596d",
          "@type": [
            "InterIoTMsg:Thing_Update",
            "InterIoTMsg:meta"
          ],
          "InterIoTMsg:conversationID": "conv85e0f5d2-cf65-4d2
3-84b1-ff1381ae01fc",
          "InterIoTMsg:dateTimeStamp": "2017-05-08T13:48:19.42
8+02:00",
          "InterIoTMsg:messageID": "msg204d0405-a6da-4054-a6db
-96d20c413746"
        }
      ],
      "@id": "InterIoTMsg:metadata"
    },
    {
      "@graph": [
        {
          "@id": "_:b0",
          "@type": "http://www.w3.org/2006/time#Instant",
          "http://www.w3.org/2006/time#inXSDDateTime": {
            "@type": "xsd:dateTime",
            "@value": "2017-05-08T13:48:18"
          }
        },
        {
          "@id": "_:b1",
          "@type": "http://ontology.tno.nl/saref/positionsenso
r#PositionProperty"
        },
        {
          "@id": "_:b2",
          "@type": "https://w3id.org/saref#SensingFunction"
        },
        {
          "@id": "_:b3",
          "@type": "http://ontology.tno.nl/saref/positionsenso
r#Position"
        },
        {
```

Figure 48: IPSM translation result

4.7 Cross-Layer solution

Having already described the different layered solutions that INTER-IoT provides we have to focus our view in the aspects that are left out of the scope of a specific layer. These aspects can be defined as transverse elements that affects more than one layer and can be divided in three main areas: security (4.7.1), layer interactions (4.7.2) and virtualization and clusterisation of the solutions (4.7.3). Security refers to the attribute of protect the system and the data being treated within them. This could be implemented directly within the solutions, together as one more piece into the software puzzle, or externally and crossed giving more resilience to the solution from an external point of work. Layer interactions refers to the modules created to communicate the solutions of each layer with each other in case a deployment needs the information from one layer to feed or actuate on another. Clusterisation is defined as the creation of system collections that work together with the same objective. This attribute in particular is highly interesting for scalability issues and to provide same solutions located in the same cluster to different tenants sharing resources.

As these elements are developed in the last states of the solution implementations, here we introduce the analysis, design and documentation of them. However, some of them are being currently implemented, the state of implementation is also contemplated in below sections.

4.7.1 Layer security integration

Security in D2D

Security in D2D gateway will be handled in four different levels:

- **Device Level:** At device level the security will be provided by the corresponding A.N/Protocol/Device Controller module. If the connected device has an option to use a secure communication channel or authorization mechanism will be handled by the corresponding module.
- **Physical-Virtual Communication Level:** The communication channel between the physical and virtual gateways will be performed through a secure websocket connection (*wss* protocol), thus providing a basic TLS encryption layer. Furthermore, there is an option to accept only TLS connection whose certificate root is specified by the user, trusting in this way both origin and destination in the connection.
- **Middleware Level:** This level is similar to the device level, in the sense that it will be the corresponding Middleware module that has to support the security mechanisms of the Middleware platform that will be connecting to.
- **API module:** The Gateway API extension module is secured using SSL connections (*https* protocol) and also providing Basic Authentication. The Basic Authentication credential store is connected to the security backend of Inter-IoT in order to manage users, roles and credentials.

Security in N2N

In case of network solution we will have a special treatment of the security. Firstly, we will analyze the treats and lacks in the technologies we utilize to implement the network interoperability solutions. And secondly, we will provide the security mechanism we implement to solve the lack we found in the previous study.

We can summarize Network solution main security problems as the lack of authorization by switches in the Southbound and application in the Northbound, the lack of secure communication in both bounds, the possibility of DoS attack and tampering of information. A summary of these threats can be found in Table 29 and will be explain below.

	Virtual Switches	Southbound Interface	Controller core	Northbound Interface	Application
Spoofing	✓		✓	✓	
Tampering		✓	✓	✓	
Repudiation	✓				✓
Information Disclosure		✓		✓	
DoS		✓	✓		

Table 29: Security analysis of threats in network layer

In the security analysis we perform on the network solution we identified clear sections or planes, as shown in the Table, where different approaches has to be taken to secure the solution. These planes are from the bottom parts to the upper parts: the control plane or virtual switches, the southbound interface and channel to connect the virtual switches, the core of the Ryu controller, the northbound interface and communication channel with the applications and the applications themselves. We can explain the threat by planes as follow:

- **Control Plane (Virtual Switches)**: due to the lack of authentication is possible to alter fraudulently a virtual switch and can be attacked by manipulating their behaviour to disrupt network operations.
- **Southbound Interface**: also because of no authentication mechanism the tampering and information disclosure threats arise. Also there is no access control implementation in Ryu which prevents denial of services attacks resulting in the crash of the system.
