# IoTDraw Modeling Framework Documentation

In this documentation, we introduce the IoT Draw. It is divided in two sections. The first present a Running Example that will be used to explain part of the framework. In the second part, the IoTDraw is presented.

# 1 Running Example

## 1.1 Introduction

In this section, we introduce a Smart City IoT system to serve as a running example throughout this work. The system is designed based on a widely used IoT infrastructure model, which is introduced in the following section.

## 1.2 IoT infrastructure

As applied by several players (e.g., Google [22] and Harbinger [13]) and identified by the literature (e.g., e.g., [11, 27]) a common end-to-end infrastructure for IoT encompasses typically three tiers, namely, *cloud*, *fog*, and *device*. The former consists of computers located in the cloud providing sets of resources (e.g., computing, storage), which can be rapidly provisioned and released on-demand. The fog tier encompasses computers that exploit capabilities at the edge of the Internet, thus, providing near-devices computing/storage resources. Edge nodes can also act as bridges to connect the devices to the cloud and providing remote access to devices through APIs offered by specific components [7]. In turn, the device tier refers to electronic platforms equipped with sensors and actuators that are attached to physical entities to enhance them with sensing, actuating, communication, and computing capabilities.

## 1.3 Padova Smart City

Our running example is based on a real smart city project deployed in the city of Padova (Italy) that adopts the infrastructure presented above. The "Padova Smarty City" (PSC) [31] was conceived to enhance the quality of the services offered to citizens while reducing the operational costs of the public administration. The project is the result of the collaboration between the municipality of Padova, the University of Padova, and the Patavina Tech – a software house specialized in the development of innovative IoT solutions. The infrastructure of PSC is depicted in Figure 1 and explained in the following. The complete explanation, including the reasoning behind all technical decisions, can be found in [31].

The PSC encompasses battery-powered devices equipped with a water level sensor and placed on the base of the streetlight poles around the downtown. The devices aim at supporting the water level monitoring in public areas across the city of Padova, in which flooding is a recurring phenomenon (almost daily in some seasons [17]). PSC also have alarm actuators attached to the streetlight poles, which can be used to alert citizens in case of flooding. The alarms, which requires a much higher power supply, are powered by the grid. Both sensing and actuating devices have a CC2420 transceiver that implements the IEEE 802.15.4 standard [14], providing them wireless communication.

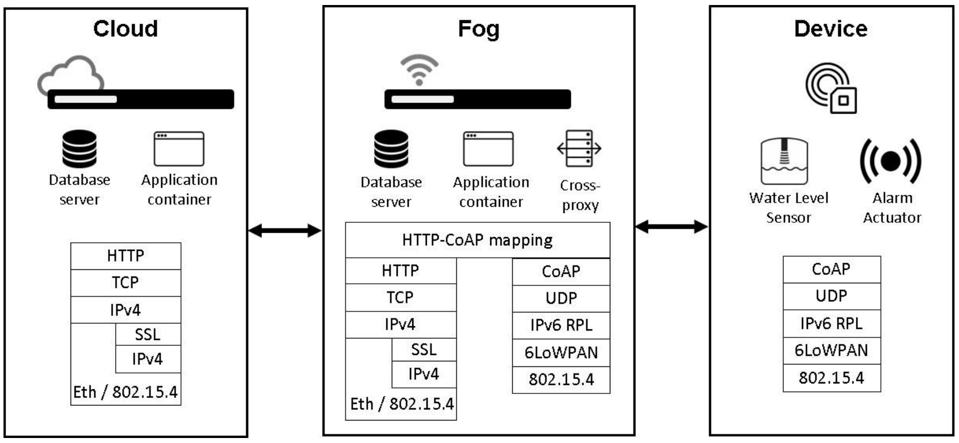


Figure 1. The infrastructure of “Padova Smart City” (based on [31])

The fog and cloud nodes provide database servers and application containers, allowing data storage and execution environment for software components, although, in the cloud, such capabilities can be provisioned and released on-demand. The fog nodes also provide protocol translation and functional mapping between unconstrained protocols and their constrained counterparts, which are used by the devices (e.g., CoAP [29], 6LoWPAN [19]), through a *cross-proxy*. The fog node performs protocol translation and forwards the requests directly to the required device by using 6LoWPAN and the RPL routing protocol [30]. Such a protocol is used in the communication between fog nodes and devices. In our example, we consider the following infrastructure: two cloud nodes: one located in *Michigan* and the second in *Stuttgart;* three fog nodes (*fog_1*, *fog_2*, and *fog_3*): located in the city of Padova, and; two types of devices, that is, *water sensor* (two instances) and *alarm* (two instances).

## 1.4 Flood Warning application

On the PSC infrastructure, we want to design a Flooding Warning (FW) IoT application. It consists of a *periodic application*[1] that aims at sensing the water level at regular time intervals and send alerts to the citizens, in case of flooding, to enable the fast evacuation near the flooding area. The FW application also aims at collecting and storing water level data in a database, and further perform analytics aiming at examining different flooding scenarios, which will be helpful for urban planning.

The SOA style is suitable for architecting the FW application due to two main reasons. Firstly, it allows providing the capabilities of the heterogeneous entities composing the application through well-defined interfaces, thus, furnishing the interoperability between the different hardware platforms and software components. Secondly, the water level sensor will also be exposed as a service to be requested by external applications based on the *request-response* model, following the Sensing as a Service ($S^2$aaS) approach [23]. In $S^2$aaS, the sensing data is available for external users and can be provided on demand based on a *pay as you go* model, in the same way as traditional services. In short, when a user needs data of a certain object or environment, he/she requests it to the device's service, which answers with the required data. In the context of FW application, by providing water level sensor as a service, public/private organizations or individual developers can create their own flooding monitoring applications. For example, the University of Padova may create an application aiming to measure the time it takes for an area to be flooded after a storm starts. On the other hand, Patavina Tech may create a mobile app that informs the citizens about flooded areas.

### 1.4.1 QoS Requirements

Constrained batteries power the devices providing the water level data; thus, a Quality of Service (QoS) requirement that must be analyzed in the FW applications is the operational *lifetime* of devices providing the required services. Based on [1], we consider the lifetime as the time spanning from the instant when the device starts functioning until it runs out of energy, making the service provided by such a device unavailable. In this sense, the

[1] As presented by Basha and colleagues [3], flooding detection can also be modeled as an *event-based application* by using pressure sensors, in which the event that triggers the alarm is when the water pressure exceeds a certain threshold. However, in the example, we model the application as periodic to allow a second (redundant) monitoring system.

ability of the service, platform, or component to perform its required function over an agreed period may be impacted. Thus, FW application shall also address *availability* requirements [4].

# 2 IoTDraw Modeling Framework

## 2.1 Introduction

In this chapter, we introduce our proposed modeling framework. The IoTDraw is structured upon the elements depicted in Figure 2 and described in the following.

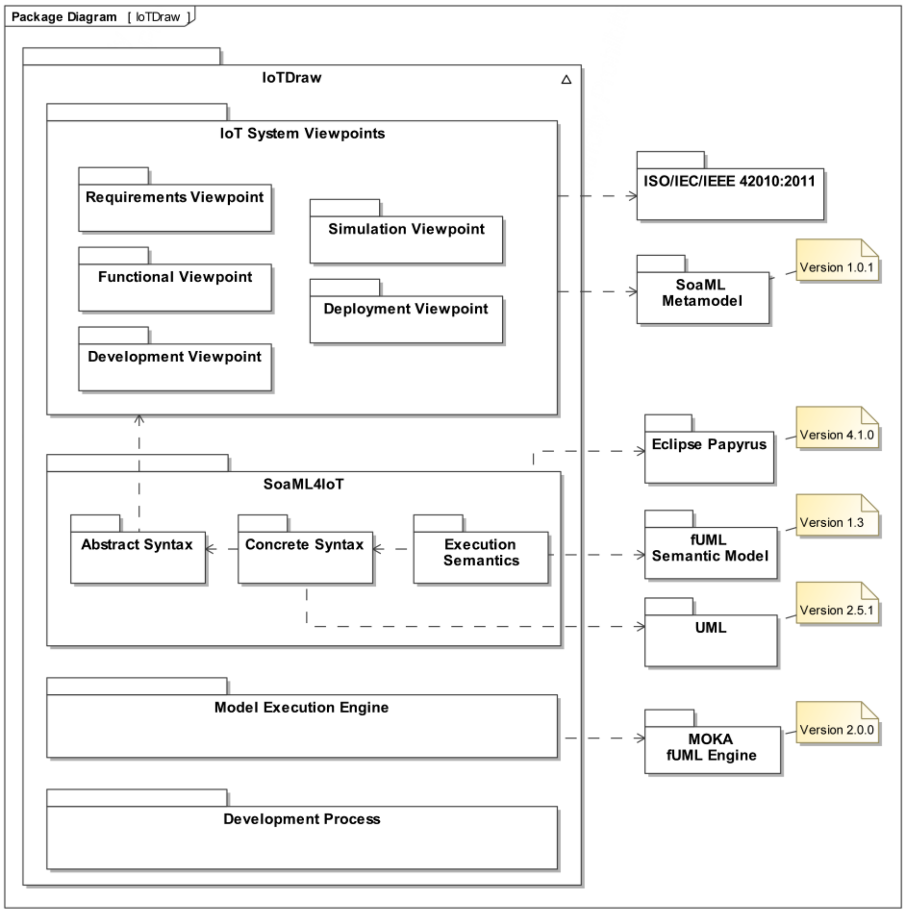


Figure 2. The IoTDraw Modeling Framework

The framework is composed of four main cornerstones, namely, the *IoT System Viewpoints*, the *SoaML4IoT* modeling language, the *Model Execution Engine*, and the *Development Process*. The viewpoints were conceived based on the ISO/IEC/IEEE 42010 standard [16]. The modeling language was designed by following the methodology presented by Brambilla and colleagues [6]. The Model Execution Engine, which implements the SoaML4IoT execution semantics, were developed based on the process of formalizing the execution semantics of UML profiles proposed by Tatibouët and colleagues [28]. Finally, the Development process is based on the activities of service-oriented design proposed by Erl [10].

## 2.2 IoT System Viewpoints

Based on Rozanski and Woods [25], we identify five viewpoints, each one related to a specific aspect of the IoT system's architecture, namely, *Functional*, *Development*, *Deployment*, *Requirements*, and *Simulation*. The Functional Viewpoint deals with the main stakeholder concern related to the IoT system's components and their responsibilities, interfaces, and primary interactions. The Development Viewpoint is related to the software architecture that supports the development process. The Deployment Viewpoint is related to the runtime platforms in which the software components are deployed, as well as the network connections between them. The Requirements Viewpoint deals with the stakeholders' requirements that aim to be fulfilled by the IoT applications. Finally, the Simulation Viewpoint is related to aspects regarding the behavior of the system at runtime. In the following, we detail each viewpoint.

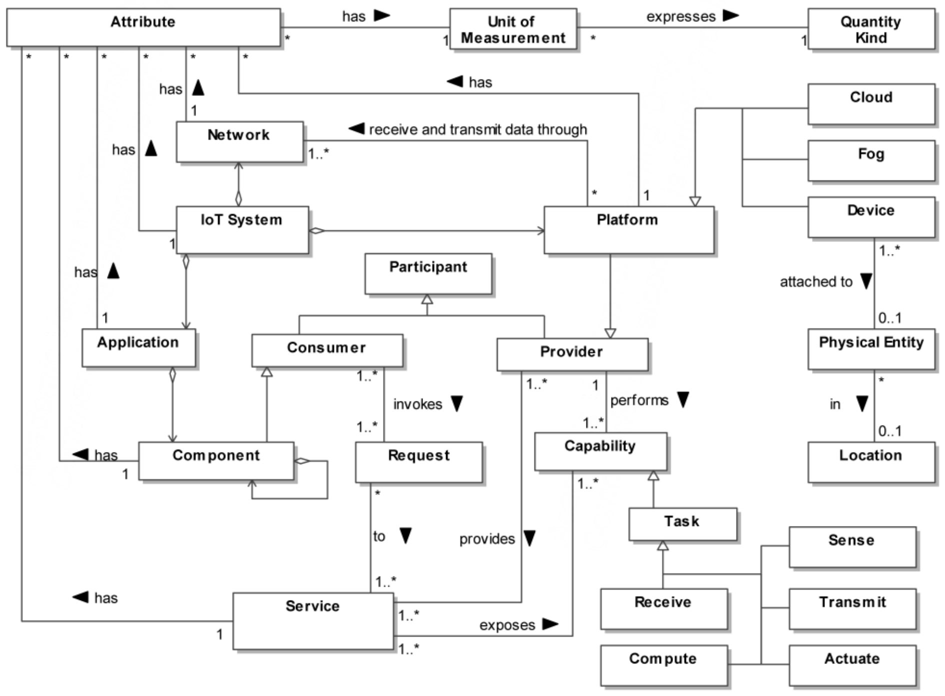
### 2.2.1 Functional Viewpoint

Table 1 gives an overview of the functional viewpoint.

Table 1. Functional Viewpoint for IoT Systems

| **Name** | Functional Viewpoint |
|---|---|
| **Overview** | The architecture viewpoint deals with the main stakeholder concern related to the IoT system's components and their responsibilities, interfaces, and primary interactions. |
| **Stakeholders** | *Software architect* who design and describe the IoT system architecture.<br>*Platform specialist* who specify the platforms composing the IoT system.<br>*Owner* who define the IoT applications' purpose.<br>*Actors* who access and contribute to the IoT applications. |
| **Concerns** | *C1- Structure*: what is the structure of the IoT system?<br>*C2- Components*: what kind of functional components structure the IoT system? |
| **Model Kind** | *IoT System Domain Model (deals with both concerns C1 and C2)*: a model that describes the components of the IoT system, their responsibilities and possible interactions. |

**Metamodel**



The IoT system domain model was conceived as an extension of the SoaML metamodel. The concepts related to the IoT domain, which extend the SoaML metamodel, were elicited from proven domain models for IoT, namely, the IoT Architectural Reference Model (IoT-ARM) [5], the WSO2 IoT Reference Architecture [18], and the IEEE Standard for an Architectural Framework for Internet of Things (P2413) [15].

An *IoT System* is a cyber-physical system composed of *Platforms*, *Applications*, and *Networks*. A platform refers to computer nodes, which can be of type *Cloud*, *Fog*, or *Device*. Nodes located in the cloud provide a set of resources (i.e., processing, storage), which can be provisioned and paid on demand. Fog nodes are computers that act as near- devices computing/storage resources or as bridges to connect the devices to the cloud. A device is a computer attached to *physical entities*, which can be anything of the real world from objects and cars to animals and human beings. A physical entity lies in a specific geographic *location*.

The devices enhance physical entities with sensing, actuating, communication, and computing capabilities. *Sense* tasks aim to collect data from physical entities (e.g., car speed, human body's temperature) or the environment it is inserted into (e.g., room's temperature or lighting level); *actuate* tasks can affect the physical realm (e.g., turn on/off a heater, sound alarm); *transmit* and *receive* tasks regard the communication of the device; finally, the *compute* task refers to the processing capability of the device. The platforms communicate with each other and eventually with the Internet by using wired or, more often, wireless networks. A *Network* refers to the set of nodes and links providing the communication path by with the platforms receive and transmit data. An *Application* refers to a set of specialized algorithms that request services and process data aiming at fulfilling user-defined requirements. An application is composed of one or more *Components*. Note that the metamodel represents the IoT applications as a set of components instead of a monolithic structure.

