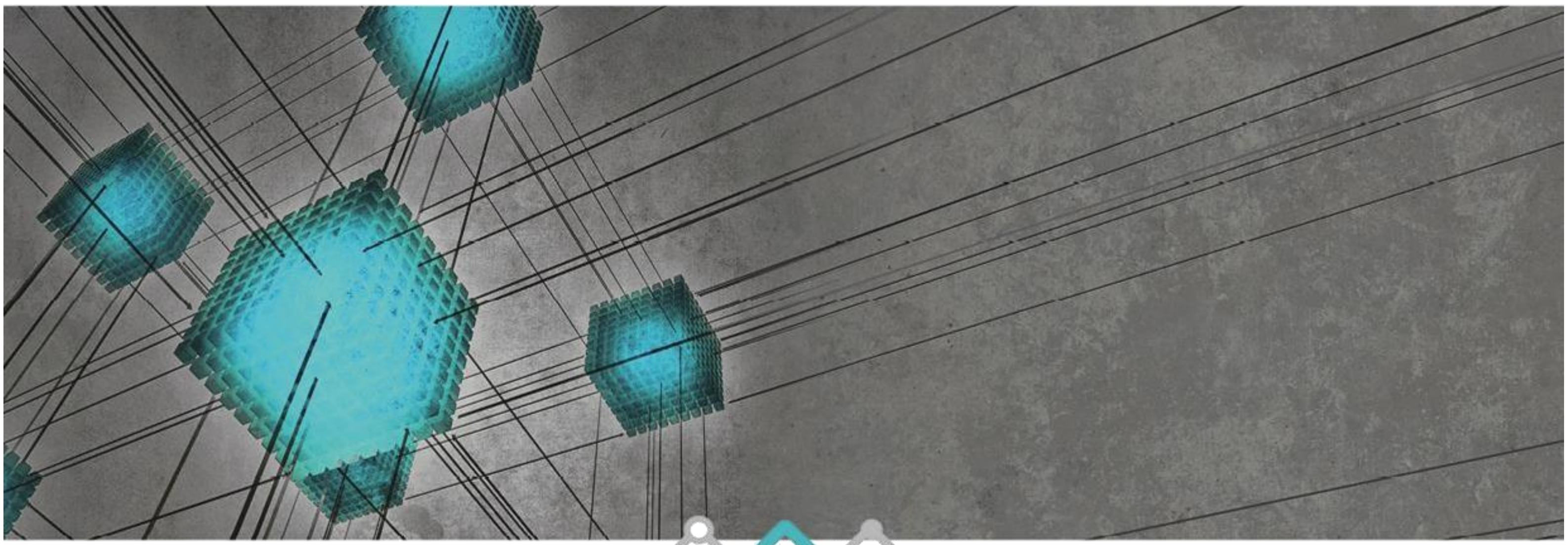


interiot

INTEROPERABILITY
OF HETEREOGENEUS
IOT PLATFORMS.

D5.1

Design Patterns for Interoperable IoT Systems

December 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 will enhance different IoT scenarios. Moreover, reuse and integration of existing and future IoT systems will be facilitated, creating a defacto 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 them to market as fast as possible.

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

INTER-IoT

Design Patterns for Interoperable IoT Systems

Version: 2.3

Security: Public

27 December 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 this document is to provide the background to the research in design of patterns for interoperable IoT systems. The work is done within “T5.1 Definitions of Design Patterns for Interoperable IoT Systems” task:

“To enable an agile design of the integration of heterogeneous IoT platforms into an integrated and interoperable IoT platform, in this task, we focus on the definition of a collection of design patterns with the aim of driving the integration designer to provide the most effective solutions. Like in related contexts (e.g. software engineering), design patterns are fundamental to ease the design phase by reusing patterns well documented with example schemas and application guidelines. The design patterns will incorporate the methods defined in WP3 and will be contextualized in different application domains (e.g. m-Health, Transportation and Logistics) to furnish well-formalized domain-specific guidelines. Thus, according to such guidelines, designers can produce rapid and effective specifications for the IoT platform integration at any desired layer and also cross-layer.”

Very important is fact, that this document describes the design patterns specific for INTER-IoT, not every pattern, used in the design and development phases.

The structure of this deliverable is as follows. Firstly, section *“Introduction*” explains the deliverable's issue and necessary definitions. Moreover, this section presents methodology, used in the process of defining new INTER-IoT design patterns. Next, section*“State of the art - research and analysis”* contains the SotA analysis, i.e. presentation of common patterns catalog (including comments about utilities for INTER-IoT). Section *“INTER-IoT patterns catalog”*presents final INTER-IoT patterns catalog. Every design pattern contains detailed description, including the solving problem, sources of inspiration and example usage in the project implementation. The aim of section *“Ethics”* is to describe the role of design patterns within the ethics aspects. Finally, section*“Conclusions”* contains conclusions of whole document content.

List of Authors

Organisation	Authors	Main organisations' contributions
SRIPAS	Katarzyna Wasielewska-Michniewska, Rafał Tkaczyk	Document structure, Executive summary, Introduction, Ethics, Conclusions, Document formatting, Initial version, SOTA of design patterns: ontology, reactive, integration. Design patterns of DS2DS layer
UNICAL	Giancarlo Fortino	SotA of design patterns: agent-oriented, IoT
PRODEVELOP	Miguel Montesinos, Miguel Ángel Llorente Carmona	SotA of design patterns: GoF. Design patterns of INTER-FW
VPF	Pablo Giménez Salazar	SotA of port logistics use case specific patterns. Design pattern of INTER-LogP
XLAB	Matevž Markovič	SotA of design patterns: Enterprise Bus, Micro-Services. Design patterns of MW2MW layer
UPV	Regel González-Usach, Alfonso Camanes Navarro, Eneko Olivares, Carlos E. Palau	SotA of design patterns: GoF. Design patterns of D2D, N2N, AS2AS layers, and Cross-Layer Final Review
UPV-SABIEN	Álvaro Fides Valero, Maria Jesus Arnal Parada	SotA of (e/m) Health use case specific patterns. Design patterns of INTER-Health

Change control datasheet

Version	Changes	Pages
1.0	SotA analysis and redefining	62
1.1	Document formatting; SotA analysis and redefining, Analysis and refinement the patterns' descriptions; Reference SotA to patterns' descriptions; Design Patterns Template	81
1.2	Review of all patterns (after redefining, including modified diagrams). Submit new patterns in sections: CROSS-Layer, INTER-FW, INTER-Health and INTER-LogP	94
1.3	Redefined all the document (changed document structure, make shorter SotA, add sections about analysis of patterns defining process, insert correlations between SotA analysis and design patterns, etc.)	77
1.4	Add Ethics section	79
2.0	Ready for internal review	80
2.1	Reviewed by internal reviewers	80
2.2	Addressed review comments	83
2.3	Final review	85

Contents

List of Figures

List of Tables

Acronyms

AIDC	Automatic Identification and Data Capture
API	Application Programming Interface
AS2AS	Application & Services Interoperability
CODePs	Conceptual Ontology Design Patterns
D2D	Device-to-Device
DS2DS	Data & Semantics-to-Data & Semantics
EDOAL	Expressive Declarative Ontology Alignment Language
FBP	Flow-based programming
GDPR	General Data Protection Regulation
GoF	Gang of Four
GOIoTP	Generic IoT Platform Ontology
GUI	Graphical User Interface
GW	Gateway
HCIS	Health Care Information Systems
INTER-FW	INTER-IoT Interoperable IoT Framework
INTER-Health	INTER-IoT Platform for Health monitoring
INTER-IoT	Interoperability of Heterogeneous IoT Platform
INTER-LAYER	INTER-IoT Layer integration tools
INTER-LogP	INTER-IoT Platform for Transport and Logistics
INTER-MW	INTER-IoT Middleware
IoT	Internet of Things
IoT-EPI	IoT-European Platforms
MW2MW	Middleware-to-Middleware
N2N	Networking-to-Networking
ODPA	Ontology Design & Patterns
ODPs	Ontology design patterns
OWL	Web Ontology Language
PCS	Port Community System
QoS	Quality of service
REST	Representational State Transfer
RDF	Resource Description Framework
SDN	Software-defined networking
SOA	Service-Oriented Architecture
SotA	State-of- the-Art
SSL	Secure Sockets Layer
UML	Unified Modeling Language

1 Introduction

Design patterns provide the most effective solutions, used and tested many times. Therefore, it was necessary to select an appropriate set of design patterns in order to design and develop approaches defined in WP3, WP4 and WP6. It was necessary to take into account every INTER-IoT element, i.e. INTER-LAYER (D2D, N2N, MW2MW, AS2AS, DS2DS, and CROSS-Layer), INTER-FW, INTER-Health and INTER-LogP.

During the process of designing INTER-IoT elements (INTER-LAYER, INTER-FW, INTER-Health, and INTER-LogP)[1], most issues did not bring the difficulties and were able to solve, using the already defined, well known approaches. However, it was not possible with some problems. The reason of this was that every INTER-IoT element generates the specific approach to the problem. Specific, means typical for the INTER-IoT paradigms. The most common and problematic aspects were: architecture, interoperability, integration and communication. Therefore, it was necessary to prepare new design patterns (INTER-IoT Layer Patterns). Most of them are modifications of existing solutions, but some of them are brand new approaches. The section titled *“INTER-IoT issues and solutions (final patterns catalog)”*describes in details the issues that have arisen and explain the solution, designed to solve them.

1.1 Definitions and terminology

Table 1: Important definitions

Term	Definition
Design pattern	A general reusable solution to a problem that recurs repeatedly within a given context in software design. It is a written document (template) describing how to solve a problem that can be used in many different situations. Design patterns are formalized best practices. The purpose is to increase re-use and quality of code and at the same time reduce the effort of development of software systems.
Pattern system	A pattern system is a cohesive set of related patterns which work together to support the construction and evolution of whole architectures. It describes many interrelationships between the patterns and their groupings and how they maybe combined and composed to solve more complex problems.[2]
Pattern language	A collection of patterns and the rules to combine them into an architectural style. Pattern Languages describe software frameworks or families of related systems. [3]
Pattern catalog	A collection of related patterns, where patterns are subdivided into a small number of broad categories, which usually include some amount of cross referencing between patterns. [2]
Ontology	Explicit specification of shared conceptualization.
Ontology alignment	The process of finding of correspondences between two or more ontologies. The result of this process is an alignment of a set of correspondences between entities (atomic alignment) or groups of entities and sub-structures (complex alignment) from different ontologies. A correspondence can be either a predicate about similarity, called a matching, or a logical axiom mapping.

1.2 INTER-IoT Design Patterns defining process – methodology

The process of defining the INTER-IoT Layer Patterns is depicted in Figure 1. The final set of patterns is a result of research of already existed solutions, good practices and also knowledge and experience of specialists working on the INTER-IoT project.

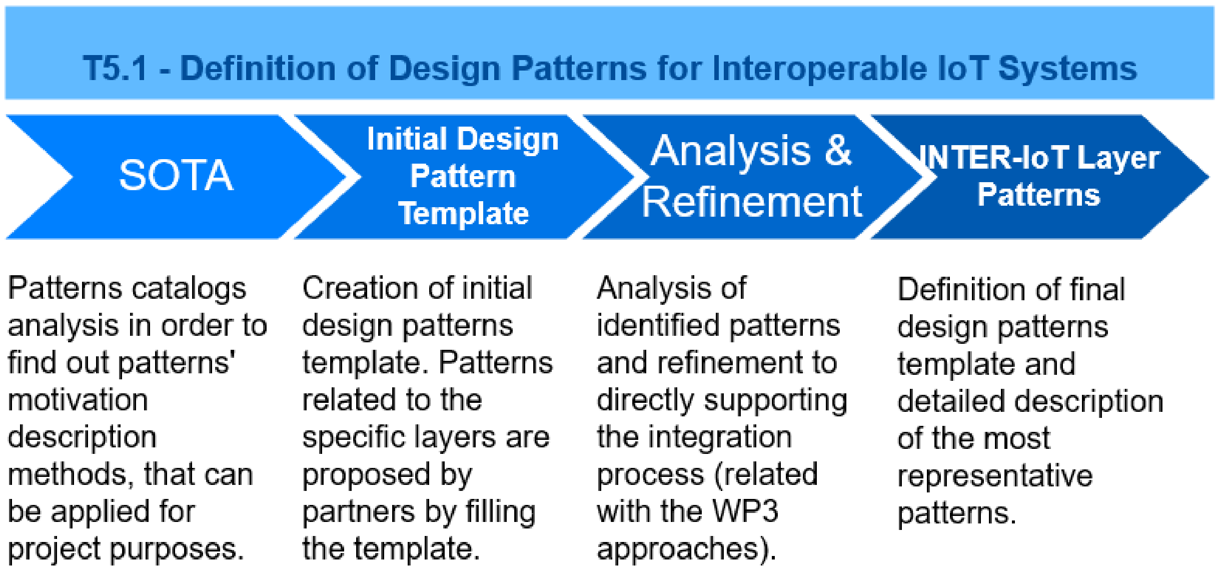

T5.1 - Definition of Design Patterns for Interoperable IoT Systems
SOTA
Initial Design Pattern Template
Analysis & Refinement
INTER-IoT Layer Patterns
Patterns catalogs analysis in order to find out patterns' motivation description methods, that can be applied for project purposes.
Creation of initial design patterns template. Patterns related to the specific layers are proposed by partners by filling the template.
Analysis of identified patterns and refinement to directly supporting the integration process (related with the WP3 approaches).
Definition of final design patterns template and detailed description of the most representative patterns.

Figure 1: INTER-IoT Design Patterns creation process

First step of INTER-IoT Layer Patterns definition process was state-of-the-art study (described in details in Section: *"State of the art - research and analysis"*). The main goal of this step was to examine the common, well known pattern catalogs (object-oriented Patterns, integration patterns, reactive patterns, agent design patterns, ontology patterns, IoT patterns, security patterns, use case patterns) and extract the knowledge (e.g. verify the applicability to the IoT domain) useful to define an INTER-IoT Design Patterns.

On the basis of SotA conclusions, the next step was to define an initial set of INTER-IoT Design Patterns. It was decided, that draft template should provide a subcategorisation for INTER-LAYER (D2D, N2N, MW2MW, AS2AS, DS2DS, CROSS-Layer), INTER-MW, INTER-Health and INTER-LogP. After extraction of the initial set of patterns, a detailed analysis was made. The main idea was to extract the patterns that directly corresponding to the integration solutions, already achieved in the WP3. Therefore, the set of patterns has been narrowed down to a smaller number of entities. Moreover, in final version, the design patterns template structure was changed in a way to describe the content in a more clearer form.

Section *"INTER-IoT patterns catalog"* shows the output result of all the design patterns creation process, i.e. defining initial set of patterns, analysis and refinement and the final product.

2 State of the art - research and analysis

Design patterns provide a way to build an end-to-end solution in well-specified ways and to provide an understanding of the use of different components of the system in a system context. A specific architecture can be constructed from a set of design patterns, and from this the (dynamic) behaviour of the system may be modelled and analyzed.

In the following subsections we introduce pattern catalogs that address different issues to be considered when designing IT solutions, from data modelling to architectural patterns.

Among these we will try to identify patterns that can be potentially reused in our context, or that can provide an inspiration for the definition of new design patterns. It should be noted, that we are particularly interested in patterns for achieving interoperability between platforms in the IoT domain. Interoperability can be defined as *the ability of two or more systems or components to exchange data and use information*, so in most cases we will be interested in communication/ integration/ interoperability patterns that should support the design and development of an integrated IoT platforms ecosystem.

Specifically, we are interested in design patterns for:

- integration of IoT platforms,
- communication of IoT platforms,
- security,
- architecture of software components, required to provide interoperability, that should be implemented for specific deployment,
- domain use case solutions.

2.1 Object-oriented Patterns “Gang of Four”

In 1994, the publication of the book [4] explained the usefulness of patterns and resulted in the widespread popularity for design patterns. The authors together are referred to as the Gang of Four (GoF). The authors documented the 23 patterns classified into three groups and two scopes ([4][5]):

- Creational Patterns: Used to construct objects so that they can be decoupled from their implementing system. They help make a system independent of how its objects are created, composed, and represented.
- Structural Patterns: Used to form large object structures between many disparate objects. They are concerned with how classes and objects are composed to form larger structures.
- Behavioral Patterns: Used to manage algorithms, relationships, and responsibilities between objects.
- Object Scope: Deals with object relationships that can be changed at runtime.
- Class Scope: Deals with class relationships that can be changed at compile time.

In [6] GoF are interviewed and give the following remarks about their original work. The originally used pattern description template was sufficient for low-level object-oriented patterns like the ones in the book. It was not good for other purposes, that resulted in creation of new templates for other purposes. Every collection of patterns needs a standard template, but no template will ever be good for all patterns. GoF had suggested some changes to the original catalog.

2.1.1 Analysis of GoF

Object-oriented patterns, proposed by GoF, form the basis in software engineering design patterns and serve as an important source for object-oriented design theory and practice. However, it should be noted that they address only object-oriented programming. They are sufficient to model internal architecture of a software system, but they are not enough to model integration or communication between components that are crucial in distributed domains such as IoT. In the scope of our work, GoF should be considered as reference when there is a need to provide design patterns for a software component that needs to be developed during IoT platforms integration to achieve interoperability.

The valuable input of GoF is the pattern specification template that was also reused by other initiatives. It is described in details in section*"Pattern Forms"*.

2.2 Integration Patterns

The most notable work in the integration patterns field is a classic book by Gregor Hohpe and Bobby Woolf [7] that describes 65 design patterns (divided into 7 categories) for the use in enterprise application integration and message-oriented middleware. The patterns provide technology-independent design guidance for developers and architects to describe and develop robust integration solutions. Specifically, these patterns are also applicable to IoT domain where platforms are integrated with message-based communication.

Integration patterns model the flow of a message from one system to the next through channels, routing, and transformations, and are implemented in software such as ESB, Apache Camel, Mule ESB, Red Hat JBoss Fuse etc.

2.2.1 Enterprise Integration Patterns

Integration patterns classification is as follows[8] (underlined pattern is a base in a category):

Table 2: Enterprise Integration Patterns

Pattern category	Description	Patterns
Integration Styles	Different ways applications can be integrated, providing a historical account of integration technologies.	• File Transfer • Shared Database • Remote Procedure Invocation • Messaging
Messaging Channels	How messages are transported across a message channel (virtual pipe that connects a sender to a receiver); implemented by most commercial and open source messaging systems.	• Message Channel • Point-to-Point Channel • Publish-Subscribe Channel • Datatype Channel • Invalid Message Channel • Dead Letter Channel • Guaranteed Delivery • Channel Adapter • Messaging Bridge • Messaging Bus
Messaging Patterns	Describe the intent, form and content of the messages that travel across the messaging system.	• Message • Command Message • Document Message • Event Message • Request-Reply • Return Address • Correlation Identifier • Message Sequence • Message Expiration • Format Indicator

Pattern category	Description	Patterns
Messaging Routing	How messages are routed from a sender to the correct receiver; message routing patterns consume a message from one channel and republish it message, usually without modification, to another channel based on a set of conditions.	• Pipes and Filters • Message Router • Content-based Router • Message Filter • Dynamic Router • Recipient List • Splitter • Aggregator • Resequencer • Composed Message Processor • Scatter Gather • Routing Slip • Process Manager • Message Broker
Message Transformation	How the content of a message can be changed, for example to accommodate different data formats used by the sending and the receiving system; data may be added, removed or rearranged.	• Message Translator • Envelope Wrapper • Content Enricher • Content Filter • Claim Check • Normalizer • Canonical Data Model
Message Endpoints	How messaging system clients produce or consume messages.	• Message Endpoint • Messaging Gateway • Messaging Mapper • Transactional Channel • Pooling Consumer • Event-driven Consumer • Competing Consumers • Message Dispatcher • Selective Consumer • Durable Subscriber • Idempotent Receiver • Service Activator
System Management	Describe the tools to keep a complex message-based system running, including dealing with error conditions, performance bottlenecks and changes in the participating systems.	• Control Bus • Detour • Wire Tap • Message History • Message Store • Smart Proxy • Test Message • Channel Purger

2.2.1.1 Analysis of Enterprise Integration Patterns

Enterprise Integration Patterns are widely accepted and implemented in many software solutions. They provide guidance when integrating systems or designing a distributed systems. Since INTER-IoT is focused on interoperability, some patterns from these groups shall be identified in the architecture, specifically, cooperation between components. Selected patterns from integration patterns should be identified in the proposed architecture of interoperable IoT Platforms. Patterns from “Message transformation”, “Messaging Patterns”, “Messaging Routing”, are candidates to be applicable e.g. in achieving INTER-IoT artifacts interoperability.

2.2.2 SOA Patterns

SOA (Service-Oriented Architecture) stems from distributed computing paradigm and, as a result, SOA design patterns[9], can be rooted in already existing pattern catalogs (Enterprise Integration Patterns, Enterprise Application Architecture Patterns, Software Architecture Patterns, Object-Oriented Design Patterns, etc.). SOA is a paradigm of a software design in which services are provided to the other components by application components, through a communication protocol over a network. Even though SOA approach is traditionally used to couple functionality of heavyweight enterprise systems, it can as well become applicable to embedded/smart devices.

Table 3 shows how design patterns reference SOA design principles. Design principles can be understood as „meta-patterns" and can form a basis to structured analysis of the proposed solution with respect to design patterns.

Table 3: Summary of SOA design patterns

Design Principle	Related Design Patterns	Description
Abstraction	Capability Composition, Capability Recomposition, Decomposed Capability, Domain Inventory, Dual Protocols, Enterprise Inventory, Entity Abstraction, Exception Shielding, Inventory Endpoint, Legacy Wrapper, Policy Centralization, Process Abstraction, Service Perimeter Guard, Service Refactoring, Utility Abstraction, Validation Abstraction	*"Service contracts only contain essential information and information about services is limited to what is published in service contracts."*
Autonomy	Canonical Resources, Capability Composition, Capability Recomposition, Composition Autonomy, Distributed Capability, Dual Protocols, Event-Driven Messaging, Process Centralization, Redundant Implementation, Service Data Replication, Service Normalization	*"Services exercise a high level of control over their underlying runtime execution environment."*
Composability	Agnostic Capability, Agnostic Sub-Controller, Brokered Authentication, Capability Composition, Capability Recomposition, Composition Autonomy, Cross-Domain Utility Layer, Data Confidentiality, Data Model Transformation, Data Origin Authentication, Direct Authentication, Domain Inventory, Dual Protocols, Enterprise Inventory, Entity Abstraction, Intermediate Routing, Logic Centralization, Non-Agnostic Context, Process Abstraction, Process Centralization, Protocol Bridging, Reliable Messaging, Service Callback, Service Decomposition, Service Instance Routing, Service Layers, State Messaging, Utility Abstraction	*"Services are effective composition participants, regardless of the size and complexity of the composition."*

Design Principle	Related Design Patterns	Description
Discoverability	Canonical Expression, Capability Composition, Capability Recomposition, Metadata Centralization	*"Services are supplemented with communicative metadata by which they can be effectively discovered and interpreted."*
Loose Coupling	Asynchronous Queuing, Capability Composition, Capability Recomposition, Compatible Change, Compensating Service Transaction, Concurrent Contracts, Contract Centralization, Contract Denormalization, Data Format Transformation, Decoupled Contract, Dual Protocols, Entity Abstraction, Event-Driven Messaging, File Gateway, Intermediate Routing, Inventory Endpoint, Legacy Wrapper, Messaging Metadata, Multi-Channel Endpoint, Partial Validation, Policy Centralization, Process Abstraction, Proxy Capability, Schema Centralization, Service Agent, Service Callback, Service Decomposition, Service Facade, Service Instance Routing, Service Messaging, Service Perimeter Guard, Service Refactoring, Trusted Subsystem, UI Mediator, Utility Abstraction, Validation Abstraction	*"Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment."*
Reusability	Agnostic Capability, Agnostic Context, Agnostic Sub-Controller, Capability Composition, Capability Recomposition, Composition Autonomy, Concurrent Contracts, Cross-Domain Utility Layer, Data Model Transformation, Entity Abstraction, Intermediate Routing, Logic Centralization, Multi-Channel Endpoint, Rules Centralization, Service Agent, Service Layers, Utility Abstraction	*"Services contain and express agnostic logic and can be positioned as reusable enterprise resources."*
Statelessness	Asynchronous Queuing, Atomic Service Transaction, Capability Composition, Capability Recomposition, Messaging Metadata, Partial State Deferral, Process Centralization, Service Grid, Service Instance Routing, State Messaging, State Repository, Stateful Services	*"Services minimize resource consumption by deferring the management of state information when necessary."*

Design Principle	Related Design Patterns	Description
Standardized Service Contract	Agnostic Capability, Asynchronous Queuing, Canonical Expression, Canonical Protocol, Canonical Schema, Canonical Versioning, Capability Composition, Capability Recomposition, Compatible Change, Concurrent Contracts, Contract Centralization, Contract Denormalization, Data Format Transformation, Data Model Transformation, Decomposed Capability, Decoupled Contract, Distributed Capability, Domain Inventory, Dual Protocols, Enterprise Inventory, Event-Driven Messaging, Inventory Endpoint, Legacy Wrapper, Message Screening, Non-Agnostic Context, Partial Validation, Policy Centralization, Protocol Bridging, Schema Centralization, Service Callback, Service Facade, Service Messaging, Service Refactoring, State Messaging, Termination Notification, Validation Abstraction, Version Identification	*"Services within the same service inventory are in compliance with the same contract design standards."*

Additionally, SOA patterns are grouped into the categories from Table 4. Design patterns marked bold have been identified as interesting for INTER-IoT approach at the moment of writing this SotA.