- **Controller Core**: As no authentication is provided, no trusted sources from both SBI and NBI can tamper information to be send to the controller. Also a major problem within the controller core is the lack of isolation of applications within the controller. Then the control core process and the applications run in the same context and the failure of one application could endanger the whole system. Additionally, no role based security for applications is implemented.
- **Northbound Interface**: Information Disclosure is feasible from a data flow between the controller and NBI since the channel is not encrypted and uses only HTTP instead of HTTPS. An intruder can avail all information from the traffic between controller and a NB application with a simple ARP spoofing and Man-In-The-Middle (MITM) attack. Also, the content of packets can be so modified being this plane vulnerable to data tampering.
- **Application**: as in the first case, the missing authentication mechanist to trust in the correct application could drive into Spoofing or Repudiations.

List of best practices to implement for improving the security in the solution:

- **Isolation** of the user process context from the core context is very important to achieve role based security for individual layers. Ryu uses a single context for execution, to secure this it is necessary to have a clear separation between core process and applications.

- **Monitoring of resources:** a single malicious process can lead to DoS the entire SDN network. These threats can be found in SBI and NBI, hence the controller should have distributed monitoring for SBI (monitoring the data path, already done in Ryu) and centralized monitoring for NBI. For the NBI take in considerations some monitoring applications like Snort. There is a need of an active monitoring with real time information analysis to detect if there is any abnormality in the system or in the traffic.
- **Access Control or Authentication**: Implementation of authentication mechanisms in both SBI and NBI. To avoid repudiation and spoofing. In the SBI this is mitigated recording the data path id (dpid) of individual message received. But in the NBI elevation of privileges for the applications is recommended or another access control implementation.
- **IDS:** easiest solution should be Snort that could be running in the same controller machine, so Ryu receives the notification through Unix Domain Sockets or Snort could be directly located in other machine so Ryu receives the notifications alerts via Network Socket. For both cases, you need to install a mirror flows to redirect the packets to Snort. Also, Snort will avoid the DoS attacks from both interfaces.
- **Secure communication channel:** in the SBI activating the TLS communication to avoid tampering and Information disclosure and in the NBI activating the HTTPS communication to avoid ARP spoofing, the modification of the content of the packets and MITM attack.
- **Safe model or partial restart:** for recovery purposes. To have a faster recovery mechanisms install a solution that brings only critical systems up as a first phase (the core process ryu-manager) of recovery followed by non-critical components (the application process). This is only possible if we fulfil the first practice of isolation.

As we have seen, most of the security risks can be mitigated or eliminated activating simple mechanism as TLS or HTTPS in the South and North bounds. Others, need an external software to help securing the system. An example of this is Snort [84]. Snort is a free and open source network intrusion prevention system (NIPS)and network intrusion detection system (NIDS) owned by Cisco since 2013. Snort has the ability to perform real-time traffic analysis and packet logging on IP networks. Additionally, it performs protocol analysis, content searching and matching.

Security in MW2MW

The INTERMW solution needs to take into account security at several levels, as it acts as a mediator between platforms and applications. This, security should be considered as follows:

- **INTERMW components** communicate through a message broker. This means that, no matter of the concrete message broker implementation used, communication between each component and the message broker should be secure. To this end, the security back-end will be used to generate trusted digital certificates for secure communication.
- **REST API security** is achieved through the deployment of an API Request Manager (See deliverables D4.3 and D4.5 for details). API request manager supports integration with different identity/authentication servers. In this case, the API Request Manager will be integrated with the Cross-Layer security back-end.
- **IoT Platform authentication** will be analyzed on case-by-case basis and implemented in bridges. During this process, where appropriate, cross-layer security mechanisms will be used.

[84] https://www.snort.org/.

Security in AS2AS

The AS2AS solution needs to apply security to the following levels:

- **Modeller and Orchestrator level:** The editor to provide access to the graphical environment that allows users to edit and configure service composition, is not secured by default. It is necessary to apply one of these types of authentication available:
  - Username/password credential based authentication.
  - Use an external authentication source through OAuth/OpenID based authentication. For example, Node-RED provide use a wide range of the strategies provided by Passport.

  The users have two type of permissions, full or read-only access.

  Furthermore, each instance should use HTTPS to encrypt the traffic between the client browser and the Modeller/Orchestrator and a non-standard IP PORT to implement the instances. The use of containers will provide other security features to protect the instances by adapting the security mechanisms of the Docker platform.
- **API level:** API is secured by access token. To access to the APIs, the users can obtain an access token and all subsequent API calls should then provide this token in the Authorization header.
- **Node level:** Nodes must manage internal security mechanisms to access to the services. These mechanisms will be those that the service provides to access it and it must be taken into account in the creation of the node. For example, if an authentication parameter is required in a service to access to it, this parameter must be provided as an input parameter of the node. Configuration nodes allow these tasks to be performed only once for each service.