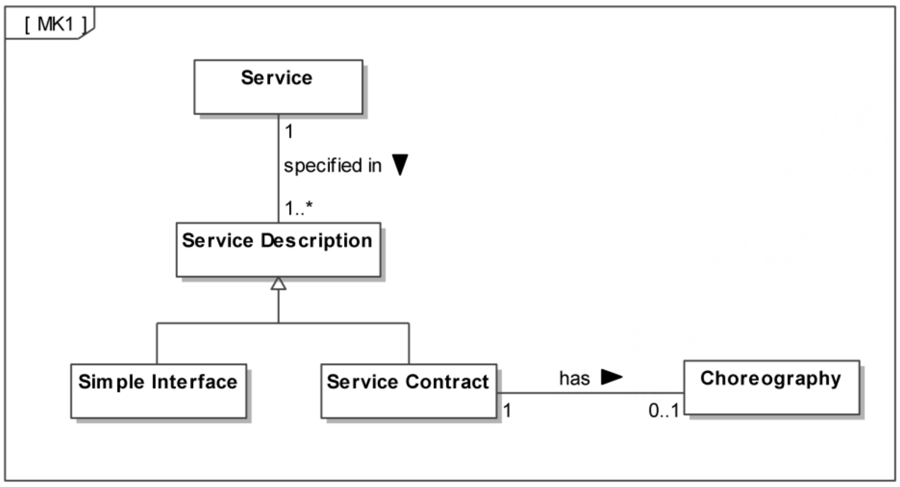
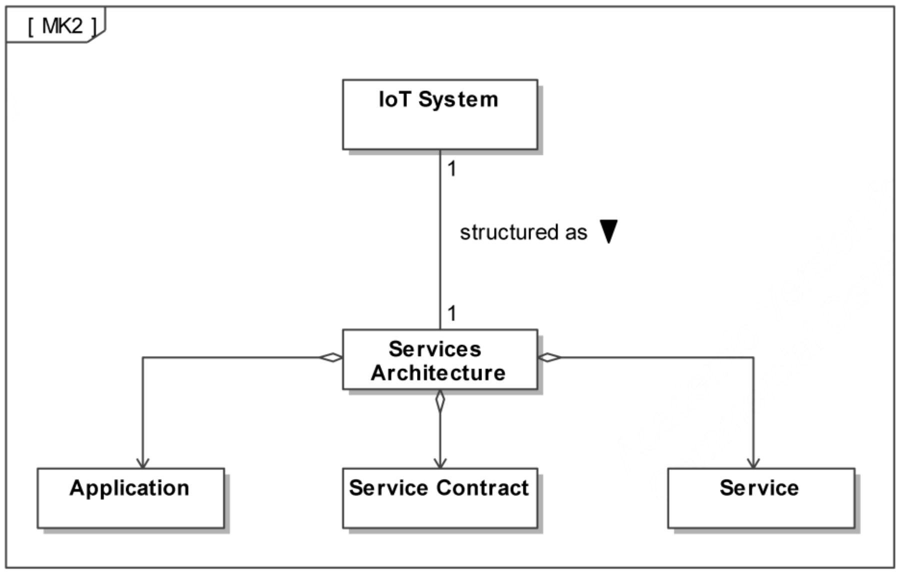
In SoaML4IoT metamodel, the capabilities performed by the providers are specialized as tasks (sense, actuate, transmit, receive, and compute). In turn, the providers are specialized as platforms. Components act as consumers, requesting the required tasks exposed through the devices' services. In other words, devices attached to physical entities expose their provided tasks through services, which are consumed by application's components aiming at fulfilling the stakeholders' requirements.

An essential concept of the metamodel is the *Attribute*. In the MOF standard, attributes are defined as *properties of a class*. In the IoT system domain model, we specialize such a concept. Besides representing properties of classes, an attribute is refined as *a property that may be useful for the architectural decision-making process*. Therefore, an attribute is a property that may be predicted or estimated at design-time to help to answer a design question. By specializing the concept of *attribute* as defined in the MOF standard, we aim at supporting the simplest SoaML4IoT model that answers a given design question. The reasoning behind this objective is to avoid creating too-detailed models, which are hard to manage [24]. Thus, the modelers would specify only attributes that are strictly necessary to answer design questions. The elements of the metamodel that have such specialized attribute are *IoT System*, *Platform*, *Network*, *Application*, *Component*, and *Service*, which are the main structural elements of a SOA-based IoT solution. The specialized attribute is specified in the metamodel as a first-class citizen. Thus, the concrete representation of these elements is realized in the UML profile as tagged values. An attribute may have a *Unit of Measurement* (e.g., watt) expressing a *Quantity Kind* (e.g., electric power).

### 2.2.2 Development Viewpoint

Table 2 gives an overview of the development viewpoint.

Table 2. Development Viewpoint for IoT Systems

| | |
|---|---|
| **Name** | Development Viewpoint |
| **Overview** | The architecture viewpoint deals with the main stakeholder concern related to the software architecture that supports the development process. |
| **Stakeholders** | *Application developer* who define the components' structure and behavior. *Software architect* who design and describe the IoT system architecture. |
| **Concerns** | *C1 - Services contracts*: what are the service contracts, providers and consumers? *C2 - Services architecture*: how the components of the IoT system are structured? |
| **Model Kinds** | *MK1 - Service contracts model (deals with concern C1)*: a model that describes the contracts, consumers, and providers. *MK2 - Architectural Model (deals with concern C2)*: a model that describes the architectural structure of components that are part of the IoT system. |
| **Metamodel** |   [ MK1 ]<br>Service<br>1<br>specified in ▼<br>1..*<br>Service Description<br>Simple Interface<br>Service Contract<br>has ▶<br>1<br>0..1<br>Choreography   [ MK2 ]<br>IoT System<br>1<br>structured as ▼<br>1<br>Services Architecture<br>Application<br>Service Contract<br>Service |

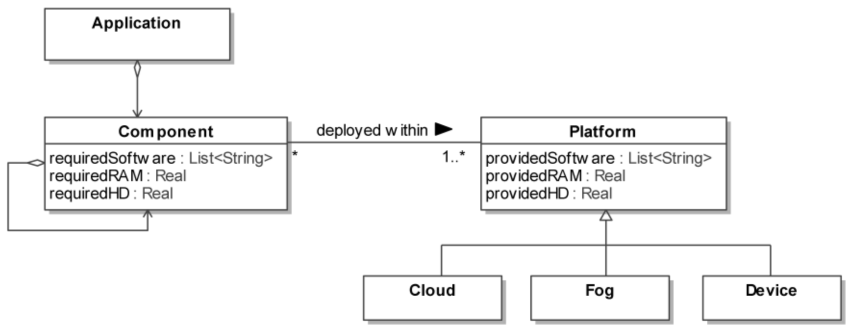
The IoT System is structured as a *Services Architecture*. Based on SoaML, such structure is composed of *Applications*, *Services*, and *Service Contracts*. Recall that an application is composed of one or more components, which are software units that act as service consumers. In turn, the services provide the interfaces for the capabilities (i.e., sense, actuate, transmit, and receive) performed by the providers (i.e., platforms – cloud, fog, and device).

A service is specified in a *service description*, that is, how the participants interact to provide or use a service. A service description can be specified as Service Interface or Service Contract. The service interface specifies a bi-directional service, which has call-backs from the provider to the consumer. In turn, the service contract specifies the roles each participant plays in the service – provider or consumer – and the choreography of the service, that is, what information is sent between the provider and consumer and in what order.

### 2.2.3 Deployment Viewpoint

Table 3 gives an overview of the deployment viewpoint.

Table 3. Deployment Viewpoint for IoT Systems

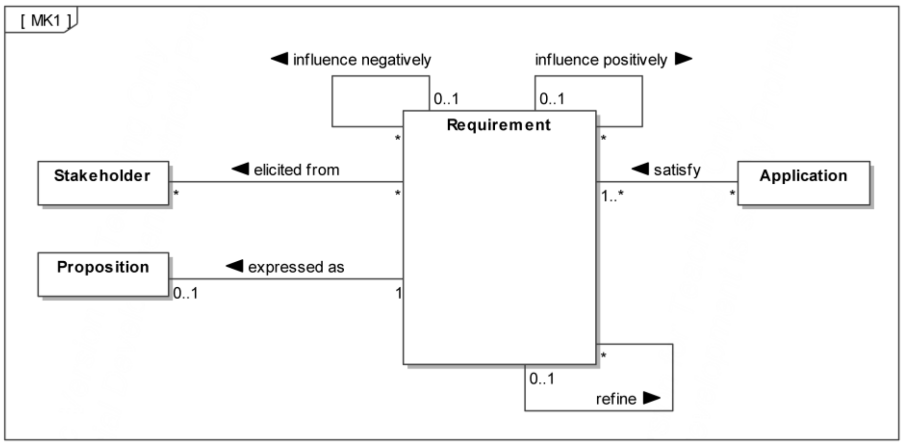
| **Name** | Deployment Viewpoint |
|---|---|
| **Overview** | The architecture viewpoint deals with the main stakeholder concern related to the runtime platforms in which the software components are deployed, as well as the network connections between them. |
| **Stakeholders** | *Deployment manager* who specifies the deployment configuration of multiple components that are part of the IoT applications. |
| **Concerns** | *Deployment configuration*: where the components of the IoT applications are deployed? |
| **Model Kind** | *Deployment Model*: a model that describes where the architectural components that are part of the IoT system are deployed. |
| **Metamodel** |   Application Component requiredSoftware : List<String> requiredRAM : Real requiredHD : Real deployed within * 1..* Platform providedSoftware : List<String> providedRAM : Real providedHD : Real Cloud Fog Device |

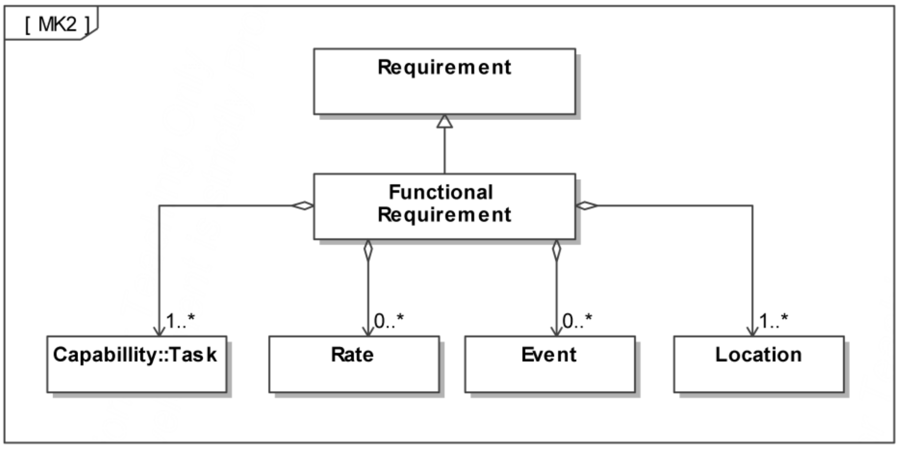
In our approach, the components that comprise the applications can be deployed within any cloud, fog, or device platforms. Components have software and hardware requirements. Software requirements refer to platforms that provide the execution environments with the libraries used by the component. For example, whether a component was developed with C#, its required software is the .NET framework. Considering the hardware requirements, it refers to the minimum memory (random and secondary) to run the respective component. When specifying the deployment configurations, both software and hardware requirements must be addressed by the platform in which the component will be deployed.

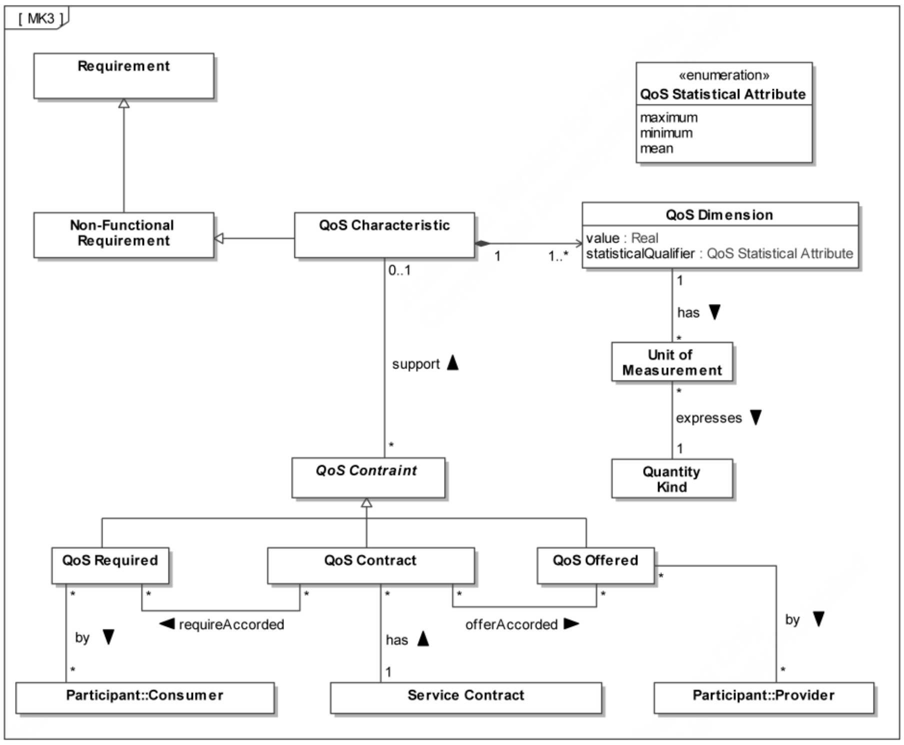
### 2.2.4 Requirements Viewpoint

Table 4 gives an overview of the requirements viewpoint.

Table 4. Requirements Viewpoint for IoT Systems

| | |
|---|---|
| **Name** | Requirements Viewpoint |
| **Overview** | The architecture viewpoint deals with the main stakeholder concern related to requirements that aim to be fulfilled by the IoT applications. |
| **Stakeholders** | *Requirements Engineer* who elicit the application's requirements.<br>*Owner* who define the IoT applications' purpose.<br>*Actors* who access and contribute to the IoT applications. |
| **Concerns** | *C1- Functional Requirements*: what the system must do, and how it must behave or react to runtime stimuli?<br>*C2- Non-Functional Requirements*: what are the qualifications or constraints of the functional requirements or the overall system? |
| **Model Kinds** | *MK1 – General Requirements*: a model that describes the elements related to requirements engineering in general.<br>*MK2 – Functional Requirements (deals with concern C1)*: a model that describes the functional requirements.<br>*MK3 – Non-Functional Requirements (deals with concern C2)*: a model that describes the Quality of Service (QoS) requirements. |
| **Metamodels** |  [ MK1 ]<br>◀ influence negatively<br>influence positively ▶<br>0..1<br>0..1<br>Requirement<br>Stakeholder<br>◀ elicited from<br>◀ satisfy<br>Application<br>1..*<br>Proposition<br>◀ expressed as<br>0..1<br>1<br>0..1<br>refine ▶<br>* |





Starting the explanation by the elements of *MK1* metamodel, *Stakeholders* are parties being influenced by or having an influence on the development of an IoT application, for example, domain specialists (e.g. structural engineers, biologists), customers, device experts, requirements engineers, and architects. The stakeholders have *requirements* regarding the application. A requirement refers to any desire of the stakeholders that is aimed to be satisfied by the IoT *application* under design.