Table 4: SOA pattern categories

Category	Design Patterns
Foundational Inventory Patterns	Canonical Protocol, Canonical Schema, **Domain Inventory**[1], **Enterprise Inventory**[2], Logic Centralization, Service Layers, Service Normalization
Logical Inventory Layer Patterns	Entity Abstraction, Process Abstraction, Utility Abstraction
Inventory Centralization Patterns	Policy Centralization, Process Centralization, Rules Centralization, Schema Centralization
Inventory Implementation Patterns	Augmented Protocols, Canonical Resources, Cross-Domain Utility Layer, Dual Protocols, **Inventory Endpoint**[3], Service Grid, State Repository, Stateful Services
Inventory Governance Patterns	Canonical Expression, Canonical Versioning, **Metadata Centralization**[4]
Foundational Service Patterns	Agnostic Capability, Agnostic Context, Functional Decomposition, Non-Agnostic Context, **Service Encapsulation**[5]

[1] http://soapatterns.org/design_patterns/domain_inventory
[2] http://soapatterns.org/design_patterns/enterprise_inventory
[3] http://soapatterns.org/design_patterns/inventory_endpoint
[4] http://soapatterns.org/design_patterns/metadata_centralization
[5] http://soapatterns.org/design_patterns/service_encapsulation

Category	Design Patterns
Service Implementation Patterns	Containerization, **Microservice Deployment**[6], Partial State Deferral, Partial Validation, Redundant Implementation, Service Data Replication, Service Facade, UI Mediator
Service Security Patterns	Exception Shielding, Message Screening, Service Perimeter Guard, Trusted Subsystem
Service Contract Design Patterns	Concurrent Contract, Contract Centralization, Contract Denormalization, Decoupled Contract, Validation Abstraction
Legacy Encapsulation Patterns	File Gateway, Legacy Wrapper, Multi-Channel Endpoint
Service Governance Patterns	Compatible Change, Decomposed Capability, Distributed Capability, Proxy Capability, Service Decomposition, Service Refactoring, Termination Notification, Version Identification
Capability Composition Patterns	**Capability Composition**[7], Capability Recomposition
Service Messaging Patterns	Asynchronous Queing, Event-Driven Messaging, Intermediate Routing, Messaging Metadata, Reliable Messaging, Service Agent, Service Callback, Service Instance Routing, **Service Messaging**[8], State Messaging
Service Interaction Security Patterns	Brokered Authentication, Data Confidentiality, Data Origin Authentication, Direct Authentication
Transformation Patterns	**Data Format Transformation**[9], **Data Model Transformation**[10], Protocol Bridging
REST-inspired Patterns	Entity Linking, Lightweight Endpoint, Reusable Contract

2.2.2.1 Micro-services

Micro-services are a modern interpretation of service-oriented architectures (SOA) used to build distributed software systems.

Micro-service is a conceptual service design in which services are decomposed into atomic, or at least smaller, functional components that can interact to compose the desired service. They are usually implemented as replicable instances, with little or no state, which make use of a REST API to present their functionalities to the other components.

Each micro-service instance runs as a separate process in a virtualized (or containerized) environment that allows them to be started easily and with a minimum amount of additional resources used, yet each service must be elastically scalable, resilient to failures, composable, and complete.

The challenge of creating a micro-service solution is thus to be able to decouple the functional components that make up a service or backend application, and to implement

[6] http://soapatterns.org/design_patterns/microservice_deployment
[7] http://soapatterns.org/design_patterns/capability_composition
[8] http://soapatterns.org/design_patterns/service_messaging
[9] http://soapatterns.org/design_patterns/data_format_transformation
[10]http://soapatterns.org/design_patterns/data_model_transformation

them as web services (ideally as a stateless services) using the most appropriate technology available.

Each of the underlying micro-services can be replicated in order to cope with an increase in the load for such particular service, by creating additional instances of said service. In turn, when the load of that sub-service decreases significantly, the number of instances can be reduced in order to avoid overprovision and minimizing operational costs.

Even though monolithic services can be also replicated, their lack of granularity might result in less efficient scalability, since all internal components are replicated in the same proportion, when perhaps only one of them is actually needed to be replicated.

2.2.2.2 Analysis of SOA Patterns

It can be noted that SOA patterns will be identified in INTER-IoT approach. It should be stressed that these patterns relate to: (i) solutions within INTER-IoT AS2AS and D2D layers, (ii) characteristics of services exposed by IoT platforms that want to join the ecosystem. We are mainly interested in the former, since the latter can be treated as good practices for designing system with SOA architecture.

The micro-services approach have the following advantages over monolithic designs and implementations:

- Fine grained scalability (better use of resources),
- Smaller development projects (easier to manage, and more productive),
- Easier DevOps (containerization and continuous delivery friendly),
- Technology flexibility (can implement each service independently),
- Higher mobility (can be easily moved to best suiting infrastructure),
- More resilient to failures (fine grained replaceable parts).

On the other hand, potential problems of a micro-service design for some particular cases are:

- Overall higher complexity,
- More complex backwards incompatible updates,
- Hard to define component boundaries (avoid nano-services antipattern),
- Problems with tightly coupled components,
- Expensive inter-process/machine communication (higher bandwidth usage),
- Additional communication latency,
- Can create communication barriers between teams.

We can try to counterbalance some of the mentioned problems by not separating logical components that require a fast and wide communication channel between them, and avoiding nano-services that increase the complexity of the system by separating the application in one a few data-wise loosely coupled parts.

2.2.3 Service Orchestration

Composition of services [10][11] encompasses all those processes that create added-value services, making different individual services work together in a sensible way and shows how the composition has to act in order to integrate distributed components.

In particular, in the context of Internet of Things, service composition can be understood as allowing for routes for the data to be treated before reaching an application or end-user, or as agents requiring information or processes to other such agents. Such compositions can help to provide more valuable information and actions than plain raw data, tailored to a particular receiver or purpose.

Two core approaches in service composition can be identified, providing different means of combining individual services, namely: orchestration paradigm (centralized service composition approach) and choreography paradigm (distributed coordination). The goal is to come up with the appropriate coordination paradigm in the IoT.

Orchestration[12] is the name of a design pattern authored by Thomas Erl, Brian Loesgen and published as part of the SOA Design Patterns catalogue as one of the Common Compound Design Patterns i.e. a coarse-grained pattern comprised of a set of finer-grained patterns.

Orchestration is a service composition strategy based on the use of a central module that controls all inputs and outputs of the atomic services (components) and perform the service composition logic. This component is called the orchestrator and needs to have control over all the services composing the business processes.

The service orchestration concept must do justice to the scalability issues typically found in IoT systems, so that the concept has to take decentralised autonomous configuration, organisation, management, and repairing capabilities into account. If these self-* properties are reflected, a deployment in real world large scale systems becomes possible.

Service orchestration model outlines the phases of modelling, resolution, binding, and execution of services.

2.2.3.1 Analysis of Orchestration Pattern

Service composition through orchestration is the main solution for service composition, if we focus on other relevant IoT platforms. Most platforms offer service orchestration, however centralized service orchestration has its limitations. Choreography is more appropriate as a means of coordination among different organizations, with the changing contexts of the augmented entities. Furthermore, it offers advantages in terms of scalability and resiliency. However, the augmented entities must have more computing capabilities, more memory, and more application logic implemented in order to execute more demanding business tasks.

Table 5: Selected IoT solutions service composition approach

IoT platform/architecture	Orchestration	Choreography	Comment
FIWARE	x		Orchestration service inside the PaaS generic enabler. This is based on OpenStack Heat project
SOFIA2	x		API orchestrator within the API Manager in 2.12
Sensinact	x		Composition and orchestration function for easing the development of custom business logic
UniversAAL	x		Service composition tool using orchestration (one using OWL-S and other based on JavaScript)

IoT platform/architecture	Orchestration	Choreography	Comment
oneM2M	x		The main open implementation of the OpenM2M standard, Eclipse OM2M, does not have any implementation for service orchestration
OpenIoT	x		Not define a component for service orchestration itself, relying in applications to develop their own service composition, if needed.
IoT-A		x	Define services, data model but interoperability is weakly introduced. Basic reference for IoT platforms, including AIOTI and IoT-EPI.

Service orchestration is significant from the point of view of INTER-IoT due to the fact that it can be applied in the context of AS2AS layer as a basic feature, similar to most of the IoT platforms analysed.

2.2.4 System test setup, Test applications and tooling

The test setup as described in this document should be present and checked for completeness. See Table 5 for the system setup.

2.3 Reactive Patterns

Systems that are responsive, resilient, elastic and message-driven are called Reactive Systems. Following the Reactive Manifesto[13] system design paradigm, several architecture elements that are commonly found in reactive systems were identified including: circuit breaker, various replication techniques, and flow control protocols. Reactive systems are distributed, what requires new and modified architectural patterns, that may be based on already existing code patterns and abstractions. Reactive patterns (defined in [14]) are language-agnostic and also independent of the abundant choice of reactive programming frameworks and libraries.

The paradigm of reactive systems is close to the architectural concepts behind IoT – message-based distributed, asynchronous and reliable. The architecture of reactive system is often based on micro-services which corresponding pattern is introduced in 2.2.7.

Reactive patterns are divided into six categories (Table 5).

Table 6: Reactive patterns summary

Pattern category	**Description**	**Patterns** (definitions from [14])
Fault Tolerance and Recovery	Address how to incorporate the possibility of failure into the design of the application.	• *Simple component*- A component should be only one thing, but do it in full. • *Circuit breaker* - Protect services by breaking the connection to their users during prolonged failure conditions. • *Let-it-crash* - Prefer full component restart to complex internal failure handling. • *Error kernel* - In a supervision hierarchy keep important application state or functionality near the root while delegating risky operations towards the leaves.
Replication	The problem of how to distribute the functionality of a component such that it can withstand hardware and infrastructure outages without loss of availability.	• *Active-Passive* - Keep multiple copies of the service running in different locations, but only accept modifications to the state in one location at any given time. • *Active-Active* - Keep multiple copies of a service running in different locations and perform all modifications at all of them. • *Multiple-Master* - Keep multiple copies of a service running in different locations, accept modifications everywhere and disseminate all modifications among them.

Pattern category	**Description**	**Patterns** (definitions from [14])
Message Flow	Communication that occurs between reactive components.	• *Request-Response* - Include a return address in the message in order to receive a response. • *Self-contained Message* - Each message shall contain all information needed to process a request as well as to understand its response. • *Ask* - Delegate the handling of a response to a dedicated ephemeral component. • *Forward Flow* - Let the information and the messages flow directly towards their destination where possible. • *Aggregator* - Create an ephemeral component if multiple service responses are needed in order to compute a service call's result. • *Saga* - Divide long-lived distributed transactions into quick local ones with compensating actions for recovery. • *Business Handshake* - Include identifying and/or sequencing information in the message and keep retrying until confirmation is received.
Flow Control	Address the issue of timeliness of the communication.	• *Pull* - Have the consumer ask the producer for batches of data. • *Managed Queue* - Manage an explicit input queue and react to its fill level. • *Throttling* - Throttle your own output rate according to contracts with other services. • *Drop* - Dropping requests is preferable to failing uncontrollably.
State Management and Persistence	How component's state can be managed.	• *Sharding* - Scale out the management of a large number of domain objects by grouping them into shards based on unique and stable object properties. • *Domain Object* - Separate the business domain logic from communication and state management. • *Event Sourcing* - Perform state changes only by applying events, make them durable by storing the events in a log. • *Event Stream* - Publish the events emitted by a component so that the rest of the system can derive knowledge from them.

Pattern category	Description	Patterns (definitions from [14])
Resource Management	Dealing with resources (file storage space, computation power, access to databases or web APIs, physical devices like printers or card readers, etc.) in reactive applications.	• *Resource Encapsulation* - A resource and its lifecycle is a responsibility that must be owned by one component. • *Resource Loan* - Give a client exclusive transient access to a scarce resource without transferring ownership. • *Resource Pool* - Hide an elastic pool of resources behind their owner. • *Complex Command* - Send compound instructions to the resource to avoid excessive network usage. • *Managed Blocking* - Blocking a resource requires consideration and ownership.

2.3.1 Analysis of Reactive Patterns

Reactive patterns are applicable to any distributed application, and therefore should be considered for INTER-IoT project. Messages, Message Flow and Flow Control are groups of patterns that are candidates to guide the development of system components, abstracting from the chosen technology. The advantage of reactive patterns is that they were described in a semi-formal template proposed in [14], containing four sections: short introduction with the problem setting (definition of task), guidelines of applying the pattern, the pattern revisited and applicability.

2.4 Agent Design Patterns

Agent design patterns were developed in the scope of multi-agent systems i.e. a system composed of multiple interacting intelligent, and autonomous agents. Agent patterns can also be used between non-agent systems as the agent metaphor could be considered as a design metaphor and not an implementation one. Thus, if we substitute the word “agent” with the word “component”, “system”, or “subsystem”, we can analyze the patterns as applicable to generic distributed systems.

Agent interaction design represents a very important stage during the design process of an agent-based distributed system as it influences the efficiency of the developed agent system. As it initially happened for the agents behaviour design, the use of patterns (see Table 7) to drive the agent interaction design is notably increased and a lot of contributions have been provided in literature [15][16][17][18][19][20]. On the other hand, several coordination models [21] have been introduced in literature to allow the agents’ interaction design according to specific interaction scenarios.

Table 7: Software agent patterns

Authors	Pattern	Description
Aridor and Lange	Meeting	Provides a way for two or more agents to initiate local interaction at a given host.
	Locker	Define a private storage space for data left by an agent before it is temporarily dispatched (send) to another destination.

Authors	Pattern	Description
	Messenger	Defines a surrogate agent to carry a remote message from one agent to another.
	Facilitator	Define an agent that provides services for naming and locating agents with specific capabilities.
	Organized Group	Compose agents into groups in which all members of a group travel together.
Kendal et al.	Conversation	Concerns with a sequence of messages between two agents, taking place over a period of time: agent messaging may occur within a context established by previous messages.
	Facilitator	Allows for interaction among agents which do not have to have direct knowledge of one another as it is based on a Mediator agent which provides a gateway or clearing house for agent collaboration.
	Agent Proxy	Enables agents to collaborate directly with one another through a proxy agent which provides distinct interfaces and allows agent to be engaged in multiple conversations.
	Protocol	Establishes conversation policies that explicitly characterize communication sequences.
	Emergent Society	Enables reactive agents to collaborate without known protocols as actions performed by agents can stimulate behaviour of neighbour agents.
Deugo et al.	Blackboard	Decouples interacting agents from each other as instead of communicating directly, agents interact through an intermediary which provides both time and location transparency to the interacting agents.
	Meeting	Allows for interaction among agents without the need for explicitly naming among them as they know a meeting point in which agent can coordinate themselves through a statically located agent.
	Market Maker	Allows for interaction among agents through a third party agent which takes an active role in the coordination process enforcing the house rules of agent interaction.
	Master/Slave	Allows for vertical coordination which is used to coordinate the activity of a delegating agent and two or more delegated agents in which delegated agents carry out a subtask for delegating agent.
	Negotiating Agents	Deals with the situation where the interacting agents appear as peers to each other, but need to align their actions for some reason.

There is by now a growing literature on the use of patterns to capture common design practices for agent systems [20][22][23]. In the following, some pattern-based agent design approaches, which also cover issues related to the design of interaction among agents, are summarized (see Table 7 for a brief description of each proposed patterns).

Aridor and Lange [15] describe a set of domain-independent patterns for the design of mobile agent systems. They classify mobile agent patterns into travelling, task, and interaction patterns and propose some patterns belonging to each the classes. Patterns in the travelling class specify features for agents that move between various environments, patterns of the task class specify how agents can perform tasks and patterns of the interaction class specify how agents can communicate and cooperate. In particular, with reference to the interaction patterns authors present the following ones: Meeting, Locker, Messenger, Facilitator, and Organized Group which concern with locating agents and facilitating their interactions.

Kendall et al. [17] capture common building blocks for the internal architecture of agents in patterns. Authors suggest a seven-layer architecture pattern for agents, and sets of patterns belonging to each of the layers. The presented seven layers are: mobility, translation, collaboration, actions, reasoning, beliefs and sensory but the exact number of layer may vary. Compared to the previously mentioned pattern classification scheme in the work by Aridor and Lange, the layered architecture has a similar logical grouping of patterns. The mobility layer together with the translation layer corresponds to the class of travelling, the collaboration layer corresponds to the class of interaction, and the actions layer corresponds to the class of task. In particular, with reference to the interaction patterns authors present the following ones: Conversation, Facilitator, Agent Proxy, Protocol and Emergent Society which concern how agents cooperate and work with other agents. The main difference between this and the previously mentioned approaches for mobile agents, is that this one aims to cover all main types of agent design patterns.

Deugo et al. [16] identify a set of patterns for agent coordination, which are, again, domain-independent. Authors classify agent patterns into architectural, communication, traveling, and coordination patterns. Moreover, they identify an initial set of global forces (Mobility and Communication, Standardization, Temporal and Spatial Coupling, Problem Partitioning and Failures) which are different types of criteria that engineers use to justify their designs and implementations. In particular, with reference to the coordination patterns authors present the following ones: Blackboard, Meeting, Market Maker, Master/Slave and Negotiating Agents which are well-documented solutions to recurrent problems related to the coordination among agents.

Kolp et al. [24] propose a catalogue of architectural styles and agent patterns for designing MAS architectures at a macro- and micro- level adopting concepts from organization theory and strategic alliances literature. Although interesting, these patterns define how goals assigned to actors participating in an organizational architecture will be fulfilled by agents without focus on coordination issues.

2.4.1 Analysis of Agent Design Patterns

On the basis of the introduced design patterns (see Table 7), we can state that they could be used to allow integration, interconnection, and interaction between agent-based and non-agent software components and systems.

In particular, the Facilitator and Agent Proxy patterns by Kendall [17] can be useful/effective to respectively support the design of a Gateway and a Proxy between two subsystems or layers of IoT systems (please refer also to INTER-LAYER).

More specifically:

- Facilitator: allows for interaction among system components that do not have to have direct knowledge of one another as the interaction is based on a Mediator component that provides a gateway or clearing-house for collaboration among systems.

- Proxy: enables systems to collaborate directly with one another through a proxy that provides distinct interfaces and allows such systems to be engaged in multiple conversations.

2.5 Ontology Patterns

Ontology design patterns are a reusable successful solutions to a recurrent (ontology) modeling problem. In [25] authors propose the use of semantic patterns to engineer ontologies while remaining independent from the underlying ontology language. The library of ontology patterns is still being developed, contrary to software engineering patterns where a set of coherent and consensual patterns is already available.

There are several initiatives that are working on and preparing catalogs of ontology design patterns. Naturally, their outcomes contain similarities, however, in the following sections they are presented separately.

2.5.1 ODP Wiki

OntologyDesignPatterns.org[26] is a Semantic Web portal dedicated to ontology design patterns (ODPs), and started under the NeOn Project. It is run by Association for Ontology Design & Patterns (ODPA). The Wiki contains list of patterns grouped into catalogs. Patterns can be submitted by users, and after review that change status from proposed to certified. Unfortunately, many of the pattern catalogs are at the moment empty. The following patterns descriptions come from the ODP Wiki website (note that some of them are empty at the moment of writing):

Table 8: ODP Wiki patterns summary

Catalog	Description	Subcatalogs
Content	For solving design problems for the domain classes and properties that populate an ontology; content-dependent.	
Correspondences	Include reengineering (provide designers with solutions to the problem of transforming a conceptual model) and alignment patterns (creating semantic associations between two existing ontologies).	• Reengineering • Alignments
Presentation	Usability and readability of ontologies from a user perspective.	• Naming • Annotation
Reasoning	Logical patterns oriented to obtain certain reasoning results.	
Lexico-syntactic	Linguistic structures or schemas that consist of certain types of words following a specific order, and that permit to generalize and extract some conclusions about the meaning they express.	
Structural	Logical patterns help to solve design problems where the primitives of the representation language do not directly support certain logical constructs; architectural patterns affect the overall shape of the ontology.	• Logical • Architectural

2.5.2 Ontology Design Patterns Public Catalog

This is a public catalog of ODPs focused on the biological knowledge domain[26].

The following patterns descriptions come from the website:

Table 9: Ontology Design Public Catalog patterns summary

Catalog	Description	Patterns
Extension	By-pass the limitations of OWL.	• Nary Datatype Relationship • Exception • Nary Relationship
Good practice	Obtain a more robust, cleaner and easier to maintain ontology.	• Entity Feature Value • Selector • Normalization • Upper Level Ontology • Closure • Entity Quality • Value Partition • Entity Property Quality • Defined Class Description
Domain modeling	Solutions for concrete modelling problems in biology.	• Out-of-scope

The group maintains a list of Semantic Web applications and demos for promoting the Semantic Web and for use by developers with the aim to provide hands-on support for developers of Semantic Web applications. The aim of the task force is to provide guidance for developers of Semantic Web applications. In particular, it focuses on the engineering of semantic web ontologies, through the publication of notes that document common and reusable ontology patterns, and general ontology engineering best practices. At the moment of writing, only three documents were published (in 2004): Representing Classes as Property Values on the Semantic Web, Representing specified Values in OWL “value partition” and “value sets”, Defining N-ary Relations on the Semantic Web: Use with Individuals.

2.5.3 Publications

In [27] authors focus on patterns in the field of Ontology Engineering and proposes a classification scheme for ontology patterns. The scheme divides ontology patterns into five levels:

- Application Patterns - Purpose, scope, usage and context of the implemented ontology, including interfaces and relations to other systems. No Application Patterns have so far been formalised for ontologies, as noted earlier, but there exist many models of ontology usage.
- Architecture Patterns - A description of how to combine or arrange implemented Design Patterns in order to fulfill the overall goal of the ontology.
- Design Patterns - A small collection of Semantic Patterns that together create a common and generic construct for ontology development.
- Semantic Patterns - Language independent description of a certain concept, relation or axiom. A meta-description of a Syntactic Pattern.
- Syntactic Patterns - Language specific ways to arrange representation symbols, to create a certain concept, relation or axiom.

There are some ontology pattern approaches present at the moment of writing, but these are mostly connected to the lower levels, such as Syntactic Patterns, Semantic Patterns, and Design Patterns.

In [28] author presents a framework for introducing design patterns that facilitate or improve the techniques used during ontology lifecycle. The proposed framework and the initial set of patterns are designed in order to function as a pipeline connecting domain modelling, user requirements, and ontology-driven tasks/queries to be executed. Conceptual Ontology Design Patterns (CODePs) have been introduced as a useful resource and design method for engineering ontology content over the Semantic Web. CODePs are distinguished from architectural, software engineering, and logic-oriented design patterns, and a template has been proposed to describe, visualize, and make operations over them.

2.5.4 Alignment Patterns

Based on ontology mismatches discussed in literature, patterns in correspondences can be identified and modeled. Patterns can be organized hierarchically according to their degree of generality. The deeper a pattern in the hierarchy, the more specialized for a specific problem. Specialization is reflected in the representation of the patterns using the alignment representation language. In the classification proposed in [29], authors define a hierarchy with top composed of the three basic patterns between classes, relations, and attributes. Then, each type of pattern is refined between equivalence, or subsumption patterns, and patterns specific for the entities types. These research has been started as PhD thesis but is declared to be continued through ontology community portal [30]

Pattern template selected by the authors follows [4] and [25] and includes information divided into two parts: classical elements (name, problem, solution, consequences) and grounding part (name of target language/system, applicability, purpose, example grounding, comment). Ontology alignment patterns are represented using Expressive Declarative Ontology Alignment Language (EDOAL) [31].