- **Flow level:** There are also nodes that are developed to provide security functions in the communication between the elements of the flow. Such as nodes that encrypt and decrypt messages, to create secured HTTP endpoint or to interact with the security functions available in the IoT platforms.

Security in DS2DS

The *IPSM* component needs to consider security at two levels:

- **Data-Flow level:** where it receives/sends JSON-LD messages for/from translation through Apache Kafka message broker, the security is based on the SSL and uses digital certificates.
- **Configuration level:** the IPSM REST API will be protected by suitable security token mechanism. Both the digital certificates and security tokens will be generated/managed by the security back-end.

4.7.2 Layer Interactions

Part of the work done in Cross-Layer has been the identification and definition of the interaction between the layers. Not every layer interacts with each other, and each interaction is different. In this section are described and explained the main interactions between layers.

D2D ⇔ N2N

The relationship between these lower layers is specially close due to is sharing of common protocols and technologies. When we talk about access network protocols or network technologies we used to reference the two or third first layers of the network stack; hence, protocols as WiFi, Bluetooth, etc. affects these layers sometimes providing new approaches to the access networks. These different access network modules are located in the Physical part of the Gateway and, in the end, the information will be translated and encapsulated to be sent through a network over IP to the Virtual part, where the connection to the IoT Platform will be done as well through an IP-based network. Here is when our software defined network solution takes its role and manage the packets coming from the virtual gateways to the platforms.

So, as we demonstrate in our first demo the virtual gateways can communicate with the external elements among the software defined network created in the network solution. Hence, the routing between the different virtual machines that allocate the virtual gateways or other resources is managed by the controller.

Additionally, in a further approach we want to strengthen the bond between device and network layer and we will implement a module inside the virtual gateway to mark the packets sent using the IPv4 field of Differentiated Services Code Point (DSCP) previously known Type of Service (ToS), defining different services in order to treat them in a different manner within the network (QoS).

D2D ⇔ MW2MW

There is no direct iteration between the two INTER-Layer components. The reasons is, that D2D has been designed to communicate (northbound) to an IoT middleware platform. However, INTERMW is not "yet another IoT platform", but rather a set of components that allows IoT platform interoperability. This means that D2D may be "attached" to INTERMW indirectly, through another IoT platform. One such setup would be, for example, D2D attached to FIWARE, which is then attached to INTERMW.

N2N ⇔ MW2MW

The same way in the interaction between device layer and network layer the network solution in charge of routing the information between the virtual resources where virtual gateways are allocated, here is also in charge of routing the information between virtual resources where platforms are located. In many cases, there is no direct connection with the middleware to network and gateways but in some specific scenarios this could be beneficial.

MW2MW ⇔ DS2DS

INTERMW translates messages through the IPSM component for all communication with IoT platforms. This means that, in principle, all messages coming from an IoT platform get first syntactically transformed into platform-specific semantic representation (expressed in JSON-LD) and sent to IPSM for translation into the appropriate target semantics/ontology. In the opposite direction a similar procedure is in place: a message gets semantically translated into the platform-specific ontology, and later, syntactically transformed into a data format recognized by the receiving platform and sent to that platform.

D2D ⇔ AS2AS

The interaction between the D2D Gateway and AS2AS component is performed through specific nodes designed to invoke functions of the REST API published by the Gateway API extension module. All functions of the REST API are supported but the most notable nodes are the ones that allow the control and management of IoT Devices, those are: **Device Status**, **Device Start**, **Device Stop**, **Read device**, **Write device**.

It is worth noticing that this interaction process can be fully automatized since the Swagger documentation of the Gateway is already automatically generated from the code and there is the possibility (still under development) of generating nodes automatically from a swagger definition.

N2N ⇔ AS2AS

It is important to notice that even if there is a portal to access graphically the configuration options and information that the network solution provides, may be interesting to create nodes on the Application and Services solution layer to be used in concordance with other services. Thus, you can create composed services using the information of the network or create flows that acts over the network following criteria coming from different applications.

MW2MW ⇔ AS2AS

INTERMW exposes all its functionality through the REST API interface defined in `https://git.inter-iot.eu/Inter-IoT/layer_apis/src/master/middleware/intermw-swagger.json`. Although any REST endpoint could be exposed as an AS2AS node, in this phase the most suitable candidate is the *subscription* flow. A subscription, which provides a series of *observations* form a set of sensors can provide information to consumer nodes, like visualisation dashboards or CEP services of other IoT platforms.

AS2AS ⇔ DS2DS

In the solution proposed for AS2AS layer, based on Node-RED, a set of nodes dedicated for interaction with IPSM is offered. They allow to send messages to any instance of IPSM (publish them to input topics of semantic translation channels) for translation, and receive translated messages from IPSM (consume them from output topics of semantic translation channels). The assumption is that input and output messages have to be in RDF, specifically in JSON-LD message format.