Requirements can be expressed in terms of *propositions*. The concept of proposition is the same as in classical propositional logic – a statement that gets either *True* of *False* as its value. For example, the requirement (*r1*) “monitor the temperature” is True whether the application addresses it, or False, otherwise. A requirement may *influence* (positively or negatively) on or *refine* other requirements. For example, the requirement (*r2*) "the devices shall be deployed inside the rooms" may influence the requirement (*r3*) "the transmission power of the devices shall be *n* [unit of measurement]" (once the walls may obstruct the signal, the transmission power should be high).

With respect to MK2 metamodel, the functional requirements state what the system must do in terms of the *Task*, *Rate*, *Event*, or *Location*. The task refers to the desired capability, e.g., sense or actuate, that is required by the application. The rate expresses the number of requests per time span (periodic applications). On the other hand, the event refers to the event of interest that triggers the required task, i.e., event-driven applications. Finally, the location is the geographic region on which the tasks are meant to be performed.

The MK3 metamodel was conceived based on the QFTP, the UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification [20]. Non-functional requirements refer to attributes that address issues of Quality of Service (QoS) for the IoT applications. A *QoS Characteristic* represents non-functional aspects of system elements, such as latency, throughput or availability. *QoS Dimensions* are metrics for the quantification of QoS Characteristics. The QoS Characteristic can be quantified in different ways. For example, we can quantify the latency of a system function as the end-to-end delay of that function or the mean time of all executions. Furthermore, a QoS Characteristic can require more than one type of value for its quantification. For example, the availability may be measured by using two arguments, the mean-time-to-repair (MTTR) and the mean-time-to-failure (MTTF), and the availability is given by the analytic model MTTF / (MTTF + MTTR).

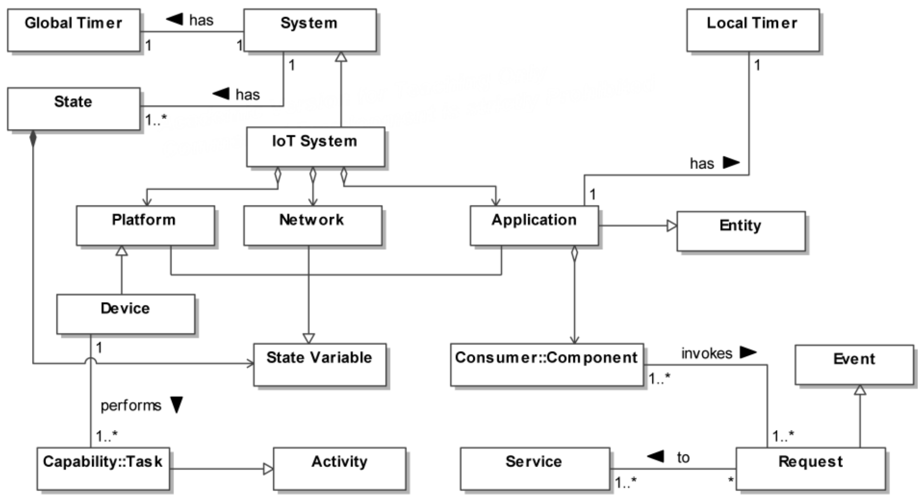
The QoS Constraint is an abstract class that is realized by the *QoS Required*, *QoS Offered*, and *QoS Contract*. The former refers to the QoS that is required by the participant consumer. The second represents the QoS that is offered by the participant provider. Both QoS required and offered are modeled as a QoS contract, which is related to the service contract, introduced earlier.

### 2.2.5 Simulation Viewpoint

Table 5 gives an overview of the deployment viewpoint.

Table 5. Simulation Viewpoint for IoT Systems

| **Name** | Simulation Viewpoint |
|---|---|

| | |
|---|---|
| **Overview** | The architecture viewpoint deals with the main stakeholder concern related to the behavior of the system at runtime. |
| **Stakeholders** | *Software architect* who design and describe the IoT system architecture; *Application developer* who define the components' structure and behavior *Deployment manager* who specifies the deployment configuration of multiple components. |
| **Concerns** | *C1- Properties' prediction*: what is the value of a given property at instant *t*, at runtime? *C2- Design decision*: what is the impact of a given design decision on the system behavior? |
| **Model Kind** | *Simulation Model (deals with both concerns C1 and C2)*: a model that describes the concepts related to the IoT system simulation. |
| **Metamodel** |   Global Timer ◀ has 1 1 System 1 Local Timer 1 State ◀ has 1..* IoT System has ▶ 1 Platform Network Application Entity Device State Variable Consumer::Component invokes ▶ 1..* Event 1 performs ▼ 1..* 1..* Capability::Task Activity Service ◀ to 1..* * Request |

Computational simulation is a mature discipline with well-defined concepts. Aiming to align IoTDraw with the literature, we extend the taxonomy presented by Banks and others [2]. According to this taxonomy, a *system*, which refers to the system-of-interest of the simulation, has a *global timer* that specifies the start and end time of the simulation and keeps track of the simulation time. The *state* of a system refers to the collection of *state variables* necessary to describe the system at any time. An *entity* is an object of interest in the system with one or more *attributes*. *Activity* refers to an action performed within a time period of specified length. Moreover, an *event* is defined as an occurrence, which triggers one or more activities that might change the state of the system.

In our model, the IoT system is classified as a discrete system, in which the state variables change only at a discrete set of point in time. The event that is responsible for triggering activities that might change the state of the system is the request. Recall that in service-based IoT systems, a component (which comprises IoT applications) requests the required capabilities to the devices' services. The devices, in turn, perform tasks aiming at addressing the request. Thus, the tasks are the simulation activities in our model. The IoT system state, i.e., the composition of state variables describing the system at a given simulation time, regards platforms, networks, and applications. Since the simulation aims at verifying the conformance of applications to their requirements, the applications are the objects of interest, and the attribute we are interested in when simulating the system is the (fulfillment of) IoT requirements.

After introducing the IoT system viewpoints, in the next chapter, we present the second component of IoTDraw, that is, the SoaML4IoT modeling language.

## 2.3 SoaML4IoT

The SoaML4IoT is our proposed modeling language to be used in the specification of IoT systems. In the following, we introduce the concrete syntax and the execution semantic. The abstract syntax of SoaML4IoT (i.e., its metamodel) consists of the metamodels of the viewpoints introduced above.

### 2.3.1 Concrete syntax

The concrete syntax of SoaML4IoT (Figure 3) consists of an UML profile extending the UML elements Port, Class, Dependency, Interface, Association, and DataType. Such extensions realize the concepts with the same name from the SoaML4IoT metamodel.

Instead of specializing the stereotypes of SoaML, we decided to extend the elements from the UML with concepts of our metamodel. Note that we extend the same UML elements as the SoaML. For example, the concept *Consumer* from the SoaML metamodel extends the UML element Class. Thus, the concept *Application* from SoaML4IoT metamodel, which specializes consumer, extends the same UML element Class. The reasoning behind this design decision is that fUML engines do not handle cases were multiple stereotypes are applied over the same modeling construct [28]. Thus, if we specialize the SoaML profile, the same class could be applied with multiple stereotypes (e.g., provider and device), hampering the model execution. Another adaptation we made to allow the execution regards the *Service Contract* and *IoT System* elements (this later specializing the concept of *Services Architecture*). In SoaML, both service contract and service architecture extend the UML element *Collaboration*, which is not part of the fUML semantic model. Thus, we extended the UML element Class, which also allows the modeling of collaboration through composite structures.

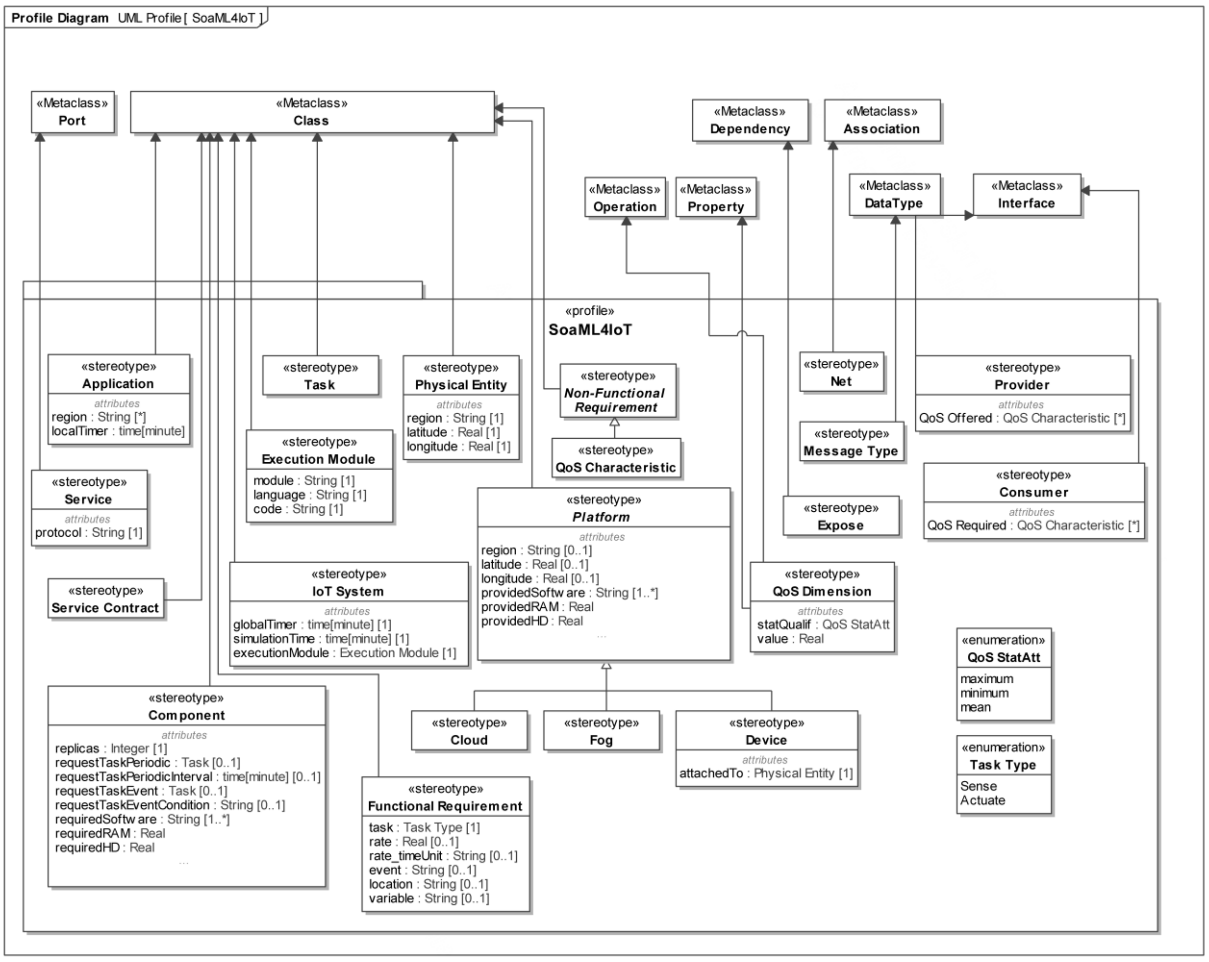


Figure 3. The SoaML4IoT UML Profile

As introduced in the SoaML4IoT metamodel (subsection 5.1), the *Attribute* concept is realized in the concrete syntax as tagged values by providing information that is necessary for the architectural decision-making process. In the presented version of SoaML4IoT, we provide essential properties to allow representing periodic and event-based IoT applications.

The *Service* has the attribute *protocol*, aiming to specify its application-level protocol (e.g., HTTP, CoAP). The *Application* also has the attributes *region* and *localTimer*. The former aims to specify the geographic location of the target environment (as requested by the application), from which the sensing data will be collected and/or actuation tasks will be performed upon. The second attribute is used by the execution semantics as a counter, considering the periodic applications. Thus, for example, when the user specifies that the application requests a given service every 2 minutes, the attribute *localTimer* is used to control the time elapsed since the last request (such behavior is specified in the execution semantics, introduced in the section).

The *Component* stereotype has the attributes for specifying periodic and/or event-based applications. Recall that an application is composed of one or more components. Thus, although the periodic or event-based model refers to the application, the components are the responsible for actually performing that required requests. The attribute *requestTaskPeriodic* models the task to be requested at regular time intervals, which is specified in the attribute *requestTaskPeriodicInterval*. In the simulation, such an attribute is used in conjunction with localtimer to trigger the request. On the other hand, the attribute *requestTaskEvent* defines the required task that aims to be requested when an event occurs. Such an event is specified in SoaML4IoT textually, in the attribute *requestTaskEventCondition*.

The *Physical Entity* stereotype allows specifying its geographic *region* through the *latitude*, *longitude* attributes. The *Platform* (abstract) also allows specifying its geographic *region* through the *latitude*, *longitude* attributes. The Device stereotype has only *attachedTo* attribute, which models the physical entity equipped with the device.

The stereotypes *QoS Characteristic* and *QoS Dimension* represents the elements of the same name in the MK3 metamodel (Requirements Viewpoint – Section 2.2.4). Also, the enumeration *QoS StatAtt* represents the concept QoS Statistical Attribute of such a metamodel.

The *IoT System* stereotype has three attributes, *globalTimer*, *simulationTime*, and *executionModule*. All these attributes are used in the model execution. The attribute *globalTimer* represents the simulation time in minutes, which is updated until it achieves the required simulation time (*simulationTime* attribute). At the beginning of the simulation, it is initialized with 0 (zero), and it is incremented until achieving the limit defined by the user or the user stops the simulation. The *Execution Module*, which is the third attribute of the IoT system, is the extension point of the execution semantics. It has three attributes, namely, *module*, *language*, and *code*. The first specifies the module name. The second defines the language used to implement the module. The third attribute is used to indicate the location of the code file to be executed in the simulation. The details about the extension of the execution semantics of SoaML4IoT through the execution module is explained in the following.

Finally, the profile has two stereotypes used to specify the application's requirements, namely, *QoS Characteristic*, which specialize the *Non-Functional Requirement* (abstract) and *Functional Requirement*. Both elements refer to the concepts of the same name of the Requirement Viewpoint (Section 2.2.4). The functional requirement is specified with the following properties: the *task* type that is required (i.e., sense, actuate); the *rate* in which such a task is performed (e.g., every 2 minutes); the *rate_timeUnit*, referring to the time unit (e.g., seconds, minutes) of the rate; the *event* that is interested to the stakeholders, considering event-based applications; the *location* on which the tasks will be performed; and, the *variable* that must be monitored (e.g., temperature, humidity).