2.5.5 Analysis of Ontology Patterns

The ontology design patterns should be considered in this research because of the INTER-IoT requirement to provide interoperability on data and semantic. The approach evaluated in INTER-IoT is based on ontologies and semantic translation using ontologies alignments. Therefore, IoT platforms are required to have semantic model of exchanged data, and alignments should be constructed during the process of integration. These activities should be guided by a collection of design patterns for ontology and alignment modeling. Moreover, the interoperability on DS2DS layer assumes the existence of central ontology that is specific to each deployment. Design patterns can be considered as references in the description of the structure of this ontology.

Even though the classification of ontology design patterns is relatively extensive, this area of research is still under development. There are at least three independent catalogs with varying structure and pattern description template (ODP Wiki and ODP Public Catalog provide structures for pattern specification).

Alignment patterns are discussed in [29], where authors declare that the work will be extended through the ontology community portal [32]. Unfortunately, many patterns present in the hierarchy are not available through portal, and the one present are in proposed and not certified status.

2.6 IoT Patterns

Internet of Things (IoT) systems and applications present design problems in many areas and at many layers (device, network, middleware, application service, data and semantics). There are many diverse use cases, with different resource constraints, and with many different standards, products, and technologies available. How do we determine which are suitable, what the specific pros and cons are without a context from which to evaluate their use?

First, there are some principles that should be met while designing an IoT solution architecture [33]:

1. The ability to ingest massive amounts of events and data generated by IoT devices.
2. Secure, real-time, two-way communication, control channel that can be initiated by either party and that guarantees to ensure the delivery of commands.
3. Communication channels supporting loose-binding between message senders and receivers due to possible intermittent connectivity.
4. Devices that do not support the IP stack shall integrate using an IoT Gateway.
5. The ability to process data streams in real-time (the hot path) or post-facto (the cold path).
6. Devices should sent periodically a heartbeat or a keep-alive signal.
7. The ability to incrementally upgrade firmware.
8. Presence of the device registry to keep track of all deployed devices.

Of course, INTER-IoT fulfills all these requirements (which is described in deliverables of WP3 and WP4). Therefore, this pattern catalog is full of inspirations for INTER-IoT Layer Patterns. There are many patterns to consider, which can be presented as a few main pattern subcatalogs:

- **Design Patterns for Connected Things**[34]. "They represent the fundamental propositions of an Internet of Things that involves connecting things through networks and software…".
- **Design Patterns for Information Model**[34]. "They consist of lower layers of data models and representation, upon which higher level encapsulation and functions are built."
- **Design Patterns for Interaction**[34]. "They describe how different parts of a system interact and communicate with each other, including communication protocols."
- **Design Patterns for Application Programming**[34]. "They describe ways that software and interfaces are created, managed, deployed, and used in IoT applications…".
- **Design Patterns for IoT Infrastructure**[34]. "They describe how different network and device technology is used to solve problems with the physical infrastructure of IoT. How do low power devices connect to wireless sensor networks and ultimately connect to services and applications…".
- **Design Patterns for IoT Security**[34]. "They describe design patterns for IoT security problems".
- **Edge-based IoT Design Pattern**[35]. Low-level patterns for designing edge-based IoT systems.
- **Edge Provisioning Pattern**. Setting up the system and getting the devices connected are hard to simplify, devices may use different types of networks and various connectivity models, so Edge Provisioning Pattern is a design challenge for automated provisioning[36][30].

Many patterns, from catalogs described above, were useful in designing INTER-IoT mechanisms, but in the process of creating new INTER-IoT Layer Patterns, the most inspiring was "Design Patterns for Interaction" subcatalog, because new solutions, solved mainly communication issues. In [34] some examples were presented:

- *Request/Response*: probably the most commonly known communication pattern. It consists of a client, that requests a service from a server. This is the pattern that HTTP uses, and it's the basis for service- oriented architecture, web services, and Representational State Transfer.
- *REST*: Representational State Transfer, design pattern allowing for externalization of application state in reusable, shareable resources.
- *Asynchronous Events*: State updates propagate through the system as they occur.
- *Resource Binding*: Associating a resource with an action, bridges REST to Asynchronous Events.
- *Observer Pattern*: A binding of resource updates to a protocol action or handler.
- *Publish/Subscribe*: A communication pattern where a client registers interest in a topic by subscribing, updates to a topic are published to all subscribers (i.e. notification).
- *Broker*: A central service to connect publishers with subscribers.
- *Proxy*: A machine that provides an interface on behalf of another interface.
- *Protocol Bridge*: A bidirectional translator between two protocols.
- *Resource Discovery*: A process where resources are found by specifying attributes.
- *Resource Registration*: An endpoint informs a resource directory of its resources.
- *Sleeping/Non-reachable Endpoint*: An endpoint is not reachable and must participate in protocol by initiating all interactions with reachable or always-on endpoints."

Worth for attention is also section "Design Patterns for IoT Security", because security aspects are also considered in INTER-IoT. In [34] following examples were described:

- *Access control using data models*: semantic hyperlinks control access to resources based on the embedded metadata.
- *Social to physical graph relationship*: well defined concepts of ownership and access delegation between people, entities, and things.
- *PGP and asymmetric public-key cryptography on devices*: ways of creating SSL sessions and signing data between devices and applications.
- *DTLS over UDP*: security for resource constrained devices.
- *End-to-end encryption*: transmitting and storing encrypted data independent of channel encryption.
- *Device Management*: using device identity, registration, and secure key exchange."

2.6.1 Analysis of IoT Patterns

It can be observed that there is no one reference architecture solution for the IoT, but rather many approaches depending on use cases. The patterns identified in [34] and quoted in previous sections form a collection of concepts that are common to IoT solutions, and provide opportunities for standardization and commonality. Optimally, architecture should be reusable within a particular class of use cases. Therefore it makes sense to talk about IoT architecture as a set of Design Patterns, working together to achieve an end-to-end solution for some problem. Patterns introduced so far are not well-formalized using typical design patterns template [2][4]; although, they could be reused at some extent, they need to be first formalized to be really useful in a systematic way. There are indeed other patterns for IoT development recently proposed [35]: they are formalized using canonical design patterns template. However, it is worth noting that none of such proposals (high-level design patterns and low-level edge-based design patterns) are related to how to integrate already existing IoT systems/ application so granting interconnection/ interoperability, but rather they are focused on the development, deployment and execution of (new) IoT systems. Nevertheless, it is worth to notice, the "Design Patterns for Interaction" catalog can be inspiration for designing the patterns, related with INTER-IoT artifacts communication, and in general handling messages mechanisms. Moreover, “Design Patterns for IoT Security” is worth to consider in the designing security aspects in CROSS-layer and also in INTER-Health (where security and data confidentiality are particularly important).

2.7 Security Patterns

Design patterns can be also considered in the area of security where goals to be fulfilled are among others: confidentiality, integrity, and availability. The secure IoT deployments can use some of the existing security technologies already on the market, however, the IoT also introduces new challenges to security engineering that need to be addressed.

The following not IoT specific security patterns catalogs are available:

- *Core Security Patterns*[37] - guidance for Java applications, XML Web Services, Identity Management and Identity provisioning. Their aim is delivering end-to-end security of a J2EE based application architecture and representing how it is related in aspects of role and responsibilities in various components and logical tiers - such as Web Tier, Business Tier, Web Services Tier, and Identity Tier. A lot of core security patterns are strictly related to GoF and Core J2EE Patterns.
- *The Open Group Security Design Patterns*[38] - technical guides to security design patterns are produced, including a catalog of design patterns for IT system architects and designers to use in verifying the completeness of designs and in designing coherent extensions to existing IT systems.
- *Munawar Hafiz and colleagues with Ward Cunningham and Microsoft Patterns and Practices group catalog*[39] - 97 security patterns written by all security experts starting from the first work on security patterns in 1997 organized into hierarchical structure.

A survey on general security patterns was given in [40], in which authors classified the patterns from the software life cycle point of view and indicated future research direction. Patterns are described using template proposed in [41]. They are divided into requirements (analysis and model based patterns), design and implementation phase patterns. The paper contains valuable references to patterns proposed in other publications.

2.7.1 Analysis of Security Patterns

In [42][43][44][45] considerations and approaches for security in IoT are presented, however these are not in the form of design patterns. The challenges for IoT include: lack of mature IoT technologies and business patterns, limited guidelines on life cycle maintenance and device management, protection of edge devices, a lack of standards for authentication and authorization of IoT edge devices, lack of control and information asymmetry (complex dataflows management). Authors of [44] provide recommendations and guidelines for protection of IoT architecture, specifically network, application, device, physical and human layers.

Note that part of the aforementioned SotA analysis is dedicated to security in IoT platforms, and not security patterns in providing interoperability within IoT platforms. These two use cases should be separated since the goal of INTER-IoT is not the design and development of an IoT platform. From the INTER-IoT perspective, crucial aspects related to security are authentication and authorization, as well as secure communication.

Besides solutions described in this section, there are also “Design Patterns for Interaction”, described in section:*“IoT Patterns”*. All of this information was very useful in designing security patterns for CROSS-layer and INTER-Health.

2.8 Use case specific patterns

Besides software engineering design patterns, we try to identify typical use cases/solutions in the two INTER-IoT pilot application domains. It can be observed that particular sequences of user intentions and system responsibilities re-occur as solutions to common problems. After establishing common characteristics of the deployment environment it may be easier to match design patterns that should be applied to achieve interoperability. Use case patterns (rather analysis than design) can describe how the system level use case is mapped onto high level design patterns, e.g. gateways and web services.

Even though there are no well-established Port Logistics and (e/m)Health design patterns, some common concepts can be identified and some papers are available dealing with the functioning of the port environment and the design of healthcare information systems. In the following, we first briefly review available resources and then we provide some overall insights.

2.8.1 Port Logistics - Port Digital Transformation Reference Model

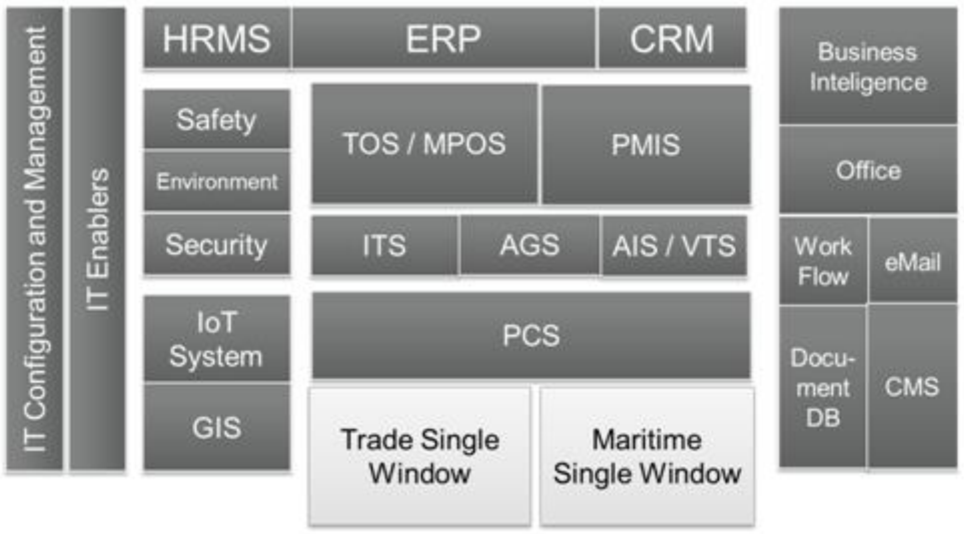


Figure 2: Port Digital Transformation Reference Model

A port is like a virtual enterprise where an assortment of specialized companies comes together to provide one face to the customer. In most cases the front-end company that provides services to the port user is not necessary the organization that provides all the underlying services. In fact, the port front-end reveals very little of the numerous processes, document exchanges and organizational arrangements that go into the delivery of the service.

An important number of actors take part in everyday port activities, serving the port traffic directly or indirectly, such as shipping lines, terminal operating companies, customs, port and maritime authorities, cross-border regulatory agencies and police, logistics service providers, freight forwarders, carriers, etc. All these independent public and private actors make up the “port community”, being considered each of them as a department of the same virtual company tied together by a common interest in maritime transportation.

The nature of relationships among the actors in the port community is the key element for the efficiency of the individual functions, port's logistics system and the import and export trades of a country.

In this sense, ICT tools, such as Single Windows or Port Community Systems, are a resource of vital importance for effective and efficient performance of port activities.

2.8.2 Port Logistics – Geo-fence

Geo-fencing (geofencing) is a feature in a software program that defines geographical boundaries. A geo-fence is a virtual perimeter for a real-world geographic area.

Programs that incorporate geo-fencing allow an administrator to set up triggers so when a device enters (or exits) the boundaries defined by the administrator, an event or action will be generated. So, for example, this activity could trigger an alert to the device's user, which could contain the location of the device, could be sent to a mobile telephone or an email account.

Many geo-fencing applications incorporate standard application as Google Earth, allowing administrators to define boundaries on top of a satellite view of a specific geographical area. Other applications define boundaries by longitude and latitude or through user-created and Web-based maps.

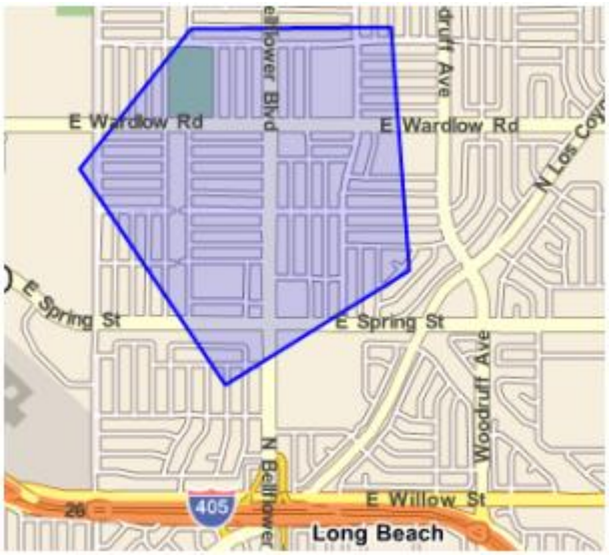


Figure 3: Geo-fencing zones

The technology has many practical uses:

- Fleet management.
- Human resource management.
- Compliance management.
- Security strategy model.

Technology:

- Global positioning system (GPS).
- Radio frequency identification (RFID).
- Bluetooth beacons.

2.8.3 Port Logistics – Automatic identification and data capture

Automatic Identification and Data Capture (AIDC) is a broad category of technologies used to collect information from an individual, object, image or sound without manual data entry.

AIDC is the process or means of obtaining external data, particularly through analysis of images, sounds or videos. To capture data, a transducer is employed which converts the actual image or a sound into a digital file. The file is then stored and at a later time it can be analyzed by a computer, or compared with other files in a database to verify identity or to provide authorization to enter a secured system. Capturing of data can be done in various ways; the best method depends on application. AIDC also refers to the methods of recognizing objects, getting information about them and entering that data or feeding it directly into computer systems without any human involvement.

AIDC systems are used to manage inventory, delivery, assets, security and documents. Sectors that use AIDC systems include distribution, manufacturing, transportation, medicine, government and retail, among many others.

Technologies typically considered as part of AIDC include:

- Bar codes.
- Radio Frequency Identification (RFID).
- Biometrics.
- Magnetic stripes.
- Optical Character Recognition (OCR).
- Smart cards.
- Voice recognition.

2.8.4 Port Logistics – Port Community System

A Port Community System (PCS) can be defined as a platform for information exchanges linked to a port, and therefore geographically restricted, which primarily seeks to serve the interests of the various companies and entities linked to port activities. A relatively wide variety of companies are involved, including terminal operators, transport operators (maritime/oceanic, road and rail), freight forwarders, customs, cross border regulatory agencies and port authorities (TrainForTrade 2009).

The main reason for creating port community systems is that port service users and customers need an increasing amount information everyday to innovate in their own processes. As a result, the transport sector must further its own metamorphosis and formalise innovation processes. Such innovations in the transport sector should not only contemplate the internal approach of each individual company, but look beyond this to see the companies and other entities related to transport as links in one single chain where the speed of the chain is determined by the slowest link. Therefore, all the parties involved in the transport chain must make a firm commitment to innovation and technological innovation processes to be prepared for the future.

Consequently, most companies are unable to tackle large innovations on their own effectively. They must be carried out under the umbrella of cooperation by creating alliances with companies and government bodies that are capable of handling such challenges. These alliances or communities take two forms:

- Chain Alliances: where a series of entities that are not themselves competitors join to cooperate in certain services to undertake innovation processes. One example is a door-to-door service with certain special features; in these cases, an alliance would be possible between an ocean carrier, their agents in the ports of origin and destination respectively, an international freight forwarder and their representative and a rail operator at origin and another at destination.

- Node Alliances: where companies and entities that work around the same node can become allies. Two examples are a Port Community or a Logistics Platform in a given area. One of the characteristics of this type of alliance is that they include companies that are competitors, but which have joined forces to obtain a common goal. They are known as "competitors".

The *European Port Community Systems Association (EPCSA)* define the PCS concept as a neutral and open electronic platform enabling intelligent and secure exchange of information between public and private stakeholders in order to improve the competitive position of the sea and air port's communities. Moreover, this entity adds that these systems optimise, manage and automate port and logistics efficient processes through a single submission of data and connecting transport and logistics chain.

Essentially, PCSs respond to the need to focus on maximizing physical infrastructure and managing the efficiency of the port operation as a whole. The system exists in an environment in where an important number of stakeholders plays different role in the transport chain.

In summary, PCS could be described as a "one-stop-shop" where all electronic shipment transactions can be performed and the many actors in the cargo network are easily accessible connecting both private and public stakeholders in a single communication channel. Therefore, the PCS is ideally placed for becoming a backbone component of the Single Windows environment (EPCSA 2011).

2.8.5 Port Logistics - The Single Window (SW) Concept

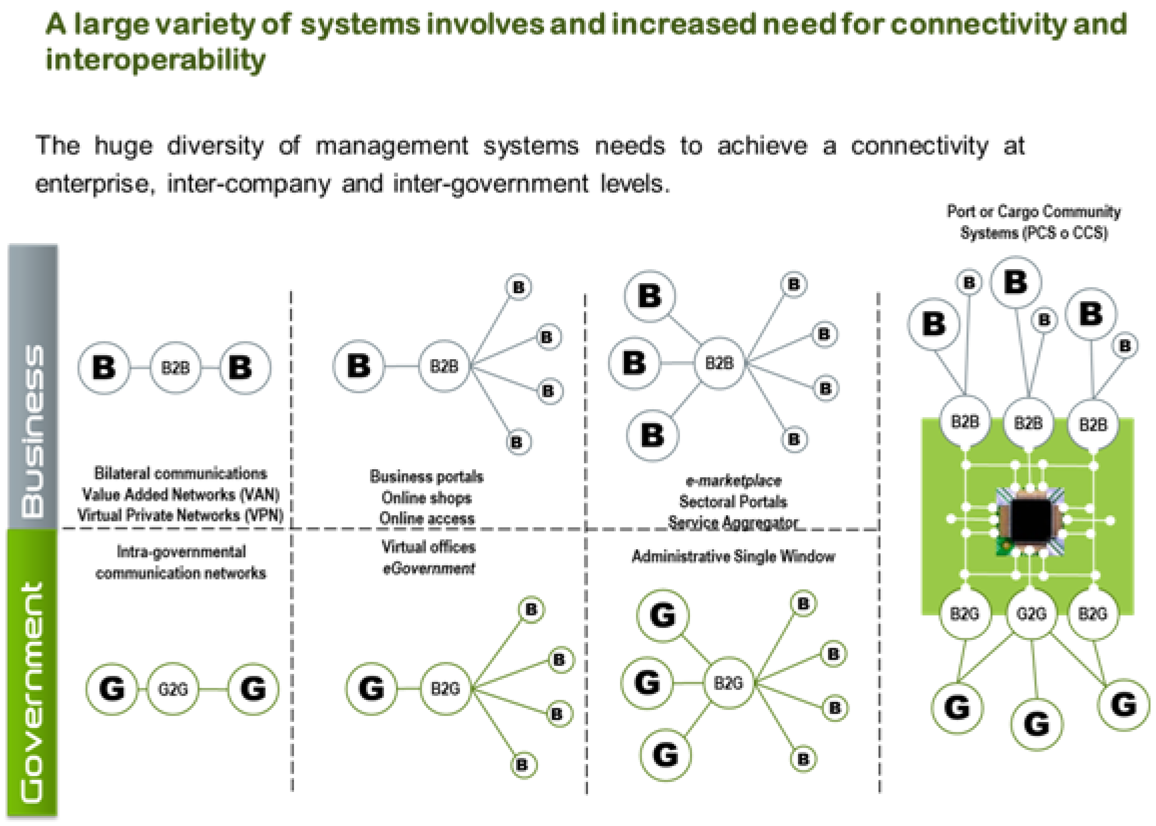
Single Window is a widely used term in the area of international trade and transport, and a big effort has been made to define and describe this term and the associated concepts. The idea of a Single Window challenges the conventional models of regulatory control of the movement of goods and means of transport.

Single Window is not an IT system but a philosophy of governance in which traditional structures of government are transformed into new arrangements that best serve the needs of citizens and businesses. Under this approach, citizens and businesses would receive government services through a single interface to government (WCO 2011).

The Single Window concept examines regulatory controls through the eyes of the port user and views all interactions between transport, trade and regulatory agencies without regard for the internal divisions within government institutions. This approach clearly brings out all the procedural redundancies, duplication in the filing of information and the wastefulness involved in the overall effort in fulfilling cross-border regulation. From this analytical approach arise a set of solutions that greatly simplify government-trade interface by reorienting procedures and reorganizing regulatory data requirements.

As concept's summary, the United Nations (UN/ECE 2005) in the Recommendation 33 describes it as "a system that allows traders to lodge information with a single body to fulfil all import- or export- related regulatory requirements". This is the most widely acknowledged definition.

This institution (UN) points that a Single Windows environment provides one entrance, either physical or electronic, for the submission and handling of all data and documents related to the release and clearance of an international transaction. This entrance is managed by one agency, which informs the appropriate agencies, and/or directs combined controls. For this reason, trade actors are strongly in favour of Single Window approaches because it creates the visions of a dramatically simplified interface to Cross-border Regulatory Agencies (CBRA), and in recent times, the value of SW, as a trade facilitation tool, has been increased enormously.



A large variety of systems involves and increased need for connectivity and interoperability

The huge diversity of management systems needs to achieve a connectivity at enterprise, inter-company and inter-government levels.

Port or Cargo Community Systems (PCS o CCS)
Business
Government
B
B2B
B2G
G2G
G
Bilateral communications
Value Added Networks (VAN)
Virtual Private Networks (VPN)
Business portals
Online shops
Online access
e-marketplace
Sectoral Portals
Service Aggregator
Intra-governmental communication networks
Virtual offices
eGovernment
Administrative Single Window

Figure 4: Port Logistics - The Single Window (SW) Concept

2.8.6 (e/m)Health - Health Care Information Systems [46]

Health Care Information Systems (HCIS) raise some specific problems related to the information requirements of the domain:

- textual data, instead of numerical data - healthcare data is mostly textual data, gathering descriptive information, by contrast with business data, dominated by numbers;
- the need for the integration of text, through a common vocabulary;
- the importance of the historical data.

In HCIS, the information is more complex than business-oriented information systems, the transactions are fairly unique, and they contain rather textual data than numbers.

The greatest problem with textual data is the integration of terms and concepts. Usually, healthcare information systems have to deal with data from different sources - hospitals, outpatient clinics, doctor's offices, emergency rooms, and so forth. In addition, the doctors that feed the database represent different disciplines – pediatrics, cardiology, epidemiology, orthopedics, gynecology, and so forth. Each of these disciplines has their own terminology. Furthermore, the information going into the database is written by different levels of people – physicians, nurses, technicians, accountants and so forth.

By ingesting all this diverse textual information, it is discovered that what is happening is a recreation of the Tower of Babel. Everyone is speaking a different language and no one really understands what anyone else is saying. That is why it is needed is a common vocabulary.