4.7.3 Virtualization and Clusterization of layers with Docker

The State of the Art has described and indicated that we are using Docker to perform issues related to virtualization. The use of Docker in Inter-Layer provides the following advantages:

- **Rapid Deployment:** Bringing up a new IoT Platform or Service resource took a considerable time, sometimes more than a day. When platforms or services offer a deployment in a container, this time slot down to minutes. This has been fundamental in AS2AS and MW2MW in which these deployments were continually required to be carried out in the development phases.
- **Consistent Environment:** The immutable nature of Docker images, provides a consistent environment for the application from development through production.
- **Improve Developer Productivity:** In the development phases of interoperability solutions, programmers have close conditions to working with real platforms and services in a production environment.
- **Portability:** The containers can be easily moved between different servers that have the Docker environment installed.

- **Multi-tenancy:** this attribute allows having different solution from different layers accessible to several clients or tenants. These tenants, however, does not realize they are sharing the resources and it seems they have the system for themselves, in a specific configured context.
- **Isolation:** a good secure practice is the creation of different context for different applications. So that, if something fails in one of them does not spread to the other making the whole system crash. With docker and the isolation mechanism this is possible and useful for the applications to have its own data space and resources to work secured and properly.
- **Multiple Instances of a service:** having one service properly configured and running, this property allows the nimbly deployment of multiple instances of the same service as a replica becoming the solution more scalable and resilience (in case of failure of one of the instances the request can be redirected to another), also, if needed, the different replicas can be configured with diverse characteristics associated with each user owner.

In the Layers is done specifically in this way:

In **D2D** only the virtual gateway is dockerised, since the physical part would not make sense for obvious reasons. Both the physical and the virtual part of the gateway have the possibility to add more functionalities adding extension modules, for this reason, to dockerise the virtual part we have three approaches:

- Create an image containing all the existing module extensions for each release.
- Create multiple images with different extensions for each release.
- Create an image without extensions and give the user the possibility of adding them in a shared folder between the host and the container.

Of the three possibilities the first one is the basic one and the image is automatically compiled in each release. The second option is not going to be considered, but documentation will be given to the user in order to instruct how to compile custom images with the desired extensions. Finally, the last option is still under development, so the user has complete control of the extensions that will be used or not in each deployment.

In **N2N** the use of Open vSwitch (OVS) bridge for Docker container has been studied. However, it provides more difficulties than advantages. It is possible to use the ovs-docker utility so that, you can manage the network and routing between containers with the technology of OVS but it is more complex the configuration of the whole network following specific requirements and the connection with the controller. Even though, we will inquire into in the possibilities of interconnecting the containers with our software defined network. There are two main modes of configuration of OVS with Docker:

- The NAT mode: the OVS bridge is a virtual interface so the docker containers added to the bridge will have an internal IP address, iptables NAT rules will be needed to communicate with outside.
- The Bridge mode: the OVS bridge is associated with a real network adapter, the docker containers added to the ovs bridge will be bridged to the external network. You have to add the physical network adapter into the OVS bridge, and configure an external IP address onto the OVS bridge.

This technology will be analysed tested and compared with the other option deployment with OpenStack and we will determine which one fits better our needs.

In **MW2MW** currently, it is used to test the Middleware IoT Platforms like Fiware Orion Context Broker, deploy middleware brokers like Kafka, RabbitMQ and ActiveMQ and running an instance of the Parliament Database. Additionally, we are deciding which components of MW2MW solution will be definitely dockerised.

In **AS2AS** it is used to deploy IoT services (for example, the services of the AS2AS Demo) and create multiple custom instances of Node-RED tool. We have exposed this need of use containers in the Section of AS2AS solution. The Figure 31 explains the configuration of a container with the interoperability solution. The Figure 32 is the current deployment of the solution and shows the multiple instances of the solution in the same host. Finally, the Figure 33 exposes the need to use Docker Swarm to handle hosts and container with different instances of Node-RED in a centralized way.

In **DS2DS** there is no need for using Docker so far.

The Docker Swarm tool provides unitary administration of a docker cluster providing also features of scheduling and management. Docker swarm provides the clustering features desired for the INTER-IoT layered solutions. We arrange several servers with Docker tool installed and with several containers running inside each one of them. Each container allocate instances of a particular Inter-Layer solution. We need Docker Swarm for allowing a centralized management property from a unique access point to our entire Docker environment. In the Figure 49 we observe an example of Docker deployment managed by Docker Swarm software. Concretely, it is observe an example with three servers where a diverse number of instances of Inter-layer solutions are running in Docker containers. These servers possess it own management console but as all of them belongs to the same swarmed deployment they are centrally managed by the console of Docker Swarm that agglutinates the characteristics of each Docker container making them a unique software element.