## 2.4 Development process

In this section, we explain how IoTDraw can be used in the specification of SOA-based applications. As introduced by Erl [10], the overall SOA process consists of several sub-processes [10], each one focused on a given aspect of the SOA style. The sub-processes are *SOA Adoption Planning*, *Service Inventory Analysis*, *Service-Oriented Design*, *Service Logic Design*, *Service Development*, *Service Usage and Monitoring*, *Service Discovery*, and *Service Versioning and Retirement*. IoTDraw is meant to be used in the specification of participants, services, interfaces, contracts, and choreographies. The activities refer to the *Service-Oriented Design* sub-process. Figure 4 depicts the service-oriented design process highlighting the activities in which SoaML4IoT can be used to specify and simulate the system model. Vertical swimlanes represents the stakeholders that are responsible for the activity; horizontal swimlanes represents the viewpoints that the activities are related (Section 2.2). The input of the process is the applications' requirements, and the output is the IoT system model, which represents the specified architecture aiming to fulfill the stakeholders' requirements.

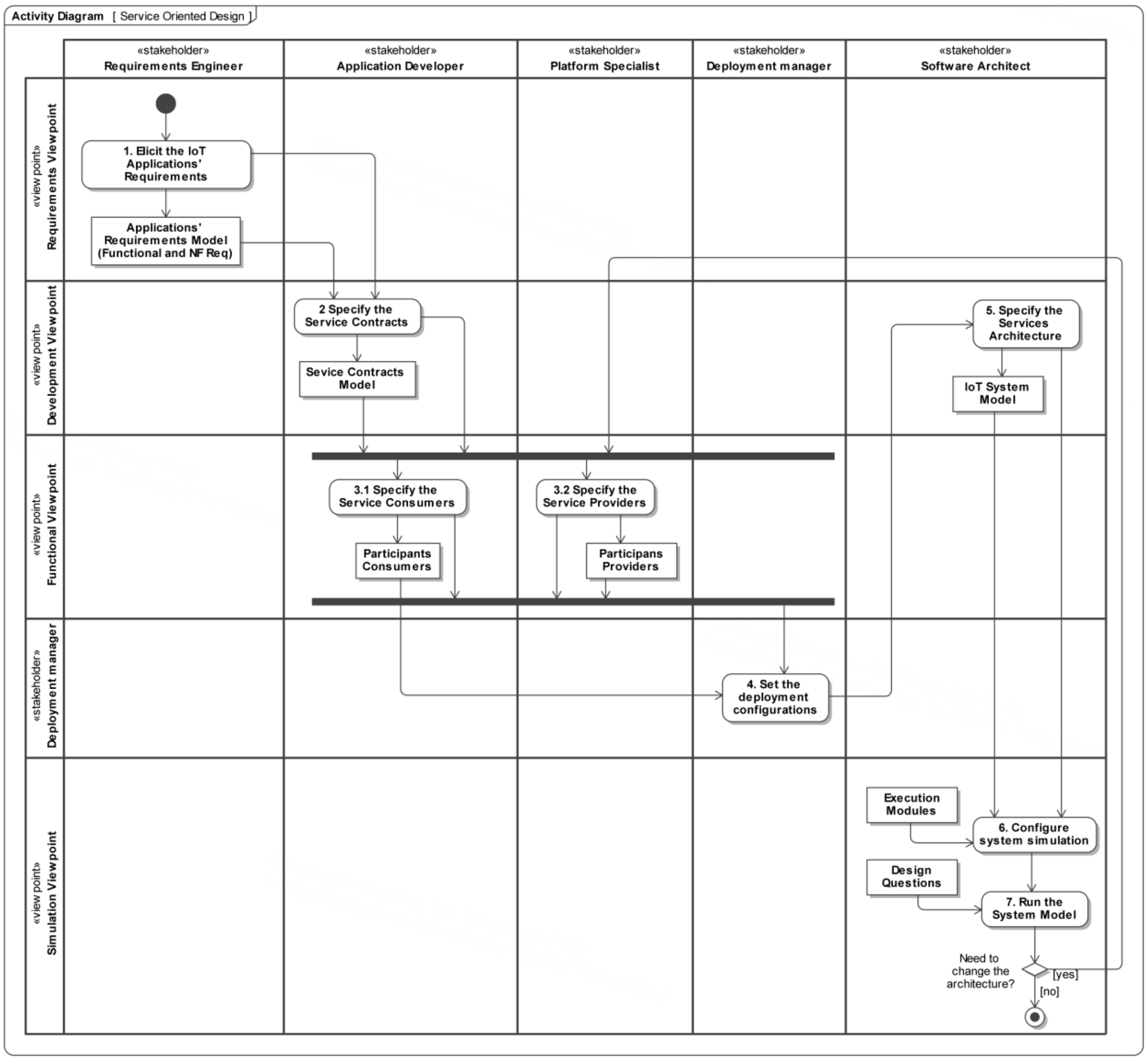

Activity Diagram [ Service Oriented Design ]
«stakeholder» Requirements Engineer
«stakeholder» Application Developer
«stakeholder» Platform Specialist
«stakeholder» Deployment manager
«stakeholder» Software Architect
«view point» Requirements Viewpoint
«view point» Development Viewpoint
«view point» Functional Viewpoint
«stakeholder» Deployment manager
«view point» Simulation Viewpoint
1. Elicit the IoT Applications' Requirements
Applications' Requirements Model (Functional and NF Req)
2 Specify the Service Contracts
Sevice Contracts Model
5. Specify the Services Architecture
IoT System Model
3.1 Specify the Service Consumers
3.2 Specify the Service Providers
Participants Consumers
Participans Providers
4. Set the deployment configurations
Execution Modules
6. Configure system simulation
Design Questions
7. Run the System Model
Need to change the architecture?
[yes]
[no]

Figure 4.Service-oriented design

To clarify the explanation of each activity, returning to the running example introduced in Chapter 0, we will design the FW IoT application as a *periodic application*. The objective is to sense the water level at regular time intervals and send alerts to the citizens, in case of flooding, to enable the fast evacuation near the flooding area. The FW application also performs analytics aiming at examining different flooding scenarios, which will be helpful for urban planning. The models of FW application, which will be created by SoaML4IoT, must support modelers to (i) identify key elements of the system, and (ii) answer important design questions regarding the services approach as well as issues related to lifetime, availability, response time, and scalability QoS requirements.

### 2.4.1 Setup IoTDraw

Before starting the specification of FW application, we must setup IoTDraw on Eclipse Papyrus. It consists of two steps, that is, (i) set the SoaML4IoT profile on a Papyrus Modeling project, and (ii) configure the Moka plugin for modeling execution. The artefacts are available at https://brccosta.github.io/iotdraw/, which must be downloaded to setup IoTDraw. We suggest the Eclipse Modeling Tools Oxygen.3a Release (4.7.3a), Papyrus version 3.3.0.201803070847, and MOKA version 3.1.0.2017.10.17.1318.

#### *2.4.1.1 Creating new IoT project in Papyrus*

In Eclipse, click on File > New > Other... In the showed dialog, chose the option Papyrus Project (Figure 5). Click on Next. In the second dialog box, select Software Engineering > UML (Figure 6). Click on Next. In the last dialog, set the project name and click on Finish.

#### *2.4.1.2 Setting SoaML4IoT profile*

The standard perspective of Eclipse is modeling. Thus, aiming to organize the modeling project, we will configure to the Papyrus perspective. Click on the menu Window > Perspective > Open Perspective > Other... In the Open Perspective dialog box, select Papyrus, and click on Open (Figure 8).

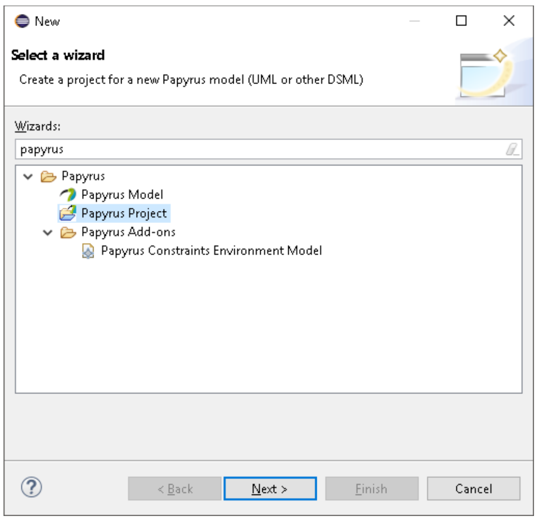


Figure 5. Creating a Papyrus Project

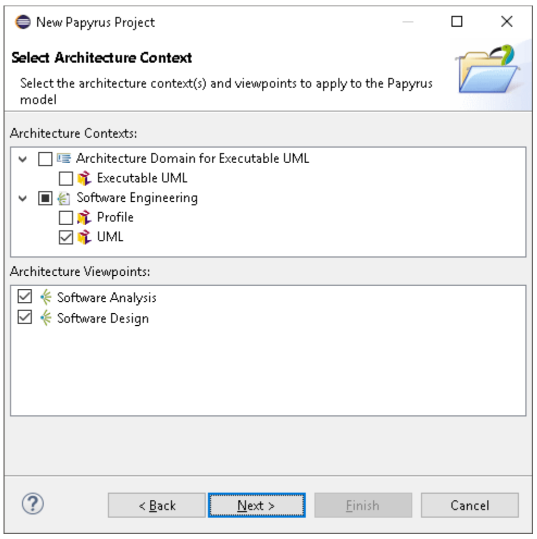
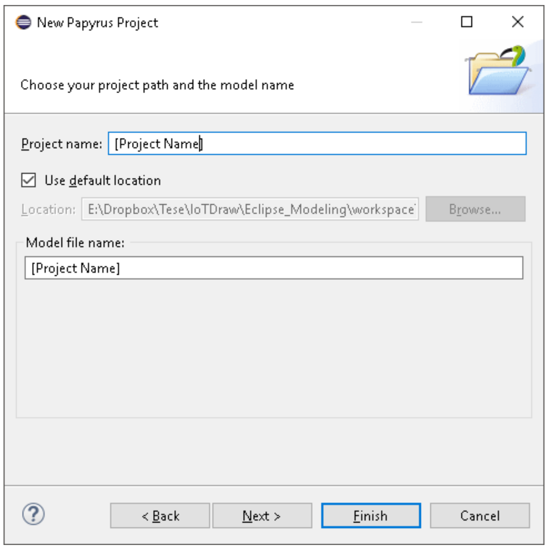


Figure 6. Selecting UML project


New Papyrus Project

Choose your project path and the model name

Project name: [Project Name]

☑ Use default location

Location: E:\Dropbox\Tese\IoTDraw\Eclipse_Modeling\workspace

Model file name: [Project Name]

Figure 7. Setting project name

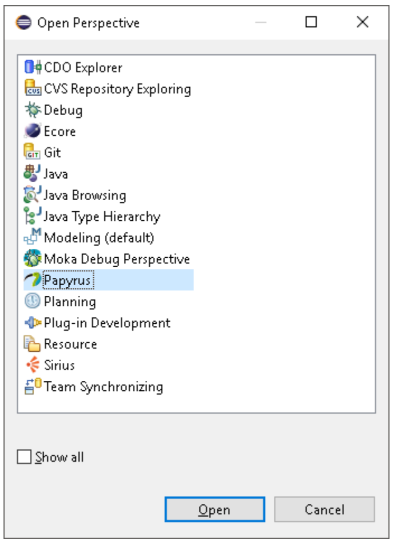


Figure 8. Setting Papyrus perspective

In the Model Explorer panel, select the project created in the previous step. In the properties, panel click on Profile, apply profile (Figure 9). Locate the SoaML4IoT profile and click OK (Figure 10).

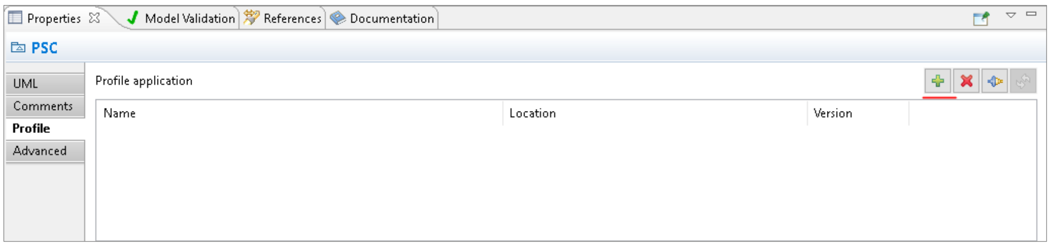


Figure 9. Applying profile

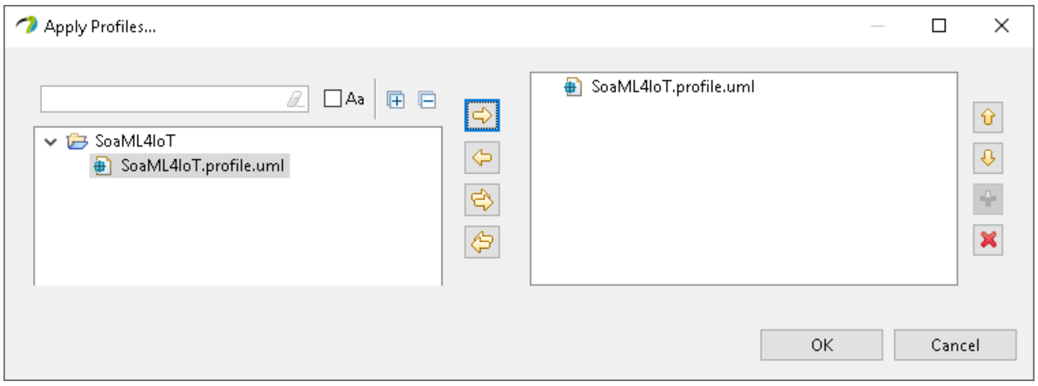
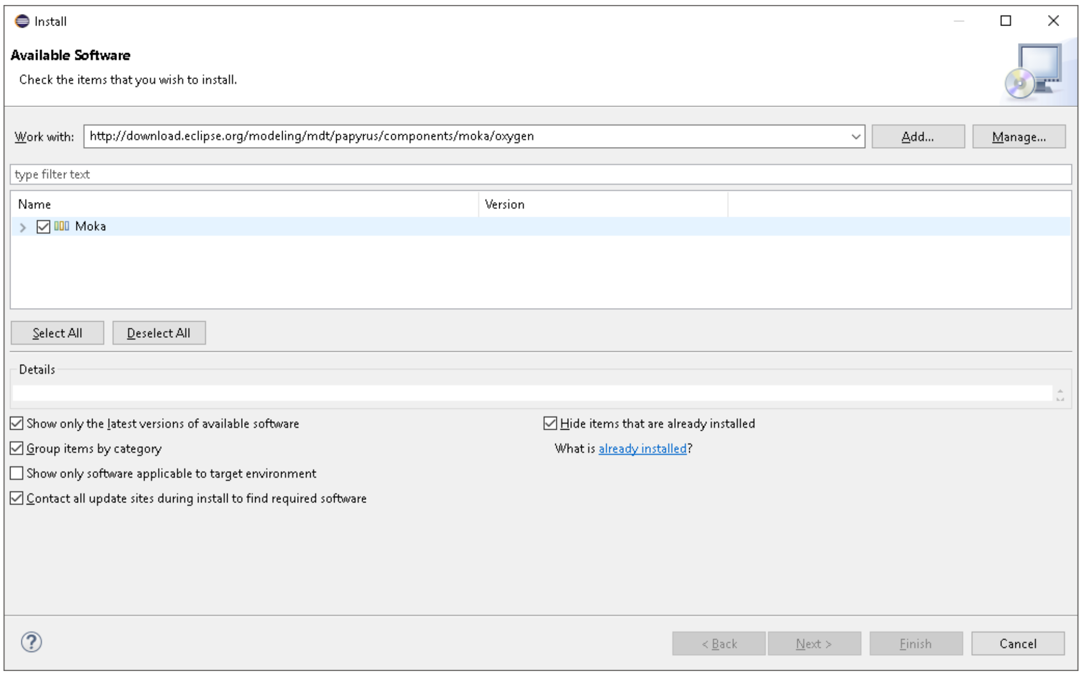


Figure 10. Applying SoaML4IoT profile

### *2.4.1.3 Setting MOKA with SoaML4IoT extension*

IoTDraw is based on the MOKA fUML engine. To install the plugin, click on Help > Install new software… (Figure 11) Enter with the repository address2 select the Moka framework. Click on Next in the following dialogs and close the Eclipse IDE after the installation.