Another important aspect is the "the time value of information": in the healthcare environment data has a long life. For instance, for the purpose of studying a disease, medical records that are 50 years old may be extremely valuable, even where many of the people whose records are represented are departed.

The HCIS should have two properties:

- **Scalability by Design**. The capability of an information system to extent its designed structure and/or functionality, without changes in the database structure or in the system's application modules. This property of the information system is generally very important, because of the requirement changes in time, determined by the natural changes of the 'business' modeled by the system. The benefits of the systems with this property consist in significant maintenance costs savings.
- **Semantic Consistency of the Historical Data**. The most challenging problem related to database consistency preservation is the semantic consistency of the data. When data were imported from legacy systems, which is often the case, or they were added in the system by different people with different specialties or approaches, there is a great chance for the same concept to bear different names in the system.

Similar with any other specialized information systems, healthcare information systems should have a data-driven architecture. The only 'business' requirements that absolutely, unconditionally must be satisfied by the information system are data requirements. However, data constraints specific for healthcare environment require special treatment in healthcare applications. The continuous evolution of the medical science determines continuous changes of the information needs, which require design flexibility for the system's data model.

While business data occurs in repetitive numerical transactions with a simple pattern, healthcare data means textual transactions, with lower frequency, but more complicated patterns. Historical value of data is also greater in health care than in business.

In order to integrate healthcare data from different institutions, different medical disciplines, or subsequent generations of information systems, is needed an integration solution, i.e. common vocabulary or semantic translator. Moreover, it is of great importance that this common vocabulary – which is in fact a dictionary with standard medical terms – is kept up to date by a medical team, so that all the data inputs would find their right match in the system.

2.8.7 (e/m)Health - Database Design Pattern for Healthcare Information Systems [47]

Healthcare Information Systems are in most cases complex systems that store and manage large amounts of medical data. When such systems are intended to be used in medical research, it presents the system designer with a double challenge: the need for complexity and flexibility at the same time. In this paper authors present a database design pattern along with a Graphical User Interface (GUI) design that empowers the researcher to build metadata structures, which are supported by relational data structures on a database server.

As most of the data gathered during a medical research is viewed as having a hierarchical structure, the presented approach is based on modeling trees and multi-trees. The developed system is based on database design patterns and GUI prototypes. Issues regarding data structuring, data entry and data retrieval are addressed.

The structure and functionality of the proposed system are presented, with emphasis on three major functions: data structuring, data entry and data retrieval. Some considerations regarding the implementation of the system are also provided.

By using the presented approach, medical researchers can quickly and efficiently configure a customizable software system for recording their data in more complex structures than tables or spreadsheets, while benefiting of the consistency of a relational database.

A database design pattern that models the medical data as a collection of trees, grouped in what it is called “structure types”. In this case each type of observation chart used in the study is represented by a “structure type”. The advantage of this approach is that each piece of information can be accessed via its parent node but also via the structure type that it belongs to. So, for instance, we can have a report with all the measurements of a variable stored within a specific observation protocol, or observation sheet. Furthermore, the same node can be included in one or more structure types, allowing the navigation from one tree to another (multi trees).

Other approaches, like OpenEHR or EHR4CR, are focused on building a semantic electronic health record technology that is primarily designed to improve the efficiency of conducting clinical trials.

2.8.8 (e/m)Health - Quality Management in Healthcare Information Systems [48]

This paper illustrates how the implementation of total data quality management (TDQM) methodologies could help to solve problems currently evident in HCIS. Information management strategies employed by healthcare systems in Europe are discussed and examined. It is our hypothesis that data quality-based management methodologies provide an efficient vehicle for reform in HCIS.

Quality can be measured through the following measurements:

- Failure to Understand Consumers’ Needs: Effective support for patients, doctors, nurses, etc.
- Poorly Defined Information Production Process: Use of interoperable EHR.
- No Product Life Cycle: lifecycle management of data and resources.
- Lack of an Information Product Manager (IPM): management of HCIS.

2.8.9 (e/m)Health - Standardized Device Services – A Design Pattern for Service Oriented Integration of Medical Devices [49]

Service oriented device architecture (SODA) is a promising approach for enabling a continuous IT support of medical processes in hospitals. However, there is a lack of specific design patterns for realizing the concept in an effective and efficient way. This paper addresses this research gap by introducing the Standardized Device Service design pattern, as a first fundamental pattern for encapsulating devices as services. The pattern is based on both established Service Oriented Architecture (SOA) best practices as well as latest research in the field of SODA. This paper contributes to a) the extension of the IT support of medical processes by devices, b) the general concept of SODA by addressing the lack of generalized design concepts, and c) the existing catalog of SOA design patterns by introducing a first pattern for device integration.

The specific elicited requirements for device services are:

- Mobility,
- Locality,
- (Human) Manual influences,
- Replacement,
- Device as resource,
- Hardware interfaces,
- Software changeability.

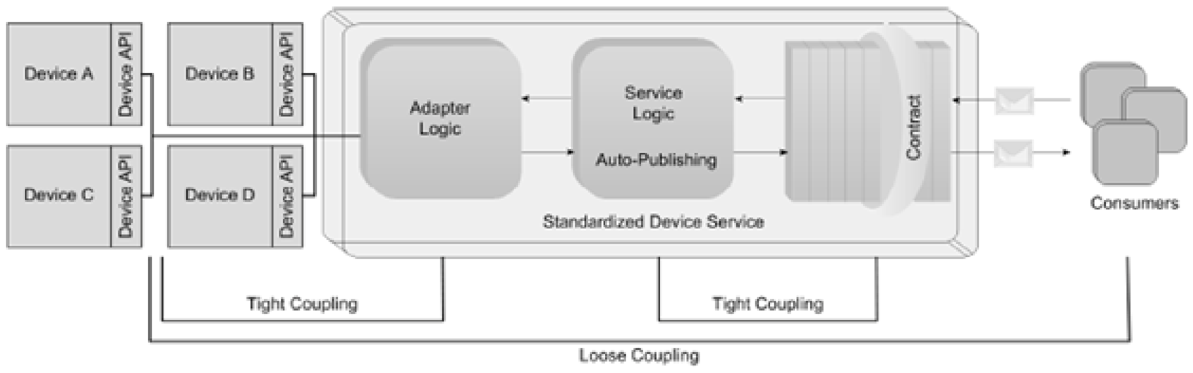


Figure 5: Standardized Device Service pattern

According to such requirements, device services must be able to:

- Dynamically handle different kinds of device interfaces, which usually cannot be influenced.
- Manage the fact that devices can suddenly be not accessible at any time.
- Provide functionality for handling devices as physical resource (e.g., exclusive access and locality) if required.
- When developing design patterns for device services, these requirements have to be taken into consideration.

This paper proposes the new Standardized Device Service pattern (see Figure 5). It is a compound pattern, i.e., it is comprised of combinations of design patterns. The name of the pattern is due to the fact that realizes device services with standardized service contracts. The following patterns are included:

- Service Encapsulation,
- Legacy Wrapper,
- Dynamical Adapter,
- Auto-Publishing.

2.9 Pattern Forms

Different approaches to design patterns specifications have been proposed that usually include narrative text with a predefined structure (usually specific to each catalog). Patterns and pattern languages have their roots in urban design and building in the work of Christopher Alexander – the inspiration for the software patterns. Nowadays, pattern writers tend to adjust the description form used to their needs and likes. Here, we analyze what forms of patterns formalization have been used in the literature, in order to select one to be used as pattern formalization technique in the INTER-IoT project.

First, lets present how design pattern can be defined which leads to the form of the pattern description that are presented next.

According to Alexander a design pattern can be described as: *Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution*.

Further explanation:

"As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.

As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant."

The goal of the form is to introduce the reader to a problem, to describe the context in which the problem might arise, to analyze the problem, and to present and a solution.

In [50] the author notices that even though different forms of pattern descriptions are used some part of it is common. Specifically:

- Name: short and evocative.
- Indication that patterns are solutions to problems: problem needs to be described, but most importantly recurring and useful solution needs to be presented.
- In what cases the pattern should be applied and how.
- Code examples as sample interpretation of the pattern.
- The pattern form is usually modified to best suit the needs of the domain being abstracted, however, some of the forms have gained greater popularity and serve as inspiration for new pattern writers.
- Following are the best known pattern forms that can be used or provide a basis for developing a new form.
- Alexandrian (A Pattern Language; APL)[51]: very narrative original pattern form without many headings. The major syntactic construct is Therefore preceding the solution. Other elements are: a clear statement of the problem, a discussion of forces, a solution and a rationale.

The following form was used in classic GoF book that introduced software patterns; it is tuned for object-oriented software design, structured with many headings[52]:

- Name and Classification: The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.
- Intent: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?
- Also Known As (optional): Other well-known names for the pattern.
- Motivation: A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.
- Applicability: What are the situations in which the design pattern can be applied?
- Structure
- Participants: The classes and/or objects participating in the design pattern and their responsibilities.
- Collaborations: How the participants collaborate to carry out their responsibilities.

- Consequence: How does the pattern support its objectives? What are the trade-offs and results of using the pattern?
- Implementation: What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?
- Sample Code: Code fragments that illustrate how you might implement the pattern.
- Known Uses: Examples of the pattern found in real systems.
- Related Patterns: What design patterns are closely related to this one?

"Portland" [27], a short textual form identified at the pattern PLoP conference ideal for summaries of large pattern languages; a one sentence summary statement that describes the problem and another that describes the solution, with A See also frequently added at the end. Each pattern in the Portland form makes a statement that goes something like: *such forces create this or that problem, therefore, build a thing to deal with them*. It is used in online Portland Pattern Repository maintained by Ward Cunningham.

"POSA", form from [4] that is structured and quite long; patterns are preceded by a narrative chapter that summarizes the pattern, then follows the headings:

- Name: meaningful and short summary.
- Example: demonstration of existence of the problem and justification for the pattern.
- Context: situation in which the pattern can be applied.
- Problem: related problem and associated forces.
- Solution: solution principles.
- Structure: structural aspects.
- Dynamics: run time behavior.
- Implementation: guidelines for implementation.
- Variations: possible variations of the pattern.
- Known Uses: example of uses of the pattern.
- Consequences: benefits and potential liabilities.
- See Also: reference to patterns that address similar problem.

"Coplien (Canonical)", the form that explicitly spells out the most basic elements (maps roughly to GoF) for describing a pattern i.e. sections with the following headings:

- Name: nouns or short verb phrase as in Alexandrian form.
- Alias (optional).
- Problem: the problem is often stated as a question or design challenge.
- Context: a history of patterns that have been applied before the current pattern; all factors that, if changed, would invalidate the pattern.
- Forces: focus of a pattern; help the designer understand how to apply a pattern effectively.
- Solution: how to solve a problem, may contain a sketch.
- Example (optional).
- Resulting Context: which forces does the pattern resolve and which remain unresolved.
- Rationale (optional): why does this pattern work?
- Known Uses.
- Related Patterns.

"Enterprise Application Architecture (EAA)" [53], narrative form from [2] including sections: how it works, when to use it, and one or more examples.

"Fowler" form includes:

- Title.
- Summary of the pattern.
- Discussion of the problem that may be addressed by implementing the pattern and potential liabilities.

"Beck" form template includes:

- Title.
- Context.
- Problem: always phrased in the form of a question the reader might have to ask themselves.
- Forces.
- Solution: the solution should include the name of the pattern in some form.
- Resulting Context.

Interesting fact is, the authors of "Enterprise Integration Patterns" use modified Beck form.

3 INTER-IoT patterns catalog

Figure *1* (Section:*"INTER-IoT Design Patterns defining process – methodology"*) shows all phases of designing the INTER-IoT Design Patterns. This gives more details and presents final solutions, i.e. INTER-IoT patterns catalog. The aim of this section is to present three aspects. First (Section:*"Designing process"*), is describing the process of defining new INTER-IoT patterns catalog. Second one (Section:*"INTER-IoT Layer Patterns template"*), presents the form of INTER-IoT design patterns. Thirdly, section: *"INTER-IoT issues and solutions (final patterns catalog)"* presents all the INTER-IoT Layer Patterns, including the references to SotA, description the solving problem and example uses in the INTER-IoT pilot implementation. Finally, section *"Analysis of INTER-IoT Design Patterns"* contains short summary of achieved solutions.

3.1 Designing process

Section: *"State of the art - research and analysis"* presents the result of first phase *"SotA analysis"*. The next step was on the creation of initial design patterns template, i.e. the patterns useful in the INTER-IoT architecture development. Every involved partner proposed patterns related to the specific INTER-LAYER (D2D, N2N, MW2MW, AS2AS, DS2DS, CROSS), INTER-FW, INTER-Health and INTER-LogP. Third phase, assumed an analysis of achieved results. It was easy to notice, that many defined patterns described very general problems, not strictly concerning to the interoperability aspects of INTER-IoT. The guiding rule for selecting the final set of patterns was the assumption that, a design patterns should reflect the solutions provided in WP3. The preliminary set of patterns did not fulfil this requirement, because they should present the solutions that support directly the integration process. So the decision was made to define a set of INTER-IoT Design Patterns, corresponding to the integration solutions, already achieved in the project. It was the main selection criterion in the process of picking the final set of patterns and refinement them or create a brand new definitions.

Table 10: Initial Design Patterns template - Analysis & Refinement (number of patterns in every step)

INTER-IoT layers	Initial Design Patterns template	INTER-IoT Layer Patterns
D2D	6	2
N2N	8	1
MW2MW	10	4
AS2AS	7	3
DS2DS	2	2
CROSS	3	1
INTER-FW	0	2
INTER-Health	0	2
INTER-LogP	2	1

Table 10 shows the comparison of preliminary and final sets of design patterns in terms of number of proposed patterns. It is easy to notice, that in the end, the final solution consists of less number of entities. In initial state was 38 patterns and in the final stage was 18 patterns. Moreover, it is worth to notice, that initial process was interrupted, because it was decided, this process generates too many and too general solutions. It is the reason why INTER-FW and INTER-Health does not contain any pattern, i.e. patterns was not yet added to the initial catalog. If the process of generating the patterns had ended, the initial catalog would have contained much more patterns.

Defining final catalog of INTER-IoT Later Patterns was complex process, that took long time. It consisted of five subtasks, described in the following paragraphs.

Step 1. Preparing the specific template, in order to define INTER-IoT patterns in a clear way. This step is described in details in section: *"INTER-IoT Layer Patterns template"*.

Step 2. Analysing project solutions. The new idea of preparing design patterns was to describe solutions defined in WP3. Because of that, it has been applied reverse engineering, i.e. patterns were extracted, using the WP3 solutions. To avoid the issue of initial catalog (generating many, general patterns), the aim of analysis was to extract the problems specific for INTER-IoT, that could not be fully solved by any existing solution.

Step 3. Analysing the initial catalog and extract patterns strictly related with the WP3. Not all the patterns from initial catalogs were declined. Some of them, solved strictly INTER-IoT issues, so they were moved to new, final catalog.

Step 4. Defining missing patterns. In this step were defined new patterns, taking into account step (2) results, i.e. issues specific for INTER-IoT.

Step 5. Analysing and refinement the content of final catalog.

3.2 INTER-IoT Layer Patterns template

In section: "Pattern Forms", was presented few, common approaches of describing the design patterns. Taking into account presented formats, it can be noticed, that structures are very similar and have the same parameters (fields) or with the same meaning, e.g. *"Title"* =

"Name", *"Applicability"* = *"Context"*, *"The body"* = *"Motivation"*, etc. For INTER-IoT was created a new template, based on the known solutions. Moreover, very helpful in process of creating format (and in general, the whole pattern) was [49][50], which give a step by step guidelines with examples on how to write a design pattern. Using all the information, extracted from the analysis, it was selected set of fields that can describe the patterns in a clear way. It consists of twelve properties, describing pattern. The names of those fields with explanation are presented below.

- **Pattern name.** The formal, unique name of the pattern. The name is very important part of pattern description. The author should try to put it in short sentence, describing the patterns action.
- **Identifier.** An individual ID within the INTER-IoT project.
- **Inspired by.** The name of the pattern(s) on which it is modeled. Most patterns are not a completely new solution, but based on the existing one. Some common patterns do not fully solve the problem, because of the specific of the INTER-IoT issue. In that case, a new pattern was created, extending the already existing one.
- **Related patterns.** Other patterns, related with the describing pattern. This field takes into account both INTER-IoT Layer Patterns and common solutions.
- **Intent** (summary). A short description of the goal behind the pattern and the reason for using it. It is an extension of the "Pattern name", explaining its action/purpose.
- **Problem & Solution.** A scenario that illustrates a problem (in the platform integration process) and how the pattern solves this problem (its setting). This is important to understand the nature of the pattern (why it was used in the INTER-IoT process). This should be taken by WP3 specific layer.
- **Applicability.** Situations in which this pattern is usable; the context for the pattern.
- **UML representation.** Structure of the pattern modeled in UML diagram (also deployment and component diagram if needed).
- **Implementation.** An extension of the "UML representation" property, i.e. the textual description of realization and architecture (not a source code, like in GoF). The aim of this property is to clarify the diagram.
- **Known uses** (within the INTER-IoT). Usages of the pattern in the platform integration process. It should described the module(s) (and scenario related with it) running within the INTER-IoT .
- **Identified by.** INTER-IoT partner who defined and submitted the pattern.
- **Registration date.** Date of contribution (required date format: dd-MM-yyyy).

3.3 INTER-IoT issues and solutions (final patterns catalog)

In section: *"INTER-IoT Design Patterns defining process – methodology"*and *"INTER-IoT patterns catalog"* was mentioned that INTER-IoT Layer Patterns defines solutions specific for INTER-IoT issues, i.e. related with integration process (described in WP3). This section presents all the created patterns catalogs, containing solutions for INTER-LAYER, INTER-FW,INTER-Health and INTER-LogP. Every pattern's template describes not only the solution but also the problem, i.e. a reason of creation pattern (property "Problem & Solution") and example of usage the pattern in the integration process (property "Known uses"). Moreover, in the property "Inspired by" is the source of inspiration for the pattern (most of the patterns extends some existing solution(s)).

3.3.1 D2D Layer

Design Pattern

Pattern name: INTER-IoT Gateway Event Subscription (Publish/Subscribe)	**Identifier:** 03
Inspired by: • *"Publish/Subscribe"* IoT Patterns: Design Patterns for Interaction (Section: *"IoT Patterns"*) • *"Publish-Subscribe Channel"* Enterprise Integration Patterns: Messaging Channels (Section: *"Enterprise Integration Patterns"*) • *"Facilitator"* Agent Design Patterns: by Kendall (Section:*"Agent Design Patterns"*) • *"Proxy"* Agent Design Patterns: by Kendall (Section:*"Agent Design Patterns"*)	
Related patterns: • D2D REST Request/Response	
Intent: INTER-IoT proposes as a potential solution, a D2D gateway that allows any type of data forwarding. It will make the device layer flexible by decoupling the gateway on two independent parts: a physical part that only handles network access and communication protocols, and a virtual part that handles all other gateway operations and services.	
Problem & Solution: In order to provide interoperability between two heterogeneous IoT devices, the solution should be applied that establishes bidirectional, asynchronous communication with the ability to publish, filter and consume the necessary data. This pattern describes the approach that uses the INTER-IoT gateway as a subscription mechanism. The INTER-IoT gateway is an intermediary element between the IoT artifacts and the IoT platforms, or, in the case of D2D communication, between two devices. It allows the transmission of data generated by sensors to its destination. This pattern allows asynchronous messaging among the gateway and other elements of the IoT system that interact with it (i.e. smart objects connected to the gateway and/or INTER-IoT middleware). If required, the gateway should perform operations of protocol conversion to enable the communication. Senders of messages (publishers) do not program the messages that are sent directly to specific receivers (subscribers). Instead, they publish messages, using defined classes, without knowledge of subscribers. Similarly, subscribers express interest in one or more classes and only receive messages that are of their interest, without necessarily knowing the publishers. Significant element of this solution is the structure of the message which contains all the necessary information about subscription (e.g. message endpoint, See design pattern 2: "D2D REST Request/ Response").	
Applicability: This pattern is used when an event-based communication model is needed, when asynchronous data needs to be pushed from/pulled to the INTER-IoT Gateway.	
UML representation:	

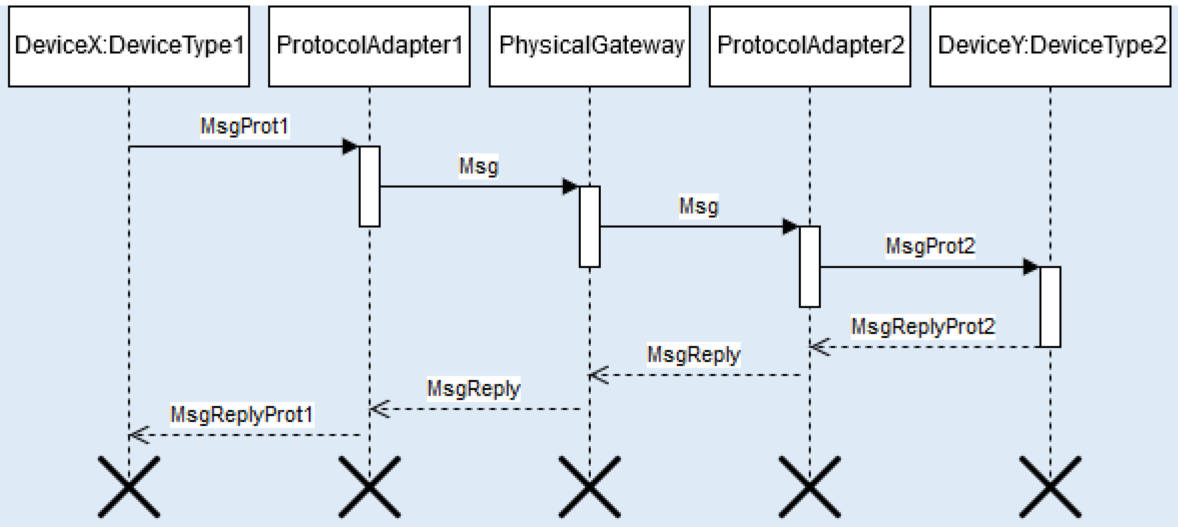


Figure 6: INTER-IoT GW Event Subscription (Publish/Subscribe) - D2D Communication

Implementation:

In this case, the Gateway is the subscriber of a publisher (Device X) and in the subscription request message an endpoint is defined (Device Y). Also the INTER-IoT gateway is subscribed to publications of Device X, and in the subscription request message the other endpoint is defined (Device X). Protocol conversion is performed if required to establish this communication. When the publisher has new events (asynchronous) it will push the event data (publication) to the defined endpoint. This example illustrates a D2D communication through the INTER-IoT gateway, that acts as an intermediary element enabling this interaction.

Known uses (within the INTER-IoT):

Identified by: UPV	**Registration Date:** 05-05-2017

Design Pattern

Pattern name: D2D REST Request/Response	**Identifier:** 02

Inspired by:

- *"Request-Response"* Reactive Patterns: Message Flow (Section: *"Reactive Patterns"*)
- *"Request-Reply"* Enterprise Integration Patterns: Messaging Patterns (Section: *"Enterprise Integration Patterns"*)
- *"Request/Response"* IoT Patterns: Design Patterns for Interaction (Section: *"IoT Patterns"*)

Related patterns:

- INTER-IoT Gateway Event Subscription (Publish/Subscribe)

Intent:

A request/response solution for gateway communication within the D2D layer.

Problem & Solution:

In an IoT scenario, the INTER-IoT Gateway needs to communicate with IoT smart objects/artifacts. INTER-IoT gateway should be accessible to authorised external elements to enable the reception of information collected by smart objects and the execution of control and configuration orders.

For example, a main INTER-IoT goal is to allow heterogeneous IoT platforms to retrieve information from the smart objects through INTER-IoT layers and framework. For this aim, the platform's MW should be able to communicate with the INTER-IoT GW to enable these information flows. Thus, it is desirable to connect IoT artifacts (if possible) through a HTTP/REST API using the Request/Response pattern. This communication pattern allows a message exchange in which a requestor (e.g. MW or GW) sends a request message to a replier system which receives and processes the request (e.g. MW or GW), ultimately returning a message in response.

Applicability:

This pattern is used when the communication between the middleware of an IoT platform and the INTER-IoT gateway (both directions, although MW->GW will be typically publish/subscribe) is performed through a REST API. Also, for management purposes the gateway will expose a REST endpoint where configuration and management actions can be performed using the Request/Response patterns.