Docker Portainer is a lightweight management UI and consists in a single container that can run on any Docker engine. It allows an easy manage of our Docker environments (Docker hosts and Swarm clusters). In our project, its purpose is to be a management tool that offers us a more comfortable way to work with the Docker Environment. We are considering to integrate Portainer in Inter-FW.

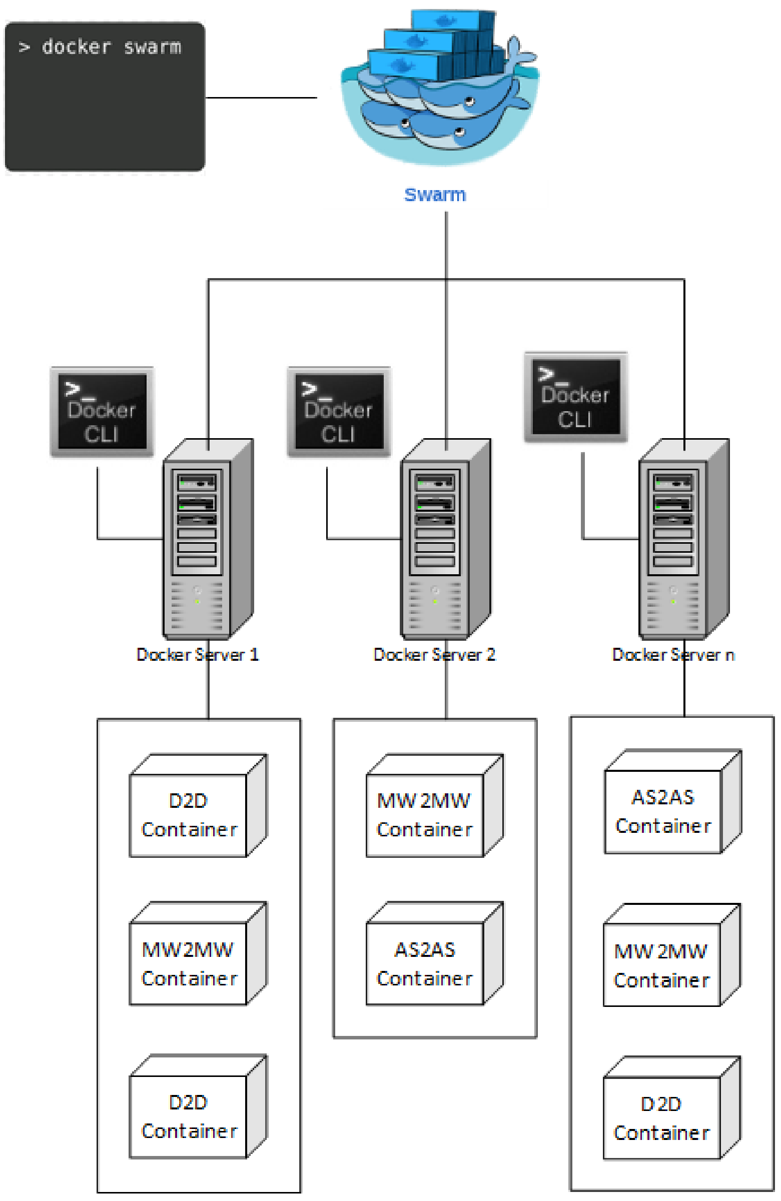


Figure 49: Example of deployment of Inter-Layer solutions with Docker swarm

4.8 INTER-Layer relation with INTER-Framework

The IoT Interoperability Framework (INTER-FW) aims at providing mechanisms, tools and helper contents to make proper use of the Layer Interoperability Infrastructures (LIIs) and Interoperability Layer Interfaces (ILI).