Install

**Available Software**

Check the items that you wish to install.

Work with: http://download.eclipse.org/modeling/mdt/papyrus/components/moka/oxygen

type filter text

| Name | Version |
|---|---|
| ☑ Moka | |

Details

☑ Show only the latest versions of available software
☑ Group items by category
☐ Show only software applicable to target environment
☑ Contact all update sites during install to find required software
☑ Hide items that are already installed
What is already installed?

Figure 11. Install MOKA plugin

Open the Eclipse installation folder and open the plugins folder. Inside the directory, search for the following plugins:

- org.eclipse.papyrus.moka.composites
- org.eclipse.papyrus.moka
- org.eclipse.papyrus.moka.fuml

Such plugins refer to the implementations of the standard fUML specification. Replace than with the following plugins[3]:

- org.eclipse.papyrus.moka.composites.extended.iotdraw
- org.eclipse.papyrus.moka.extended.iotdraw
- org.eclipse.papyrus.moka.fuml.extended.iotdraw

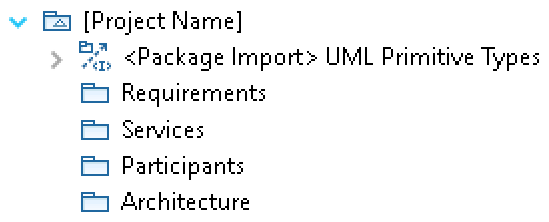
Now the Eclipse IDE is configured with the IoTDraw modeling framework.

[2] Available at Eclipse Repository: download.eclipse.org/modeling/mdt/papyrus/components/moka/oxygen

[3] Available at: https://brccosta.github.io/iotdraw/

#### *2.4.1.4 Organizing project directories*

After configuring the IoTDraw on Eclipse, we will organize the packages on which the diagrams representing the views of the system will be stored. To create a package, in the Model Explorer panel, right-click on the project > New child > Package. We suggest the following structure:



[Project Name]
<Package Import> UML Primitive Types
Requirements
Services
Participants
Architecture

Figure 12. Project package structure

The requirements package aims at storing requirements (i.e., the Requirement Viewpoint – Section 2.2.4). The Services package store the Service Contracts, including tasks, consumers and provider interfaces (i.e., the Development Viewpoint – Section 2.2.2). In the Participants package, the actual consumer and provider participants will be stored (i.e., the Functional Viewpoint – Section 2.2.1). Finally, in the Architecture folder, it will be stored the diagram representing the services architecture of the IoT system (i.e., the Development Viewpoint – Section 2.2.2).

After configuring the environment with IoTDraw, in the next sections, we present the specification of FW through each activity of the development process (Figure 4).

### 2.4.2 Elicit the IoT applications' requirements

The first step of the development process is to elicit the functional and non-functional requirements that aim to be fulfilled by the IoT applications. To achieve this goal, the requirements engineers structure the requirements over the Requirements Viewpoint (Section 2.2.4). Functional requirements are elicited according to the type of interaction model (i.e., periodic, event-driven). When the application is periodic, it is necessary to specify the rate in which a given task is required in a given location. On the other hand, whether the application is event-driven, it is specified the event that must trigger the task. At this phase, the functional requirements are represented through specific elements of SoaML4IoT, following the metamodel MK1 of Requirements viewpoint. The non-functional requirements are modeled through specific elements of SoaML4IoT that follows the MK2 metamodel (Requirements viewpoint), i.e., the stereotypes *QoS Characteristic*, and *QoS Dimension*.

In the Model Explorer, right-click on the Requirements package > New Diagram > Class Diagram. In the Palette, we will select two UML Classes and create them in the editor. Such classes represent the functional requirements of the FW applications. Thus, we must stereotype them with such an element from SoaML4IoT. To apply a stereotype of the SoaML4IoT profile, select the UML element that must be stereotyped (in the case, one of the classes created before). Next, in the properties panel, click on Apply Stereotype option (Figure 13). Next, select the desired stereotype, and click on OK (Figure 14).

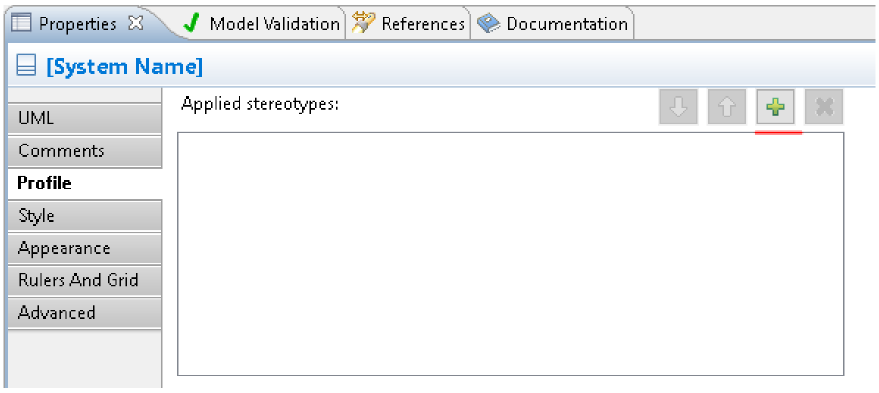
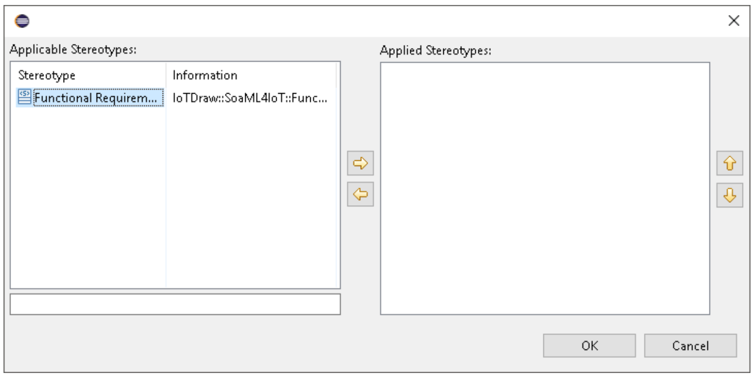


Figure 13. Applying stereotype



| Stereotype | Information |
|---|---|
| Functional Requirem... | IoTDraw::SoaML4IoT::Func... |

Figure 14. Applying Functional Requirement stereotype

After stereotyping one of the two classes representing the functional requirement of FW application, we must name such a class and set the tagged values. The first functional requirement of FW application is the monitoring of water level. It must be performed every 2 minutes. The other functional requirement refers to the alert in case of flooding. Thus, the alarm must sound aim to alter the citizens when the water level exceeds 20 cm. Figure 15 depicts the class diagram with these requirements.

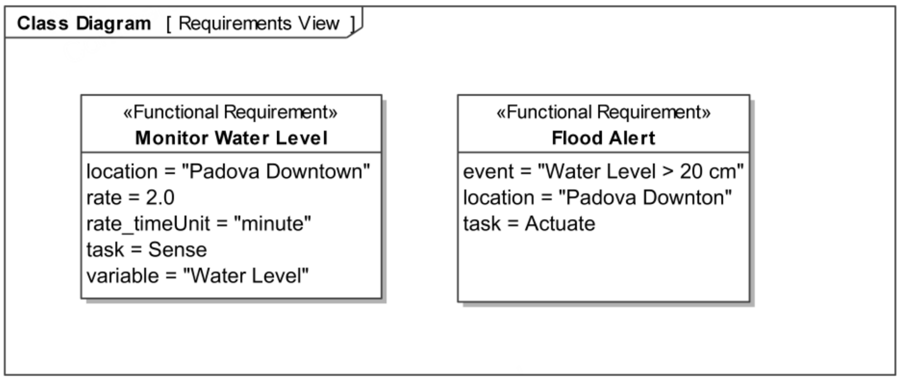


Figure 15. FW application - Requirements View (Functional Requirements)

Next, we will specify the non-functional requirements of the application. Recall that the FW application requires availability and response time QoS requirements. Following the QoS metamodel of the Requirements Viewpoint (Section 2.2.4), we structure such QoS requirements with two UML Classes stereotyped with QoS Characteristic, as depicted in Figure 16. For the application, it is required 99985% (minimum) of availability and, 1230 milliseconds (maximum) of response time.

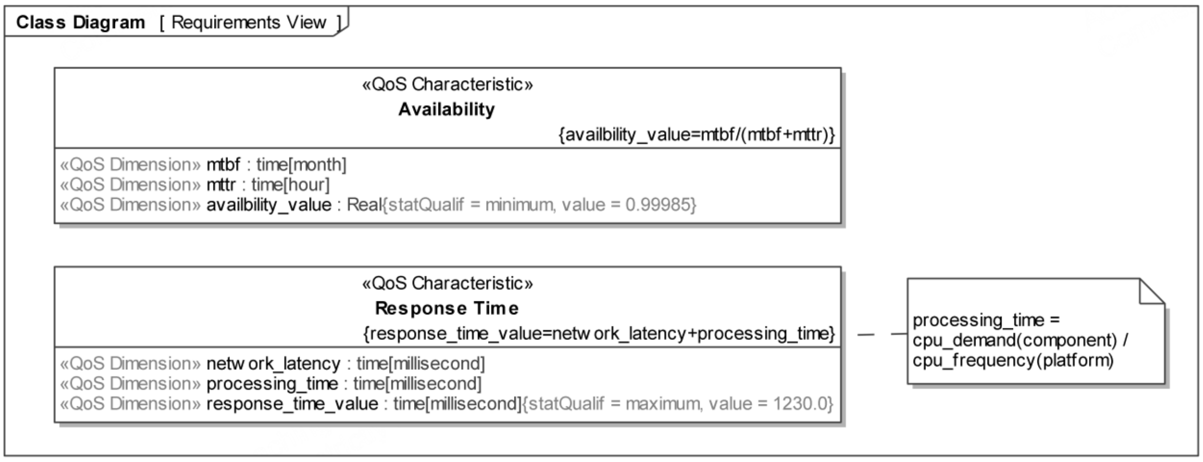

Class Diagram [ Requirements View ]
«QoS Characteristic»
Availability
{availbility_value=mtbf/(mtbf+mttr)}
«QoS Dimension» mtbf : time[month]
«QoS Dimension» mttr : time[hour]
«QoS Dimension» availbility_value : Real{statQualif = minimum, value = 0.99985}
«QoS Characteristic»
Response Time
{response_time_value=netw ork_latency+processing_time}
«QoS Dimension» netw ork_latency : time[millisecond]
«QoS Dimension» processing_time : time[millisecond]
«QoS Dimension» response_time_value : time[millisecond]{statQualif = maximum, value = 1230.0}
processing_time = cpu_demand(component) / cpu_frequency(platform)

Figure 16. FW application – Requirements View (Non-Functional Requirements)

In the specification of availability QoS requirement, we adopt the following availability model [4]:

$$Availability(P) = \frac{MTBF}{(MTBF + MTTR)} \tag{1}$$

where $P$ is the platform, $MTBF$ refers to the mean time between failures and $MTTR$ refers to the mean time to repair. In turn, the specification of response time QoS requirement is based on the following analytic model [21]:

$$ReponseTime(R) = L(Network) + P_{time}(Platform, Component) \tag{2}$$

where $R$ denotes the request. $L$ denotes the average latency of the network connection between consumer and provider participants. And, $P_{time}$ refers to the mean processing time of the platform to process the component's algorithms. And, $Q$ denotes the waiting time in the queue. $P_{time}$ is given by [24]:

$$P_{time}(Platform, Component) = \frac{CPU_{demand}(Component)}{CPU_{frequency}(Platform)} \tag{3}$$

where the average CPU demand of the component (unit: cycles) is divided by the CPU frequency (uni: GHz) of the Platform.

### 2.4.3 Specify the Service Contracts

The second step is the modeling of service contracts, providing the specifications that consumers and providers must agree with. This specification is performed by the application developers, defining which type of capabilities the application requires, as well as the service interfaces that expose them. In the service contract, the choreography of the service is also specified, modeling the interaction between consumer and provider participants. Such a choreography can be modeled as fUML activities, thus, allowing its execution.

The specification of service contracts of the FW application is created in the package Services, created before. We create a UML Class diagram named as Service Contracts, as depicted in Figure 17.

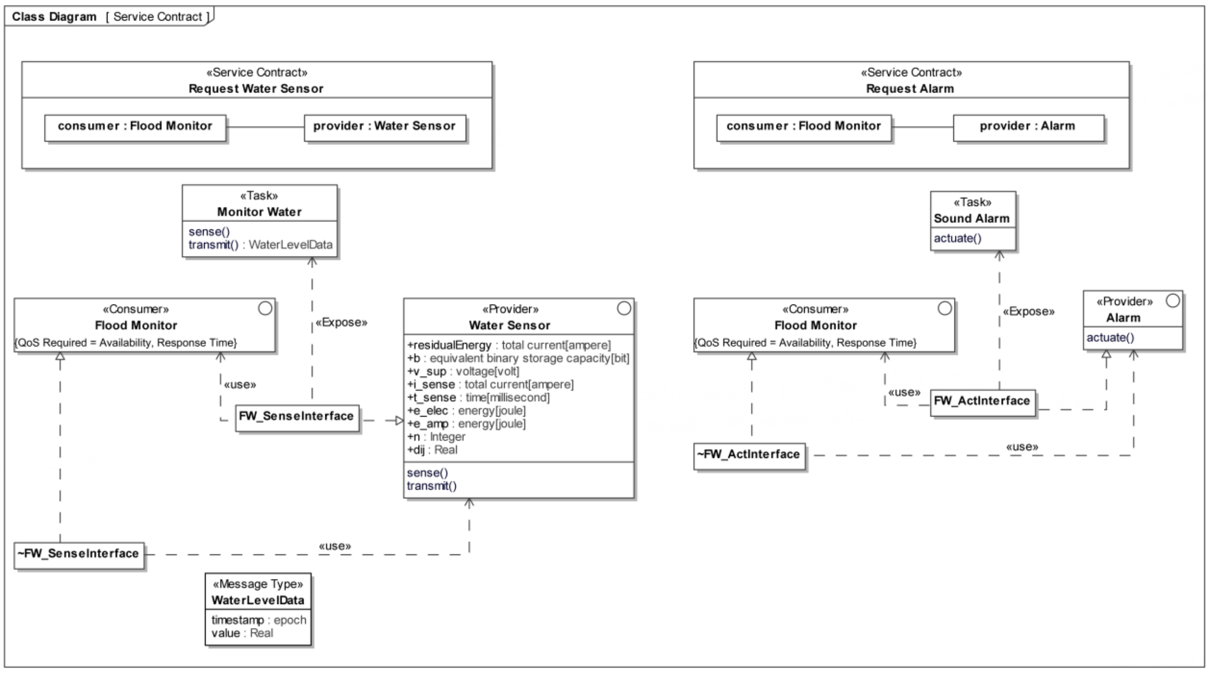


Figure 17. Contract Design for FW Application

There are two contracts that the participants must agree when designing the consumers and providers of the FW application, namely, *Request Water Sensor* and *Request Alarm* (Figure 17). The *Monitor Water* task provides two operations: *sense*() and *transmit*(). The former aims at sensing the water level while the second transmits the data to the requesting component. The data is structured as the *WaterLevelData* message type. The *Monitor Water* task is exposed by the *FW_SenseInterface*, which is realized by the *Water Sensor* (provider interface). This interface is used by *Flood Monitor* (consumer interface), which, in turn, realizes the conjugated[4] interface *~FW_SenseInterface*.