UML representation:

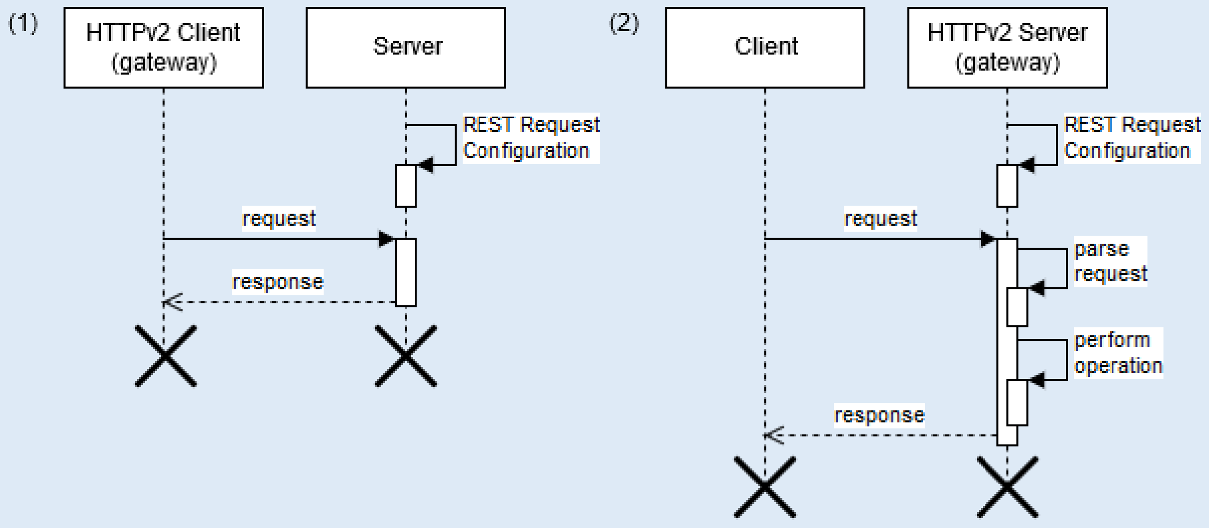


Figure 7: D2D REST Request/Response

Implementation:

There are two possible implementations of this pattern: (1) when the gateway is the requester (client) or (2) the one who receives requests (server). In the first case, the gateway is the requester and through an HTTPv2 client performs the request and gets the response. In the second case, the gateway deploys an HTTPv2 server that exposes REST endpoints and after performing the operations will create a response with the outcome.

Known uses (within the INTER-IoT):

- Register/Unregister device to a platform MW.
- Sensor data update to a platform MW.
- Configuration actions in GW.

Identified by:	**Registration Date:**
UPV	05-05-2017

3.3.2 N2N Layer

Design Pattern

Pattern name: INTER-IoT Pattern for Orchestration of SDN NetworkElements | **Identifier:**01

Inspired by:

- *“Software-defined networking (SDN) orchestration”*[54]
- *“Network virtualization (NV)”*[55]

Related patterns:

Intent:

Monitoring and configuration of SDN elements (virtual-switches) with an orchestrator component (Controller) exchanging flow and control messages. The main goal of the Orchestration of SDN Network Element pattern is to provide interoperability between different domains connected to a network or between different networks topologies and/or configurations.

Problem & Solution:

The domain-orientation of IoT deployments makes all of them to be isolated from each other. One of the solutions for the interconnection among them is, instead of making it at the device/gateway level, it is to solve it in upper layers. In particular, at network layer the interoperability and exchange of information can be done in a seamless way.

For that purpose, this INTER-IoT orchestration oriented pattern is defined to manage the elements of the network that provides connection from the different domains to the network itself. This pattern is used in the development of a virtual software defined network (SDN) where all elements are virtual resources or instances that are controlled within a central point or orchestrator. Network-to-Network (N2N) interconnection can then be performed through Software Defined Networking. The different networks, which can be on different locations, can be virtually interconnected and belong to the same Virtual LAN (VLAN). Thus, this physical separation of networks is invisible for the user and elements connected to the VLAN, which can only perceive one network. This allows network interoperability; elements can interoperate across both physical IoT networks.

- The virtualization of: (1) networks functions (NFV); (2) applications at the top of the controller; (3) nodes: perform interconnection in the network is used in this pattern to reduce the cost of the deployment and improve the efficiency of the system.
- Flow control pattern: is used for the self-awareness of the state of the network, that allows to react to specific situations.
- Message flow pattern: is a sequence of processing steps that run in the node when an input message is received. This sub-pattern is implemented in each node (virtual switch) and in the controller to process the message with a pre-defined rules.

Applicability:

This pattern is applied when a INTER-IoT Software Defined Network is deployed, to enable its functionality. This type of networks allows a total software control of the network functions, and a completely seamless and transparent Network-to-Network interoperability.

UML representation:

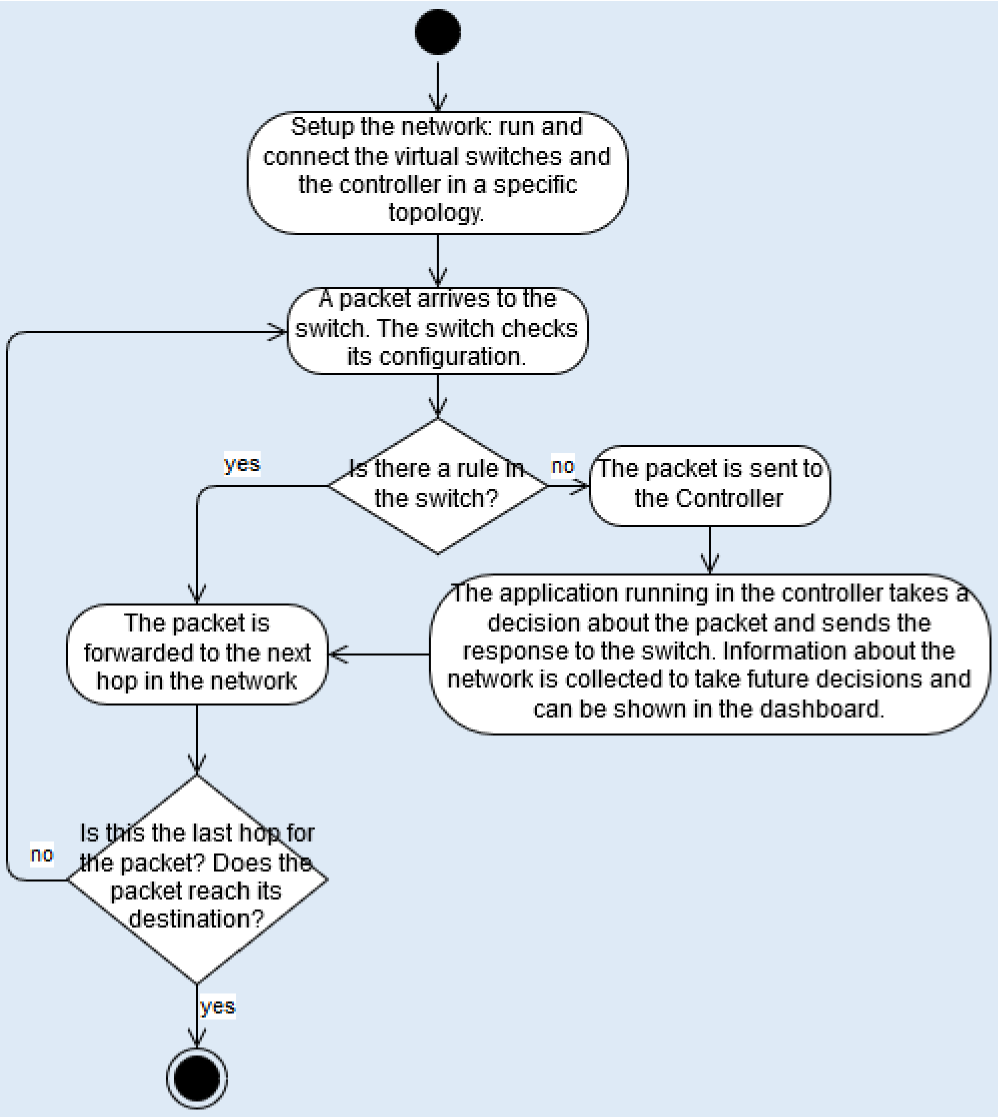

Setup the network: run and connect the virtual switches and the controller in a specific topology.
A packet arrives to the switch. The switch checks its configuration.
Is there a rule in the switch?
yes
no
The packet is sent to the Controller
The application running in the controller takes a decision about the packet and sends the response to the switch. Information about the network is collected to take future decisions and can be shown in the dashboard.
The packet is forwarded to the next hop in the network
Is this the last hop for the packet? Does the packet reach its destination?
no
yes

Figure 8: INTER-IoT Pattern for Orchestration of SDN Network Elements

Implementation:

Once the sensor defined network is created with a specific topology, the Controller manages one or several switches in the network in a dynamic way, and it is in charge of the switches' configuration. This network can be a set of several physical networks, interconnected in a virtual way, and thus acting as single network from a SDN perspective. The different parameters that are applied on the network can be changed within the controller of the network. Network traffic through the switches is redirected depending on the switches' rules or instead, on the controller application. The interconnection of the networks is represented with a virtual switch. This switch is from the SDN controller perspective no different from the virtualized switches of the networks, and receives the same type of control orders for forwarding packets or requires the same type of forwarding decisions.

After the setup of the SDN network, packets can be sent through it. Switches are responsible of routing the packets. For the virtual network, switches from different physical network are not distinguished, as well as the virtual switch that interconnects the networks. When a packet is received in a switch, this routing element checks the specific rules programmed for its forwarding. In case a rule applies to this packet, it is forwarded to its next hop in the network. In case there is no valid rule, unlike a traditional network, an upper element takes a decision regarding the packet: the SDN Controller. The Controller is aware of this undefined situation, it receives the packet and takes a decision regarding the action that must be performed with it. The Controller orders the switch to take that action. Moreover, the Controller registers information regarding the network situation to be taken into account in future decisions, and to create an accessible record through the SDN dashboard.

Each next hop within the SDN network that does not reach yet the final end-point has a similar process for deciding the forwarding of the packet. This happens in the same way even if the hop is in another network physically independent and distant, but virtually interconnected. N2N interoperability is thus achieved.

Known uses (within the INTER-IoT):

- Any use that involves the utilization of the INTER-IoT Software Defined Network (SDN) component.
- An example scenario is "Configuration of virtual switches within a SDN", which is a necessary configuration step for enabling the use of an INTER-IoT Software Defined Network.

Identified by:	**Registration Date:**
UPV	13-06-2017

3.3.3 MW2MW Layer

Design Pattern	
Pattern name: INTER-MW Simple Component Pattern	**Identifier:** 04

Inspired by:

- *"Simple component"* Reactive Patterns: Fault Tolerance and Recovery (Section: *"Reactive Patterns"*)

Related patterns:

Intent:

The intent of this pattern is to partition INTER-MW into multiple components, which operate as close as possible to the idea of doing only one thing, and doing it in full. This pattern is derived from the single responsibility principle, which states that a class should have only one reason to change.

Problem & Solution:

During the building of complex systems with multiple functions, appears the necessity to have these functions be performed by multiple different components. Responsibilities between these are to be divided recursively, until we reach a desired granularity of the component hierarchy. This would enable us to test, debug and extend the complex system more efficiently, simplifying all operations upon the system.

Applicability:

This is one of the most basic patterns that can be universally applied. It does not impose how much granularity in division of responsibilities should be achieved, but it does indicate that the analysis should be performed in order to end up with the best component decomposition for a given application. The pattern should be applied recursively, but one should remember not to divide components too far, not to end up with trivial ones.

In INTER-IoT, the main idea is to divide the responsibility of routing and translating messages between platforms and applications. These responsibilities are divided among components, joined into multiple distinct component groups. Each of these component groups contains several components.

UML representation:

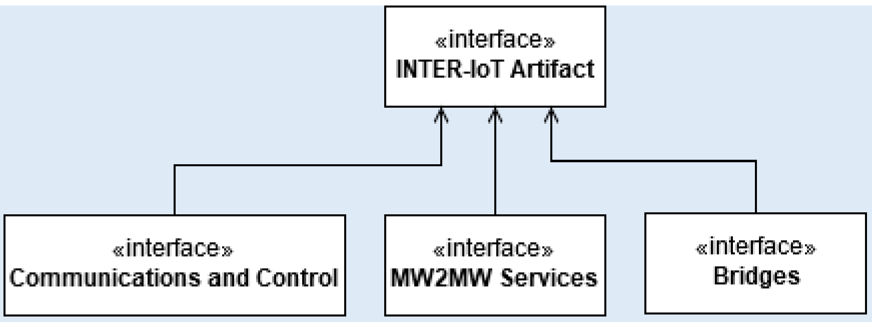


Figure 9: INTER-MW Simple Component Pattern

Implementation:

Within INTER-MW, there are three distinct component groups: (1) communications and control, (2) MW2MW services, and (3) bridges. All of them are further divided. Responsibilities of whole INTER-MW are clearly partitioned among the three component groups: (i) making communications and control component group in charge of preparing actions for execution and forwarding the results to the application, (ii) designating the bridges group to take care of communication with specific platforms, and (iii) leaving support of both two to the MW2MW services component group.

Within the communications and control group, are the Platform Request Manager and the API Request Manager. Within the MW2MW services group are the Resource Registry, Resource Discovery and Platform Registry and Capabilities components. Within the Bridges group there are various bridge components, that focus on enabling communication between the INTER-MW and various platforms.

Known uses (within the INTER-IoT):

- Three distinct component groups within INTER-MW: (1) communications and control, (2) MW2MW services and (3) bridges. Each component in each of one of these groups has only one purpose, e.g. API Request Manager takes care of application's API requests, while the bridges establish communication between INTER-MW and specific platforms.

Identified by:	**Registration Date:**
XLAB	13-06-2017

Design Pattern

Pattern name: INTER-MW Message Broker	**Identifier:** 05

Inspired by:

- *"Message Broker"* Enterprise Integration Patterns: Message Routing (Section: *"Enterprise Integration Patterns"*)
- *"Broker"* IoT Patterns: Design Patterns for Interaction (Section: *"IoT Patterns"*)

Related patterns:

- INTER-MW Self-contained Message

Intent:

A component that facilitates passing of messages between decoupled INTER-MW components.

Problem & Solution:

Building middleware, composed of several independent components, one tries to avoid making point-to-point connections between these, especially because many such connections correspond to many interfaces that expose the operations of each individual component. In general, components also tend to be proprietary, and even if that is not true in INTER-IoT architecture, having point-to-point interfaces makes not only dynamic reconfiguration, matching of different security constraints and QoS (quality of service) requirements between components difficult, but also general system's operation downright cumbersome.

The employment of a message broker, can help to overcome limitations of point-to-point connections and enforce a common messaging interface upon different middleware components. Thus allowing each component to initiate interactions with any other component (for example, a component that manages platform requests initiates interaction with the semantic mediator), no matter what their internal differences are and how big the difference in their purpose is. Each component communicates directly only with the broker, while internally in the broker, each component is represented with a logical name, abstracting the component and making its internal operation hidden from other components. Important part of this approach is the proper format of message, which consists of the payload and the label, which contains all the necessary information for the broker.

Applicability:

Central Message Broker receives messages from multiple destinations (called also message producers), determines the correct destinations (called also message consumers), and routes messages to correct channels that take them towards their destinations. It allows us to decouple the destination of a message from the sender and maintain central control over the flow of messages. This is achieved through usage of topics, to which consuming components can subscribe and proceed to consume messages, sent to those topics by the publishing components.

UML representation:

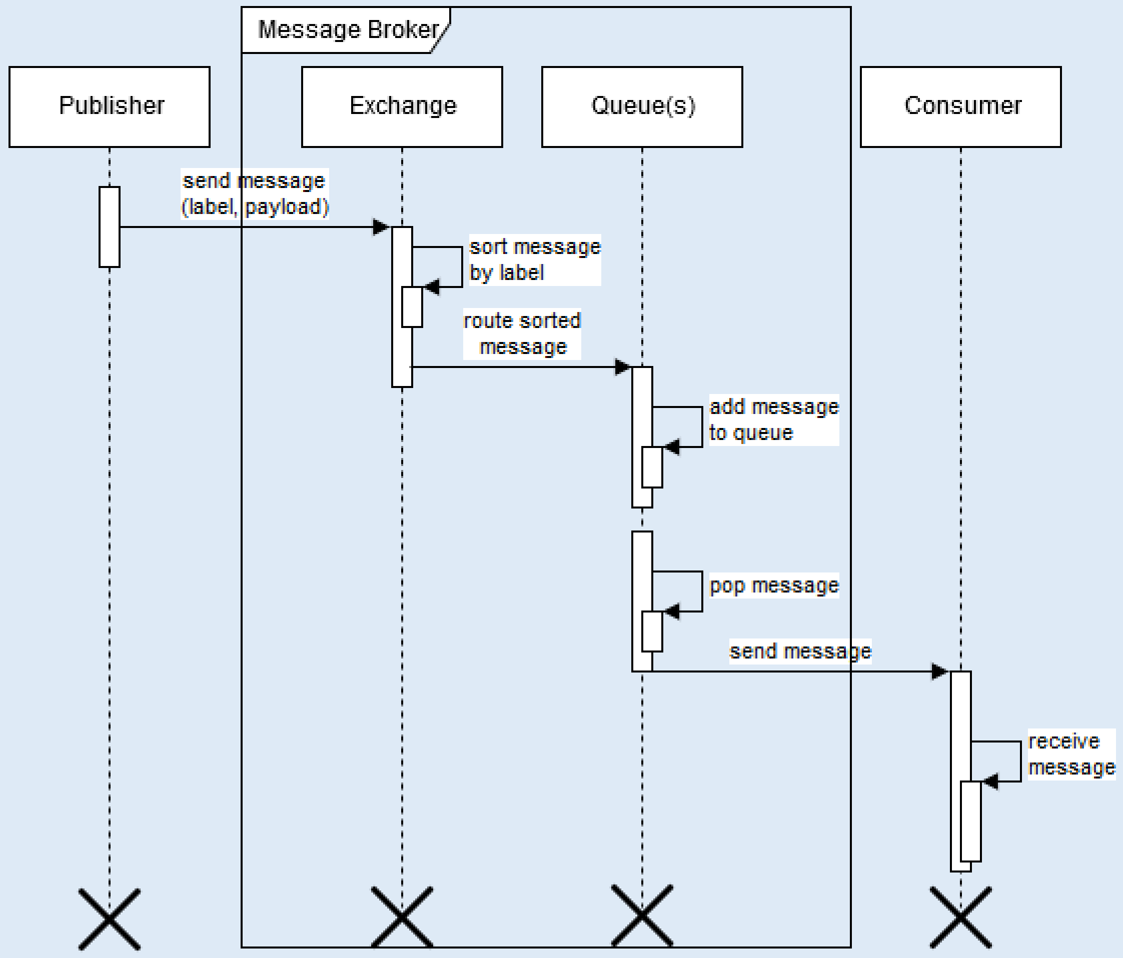


Figure: 10 INTER-MW Message Broker

Implementation:

The UML diagram depicts the message flow through the broker. Each message, which the publisher sends to the broker, is composed of a label and payload. Message payload contains the content of the message, while the label is used for message routing within the broker. When the message arrives into the broker, it is sorted in Exchange into applicable queue(s), based on its label.

INTER-MW label contains the topic that this message has to be routed to (for example, the topic that distinguishes messages that come from IoT artifact to the other one), and each topic has a queue associated with it. Message is then passed on to the consumer through the queues, associated with this topic.

Known uses (within the INTER-IoT):

- Message passing between components of INTER-MW (an example scenario: "Request query to MW2MW"). Each INTER-MW component (API Request Manager, Platform Request Manager, etc.) has at least one topic in the broker, through which other INTER-MW components can send messages to it. Topics represent information about which components are required, in order to act upon them and/or forward them. Thus, when a component receives a message on a particular topic, it knows also from which neighbouring INTER-MW component this message came, therefore it also knows the direction of the message – whether it is going downstream towards the bridges, or upstream towards the application.
- Within INTER-MW, this pattern first started life as a data flow manager, but evolved through removal of the concept of data flows into a full-fledged message-passing mechanism. The concept of the message broker naturally fits into the INTER-MW architecture, because its components both send and receive messages from one another, and they are all decoupled.

Identified by:	**Registration Date:**
XLAB	13-06-2017

Design Pattern

Pattern name: INTER-MW Self-contained Message	**Identifier:** 06

Inspired by:

- *"Self-contained message"* Reactive Patterns: Message flow (Section:*"Reactive Patterns"*)
- *"Messaging Metadata"* SOA Patterns (Section:*"SOA Patterns"*)

Related patterns:

Intent:

Each message contains all the information that is needed for execution of a particular action.

Problem & Solution:

Within middleware, the messages should be pure and complete representations of certain events or commands, regardless of whether we interpret them now or in the future. In doing so, there is no necessity to rely on additional data stores that need to be in time-sync with the event during message processing.

Each INTER-MW component must be able to extract from the message (at all times) all the information, needed for message's routing and interpretation, with minimal data stored within INTER-MW components themselves. Each message contains all data needed for its processing and understanding its purpose. Each INTER-MW message has a distinct set of message types associated with it. In each INTER-MW component, message is processed and routed based solely on this set of message types. For example, in the case of this set containing a particular message type that subscribes the application to platform's status updates, INTER-MW components that will handle the message will know, that the message has to be sent downstream, and when the message reaches the bridge for the relevant platform, this bridge will finally act upon this message type. For each message that goes downstream, there can also be a response that goes upstream. Such messages might, for example, have an additional response message type associated with them. Matching messages that go downstream with response messages that go upstream can be done through remembering and distinguishing different chains or conversations of messages.

Applicability:

This pattern is usable in the case of middleware components to be as context-free as possible, storing only a minimal amount of data needed for message processing and routing.

This pattern can be also employ in the case of no need to reference past messages, except for message responses, and even then, these are only semantically linked to original messages; one could even exist without original messages.

UML representation:

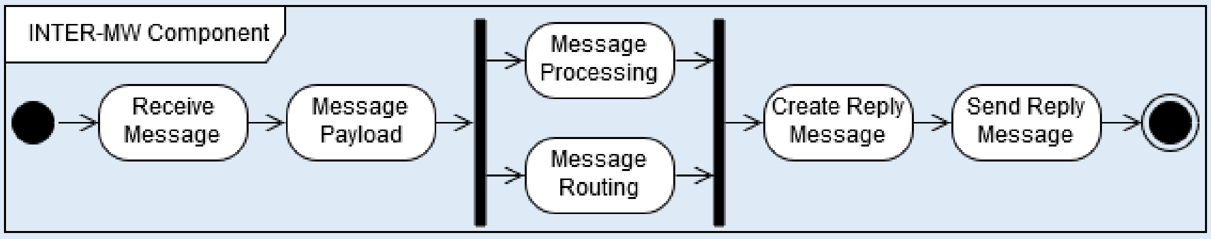


Figure 11: INTER-MW Self-contained Message

Implementation:

The activity diagram illustrates the message flow through an INTER-MW component. Message is processed based on its payload and routed based on its label (or metadata). The resulting message, often a mere copy of the original, is then routed towards the next INTER-MW component.

Known uses (within the INTER-IoT):

- "Query". Each message contains a message ID, as well as a Conversation ID. They together with message's type uniquely distinguish both the message itself and it purpose, such as SUBSCRIPTION, reply to SUBSCRIPTION, and so forth.
- Messages, that have the same Conversation ID, belong to the same conversation. Within a conversation are typically find messages that go down stream, followed by replies to those messages, that group stream. Furthermore, response messages have an additional RESPONSE type.

Identified by:	**Registration Date:**
XLAB	13-06-2017

Design Pattern	
Pattern name: INTER-MW Message Translator	**Identifier:** 07

Inspired by:

- *"Message Translator"* Enterprise Integration Patterns: Message Transformation (Section: *"Enterprise Integration Patterns"*)
- *"Data Format Transformation"* SOA Patterns (Section: *"SOA Patterns"*)

Related patterns:

Intent:

Translation of messages to and from INTER-MW internal message format and platform's proprietary data models and data formats.

Problem & Solution:

The purpose of middleware is to pass information between applications and different IoT platforms such as FIWARE[56] and universAAL[57]. However, as platforms expect to deal with messages in the format, that models their internal workings, they tend to use proprietary data models. This poses a major problem, since each platform formats the data in a different way. Both messages that go upstream from the platforms towards the application and messages that go downstream towards the platforms need to be syntactically and semantically translated, taking into account proprietary data models and data formats, used by different platforms.