As described in previous sections, each LII provides generic interoperability means for different identified technology layers in IoT platforms. The INTER-FW provides a way to 1) select the appropriate LIIs for the particular scenario (e.g. middleware and services excluding the rest); 2) abstract generic behaviours present in the most common scenarios (e.g. security administration, device discovery) even though these behaviours apply to different LIIs (e.g. device discovery); 3) extend the generic

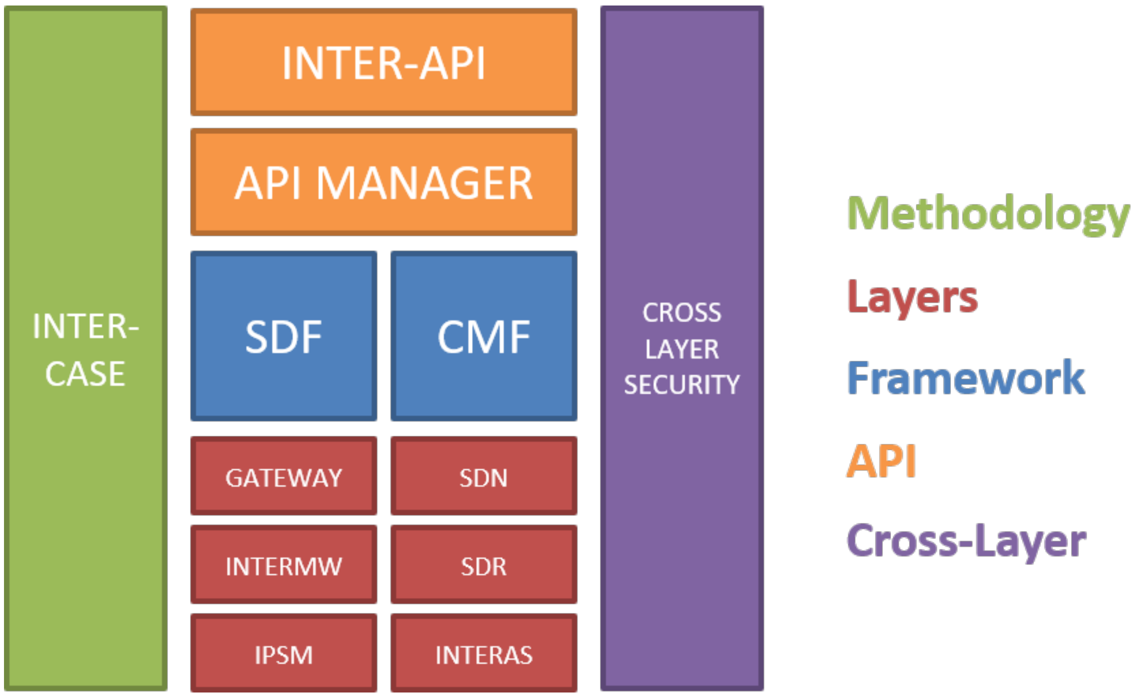


Figure 50: INTER-IoT global architecture. INTER-FW services lay over the LIIs

functionalities provided to the specific cases of the scenario (e.g. extend the service interoperability for a particular natively supported service); 4) provide a single entry point (the framework user interface) to start using the different capabilities of INTER-IoT, with a common, harmonized API with all the documentation in one place; and 5) unify when possible the development process with a homogenized approach to access libraries, develop, configure, deploy and test through the Software Development Framework.

Thus, INTER-FW provides access to INTER-LAYER structures and mechanisms through the APIs provided by components of the five layers, as described in the previous sections. In INTER-IoT, these APIs are exposed only internally (i.e., the ILIs are not accessible directly by an INTER-IoT user or third party), however, a good part of these APIs will be exposed almost identically through the INTER-FW API. For this reason, INTER-FW design and ILIs are processes that depend between them. INTER-FW specifications (especially those coming from the Reference Architecture and the Meta-Data Model tasks have an influence in the ILIs design, and, at the same time, the LIIs capabilities affect the framework definition and the global relation among ILIs.

The latest version of INTER-FW GUI is available in the `http://vmplsp04.westeurope.cloudapp.azure.com/interiot_wfk/#`. The current version of INTER-FW provides interfaces to every layer:

- D2D: The management interface of the D2D layer is under the *Gateway* tab. It contains a summary view and access to virtual gateway configurations, physical gateway configurations and a shortcut to the API operations (for testing.)
- N2N: It is managed under the section *Network*, and it contains the network topology, virtual network statistics and the QoS control interface.
- MW2MW: This interface is available in the *Platforms* tab. It shows a summary view of the

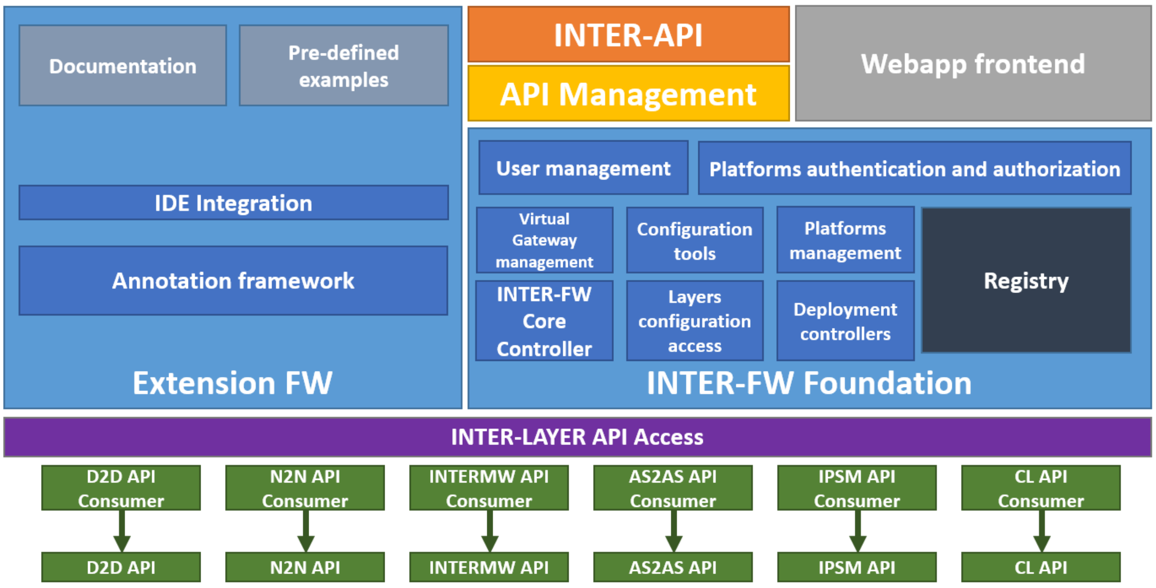


Figure 51: INTER-FW high-level architecture

platforms connected through INTER-IoT. The list of platforms is navigable, so it contains a detailed view for each platform and a list of devices connected to that platform. The list of devices allows basic operations (creation, update, removal).