In the *Request Alarm* service contract, the *Sound Alarm* task provides only the operation *actuate*(), which aims to sound the alarm in case of flooding. The Sound Alarm task is exposed by the *FW_ActInterface*, which is realized by the *Alarm* (*Provider* interface). This interface is used by *Flood Monitor* (consumer interface), which, in turn, realizes the conjugated interface *~FW_ActInterface*.

The Flood Monitor application (service consumer) has two QoS requirements, that is, Availability and Response time. As specified in the Requirements View (Figure 16), the Flood Warning application requires 99985% (minimum) of availability and, 1230 milliseconds (maximum) of response time.

[4] The conjugate property reverses the provider and required interfaces so that you can connect ports with similar types.

In the *Request Water Sensor* service contract, the *Water Sensor* (provider interface) specifies a set of properties that a participant implementing such interface must provide. The devices are powered by limited batteries; thus, these properties are required to answer the following design question: "*considering the request rate of the application, what is the operational lifetime of the Water Sensor devices?*". Recall that, in the context of the PSC, the lifetime is considered as the time spanning from the instant when the device starts functioning until it runs out of energy, becoming the service provided by such a device unavailable.

To answer this design question, we adopt an analytic model proposed by Halgamuge and colleagues [12] to predict the energy consumption of devices' tasks. In such a model, the energy consumption of sensing tasks $E_{sense}$ (unit: Joules) is given by:

$$E_{sense} = bV_{sup}I_{sense}T_{sense} \tag{1}$$

where $b$ is the bit package (unit: kilobit - kb) collected by the sensing activity, $V_{sup}$ is the supply voltage (unit: volt - v), $I_{sense}$ is the total current (unit: milli ampere) required for sensing activity, and $T_{sense}$ is the time duration (unit: milliseconds - mS) for sensing unit is collecting data from the environment.  The equation from Halgamuge's model formalize the energy consumption of transition task $E_{transmit}$ (unit: Joules), is given by

$$E_{transmit} = bE_{elec} + bd_{ij}^{n}E_{amp} \tag{2}$$

where $b$ is the bit package to be transmitted in a distance $d_{ij}$ (unit: meter - m), $E_{elec}$ (unit: Nano Joules per bit – nJ/bit) is the energy dissipated to transmit data, $E_{amp}$ is the energy dissipated by the power amplifier (unit: Pico Joules per bit per square meter – pJ/bit/m$^2$), and $n$ is the distance-based loss exponent (unit: Integer).

The battery capacity of devices is typically measured as milliampere hour (mAh), while the energy consumption is measured as Joules (J). Thus, it is necessary to convert from J to mAh in order to verify the difference between the battery capacity and the total energy consumption required for the device. The conversion from J to mAh is given by:

$$Joules\_to\_mAh = \frac{1000 \times E_{(Wh)}}{V_{(V)}} \tag{3}$$

where $E_{(Wh)}$is the energy in watt-hours (Wh) and $V_{(V)}$is the voltage in volts (V). 1 J corresponds to 0.000277778 Wh, thus $1\,J = 1000 \times 0{,}000277778/V_{(V)}$ mAh.

These energy consumption equations are modeled in the choreography of the *Request Water Sensor* service contract, specified as an fUML Activity Diagram (Figure 18). It starts with a *readSelf* action followed by the reading of the attributes (*readStructuralFeature* action) required to calculate the energy consumption of sensing task. An *opaque action* calculates the energy consumption. The result in Joules is converted with an opaque action into mAh (milli ampere hour), which is the unit of battery capacity of devices. Next, this result is used to update the residual energy. The behavior of the transmit task follow the same structure.

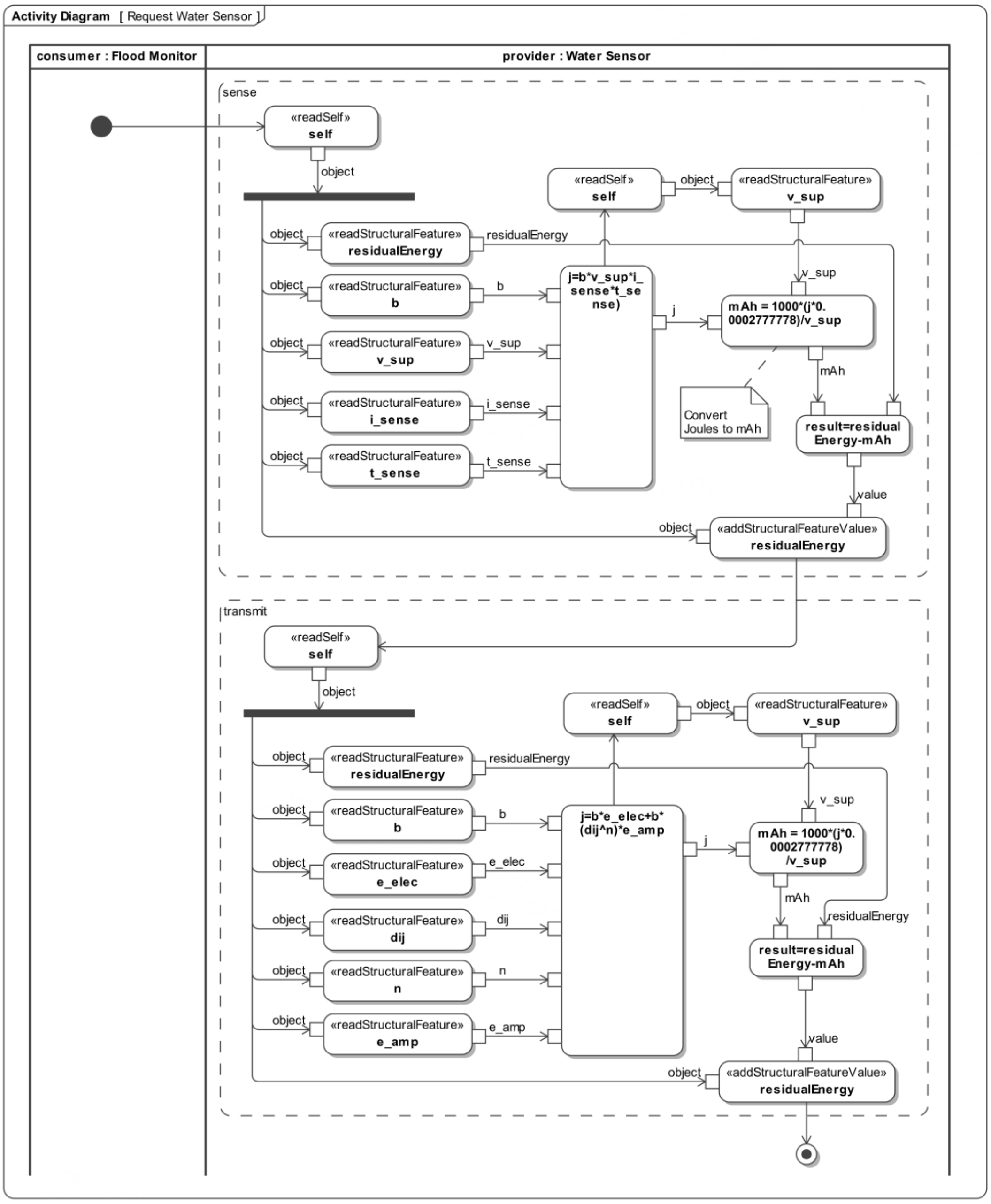

Activity Diagram [ Request Water Sensor ]
consumer : Flood Monitor
provider : Water Sensor
sense
«readSelf» self
object
«readStructuralFeature» residualEnergy
residualEnergy
«readStructuralFeature» b
b
«readStructuralFeature» v_sup
v_sup
«readStructuralFeature» i_sense
i_sense
«readStructuralFeature» t_sense
t_sense
j=b*v_sup*i_sense*t_sense)
j
mAh = 1000*(j*0.0002777778)/v_sup
mAh
Convert Joules to mAh
result=residualEnergy-mAh
value
«addStructuralFeatureValue» residualEnergy
transmit
«readStructuralFeature» e_elec
e_elec
«readStructuralFeature» dij
dij
«readStructuralFeature» n
n
«readStructuralFeature» e_amp
e_amp
j=b*e_elec+b*(dij^n)*e_amp
mAh = 1000*(j*0.0002777778)/v_sup

Figure 18. Choreography of Request Water Level Sensor Service Contract

### 2.4.4 Specify the Service Consumers and providers

In the third step, the consumer (i.e., applications) and provider (i.e., platforms) participants are identified and modeled. They must agree with the contract previously specified by providing the required capabilities through the service interfaces. Application developers model the consumer participants while the providers are modeled by platform specialists, who have a deep understanding of the protocols, configurations, and specificities of the computer nodes.

In the package Participants, we create two UML Class diagrams, one for participants consumers and the other for participant provides. The specification of consumer participants of FW application is depicted in the UML Class Diagram of Figure 19.

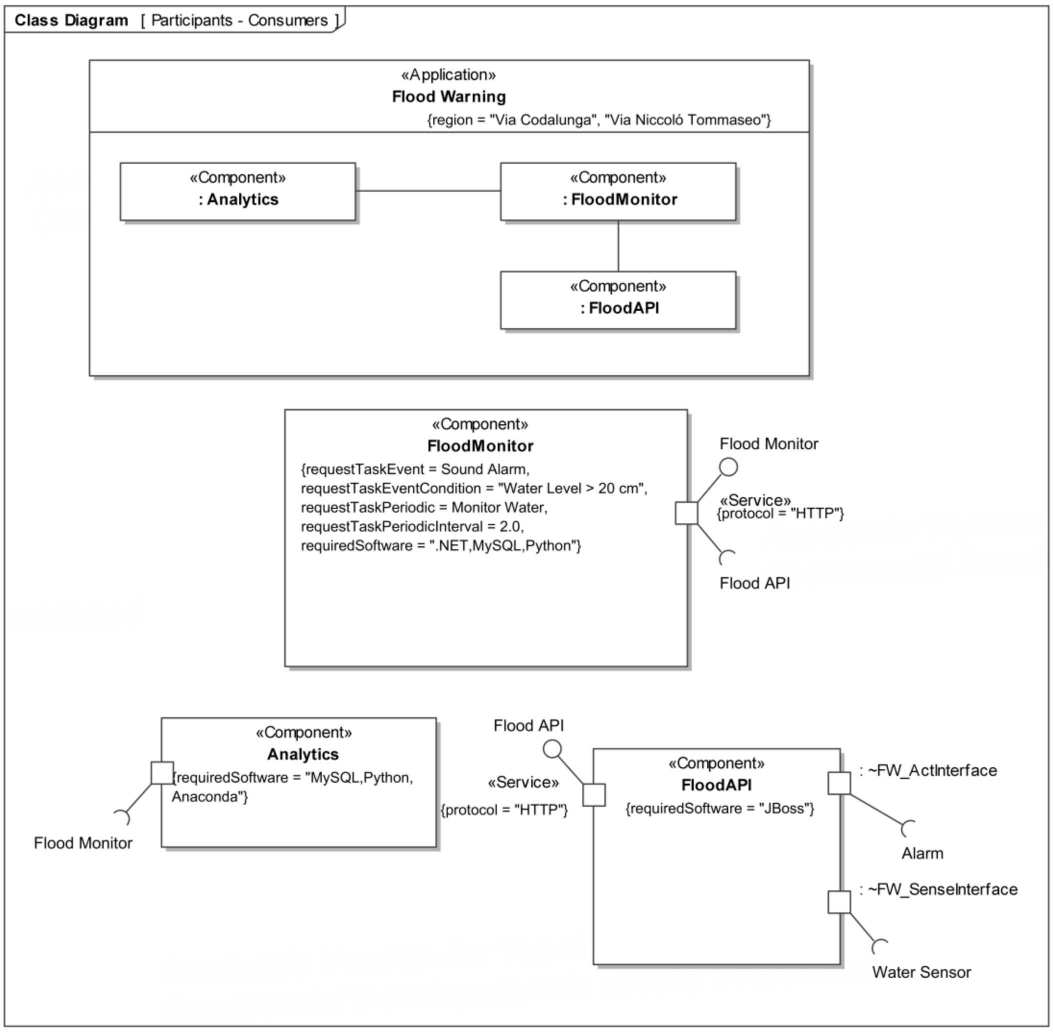

Class Diagram [ Participants - Consumers ]
«Application»
Flood Warning
{region = "Via Codalunga", "Via Niccoló Tommaseo"}
«Component»
: Analytics
«Component»
: FloodMonitor
«Component»
: FloodAPI
«Component»
FloodMonitor
{requestTaskEvent = Sound Alarm,
requestTaskEventCondition = "Water Level > 20 cm",
requestTaskPeriodic = Monitor Water,
requestTaskPeriodicInterval = 2.0,
requiredSoftware = ".NET,MySQL,Python"}
Flood Monitor
«Service»
{protocol = "HTTP"}
Flood API
«Component»
Analytics
{requiredSoftware = "MySQL,Python, Anaconda"}
Flood Monitor
Flood API
«Service»
{protocol = "HTTP"}
«Component»
FloodAPI
{requiredSoftware = "JBoss"}
: ~FW_ActInterface
Alarm
: ~FW_SenseInterface
Water Sensor

Figure 19. Participants of FW application - Consumers

The application Flood Warning aims to monitor the water level in two important avenues of Padova, namely, Via Codalunga and Via Niccoló Tommaseo. The application is composed of three components, namely, *FloodAPI*, *FloodMonitor*, and *Analytics*. The component *FloodAPI* intermediates the communication between other components and the devices' services. It has two services; each one typed as an interface as defined in the respective contract (see Section 2.4.3); thus, the *FloodAPI* requires the *Alarm* and the *Water Sensor* interfaces. The component also provides a service, based on HTTP, providing an interface for external requests.