A message translator enables such translation between proprietary data models and data formats, used by platforms, and the internal data model and data format, used by INTER-MW components.

Applicability:

INTER-MW's message translator pattern is realized through multiple INTER-MW components and enables interoperability between different platforms without needing to introduce translations between every possible pair of platforms; that is, translation into and out of the common INTER-IoT data model.

Semantic translation from and into the internal message format is done by a dedicated IoT semantic translation component, while the syntactic translation is done in individual platform's bridges, as only these know the internal platform's data format and the syntax it uses.

UML representation:

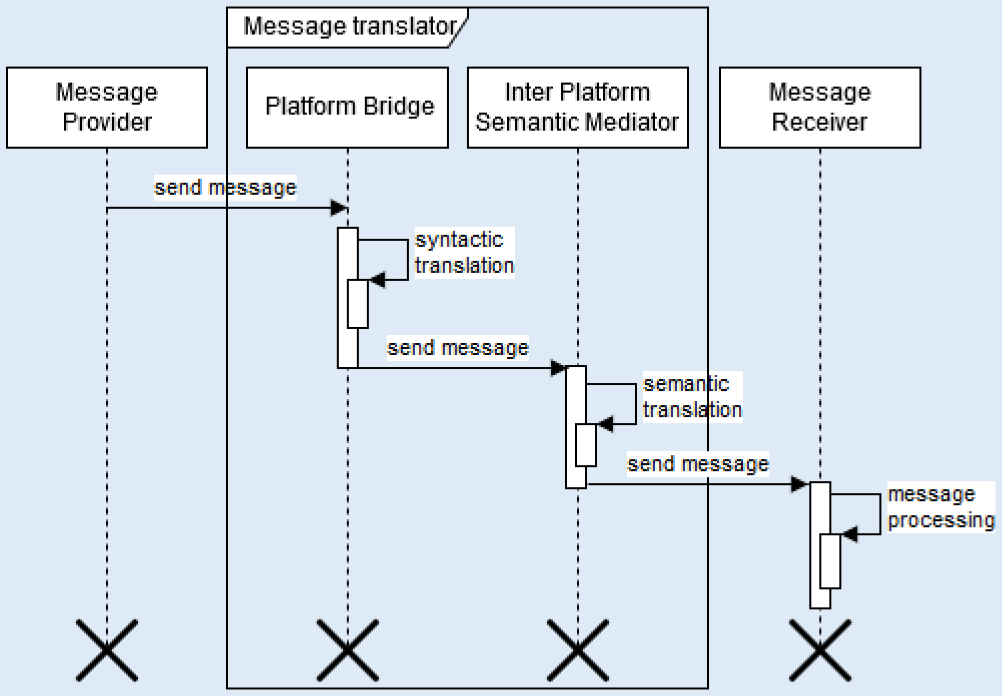


Figure 12: INTER-MW Message translator

Implementation:

The sequence diagram illustrates the flow of messages between the two INTER-IoT artifacts (e.g.

platform). Syntactic translation to/from proprietary message format is done in the bridge, while the semantic translation of the message is done in the Inter Platform Semantic Mediator.	
Known uses (within the INTER-IoT): • INTER-MW use case: "MW2MW sends information to device(s)".	
Identified by: XLAB	**Registration Date:** 13-06-2017

3.3.4 AS2AS Layer

Design Pattern	
Pattern name: AS2AS Flow Based Service Composition	**Identifier:** 11
Inspired by: • *"Flow-based programming"*[58]	
Related patterns: • AS2AS Service Orchestration • AS2AS Discovery of IoT Services	
Intent: To generate a service execution flow that allows an interoperation and composition of services from different IoT platforms.	
Problem & Solution: AS2AS aims to provide a solution to allow interoperability of services and applications from different IoT platforms. For this aim, AS2AS will furnish a solution to allow the catalog, discovery and use of different services from diverse heterogeneous IoT platforms. In this solution it is necessary to provide an execution flow that allows a specific sequence of execution of several services, the aforementioned IoT services. Flow-based programming (FBP) is a programming paradigm that defines applications and services as networks of "black box" processes, which exchange data across predefined connections by message delivery, where the connections are specified externally to the processes. These black box processes can be reconnected to form different applications without having to be changed internally. FBP is thus naturally component-oriented, considering those black box as components. The AS2AS flow-based execution pattern allows the creation of sequential execution flows using those services, thus allowing for service composition among different IoT services. Those black boxes that represent IoT services can be linked by wiring the output of a service with the input of a different one. Each defined input can only be feed by one wire, generally from an output. This creates a flow; output messages from a service are routed to a service input. Several services can be concatenated in this way, by linking an output of a first service to the input a second service, and its output to the input of a third one and so on successively. Thus, by wiring the IoT services (or a service for performing data processing), can created an execution flow of IoT services.	
Applicability: This pattern can be applied for any black boxes that represent IoT services, that can be interconnected through a FBP link, generating a flow.	
UML representation:	

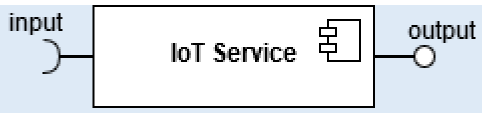


Figure 13: AS2AS Flow Based Pattern

Implementation:

The UML - component diagram depicts the IoT service as a black box. Every IoT service contains input and output to wire it with other services. This approach enables IoT services to be linked by wiring the output of a service with the input of a different one. Each defined input can only be feed by one wire, generally from an output. This creates a flow; output messages from a service are routed to a service input.

Known uses (within the INTER-IoT):

- AS2AS browser-based flow editor for joining together services from different IoT platforms available in INTER-IoT, in order to create flows (an example scenario: to design a flow composed by several services).

Identified by:	**Registration Date:**
UPV	12-06-2017

Design Pattern	
Pattern name: AS2AS Service Orchestration	**Identifier:** 09

Inspired by:

- *"Service Orchestration"* Service Orchestration (Section: *"Service Orchestration"*)

Related patterns:

Intent:

To adapt the orchestration of services to an INTER-IoT solution that is in charge of the interactions among different IoT services to produce a specific process.

Problem & Solution:

Cooperating diverse heterogeneous IoT platforms uses huge number of different services. It causes the need to find the way of making different, individual services work together in a reasonable way. The important is not only a message flow from point(s) to point(s) but also triggering the necessary actions (during the flow process). The main and common problem is that instead of reusing the existing processes/actions, they are duplicated.

This pattern allows the union and orchestration of IoT services from heterogeneous platforms. The union of several services creates a specific process. The main idea of this pattern is to define a set of INTER-IoT nodes, i.e. services and its interfaces, runs within the integrated platforms and wire. The internal, central core element wire the nodes, necessary to handle the specific task and controls all the processes. Due to this, it allows a higher range of options for this execution.

Applicability:

This pattern focuses on process fragment reuse and is designed for goal based processes. Orchestration enables the composition of IoT service workflows based on services from the underlying IoT platforms.

UML representation:

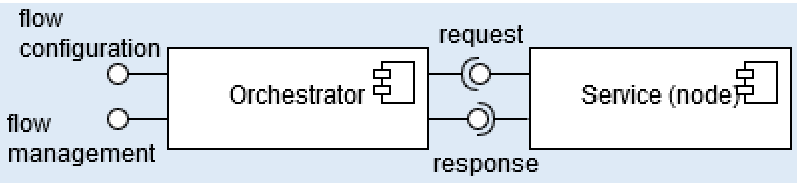


Figure 14: AS2AS Orchestration Pattern

Implementation: The UML diagram depicts the orchestrator way of action, i.e. designing the task, requiring the work of many services. It allows to wire together flows, using defined nodes (INTER-IoT services). Moreover, it is possible to manage the flow in an easy way. The orchestrator deploys the designed flow to the runtime.	
Known uses (within the INTER-IoT): • AS2AS orchestrator module. A Node-RED tool is used to create the visual model of the sequence that compose this workflow and to create the code that will perform the sequence of actions.	
Identified by: UPV	**Registration Date:** 12-06-2017

Design Pattern	
Pattern name: AS2AS Discovery of IoT Services	**Identifier:** 10
Inspired by: • *"Discovery"* IoT Patterns: Design Patterns for Interaction (Section: *"IoT Patterns"*) • *"Enterprise inventory"*: SOA Patterns (Section: *"SOA Patterns"*)	
Related patterns:	
Intent: This design pattern allows to register and claim the specific services, used by the platforms (within the INTER-IoT environment).	
Problem & Solution: There are a variety of available IoT services from different IoT platforms that provide a wide range of functionalities. It is necessary to be able to discover these services, to be aware of them and to be capable to use them appropriately in an efficient way. This design pattern enables the registration of services in the IoT environment, in order to consultation of available services from IoT platforms, as well as its potential use through the INTER-IoT AS2AS solution. To perform the Discovery of IoT services, it is necessary to have them previously registered in a catalog of available services, prior the Discovery action. Thus, only services previously registered in INTER-IoT, indicating their associated features, can be discovered.	
Applicability: This pattern is applicable for providing the services interoperability by registration process and claiming process. It is used for all services run within the INTER-IoT environment and used by other INTER-IoT artifacts.	

UML representation:

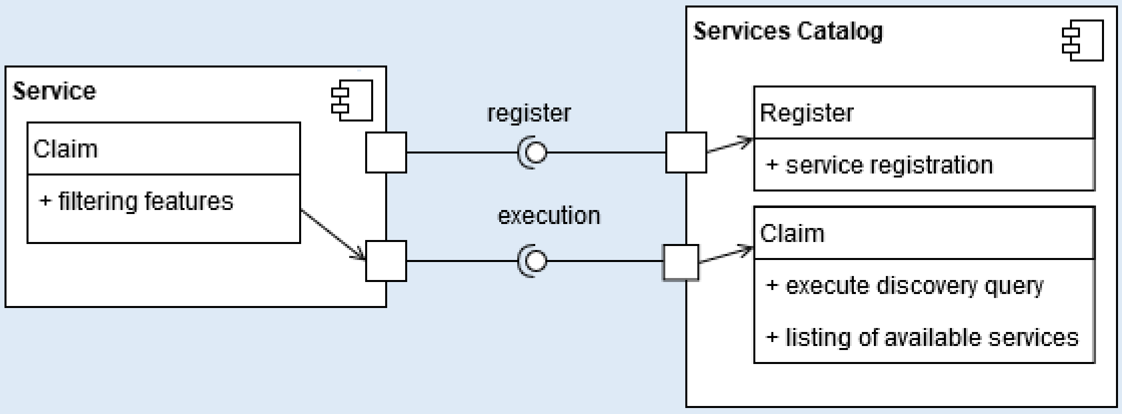


Figure 15: Discovery of IoT Services

Implementation:

The UML diagram depicts two aspects, relevant for this pattern. First is a service registration process, in order to provide the service as a public. Second main functionality is discovery action, i.e. claiming the necessary service(s) by INTER-IoT artifact.

Known uses (within the INTER-IoT):

- AS2AS discovery module.

Identified by:	**Registration Date:**
UPV	12-06-2017

3.3.5 DS2DS Layer

Design Pattern	
Pattern name: Alignment-based Translation Pattern	**Identifier:** 13

Inspired by:

- *"Message Translator"* Enterprise Integration Patterns: Message Transformation (Section: *"Enterprise Integration Patterns"*)
- *"Data Model Transformation"* SOA Patterns (Section: *"SOA Patterns"*)
- *"Metadata centralization"* SOA Patterns (Section: *"SOA Patterns"*)
- *"Market Maker"* Agent Design Patterns (Section: *"Agent Design Patterns"*)

Related patterns:

Intent:

Semantic translation of RDF messages exchanged between two IoT artifacts, based on an alignments (set of correspondences) defined between artifacts' ontologies.

Problem & Solution:

Building the IoT ecosystem involves combining already existing solutions, which (likely) belong to different owners and have been developed using different technologies (e.g. a Web Services-based application is combined with a graph database-based application which communicates using JSON messages and with an application that communicates using XML messages). Consequently, they differ both on syntactic and semantic level. Cooperation and communication between platforms should be made possible regardless of their underlying technology and the scalability of the IoT environment. Without loss of generality, the message format can be assume to be RDF since other formats can be transformed to RDF. The semantics of messages is platform specific (ontology can be natively supported, or semantics expressed in e.g. XSD can be lifted to OWL ontology). Specifically, for semantic interoperability an expressive method for defining correspondences between entities should be offered. Such correspondences should support mapping between specific URIs, parts of the RDF structure, transformations etc. The component implementing the translation should provide interfaces to submit messages to be translated and publish translated messages.

Applicability:

This pattern is applicable for providing semantic translation between RDF messages exchanged between heterogeneous IoT artifacts. The translation, based on one-to-one translation (alignment), should be possible to define for any two artifacts. The advantage of this approach is a good quality of translations between ontologies, since the alignments are generated directly between ontologies of two parties, without any “intermediate ontology”. Moreover, it is easy to understand and debug the translation process.

UML representation:

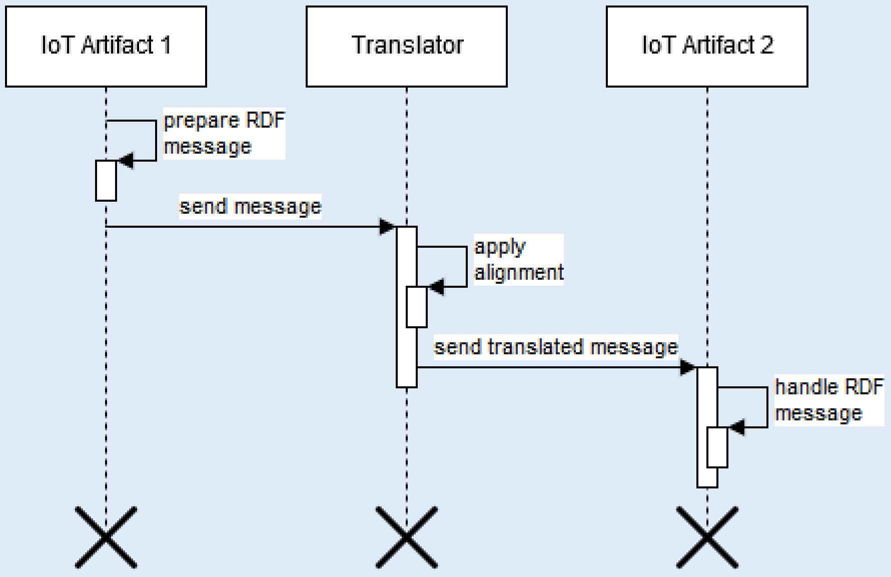


Figure 16: Alignment-based Translation Pattern

Implementation:

The UML diagram depicts a process of alignment-based translation between two IoT artifacts. Translator is an external component that applies alignment i.e. performs translation. It consumes incoming messages, applies an alignment (correspondence after correspondence), and publishes translated message.

Known uses (within the INTER-IoT):

IPSM component of INTER-IoT uses this pattern to execute translation between source IoT artifact and INTER-MW (source ontology is translated to central ontology), and INTER-MW and target IoT artifact (central ontology is translated to target ontology). The alignment based translation pattern is used in both step of semantic translation between source and target platforms. Rules for each translation step are defined in an alignment persisted in IPSM Alignment Format based on Alignment API Format level 2. The IPSM Alignment Format allows to formally specify correspondences between parts of RDF graphs with additional features such as calling functions.

IPSM uses the communication infrastructure based on channels (Apache Kafka) that provide interfaces (topics) to consume and publish messages. IPSM consumes the messages to be

translated from a predefined topics and publishes translated messages to other predefined topics.	
Identified by: SRIPAS	**Registration Date:** 20-09-2017

Design Pattern	
Pattern name: Translation with central ontology	**Identifier:** 12
Inspired by: • *"Message Translator"* Enterprise Integration Patterns: Message Transformation (Section: *"Enterprise Integration Patterns"*) • *"Data Model Transformation"* SOA Patterns (Section: *"SOA Patterns"*) • *"Metadata centralization"* SOA Patterns (Section: *"SOA Patterns"*) • *"Market Maker"* Agent Design Patterns (Section: *"Agent Design Patterns"*)	
Related patterns:	
Intent: Semantic translation of RDF messages exchanged between IoT artifacts, where one constitutes the central point for achieving interoperability.	
Problem & Solution: To provide common understanding in the semantic translation process a modularized central ontology can be created from "merged" IoT and domain ontologies. Here, a domain ontology is a conceptual model for specific domain, e.g. transportation, health, etc. IoT ontology describes different aspects related with the IoT domain e.g. platforms, devices, sensors, services, etc. The central ontology is to be specified when the initial ecosystem is put in place. First advantage of this approach is that it does not suffer from the scalability problems. After a number of artifacts is combined into a working ecosystem, it should be possible to add additional ones without decreasing performance. Moreover, joining should not involve extra effort, larger than combining the original group of artifacts. Integration of new artifacts involves creation of a single pair of translation rules (alignments; see pattern 13) with the central ontology (in case of unidirectional interoperability only one alignment is enough). Second, such approach requires less preparation/work from the semantic engineer responsible for bringing the new IoT platform to the ecosystem. This is because only a single "point of joining" has to be instantiated. Furthermore, the long-term maintenance is simplified as changes in a single platform require localized adjustments only. The component implementing the translation should provide interfaces to submit messages to be translated and publish translated messages.	
Applicability: This pattern is applicable for providing semantic translation between multiple heterogeneous IoT artifacts that want to exchange RDF messages. Note that, this pattern can be extended also to semantic translation outside of IoT domain. The translation should be provided for more than two artifacts at the same time minimizing the size of required storage for alignments.	
UML representation:	

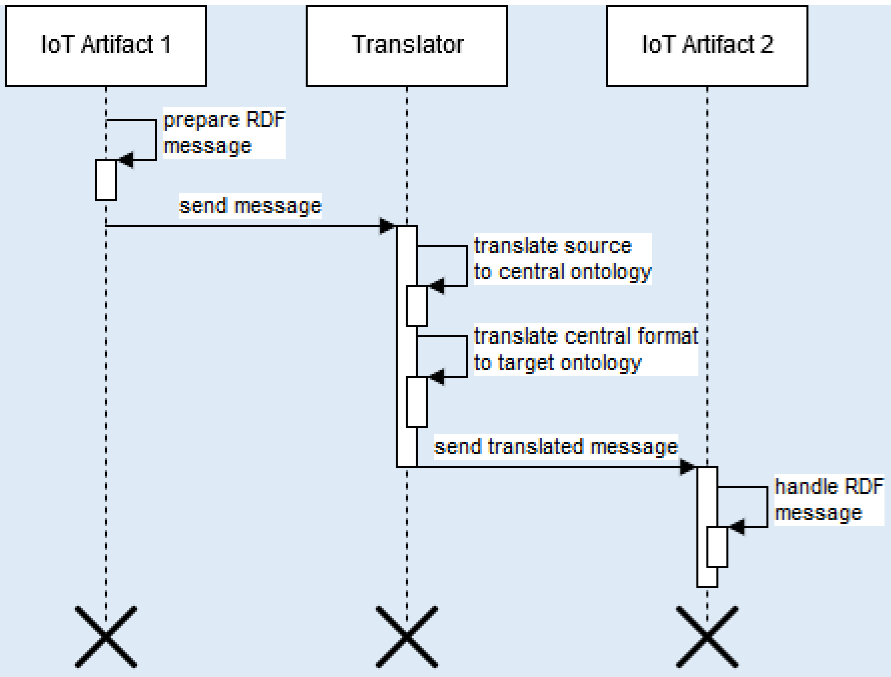


Figure 17: Translation with central ontology

Implementation: The UML diagram depicts a process of a semantic translation with central ontology between two IoT artifacts. Translator is an external component that performs translation (applies alignment). It consumes incoming messages, applies an alignment (correspondence after correspondence) from source to central and from central to target semantics, and publishes translated message.	
Known uses (within the INTER-IoT): IPSM component of INTER-IoT uses this pattern to execute translation between source IoT artifact and target IoT artifact. The translation process is divided into two steps: input message is translated from source semantics to the central ontology, message in the central ontology is translated to target semantics. The modularized central ontology is deployment specific but the core is based on GOIoTP (Generic IoT Platform Ontology) and additional modules are domain specific e.g. transportation, health.	
Identified by: SRIPAS	**Registration Date:** 20-09-2017

3.3.6 CROSS Layer

Design Pattern	
Pattern name: INTER-IoT SSL CROSS-Layer secure access	**Identifier:** 17
Inspired by: • Security Patterns (Section: *"Security Patterns"*) • IoT Patterns: Design Patterns for IoT Security (Section: *"IoT Patterns"*)	
Related patterns: • Login Authentication • Sensitive Data Encapsulation	

- Encryption and Single Point of Access

Intent:

Ensuring the security of the interactions with external interfaces (i.e. APIs) of every layer that composes INTER-IoT.

Problem & Solution:

As INTER-IoT architecture is composed by diverse layers, the access to each of these layers, as well as the interactions among them, must be secure. A gap of security in a layer of INTER-IoT jeopardizes the security and integrity of the whole layered framework and connected platforms. To ensure a sufficient level of security on each of the INTER-LAYER and INTER-FW components, different security mechanisms can be implemented: authentication of credentials, use of authentication tokens and Secure Sockets Layer (SSL).

The transmission of unencrypted data makes the information vulnerable. Even when login authorization is set, when the user enters a password if the data transmission is unprotected, a network sniffer program from another user in the network, or a man-in-the-middle, are able to receive this information and read it. Any public Wi-Fi network can be easily eavesdropped by an unwanted actor. In that case, if passwords are sent in plain text, not only the information can be seen by non-authorized users, also false information can be sent as authentic in both sides of the communication with the INTER-IoT APIs. In the case of sensitive data, such as medical, this can lead to serious hazards in the people’s privacy and personal scope, and lead to ethical issues. Integrity of data suffers a high risk in non-protected communications. It has crucial importance to guarantee the authenticity, privacy and confidentiality of data.

In INTER-IoT, layer access will be secured with Secure Sockets Layer (SSL) that employs the pattern of INTER-IoT SSL. Every INTER-IoT layer exposes a REST API that represents an external interface accessible to external actors, such as other INTER-IoT layers, users, or IoT platforms.

To enable the use of this APIs to only allowed actors the access is secured through SSL. REST APIs are accessible through a browser, which should provide a trusted certificate. Only after the establishment of a secure connection it will be accessible the authentication through login, to open the access to the layer API. Further operations on the layer API will be done under this secure connection.

Applicability:

This pattern is applied in the interactions of any actor with the INTER-IoT layers APIs. The access can also be done internally among a pair of different layers. See D3.2 for further information about interactions among the INTER-IoT layers (D2D, N2N, MW2MW, DS2DS, AS2AS).