- AS2AS: It lays under the *Services* tab, allowing to manage multiple service orchestration flows. Each flow give access to the service composition tool.
- DS2DS: It allows operations with the IPSM, accessed through the *Semantics* tab. It allows monitoring of the tool allowing start and stop translation channels, the configuration of channels and alignments and, finally a tool to test the current alignment with sample messages.
- Cross Layer: The cross layer is also represented through the *Users* tab. This tab allows to manage the INTER-IoT Users tab, which interacts with the Identity Server and allows a fine-grained authorization of the user in all the layers entities.

Apart from this direct contributions, all individual Layers APIs are indirectly exposed through INTER-API, which, at the same time, has a Swagger-based interface available in INTER-FW UI. This can be found under the *API Management* tab.

As the INTER-FW UI is a multi-user interface and the layers are interoperability tools that generally does not manage the concept of *User*, a containerization solution has been developed, to support different users to access independent instances of layer solutions. This has been achieved by using Docker containers and Docker-swarm to clusterise the access to these containers. This way, each user will be able to access a different container containing the layer solution and, at the same time, the group of container will be managed as a single entity. Docker-swarm also provides security and authorization mechanisms enabling a full control on the access of the different users.

5 Ethics

5.1 Introduction

Ethics is a central consideration to all INTER-IoT planning and development. As requested at the interim review, an ethical advisory board has been established. This board, within INTER-IoT, continuously reviews ethical issues. The aim of the committee is to ensure that ethical considerations and issues are addressed in the conduct of the research and development work undertaken within the project. The committee seeks to support and encourage the process of ethically conducted research to maintain the safety and well-being of participants and researchers to promote ethical values.

5.2 Ethics and INTER-LAYER

INTER-LAYER is designed to enable the interoperability of existing IoT systems at different levels. Ethical considerations must be taken into account from two different perspectives: From the systems being connected, and from the inner workings of each layer that provide said interoperability. INTER-LAYER cannot be responsible for the proper ethical handling of data and security within each connected IoT system, but it can and must assure that such handling in each system will not be worsened by connecting to INTER-LAYER. In addition, the inner logic of the components in each layer must comply with the proper ethical handling of data and security, as per the requirements and principles enumerated in the following sections.

5.2.1 Data types

Primary focus of the ethical review of data management focuses on personal data and sensitive personal data. Personal data means data which relate to a living individual who can be identified –

(a) from those data

(b) from those data and other information which is in the possession of, or is likely to come into the possession of, the data controller, and includes any expression of opinion about the individual and any indication of the intentions of the data controller or any other person in respect of the individual.

Sensitive personal data means personal data consisting of information as to -

(a) the racial or ethnic origin of the data subject,

(b) his political opinions,

(c) his religious beliefs or other beliefs of a similar nature,

(d) whether he is a member of a trade union,

(e) his physical or mental health or condition,

(f) his sexual life,

(g) the commission or alleged commission by him of any offence, or

(h) any proceedings for any offence committed or alleged to have been committed by him, the disposal of such proceedings or the sentence of any court in such proceedings.

It is possible that INTER-IoT and INTER-LAYER will be used during processing of these types of data, so appropriate controls have to be built into the layer components to enable systems to do this ethically by conforming to the data protection act. This is achieved by assuring the proper security of the communications and data handling within the components of each layer and across layers, as per the security considerations described in previous sections, encompassing the common aspects of integrity, privacy, authentication, authorization of information systems, and the other, more "IoT-specific" ones of pseudonimity, autonomous communication, and semantic querying.

5.2.2 Requirements for ethical data processing

The data protection act requires adherence to 8 principles:

1. Personal data shall be fairly and lawfully processed as defined in the data protection act.
2. Personal data shall be obtained only for one or more specified and lawful purposes, and shall not be further processed in any manner incompatible with that purpose or those purposes.
3. Personal data shall be adequate, relevant and not excessive in relation to the purpose or purposes for which they are processed.