The component *FloodMonitor* is responsible for managing the monitoring for flooding. It requires the interface *Flood API* and provides a service, based on HTTP, with an interface *Flood Monitor* for external requests. The task that will be requested periodically by the component (i.e., every 2 minutes), is the *Monitor Water*. In turn, the task *Sound Alarm* will be requested only when the water level is above 20 centimeters. Finally, the component *Analytics*, which requires the interface *Flood Monitor*, aims to store the water level data and further perform analytics aiming at examining different urban scenarios. This component requires the interface Flood Monitor.

The component FloodMonitor was developed in two languages, namely, C# and Python, thus, it requires an execution environment providing such capabilities. Also, temporary data is stored in a MySQL DBMS instance. The component *Analytics* performs all analysis with the Anaconda, a distribution of the Python for data science and machine learning applications and requires the MySQL DBMS for storing data. Finally, the *FloodAPI* component requires only the JBoss application server, aiming to allow addressing HTTP requests.

The specification of provider participants of FW application is depicted in the UML Class Diagram of Figure 20. There are two cloud nodes available for the system; the first is in *Stuttgart* while the second is in *Michigan*. The three available fog nodes, *fog_1*, *fog_2*, and *fog_3*, are located in different areas of the city of Padova. Both cloud nodes and fog nodes provides a specific set of software platforms. Finally, in the example, there are two devices, namely, w*ater sensor* and *alarm*. The devices have services providing interfaces for external access. Note that these services are typed as elements of the service contracts.

Since the water sensor devices realize the *Water Sensor* interface, it provides the properties required by this interface (recall that these properties are used in the choreography to estimate the energy consumption of the services). In our example, we use the values of voltage, current, etc. presented by [12], considering a generic device (e.g., Arduino, RaspberryPi). Finally, the associations stereotyped as <<Net>>, models the network connections between the platforms.

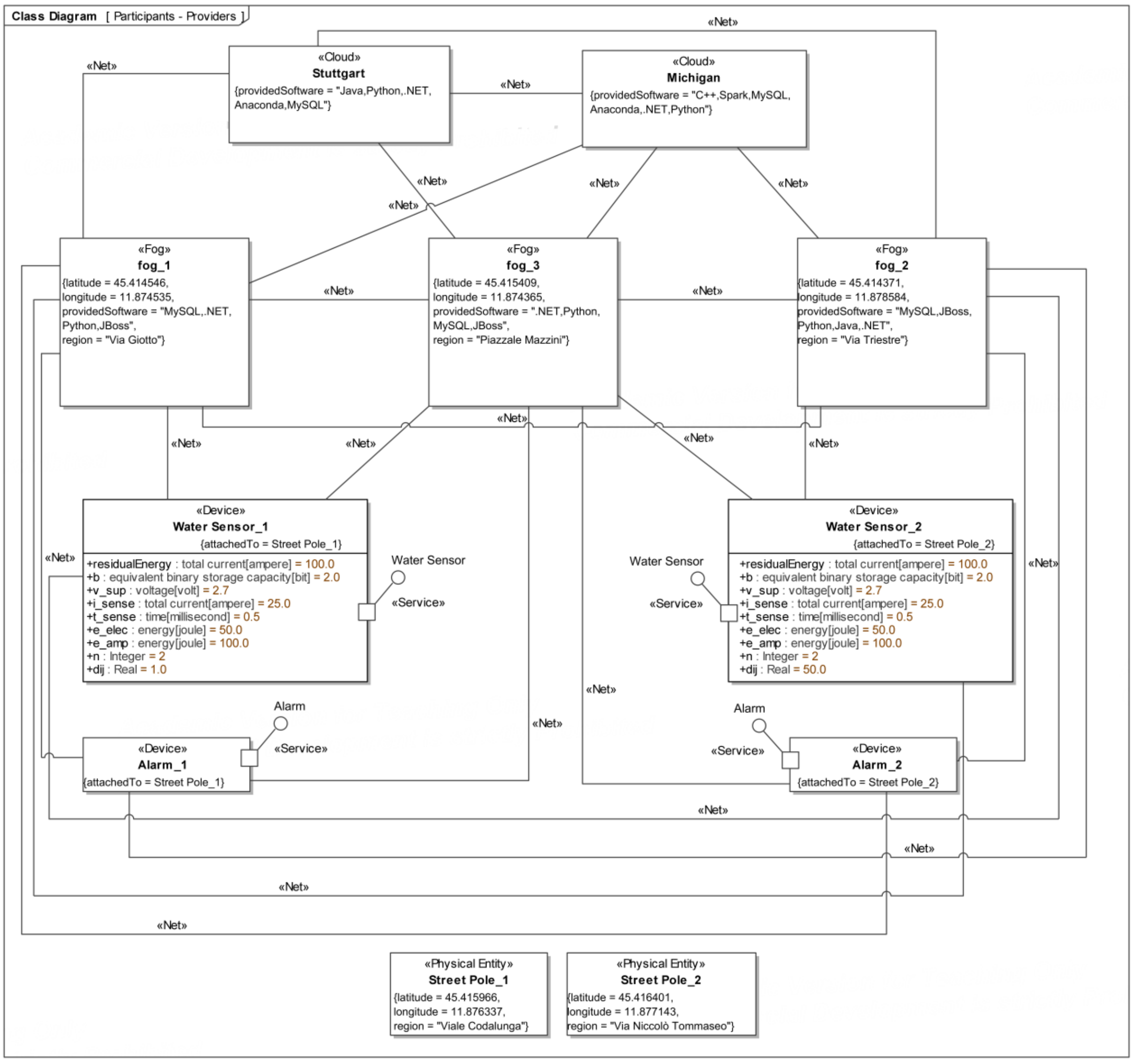

Class Diagram [ Participants - Providers ]
«Net»
«Cloud»
Stuttgart
{providedSoftware = "Java,Python,.NET, Anaconda,MySQL"}
«Cloud»
Michigan
{providedSoftware = "C++,Spark,MySQL, Anaconda,.NET,Python"}
«Fog»
fog_1
{latitude = 45.414546, longitude = 11.874535, providedSoftware = "MySQL,.NET, Python,JBoss", region = "Via Giotto"}
«Fog»
fog_3
{latitude = 45.415409, longitude = 11.874365, providedSoftware = ".NET,Python, MySQL,JBoss", region = "Piazzale Mazzini"}
«Fog»
fog_2
{latitude = 45.414371, longitude = 11.878584, providedSoftware = "MySQL,JBoss, Python,Java,.NET", region = "Via Triestre"}
«Device»
Water Sensor_1
{attachedTo = Street Pole_1}
+residualEnergy : total current[ampere] = 100.0
+b : equivalent binary storage capacity[bit] = 2.0
+v_sup : voltage[volt] = 2.7
+i_sense : total current[ampere] = 25.0
+t_sense : time[millisecond] = 0.5
+e_elec : energy[joule] = 50.0
+e_amp : energy[joule] = 100.0
+n : Integer = 2
+dij : Real = 1.0
Water Sensor
«Service»
«Device»
Water Sensor_2
{attachedTo = Street Pole_2}
+residualEnergy : total current[ampere] = 100.0
+b : equivalent binary storage capacity[bit] = 2.0
+v_sup : voltage[volt] = 2.7
+i_sense : total current[ampere] = 25.0
+t_sense : time[millisecond] = 0.5
+e_elec : energy[joule] = 50.0
+e_amp : energy[joule] = 100.0
+n : Integer = 2
+dij : Real = 50.0
Alarm
«Service»
«Device»
Alarm_1
{attachedTo = Street Pole_1}
«Device»
Alarm_2
{attachedTo = Street Pole_2}
«Physical Entity»
Street Pole_1
{latitude = 45.415966, longitude = 11.876337, region = "Viale Codalunga"}
«Physical Entity»
Street Pole_2
{latitude = 45.416401, longitude = 11.877143, region = "Via Niccolò Tommaseo"}

Figure 20. Participants of FW application – Providers

### 2.4.5 Set the Deployment Configurations

In the fourth step, the deployment manager specifies the deployment configurations of the components that are part of the IoT applications. He/she must consider the constraints regarding required software, and minimum RAM and Hard Disk. Figure 21 depicts an example of deployment configuration for the components of FW application.

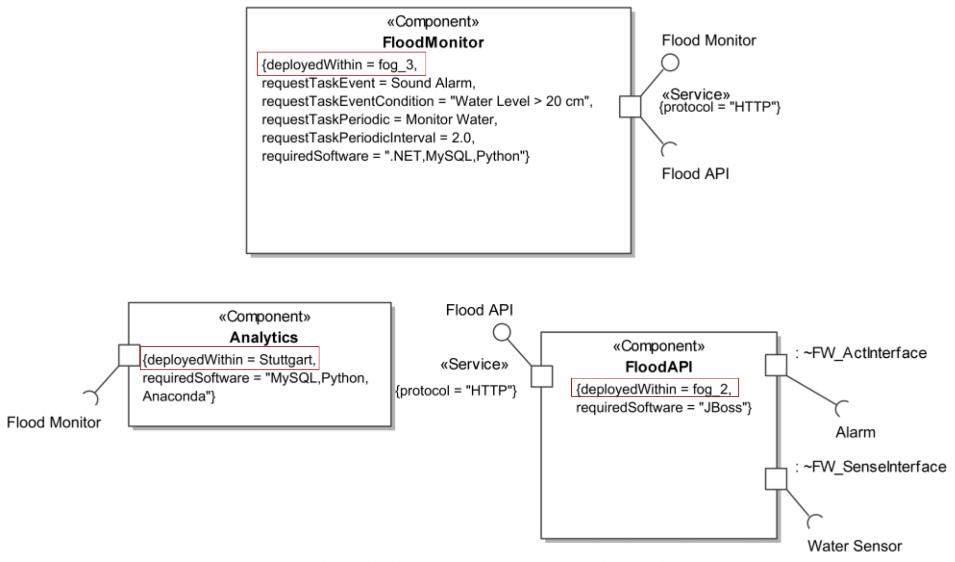

«Component»
FloodMonitor
{deployedWithin = fog_3,
requestTaskEvent = Sound Alarm,
requestTaskEventCondition = "Water Level > 20 cm",
requestTaskPeriodic = Monitor Water,
requestTaskPeriodicInterval = 2.0,
requiredSoftware = ".NET,MySQL,Python"}
Flood Monitor
«Service»
{protocol = "HTTP"}
Flood API
«Component»
Analytics
{deployedWithin = Stuttgart,
requiredSoftware = "MySQL,Python,
Anaconda"}
Flood Monitor
Flood API
«Service»
{protocol = "HTTP"}
«Component»
FloodAPI
{deployedWithin = fog_2,
requiredSoftware = "JBoss"}
: ~FW_ActInterface
Alarm
: ~FW_SenseInterface
Water Sensor

Figure 21 Deployment Configuration of FW application

The multiple components composing an application may be deployed on different platforms, which result in a considerable number of eligible deployment scenarios. Since each deployment scenario impacts differently on application requirements, it may become humanly infeasible to identify the best deployment alternative. In this sense, IoTDraw supports the deployment managers finding for eligible deployment scenarios. Furthermore, it is possible to consider QoS requirements when finding for deployment options.

### 2.4.6 Specify the Services Architecture

The fifth step aims at modeling the IoT system by integrating the consumers and providers previously created. Such an integration refers to a diagram que connects applications to devices through the service contracts. When connecting applications and devices, it is necessary for the applications having at least one component with its interface that agree with the contract.

The services architecture design of PSC, including the FW application, is depicted in the UML Class Diagram of Figure 22. The PSC services architecture aims at connecting the consumers (i.e., applications) to the providers (i.e., devices) through the specified contracts. When connecting the participants, IoTDraw checks if the applications have components that agree with the contracts, that is, if they have the ports implemented the required interfaces as specified in the contracts.

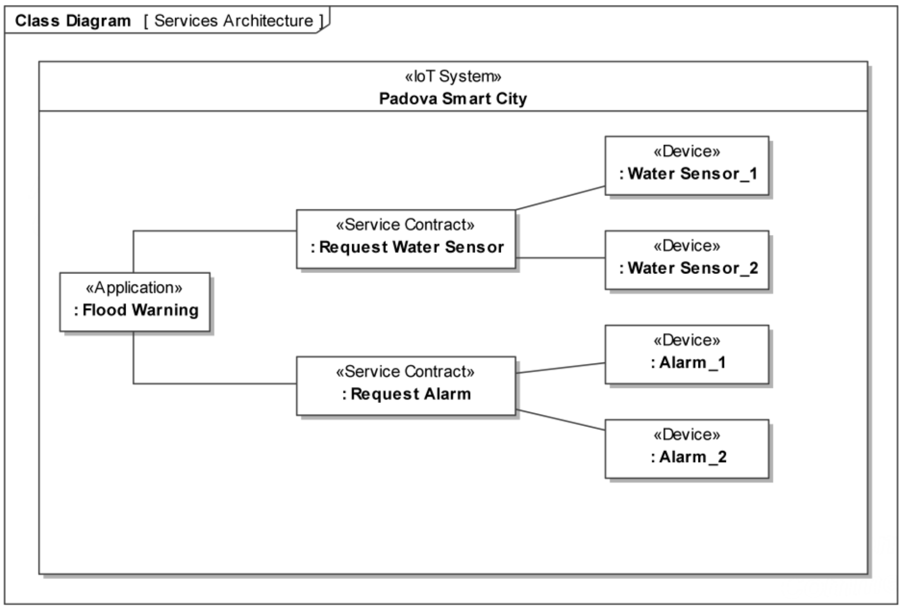


Figure 22. The services architecture of PSC System

## 2.4.7 Configure the System Simulation

The model generated in the previous activity is the input of the sixth activity, in which the model is configured to be executed. It consists of: (i) setting the time limit to execute the model; (ii) setting the execution module that aims to be invoked during the simulation. The simulation time of the model is specified as 1.051.200 minutes (i.e., 2 years). In this example, we do not apply any execution module..

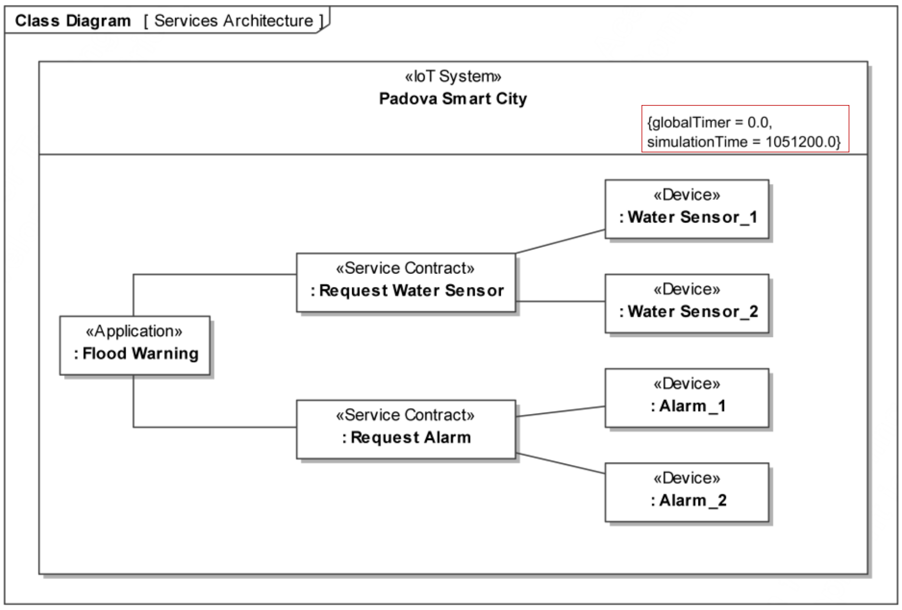


Figure 23. The services architecture of PSC System with system simulation tagged values

### 2.4.8 Run the System Model

The system model runs in the seventh activity. The model execution aims to (i) verify whether the specified architecture meets the application requirements, and/or (ii) answer design questions by predicting system properties. By answering a design question, the application developers can make architectural design decisions and tune the specification of the system. Furthermore, whether the simulation shows that the architecture does not address the applications' requirements, the process returns at the second step, aiming to change the configurations or calibrate some properties. The process ends when there is no need of changing the architecture due, for example, it fulfills all application requirements.