UML representation:

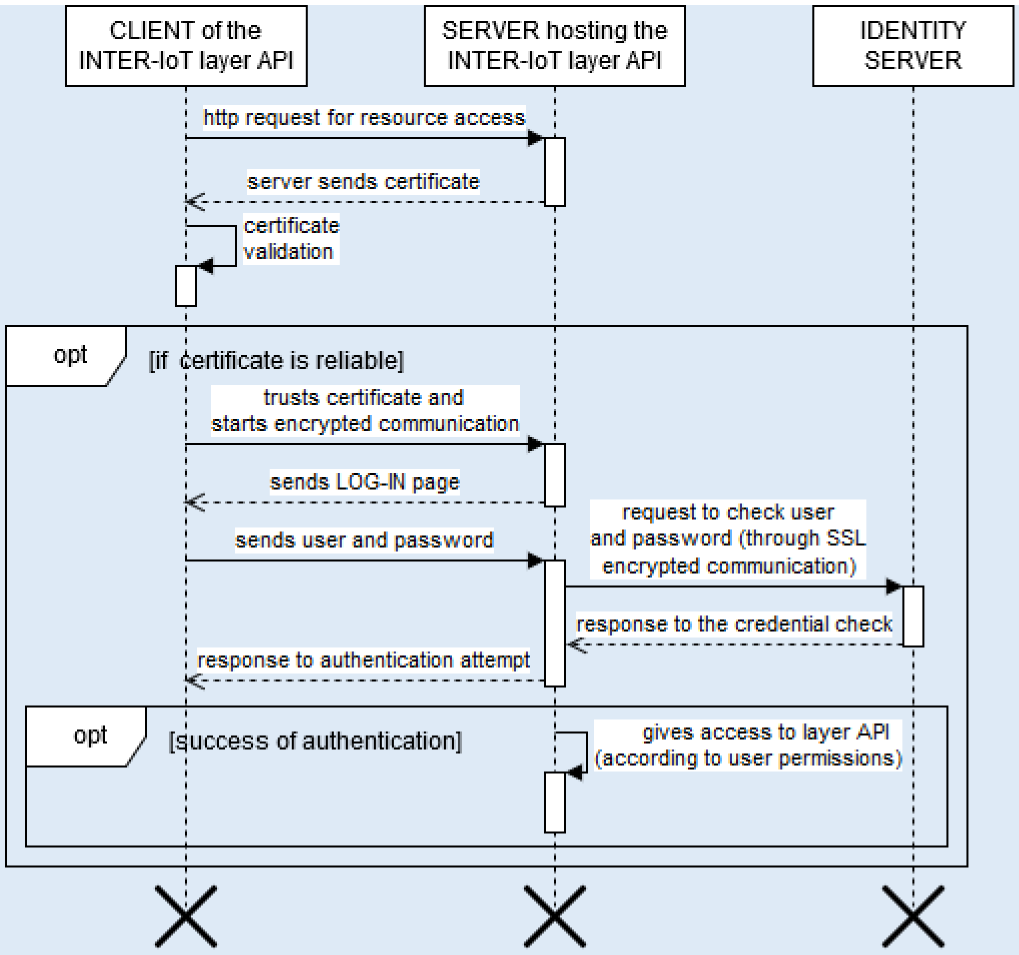

CLIENT of the INTER-IoT layer API
SERVER hosting the INTER-IoT layer API
IDENTITY SERVER
http request for resource access
server sends certificate
certificate validation
opt
[if certificate is reliable]
trusts certificate and starts encrypted communication
sends LOG-IN page
sends user and password
request to check user and password (through SSL encrypted communication)
response to the credential check
response to authentication attempt
opt
[success of authentication]
gives access to layer API (according to user permissions)

Figure 18: INTER-IoT SSL for a secure access to layers' APIs Authentication

Implementation:

A browser or an external server (acting as a client in this model) intends to connect with the INTER-IoT layer API web interface (server). This INTER-IoT web service is secured with SSL and sends a copy of its SSL certificate to identify itself. The server or browser trying to access the INTER-IoT API should trust the INTER-IoT SSL certificate, and send back a digitally signed acknowledgement to start an SSL encrypted session. Encrypted data is shared between the client and the INTER-IoT web service. Then a second security mechanism will take place, and the web interface will request an authorized user name and associated password to access to the layer API functionality. These credentials could be checked in an external identity server, that stores passwords and users' permissions, through a secured SSL connection. Thus, only authorized actors will be able to use the layer functions, with the degree of privileges and access granted to their user. Authentication after the establishment of a secure SSL connection (and not prior to) will impede password thefts and loads of accounts, enabling a higher degree of security.

Known uses (within the INTER-IoT):

- When an external user/actor intends to access and interact with a layer API (i.e. D2D, N2N, MW2MW, AS2AS, DS2DS). For instance, if a platform wants to receive sensor data flows from the INTER-IoT gateway needs to interact with the D2D API as an external user.
- With the internal interaction of INTER-IoT layers with other layers APIs (e.g AS2AS <=> MW2MW, or any other layers' interactions).

Identified by:	**Registration Date:**
UPV	17-11-2017

3.3.7 INTER-FW

Design Pattern

Pattern name: Configuration delegation pattern	**Identifier:** 18

Inspired by:

- *"Delegation Pattern"* Object-oriented Patterns (Section: *"Object-oriented Patterns "Gang of Four""*)

Related patterns:

Intent:

Configuration of multiple instances of heterogeneous IoT platforms or IoT artefacts in a single place, abstracting from singularities of each individual case and offering a global view.

Problem & Solution:

Interoperability services management implies the availability of the configuration information of the multiple heterogeneous original resources in a single place, to provide a central view of the different artifacts to be connected and evaluate properly the different strategies to follow towards the best interoperability solution.

The configuration parameters of the different platforms to be connected can differ greatly, specially considering that the INTER-IoT approach aims at developing a future-proof interoperability solution, and thus, not only the existing or most prominent platforms/artifacts must be considered, but also possible new comers, expanding the number of configuration points to virtually infinite.

Exposing the configuration of the different IoT artifacts, generalizing the common points and hiding the differences enables a general overview at different interoperability levels and, thanks to the homogenization of the view of the heterogeneous artifacts (that can also be seen as data source at different abstraction layers), it is also allowed the operation of the data access mechanisms (pub/sub, query, update) or authorization actions at the highest level.

Applicability:

This pattern is applicable for providing a common interface for elements at the same abstraction level (so-called layers in the framework of INTER-IoT). This pattern can be applied recursively, exposing a façade common for the different layers, allowing the replication of the pattern until building a single administration and management view for the convenience of the end-user.

For the correct applicability of the patterns, the artifacts that delegate their configuration to the façade interface must accomplish some aspects: 1) to expose an access interface (e.g. a REST API or some web services); 2) to share some common characteristics that are relevant for the problem being solved (in the framework of INTER-IoT, the interoperability of IoT platforms); 3) to be able to authorise external applications to access relevant data in the platforms.

UML representation:

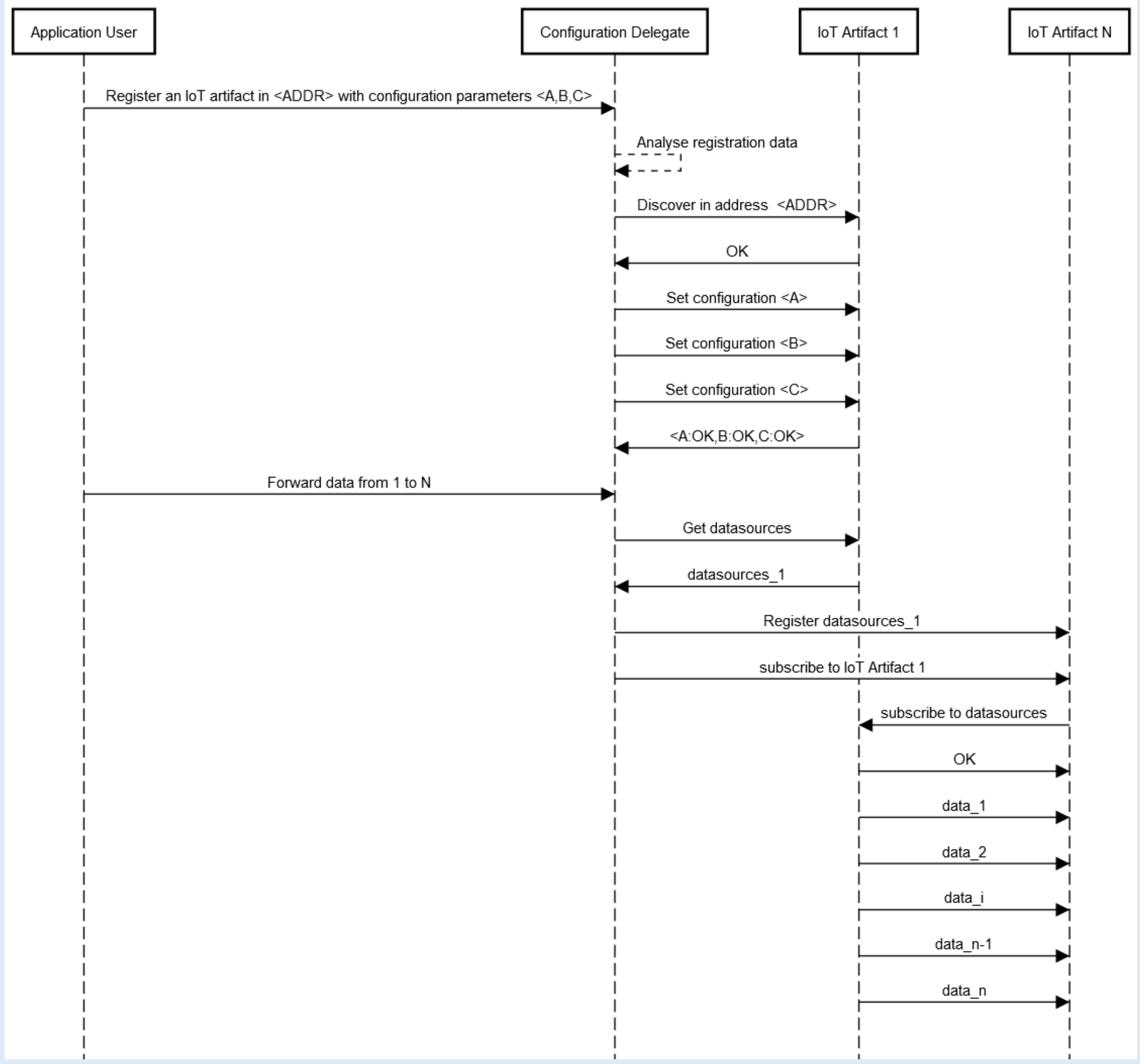


Figure 19: Configuration delegation pattern (example of a process)

Implementation:

The implementation of this pattern is done by establishing a middle point which abstracts configuration and management of the different IoT artifacts connected, as it can be seen in the UML diagram. In practise, this element has access to a number of common commands offered by the artifacts through an API or an adapter, which allows the configuration of a limited homogeneous set of aspects from a single point.

Known uses (within the INTER-IoT):

As a natural consequence of the generalization process, singularities or specialization in the artifacts are hidden, so that only generic features are covered by this pattern. A way to partially overcome this pitfall is to increase the number of endpoints in the artifacts interfaces, mocking or answering with a ‘not implemented’ like error when a particular feature is not available in an artifact.

Due to the complexity of the multi-layer artifacts, authorization mechanism are preferred to be implemented only at the same layer level. Even considering this, authorization is complex, as some layers access resources categorized in inferior layers, as in the case of services.

Identified by:	**Registration Date:**

PRODEVELOP	29-11-2017

Design Pattern	
Pattern name: API façade	**Identifier:** 19
Inspired by: • *"Web API Design: Crafting Interfaces that Developers Love"*[59]	
Related patterns:	
Intent: Create a single unique API for IoT artifacts interoperability.	
Problem & Solution: Interoperability mechanisms are complex and very heterogeneous depending on the abstraction level they are designed for. Operations at different interoperability levels can be dependent on libraries, other interoperability process or simply not be available in standard web accessible interfaces. Complex interoperability systems (such in the case of INTER-IoT) can't be represented/exposed in a single API but in an array of complementary systems that all need to be used to fulfil the user needs. This pattern gives a buffer or virtual layer between the interface on top and the API implementation on the bottom. It essentially creates a façade – a comprehensive view of what the API should be and importantly from the perspective of the app developer and end user of the apps they create. The developer and the app that consume the API are on top. The API façade isolates the developer and the application and the API. Making a clean design in the facade allows you to decompose one hard problem into a few simpler problems.	
Applicability: This pattern is applicable to homogenize the access to interfaces heterogeneous by nature, making easier and simpler the access and comprehension on the overall goal of the system. The API façade can also compound several atomic actions on the subsystems.	
UML representation:	

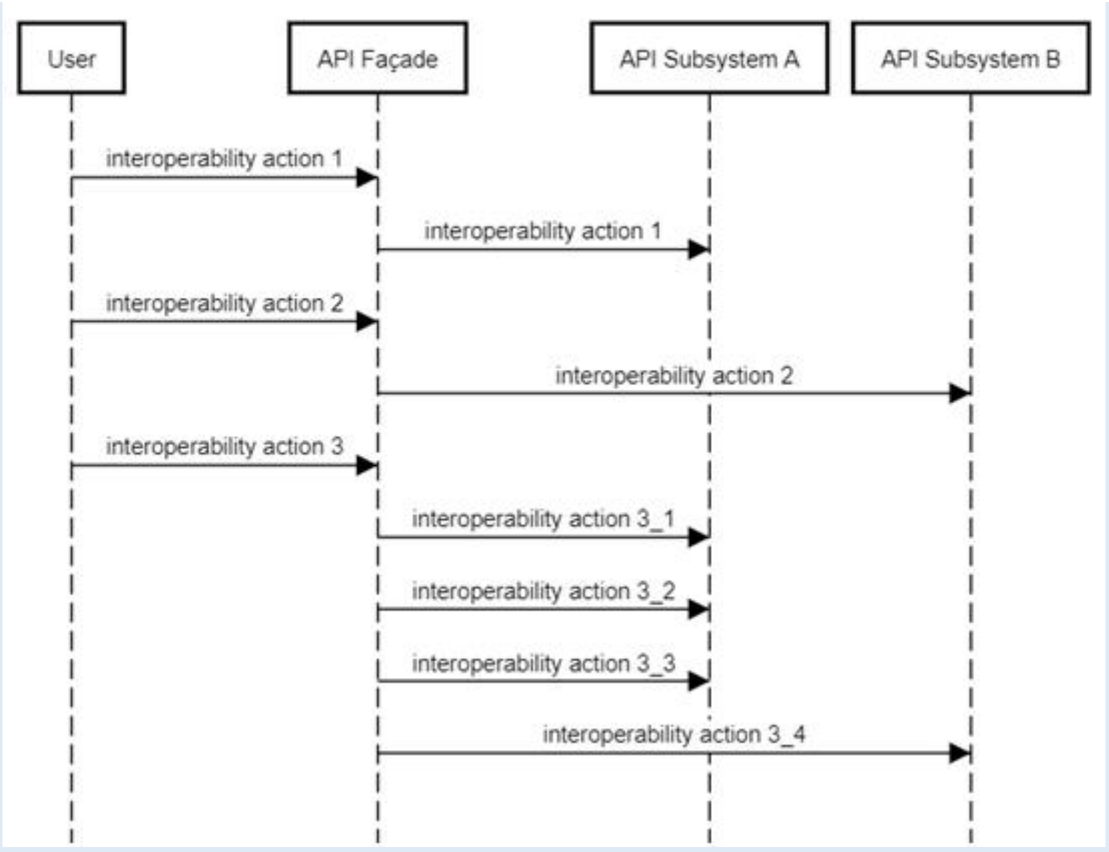

User
API Façade
API Subsystem A
API Subsystem B
interoperability action 1
interoperability action 1
interoperability action 2
interoperability action 2
interoperability action 3
interoperability action 3_1
interoperability action 3_2
interoperability action 3_3
interoperability action 3_4

Figure 20: API façade pattern (example of a process)

Implementation:

1. Design the ideal API – design the URLs, request parameters and responses, payloads,headers, query parameters, and so on. The API design should be self-consistent.
2. Implement the design with data stubs. This allows application developers to use your API and give you feedback even before your API is connected to internal systems.
3. Mediate or integrate between the façade and the systems.

Known uses (within the INTER-IoT):

The API façade can turn complex for the user depending on the number of subsystems that is composed by. In the case of INTER-IoT, the heterogeneity of these systems and the depth in the individual APIs can increase the overall learning curve.

Identified by:	**Registration Date:**
PRODEVELOP	29-11-2017

3.3.8 INTER-Health

Design Pattern	
Pattern name: INTER-Health Pilot Integration	**Identifier:** 15

Inspired by:

- (e/m) Health use case patterns (Section:*“Use case specific patterns”*)
- Enterprise Integration Patterns (Section:*“Enterprise Integration Patterns”*)

Related patterns:

- IoT Patterns: Design patterns for connected things (Section: *"IoT Patterns"*)
- IoT Patterns: Design patterns for IoT Infrastructure (Section: *"IoT Patterns"*)
- IoT Patterns: Design patterns for IoT Security (Section: *"IoT Patterns"*)
- IoT Patterns: Edge Based IoT Design Patterns (Section: *"IoT Patterns"*)

Intent:

Follow the most suitable design of the overall deployment model that fits the requirements of the INTER-Health pilot. This requires deploying all the modules of INTER-IoT that are needed, in addition to the existing IoT platforms, and the native applications, that are used in the pilot. Then all these must be connected to each other according to the defined APIs, and operate together properly.

Problem & Solution:

The INTER-Health pilot integrates two existing IoT platforms: BodyCloud and universAAL, and a native application: the Professional Web Tool. The INTER-IoT platform is used to bridge all these together. But each platform is used to cover different scenarios, and the application must be platform-agnostic. While BodyCloud is used to access a scalable number of mobile devices with sensors connected to it, universAAL is used to connect a limited set of mobile devices with sensors within the premises of the deployment. Each platform also has different installation requirements. On the other hand the native application has to access the sensor data from these two platforms, and it also has its own different set of installation requirements. To top it all, INTER-IoT, which will be used to bring all components together, also has its own requirements.

The design shown here depicts how to connect all these disparate components within the premises of the pilot partner and cover all the required sensors, in addition to provide hints as to how to install each component according to its requirements and constraints.

Applicability:

This design pattern can be used as a starting point or inspiration for similar deployments in real-life scenarios that present similar constraints: Installing INTER-IoT within the premises of the provider of a certain application or set of applications, and obtaining sensor data from several IoT platforms that cover a range of remote devices. Notice that it is not necessary that such deployment is related to e-Health scenarios. Notice as well that this pattern is suitable for on-premises server-based installations and not Cloud-based installations, although it should be relatively easy to adapt it to such environments.

UML representation:

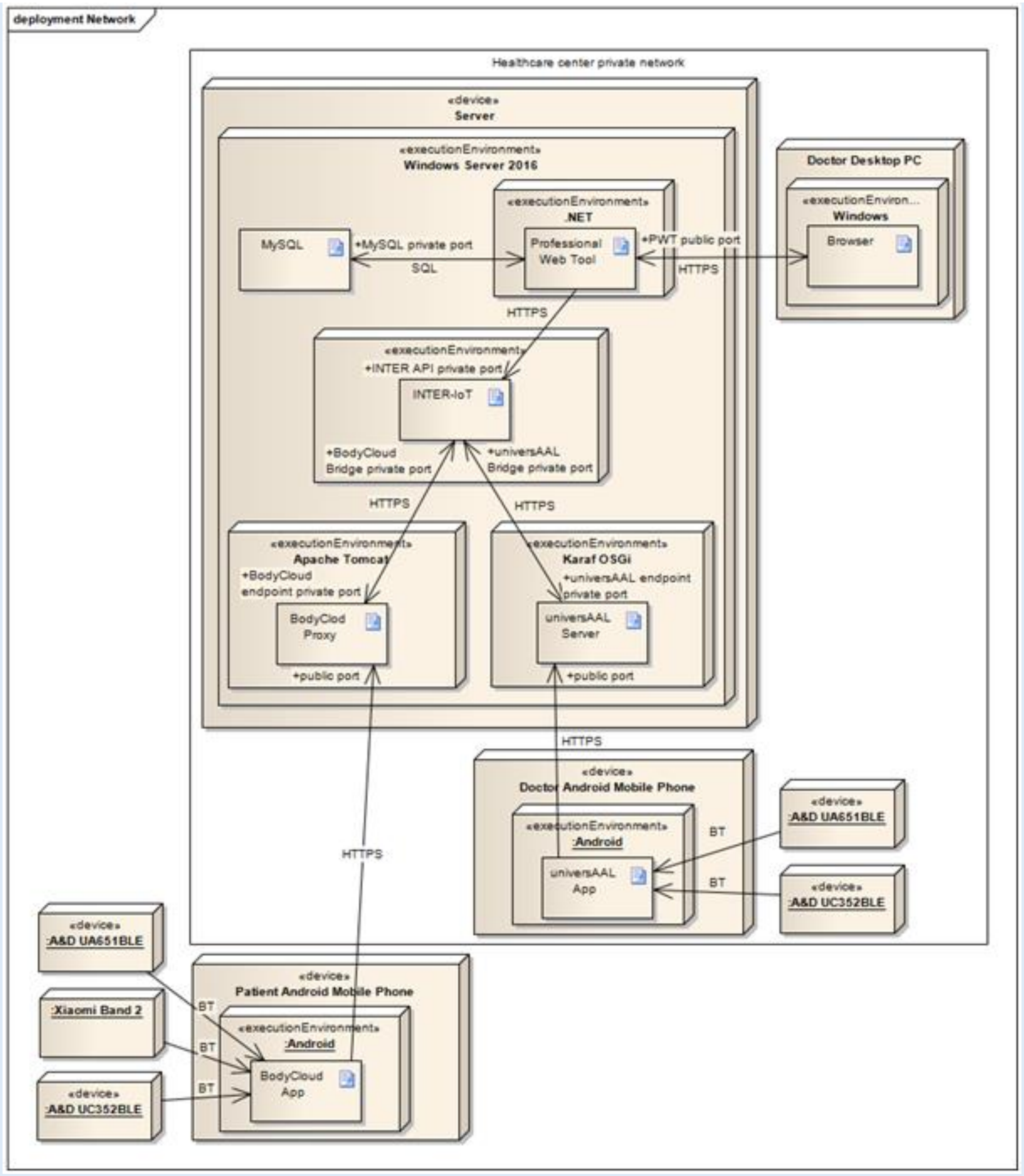

deployment Network
Healthcare center private network
«device» Server
«executionEnvironment» Windows Server 2016
«executionEnvironment» .NET
MySQL
+MySQL private port
SQL
Professional Web Tool
+PWT public port
HTTPS
Doctor Desktop PC
«executionEnviron... Windows
Browser
HTTPS
«executionEnvironment»
+INTER API private port
INTER-IoT
+BodyCloud Bridge private port
+universAAL Bridge private port
HTTPS
HTTPS
«executionEnvironment» Apache Tomcat
+BodyCloud endpoint private port
BodyClod Proxy
+public port
«executionEnvironment» Karaf OSGi
+universAAL endpoint private port
universAAL Server
+public port
HTTPS
«device» Doctor Android Mobile Phone
«executionEnvironment» :Android
universAAL App
BT
BT
«device» :A&D UA651BLE
«device» :A&D UC352BLE
HTTPS
«device» :A&D UA651BLE
:Xiaomi Band 2
«device» :A&D UC352BLE
«device» Patient Android Mobile Phone
«executionEnvironment» :Android
BodyCloud App
BT
BT
BT

Figure 21: Deployment model of INTER-Health

Implementation:

The hardware deployment consists of a central server machine hosted by the pilot partner where the main backend components of the participating platforms are installed: The Professional Web Tool native application, the BodyCloud and universAAL platforms backends, and the INTER-IoT components bridging them.