4. Personal data shall be accurate and, where necessary, kept up to date.
5. Personal data processed for any purpose or purposes shall not be kept for longer than is necessary for that purpose or those purposes.
6. Personal data shall be processed in accordance with the rights of data subjects under the data protection act.
7. Appropriate technical and organisational measures shall be taken against unauthorised or unlawful processing of personal data and against accidental loss or destruction of, or damage to, personal data.
8. Personal data shall not be transferred to a country or territory outside the European Economic Area unless that country or territory ensures an adequate level of protection for the rights and freedoms of data subjects in relation to the processing of personal data.

Because the components of each layer do not interpret the payload data being transferred, handled and/or stored within them, they do not hinder the above principles per se. It is however necessary that during the process of implementation of each component the developers are aware of the principles themselves, so as to not introduce unwanted features, workarounds, hot fixes or "hacks" that interfere with them, even if in a temporary fashion, as long as there is the chance that these are utilized in the pilots real-life deployments. It therefore becomes an official recommendation for developers to read the above principles and take them into consideration during the development process.

6 Conclusions

This document is the intermediate report (out of three) on the status of design and development of INTER-IoT interoperability mechanisms. It describes the status of development of the different components, entities and interfaces of the layered approach proposed by the project. In this report, we provide an updated state of the art section, but the real emphasis is on the technological choices made in the development of interoperability layers.

INTER-LAYER components have reached a level of maturity when they can be deployed in INTER-IoT pilots and validated in a real-world scenario. All relevant aspects of interoperability have been addressed, including syntactic, semantic and layered interoperability. Examples of functional demonstrators for each layer are provided, together with exposed APIs for each layer, thus providing the fundamentals for the implementation of Cross-Layer and INTER-FW solutions. As it can be seen from the development status, the priority in this phase of the project was on providing a MVP (Minimum Viable Product) for pilot deployment and APIs to allow further developments of INTER FW.

When layered components reached a satisfactory level of maturity, activities have started in the identification of common Cross-Layer issues and interaction among layers. Two main cross-cutting issues have been identified, and existing solutions examined in the SotA section: Security and virtualisation/clustering. Cross-Layer proposes solutions for security and virtualisation while also describing the interaction between layers.

Overall, this document reports significant advancements in the development of INTER-Layer components. It also proposes some cross-layer solutions and describes the initial implementation steps undertaken.

Bibliography

[1] G. Fortino, D. Parisi, V. Pirrone, and G. D. Fatta, “Bodycloud: A saas approach for community body sensor networks,” *Future Generation Computer Systems*, vol. 35, no. 6, pp. 62–79, 2014.

[2] “Bodycloud website.” http://bodycloud.dimes.unical.it.

[3] “The javascript object notation (json) data interchange format.” https://tools.ietf.org/html/rfc7159.

[4] “Linked data.” https://www.w3.org/DesignIssues/LinkedData.html.

[5] C. Bizer, T. Heath, and T. Berners-Lee, “Linked data – the story so far,” *Int. J. Semantic Web Inf. Syst.*, vol. 5, no. 3, pp. 1–22, 2009.

[6] “JSON-LD 1.0 – a JSON-based serialization for Linked Data.” https://www.w3.org/TR/json-ld/.

[7] “Resource description framework (RDF).” https://www.w3.org/RDF/.

[8] “Semantic Sensor Network XG final report,” 2011.

[9] M. Compton, P. Barnaghi, L. Bermudez, R. Garcia-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W. D. Kelsey, D. L. Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, and K. Taylor, “The SSN ontology of the W3C semantic sensor network incubator group,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 17, pp. 25–32, 2012.

[10] “Semantic Sensor Network Ontology.” https://www.w3.org/TR/vocab-ssn/.

[11] P. Szmeja, M. Ganzha, M. Paprzycki, W. Pawłowski, K. Wasielewska, B. Solarz-Niesłuchowski, and J. Suárez de Puga García, “Towards high throughput semantic translation,” in *3rd EAI International Conference on Interoperability in IoT (InterIoT)*, in press.

[12] M. Ganzha, M. Paprzycki, W. Pawłowski, P. Szmeja, and K. Wasielewska, “Streaming semantic translations,” in *21st International Conference on System Theory, Control and Computing ICSTCC, Proceedings*, in press.

[13] J. David, J. Euzenat, F. Scharffe, and C. Trojahn dos Santos, “The alignment API 4.0,” *Semantic Web*, vol. 2, no. 1, pp. 3–10, 2011.

[14] “EDOAL: Expressive and declarative ontology alignment language.” http://alignapi.gforge.inria.fr/edoal.html.

[15] P. Szmeja, M. Ganzha, M. Paprzycki, W. Pawłowski, and K. Wasielewska, “Declarative ontology alignment format for semantic translation,” in *6th EAI International Conference on Context-Aware Systems and Applications (ICCASA), Proceedings*, in press.