The other hardware in the model are Mobile devices that act as Gateways for the sensors that perform the actual measurements that are of interest to the Professional Web Tool. These run the software applications of BodyCloud and universAAL that connect to their respective backends in the server, and to the sensor devices to obtain their values. Notice that BodyCloud is used by the patients, who can roam freely and therefore is connected to the server through the internet, and

universAAL is used by doctors whose mobile devices will always connect through the local private network. The remaining hardware is the sensor devices. They themselves do not connect or are related to INTER-IoT but they rely on their connection to the universAAL and BodyCloud client applications in the mobile devices, to which they are connected through Bluetooth. Take into account as well that there is a “virtual” sensor that represents a questionnaire. This pattern is applicable to any other software-only source of information, that will be managed by the appropriate client application (in this case BodyCloud mobile app).	
Known uses (within the INTER-IoT): This pattern is followed exactly in INTER-Health, and, to some degree and with appropriate modifications, in INTER-Domain.	
Identified by: UPV-SABIEN	**Registration Date:** 24-11-2017

Design Pattern	
Pattern name: Integrated deployment in security-constrained environments	**Identifier:**16
Inspired by: • (e/m) Health use case patterns (Section:*“Use case specific patterns”*) • Security Patterns (Section:*“Security Patterns”*)	
Related patterns: • IoT Patterns: Design patterns for IoT Infrastructure (Section:*“IoT Patterns”*) • IoT Patterns: Design patterns for IoT Security (Section: *“IoT Patterns”*) • Enterprise integration patterns (Section:*“Enterprise Integration Patterns”*)	
Intent: Determine the security restrictions imposed by the INTER-Health pilot in particular. Design the integration and deployment of the overall pilot system to fit within these restrictions.	
Problem & Solution: In addition to the security requirements imposed by common sense, ethics, privacy concerns and legal framework of the INTER-IoT project, there are some additional security restrictions imposed by the scope of the INTER-Health pilot. These are determined by two main factors: the applicable laws in the location of the deployment (Torino, Italy) and the nature of the treated data (health). Because of these restrictions the design of the pilot deployment has to face some changes compared to what could have been used if these restrictions did not apply. It impacts the actual physical location of the deployed components, the technologies to use, who has access to them and how, and the connections between them.	
Applicability: This design pattern can be used as a starting point or inspiration for similar deployments in real-life scenarios that present similar constraints: Installing INTER-IoT for an application that deals with personal user health data. Although the particularities of INTER-Health applied only to the legal conditions of the pilot in Italy, it is foreseeable that a similar approach can be followed in other EU countries, especially once the GDPR becomes active.	
UML representation:	

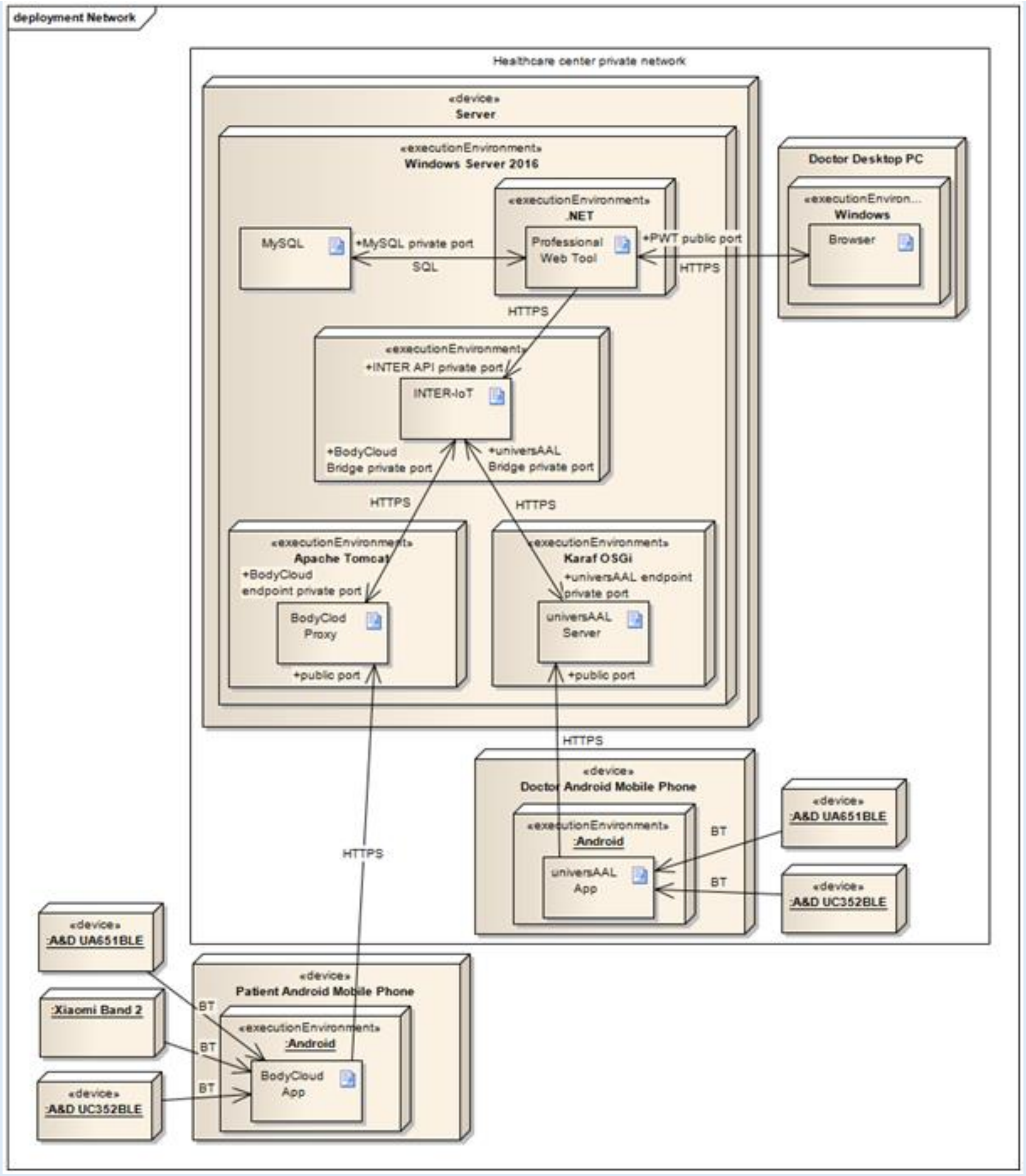

deployment Network
Healthcare center private network
«device» Server
«executionEnvironment» Windows Server 2016
«executionEnvironment» .NET
MySQL
+MySQL private port
SQL
Professional Web Tool
+PWT public port
HTTPS
Doctor Desktop PC
«executionEnviron... Windows
Browser
HTTPS
«executionEnvironment»
+INTER API private port
INTER-IoT
+BodyCloud Bridge private port
+universAAL Bridge private port
HTTPS
HTTPS
«executionEnvironment» Apache Tomcat
+BodyCloud endpoint private port
BodyClod Proxy
+public port
«executionEnvironment» Karaf OSGi
+universAAL endpoint private port
universAAL Server
+public port
HTTPS
«device» Doctor Android Mobile Phone
«executionEnvironment» :Android
universAAL App
BT
BT
«device» :A&D UA651BLE
«device» :A&D UC352BLE
HTTPS
«device» :A&D UA651BLE
:Xiaomi Band 2
«device» :A&D UC352BLE
«device» Patient Android Mobile Phone
«executionEnvironment» :Android
BodyCloud App
BT
BT
BT

Figure 22: Security restrictions in INTER-Health deployment model

Implementation:

An extension of the "UML representation", i.e. the textual description of realization and architecture (not a source code, like in GoF). The goal of this property is to clarify the diagram.

One of the main limitations imposed by the legal framework is that all components that store, to any degree, medical or health-related personal data, must be located within the physical premises of the responsible of the pilot. This means that all INTER-IoT components, native pilot application (the Professional Web Tool) and the existing IoT platform backends must all run on premises. This discards Cloud-based deployments and forces a local server-based approach.

The physical server, even though it is not necessarily located in the same physical location of the

users (doctors) accessing it, is located in the same private network, properly secured to give access only to authorized personnel. This "shields" the components running on it from security threats by leaving the protection of the data and communication between the software components therein up to the security measures taken to protect the private, corporate network. By securing as well network-based communications between components within that network, using HTTPS and dedicated certificates, it achieves an additional layer of security, preventing even authorized users, or potential attackers that infiltrate the network, from being able to intercept the data travelling through those connections. HTTPS is also used to secure the connections between the server and the additional devices: both the doctors' mobile phones, which in any case would connect from within the same private network, and the patients' mobile phones, which connect through the internet.	
Known uses (within the INTER-IoT): This pattern is followed exactly in INTER-Health, and, to some degree and with appropriate modifications, in INTER-Domain.	
Identified by: UPV-SABIEN	**Registration Date:** 24-09-2017

3.3.9 INTER-LogP

Design Pattern	
Pattern name: Geofencing pattern	**Identifier:**14
Inspired by: • *"Geo-fence"* Use Case Pattern (Section:*"Use case specific patterns"*)	
Related patterns:	
Intent: Nowadays, geofencing of objects is a common issue of IoT systems. Within the INTER-LogP, there is a need to detect the presence of objects inside a restricted area, e.g. truck in the port.	
Problem & Solution: The cooperation of platforms requires the triggering of certain processes/actions, as a result of detection a specific object within a restricted/defined area. For this purpose, four main entities must cooperate. First one is a Haulier Platform, which is publisher of the trucks' data (including positions). Second element is the INTER-MW, which is necessary to be the communication channel. Third entity is Port Platform, which contains two important modules: (1) list of defined restricted areas, and (2) process that detects the event of object presence within the defined area. This event contains a collection of information, such as time of event, object's details, type of event, etc. Fourth element is application which monitors the events. Of course many applications can monitor the events.	
Applicability: This pattern is used when it is necessary to detect the event of vehicle's access to an area in real time, in order to trigger a specific action, associated with the object.	
UML representation:	

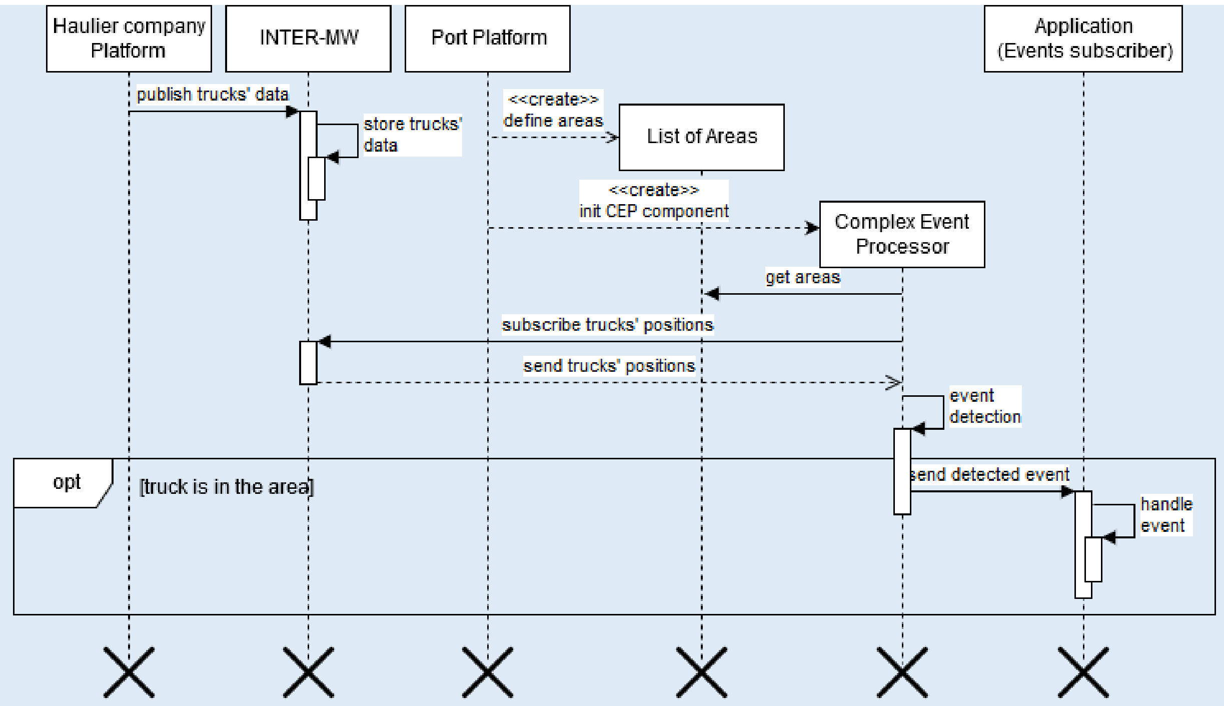


Figure 23: INTER-LogP Geofencing Pattern

Implementation:

The data of the trucks is provided by the IoT platform of the haulier company, so the port IoT platform needs to subscribe to this data through the INTER-MW. In the IoT platform in the port there are defined and stored all the areas where a truck can access and it has a Complex Event Processor, which checks the information in real time. The information is stored and received by the applications that may need it.

Known uses (within the INTER-IoT):

In the defined transport pilot, it is needed to know when a truck is near to the port (10 km distance), inside the port, and inside the terminal. For each of these cases it is needed to define an area and detect when the truck is inside. This information can be checked with other system, for instance, with the automatic identification system at the port access.

Identified by:	**Registration Date:**
VPF	16-11-2017

3.4 Analysis of INTER-IoT Design Patterns

Analysis of all INTER-IoT Design Patterns shows that every catalog, described in section: *"State of the art - research and analysis"* was helpful in the process of defining new design patterns. It proofs, that the analysis presented in sections *"Analysis of..."* (subsections of every described catalog) were accurate. It is hard to say what inspiration was the most important. For instance, the most often used catalogs were: *"Enterprise Integration Patterns"* and*"SOA Patterns"*, because they described most common issues, related with communication, cooperation, etc. Nevertheless, *"Security Patterns"*were used only twice because only two patterns described security issues, which are very important problem, for example because of ethical issues. Moreover, Object-oriented Pattern was used only once as an extension, but format of*“Object-oriented Patterns “Gang of Four””* catalog were crucial in defining INTER-IoT Layer Patterns format.

Very important is fact, that catalogs (described in SotA) were not the only inspiration in defining new design patterns. Other sources were common paradigms and technologies, i.e.: “Software - defined networking (SDN) orchestration”, “Network virtualization (NV)”, “Flow - based programming” and “Web API Design: Crafting Interfaces that Developers Love”.

Notice, that *“Ontology Patterns”* were not extended in new INTER-IoT Layer Patterns catalog. Of course, they were used in the project, but were not extended as a specific INTER-IoT patterns. Nevertheless, ontology patterns (mainly alignment patterns) are an integral part of translation patterns (i.e. “Alignment - based Translation Pattern” and “Translation with central ontology”).

4 Ethics

Ethics in INTER-IoT concerns issues related with General Data Protection Regulation (GDPR)[54]. It is EU regulation containing provisions on the protection of individuals with regard to the processing of personal data and free flow of personal data. The purpose of the regulation is to achieve full harmonization of substantive law within the EU and the free flow of personal data.

The GDPR defines two types of entities that have obligations: data controllers and data processors. Under the previous EU Data Protection Directive, only controllers could be held liable. With the GDPR, processors now also face serious data protection requirements and obligations. It is important, that a single entity (e.g. company) can be both both controller and a processor, depending on the exact type and usage of data. GDPR gives definitions of controllers and processors:

- A **controller** is an entity that determines the purposes, conditions, and means of the processing of personal data. For example, educational and research private and public institutions, healthcare services, or any business that manages the personal data of their employees and customers.
- A **data processor** is an entity which processes personal data on behalf of the controller, such as a cloud provider (for example, Software-as-a-Service companies like a CRM software).

Notice, that INTER-IoT acts both as a controller and processor. It is a data controller, because it handles the data related with processing of healthcare services and manages the personal data of patients and personnel. It is also a processor, because it allows any platform to process the mentioned healthcare data.

The main idea of GDPR is to protect personal data at every phase of data processing. INTER-IoT Layer Patterns contains solutions related with data processing (e.g. exchange data, translate data). Nevertheless, the most important issue that really affects the data are patterns related with security (*“CROSS Layer”*) and INTER-Health (*“INTER-Health”*). Using these patterns, it is possible to fulfill the main principles of the GDPR:

- Lawfulness, fairness, and transparency: personal data should be processed lawfully, fairly, and in a transparent manner.
- Limited purpose: personal data should be collected for specified, explicit, and legitimate purposes and not further processed in a manner that is incompatible with those purposes

- Data minimisation: personal data should be adequate, relevant, and limited to which it is necessary in relation to the purposes for which they are collected.
- Accuracy: personal data stored and managed should be accurate and, where necessary, kept up to date.
- Storage limitation: personal data should be kept in a form which permits the identification of data subjects for no longer than is necessary for the purposes for which the personal data is processed.
- Confidentiality and integrity: personal data should be processed in a manner that ensures appropriate security of the personal data, including protection against unauthorized or unlawful processing and against accidental loss, destruction or damage, using appropriate technical or organisational measures.

5 Conclusions

This document is part of *"WP5 Methodology for the Integration of IoT Platforms"* and is strictly related with deliverables *"D5.2 Full-fledged Methodology for IoT Platforms Integration (INTER-METH)"* and *"D5.3 CASE tool for Automated Application of INTER-METH Methodology"*. All deliverables considers the platform integrations process.

This document presented all the process of defining INTER-IoT Layer Patterns, starting with SotA analysis, via defining requirements based on WP3, to achieve final catalog. Every pattern were described, using new INTER-IoT Layer Patterns format. Description contains the solution and also the reason of necessity of creating brand new pattern, instead of using common approaches. Moreover, pattern's template presents its inspiration (mainly related with SotA analysis) and example of usage within the project.

It is worth to notice, that defined design patterns are strictly related with INTER-IoT development phase. They are very important part of documentation which allows to understand the INTER-IoT architecture and paradigms and also enables to prepare new components as part of the project. Described design patterns are also related with project ethics issues.

6 References

[1] *D3.1 Methods for Interoperability and Integration.*

[2] **F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal.** *Pattern-oriented Software Architecture: A System of Patterns.* New York, NY, USA : John Wiley & Sons, Inc., 1996.

[3] What are Patterns. [Online] http://hillside.net/patterns/50-patterns-library/patterns/222-design-pattern-definition.

[4] **E. Gamma, R. Helm, R. Johnson, J. Vlissides.** *Design Patterns: Elements of Reusable Object-oriented Software.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1955.

[5] **Kuchana, P.***Software Architecture Design Patterns in Java.* Auerbach Publications, 2004.

[6] **E. Gamma, R, Helm, R. Johnson, L. O'Brien.** *Design Patterns 15 Years Later: An Interview with Erich Gamma, Richard Helm, and Ralph Johnson.* Software Development & Management, 2009.

[7] **G. Hohpe, B. Woolf.** *Enterprise Integration Patterns.* Boston, MA, USA : Pearson Education Inc., 2004.

[8] Enterprise Integration Patterns. [Online] http://www.enterpriseintegrationpatterns.com/patterns/messaging/toc.html.

[9] Service Oriented Architecture Patterns. [Online] http://soapatterns.org/introduction.

[10] IoT-A Project Deliverable D2.3 - Orchestration of distributed IoT service interactions. [Online] http://www.iot-a.eu/public/public-documents/documents-1.

[11] **A. L. Lemos, F. Daniel, B. Benatallah.** Web Service Composition: A Survey of Techniques and Tools. *ACM Computing Surveys.* 2016, Tom 48.

[12] **Cancarini, P.** Coordination models and languages as software integrators. *ACM Computing Surveys.* 1996, Tom 28.

[13] The Reactive Manifesto. [Online] http://www.reactivemanifesto.org.

[14] **R. Kuhn, B. Hanafee, J. Allen.** *Reactive Design Patterns.* Manning Publications, 2016.

[15] **Y. Aridor, D. Lange.** Agent Design Patterns: Elements of Agent Application Design. *AGENTS '98 Proceedings of the Second International Conference on Autonomous Agents.* 1988.

[16] **D. Duego, M. Weiss, E. Kendall.** Reusable Patterns for Agent Coordination. [aut. książki] M. Klusch and R. Tolksdorf A. Omicini F. Zambonelli. *Coordination of Internet Agents: Models, Technologies, and Applications.* Springer, 2001.

[17] **Kendall, E.A.** Patterns of Intelligent and Mobile Agents. *Second Intl. Conference on Autonomous Agents.* 1998.

[18] Role Models: Patterns of Agent System Analysis and Design. *ACM, Agent Systems and Applications/Mobile Agents (ASA/MA-99).* 1999.

[19] **Tolksdorf, R.** Coordination patterns of mobile information agents. *Cooperative Information Agents II.* 1998, Volume: 1435.

[20] **Weiss, M.** Pattern-Driven Design of Agent Systems: Approach and Case. *Conference on Advanced Information Systems Engineering (CAiSE).* 2003, Volume: LNCS 2681.

[21] **Cancarini, P.** Coordination models and languages as software integrators. *ACM Computing Surveys.* 1996, Volume: 28.

[22] **M. Luck, P. McBurney, C. Preist.** A Manifesto for Agent Technology: Towards Next Generation Computing. *Autonomous Agents and Multi-Agent Systems.* 2004, Volume: 9.

[23] **Tveit, A.** A survey of Agent-Oriented Software Engineering. *First NTNU CSGS Conference.* 2001.

[24] **M. Kolp, P. Giorgini, J. Mylopoulos.** A Goal-Based Organizational Perspective on Multi-Agent Architectures. *Eighth Intl. Workshop on Agent.* 2001.

[25] **S. Staab, M. Erdmann, A. Maedche.** Engineering ontologies using semantic patterns. *Proceedings of the IJCAI-01 Workshop on E-Business & the Intelligent Web.* 2001.

[26] ONTOLOGY DESIGN PATTERNS (ODPs) PUBLIC CATALOG. [Online] http://odps.sourceforge.net/odp/html/index.html.

[27] **E. Blomqvist, K. Sandkuhl.** Patterns in Ontology Engineering: Classification of Ontology Patterns. *Proceedings of the Seventh International Conference on Enterprise Information Systems.* 2005.

[28] **Gangemi, A.** Ontology Design Patterns for Semantic Web Content. *International Semantic Web Conference.* 2005, Volume 3729.

[29] **F. Scharffe, O. Zamazal, D. Fensel.** Ontology alignment design patterns. *Knowledge and Information Systems.* 2014, Volume: 40.

[30] Ontology community portal. [Online] http://plasma.dimes.unical.it/events/I4T2016/PDF/SoheilQanbari.pdf.

[31] Expressive and Declarative Ontology Alignment Language. [Online] http://alignapi.gforge.inria.fr/edoal.html.

[32] Proposed Alignment ODPs. [Online] http://ontologydesignpatterns.org/wiki/Submissions:AlignmentODPs.

[33] Architectural Design Patterns for an IoT Platform. [Online] http://insights.mindstix.com/design-patterns-for-your-iot-architecture/.

[34] **Koster, M.** Design Patterns for an Internet Of Things A Design Pattern Framework for IoT Architecture. [Online] http://community.arm.com/groups/internet-of-things/blog/2014/05/27/design-patterns-for-an-internet-of-things.

[35] **Qanbari, S.***IoT Design Patterns: Computational Constructs to Design, Build and Engineer Edge Applications.* Proc. of IEEE IoTDI, 2015.

[36] IoT Overview - Provisioning. [Online] https://cloud.google.com/solutions/iot-overview#provisioning.

[37] Core Security Patterns. [Online] http://www.coresecuritypatterns.com/patterns.htm.

[38] The Open Group Security Design Patterns. [Online] http://www.opengroup.org/security/gsp.htm.

[39] Security Pattern Catalog. [Online] http://www.munawarhafiz.com/securitypatterncatalog/index.php.

[40] **N. Yoshioka, H. Washizaki, K. Maruyama.** A survey on security patterns. *Progress in Informatics.* 2008, Volume: 5.

[41] **Schumacher, M.***Security Patterns: Integrating Security And Systems Engineering.* John Wiley & Sons Inc., 2006.

[42] **Escribano, B.** Privacy and security in the Internet of Thing: challenge or opportunity? [Online] http://datonomy.eu/2014/11/28/privacy-and-security-in-the-internet-of-things-challenge-or-opportunity/.

[43] **Ajit Jha, Sunil M C.** Security considerations for Internet of Things. [Online] http://www.lnttechservices.com/sites/default/files/whitepapers/2017-07/whitepaper_security-considerations-for-internet-of-things.pdf.

[44] Security Guidance for the Early Adopters of the Internet of Things (IoT). [Online] https://downloads.cloudsecurityalliance.org/whitepapers/Security_Guidance_for_Early_Adopters_of_the_Internet_of_Things.pdf.

[45] **R. Rodrigo, P. Najera, J. Lopez.**Securing the Internet of Things. *Computer Journal.* 2011, Volume: 44.

[46] **Muji, M.** Best Practices in the Design and Development of Health Care Information Systems. *1st International Conference on Advancements of Medicine and Health Care through Technology, MediTech2007.* 2007.

[47] **Oláh, P.** A Database Design Pattern for Structuring Hierarchical Medical Data. *Acta Medica Marisiensis.* 2012, Volume: 58.

[48] **C. Ó Riain, M. Helfert.** An evaluation of data quality related. [Online] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.6924&rep=rep1&type=pdf.

[49] **Mauro, C.** Standardized Device Services – A Design Pattern for Service Oriented Integration of Medical Devices. *Proceedings of the 43rd Hawaii International Conference on System Sciences.* 2010.

[50] **Fowler, M.** Writing Software Patterns. [Online] 2006. https://martinfowler.com/articles/writingPatterns.html#CommonPatternForms.

[51] Pattern Language. [Online] http://www.patternlanguage.com/.

[52] Gang-of-four template. [Online] http://hillside.net/index.php/gang-of-four-template.

[53] Catalog of Patterns of Enterprise Application Architecture. [Online] https://martinfowler.com/eaaCatalog/.

[54] Software-defined networking (SDN) orchestration. [Online] https://www.sdxcentral.com/sdn/definitions/what-is-sdn-orchestration/.

[55] Network virtualization (NV). [Online] https://www.sdxcentral.com/sdn/network-virtualization/definitions/whats-network-virtualization/.

[56] FIWARE homepag. [Online] https://www.fiware.org/.

[57] Universal project homepage. [Online] http://universaal.sintef9013.com/entry/.

[58] Flow-based programming. [Online] http://www.jpaulmorrison.com/fbp/.

[59] Web API Design: Crafting Interfaces that Developers Love. [Online] https://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf.

[60] General Data Protection Regulation. [Online] https://www.eugdpr.org/.