We execute the PSC model aiming to answer the following design questions (i) - *considering the request rate of the FW application, what is the operational lifetime of the Water Sensor devices?*, and (ii) - *in the $S^2$aaS model, what is the operational lifetime of the devices considering the required data freshness of service consumers?*" In the analysis, we execute the model in Intel Core i7-2640M, 2.80GHz, 6GB of RAM, JRE version 1.8.0_191..

It is worth to highlight that, as demonstrated by several studies (e.g., [9, 26]), predicting the lifetime of devices is critical in many IoT systems, since limited batteries may power the devices composing the systems. Without a previous notion on how long the devices will keep working and providing data, the planning for redundant monitoring or replacing batteries may be inappropriate. Consequently, the applications requiring the data may fail unexpectedly.

To simulate the model, Moka provides helpers that generates the required classes to allow model execution and animation. Thus, the first step to simulate the model is to generate such classes. On Eclipse IDE, right-click on the class stereotyped with IoT System > Moka > Modeling Utils > Generate Factory (Figure 24). Note that in the Model Explorer, Moka generates the required elements for simulating the model (Figure 25). The next step is to configure the execution. Click on the menu Run > Run Configurations… Next, select Moka launch configuration and click on "New lunch configuration". In Figure 26, we present the final configuration of the simulation. The execution engine as set as the extended version of Moka, that is, org.eclipse.papyrus.moka.composites.extended.iotdraw. Finally, click on Run to perform the model execution.

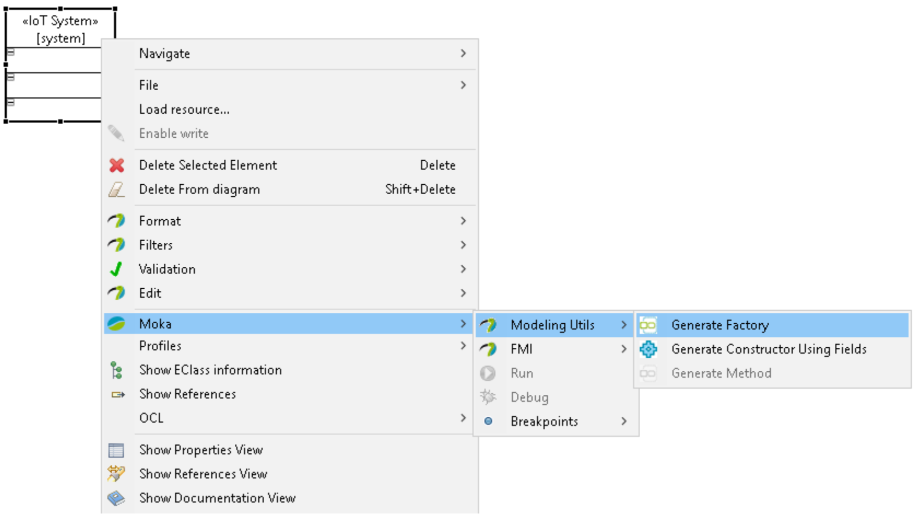


Figure 24. Generating factory

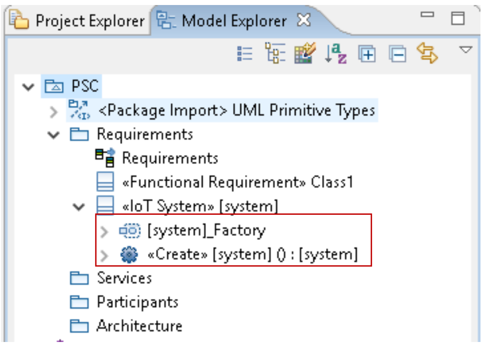
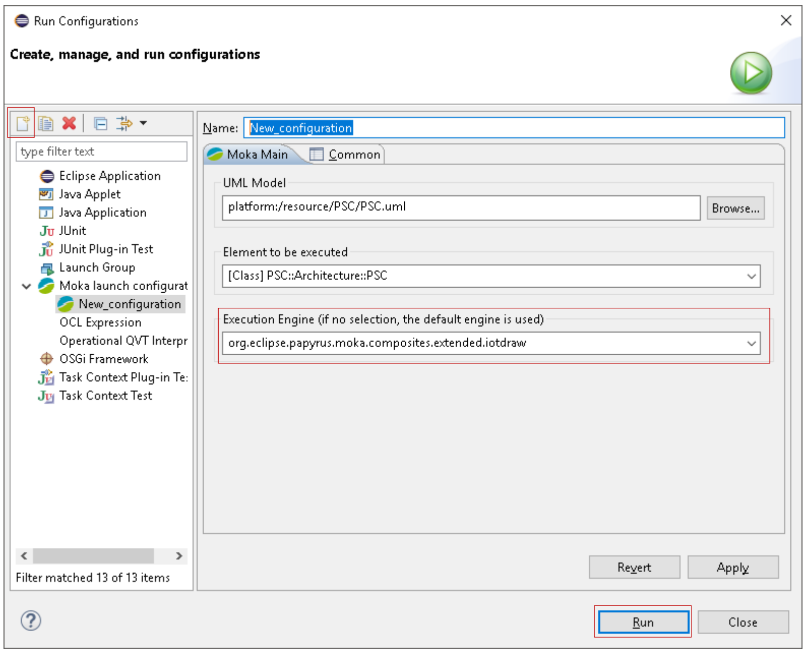


Figure 25. Generated elements for model simulation


Run Configurations

Create, manage, and run configurations

Name: New_configuration

Moka Main | Common

UML Model: platform:/resource/PSC/PSC.uml

Element to be executed: [Class] PSC::Architecture::PSC

Execution Engine (if no selection, the default engine is used): org.eclipse.papyrus.moka.composites.extended.iotdraw

Filter matched 13 of 13 items

Figure 26. Simulation configuration

Based on the participants design (see Section 2.4.4), water sensor devices can communicate with *fog_1*, *fog2*, or *fog_3*. Each of this fog nodes are located in a different place of Padova. Thus, for each simulation scenario, we vary the distance of the devices (i.e., the variable $d_{ij}$ of equation (2)), setting a random value from 1 to 50 (meters) for the attribute $d_{ij}$. After starting the model execution, we monitor the variable *residualEnergy*. When the value of the variable is above a given constant (i.e., 5 mAh), we stop the simulation and verify the value of the tagged value *globalTime*. We also vary the request rate from 2 to 6, aiming to analyze the lifetime of the water sensor device with different request rates. For each simulation scenario, we perform 30 execution rounds. The result is depicted in Figure 27.

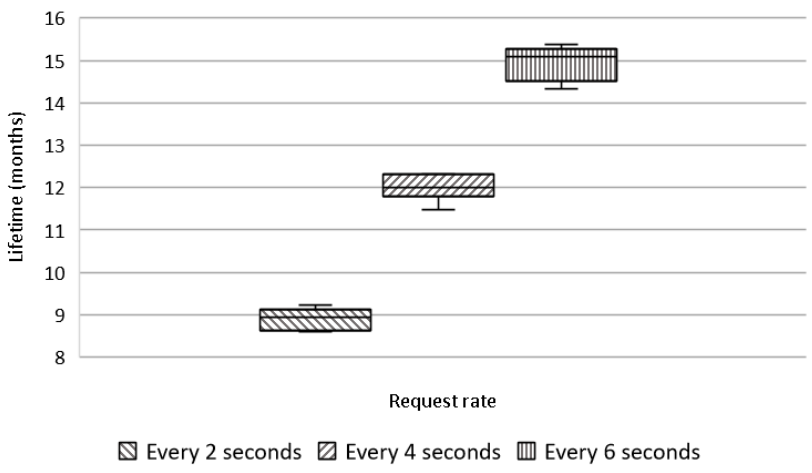


Figure 27. Operational lifetime of water sensor devices considering different request rates of FW application

The simulation shows that requesting the water level data every 2 seconds, the average lifetime of the devices is 9 months. On the other hand, by increasing the request rate to 4 and 6 seconds, the lifetime of the water sensor device increases to 12 and 15 months.

The second design question related to the devices' lifetime we aim to answer is "(ii) *in the S$^2$aaS model, what is the operational lifetime of the devices considering the required data freshness of service consumers?"*. A request for a given device's service requires it performing at least one sensing tasks. However, the fewer tasks performed by a device, the less energy consumed and, consequently, potentially the higher is its operational lifetime. A common strategy to reduce performing sensing tasks is to cache data to reuse a prior response message to satisfy a current request.

CoAP, which is the application protocol used in the devices' services of PSC, allows the specification of freshness in the request and response messages by using the option *max-age* [8]. This option indicates that the response is to be considered not fresh after its age is greater than the specified number of seconds. Thus, whether the cached data is less than the max-age value, it can be used to satisfy the request without requiring the device to perform (new) sensing tasks (which would reduce the lifetime of the device).

We add the max-age (unit: minute) property in the Water Sensor components and change the choreography of Request Water Level Sensor Service Contract (Figure 18) to represent this scenario. In the simulation, we consider one request for water level data per minute (a higher workload is analyzed later, regarding the scalability requirement). When an application requests for water level, if the cached data is less than the max-age value, the device does not perform the sensing task, i.e., only transmit the data. Else, the device performs sensing, cache, and transmit task.

As the previous simulation scenario, we vary the distance of the devices of the fog nodes. After starting the model execution, we monitor the variable *residualEnergy*. When the value of the variable is above a given constant (i.e., 5 mAh), we stop the simulation and verify the value of the tagged value *globalTime*. We also vary the max-age from 1 to 4, aiming to analyze the lifetime of the water sensor device with different data freshness. For each simulation scenario, we perform 30 execution rounds. Figure 28 shows the results.

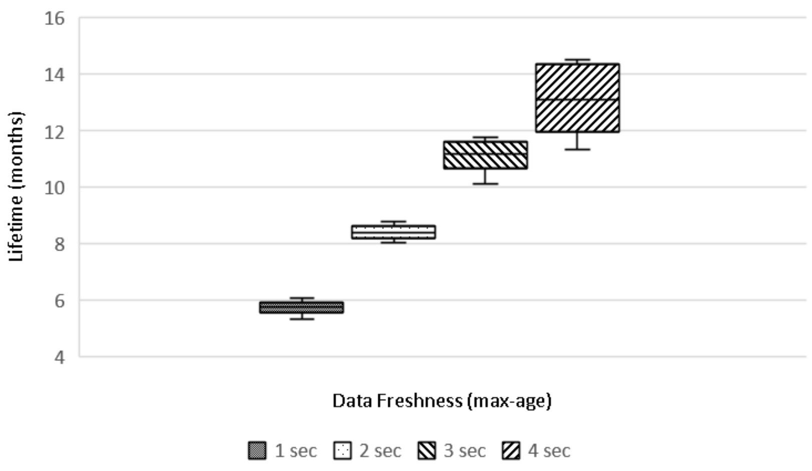


Figure 28. Operational lifetime of water sensor devices considering different data freshness requirements

By allowing to request water level with higher max-ages, the lifetime of the devices increases considerably. For example, in a scenario that the max-age is 2 seconds, it increases the lifetime of the devices in approximately 45% when compared with 1 second of max-age.

Whether the requests allow 4 seconds of max-age, it could more than duplicate the operational lifetime.

In the context of S$^2$aaS, an important design decision that could be made based on such analysis is to allow requesting water level sensor with a desired max-age. Furthermore, given that the service providers may want to maximize the operational lifetime of the devices, the goal is to reduce the number of tasks by sending cached data to consumers. In this sense, the higher max-age a consumer accepts, the cheaper he/she would pay for the water level data. On the other hand, the lower max-age, the more sensing tasks are required. Thus, the service provider may charge more. Therefore, an architectural design decision is to address the requests with max-age. Figure 29 depicts the model representing such an architectural decision.

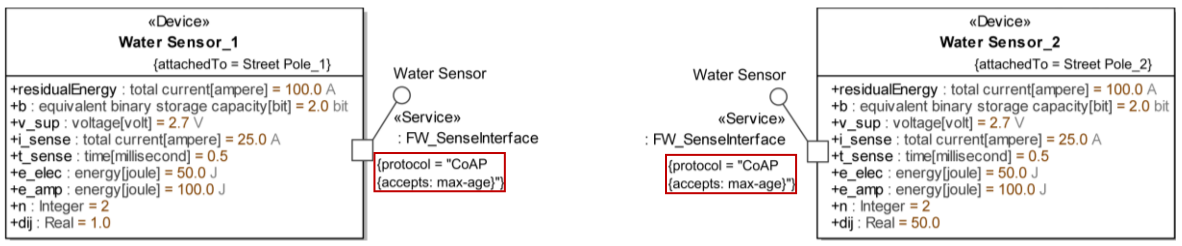

«Device»
Water Sensor_1
{attachedTo = Street Pole_1}
+residualEnergy : total current[ampere] = 100.0 A
+b : equivalent binary storage capacity[bit] = 2.0 bit
+v_sup : voltage[volt] = 2.7 V
+i_sense : total current[ampere] = 25.0 A
+t_sense : time[millisecond] = 0.5
+e_elec : energy[joule] = 50.0 J
+e_amp : energy[joule] = 100.0 J
+n : Integer = 2
+dij : Real = 1.0
Water Sensor
«Service»
: FW_SenseInterface
{protocol = "CoAP
{accepts: max-age}"}
Water Sensor
«Service»
: FW_SenseInterface
{protocol = "CoAP
{accepts: max-age}"}
«Device»
Water Sensor_2
{attachedTo = Street Pole_2}
+residualEnergy : total current[ampere] = 100.0 A
+b : equivalent binary storage capacity[bit] = 2.0 bit
+v_sup : voltage[volt] = 2.7 V
+i_sense : total current[ampere] = 25.0 A
+t_sense : time[millisecond] = 0.5
+e_elec : energy[joule] = 50.0 J
+e_amp : energy[joule] = 100.0 J
+n : Integer = 2
+dij : Real = 50.0

Figure 29. Water sensor devices with CoAP protocol configured to accept requests with max-age.

By executing the system model, it was possible to answer design questions related to lifetime and data freshness. However, there are still the availability and response time QoS requirements that are required by the service consumers (see Figure 17). Aiming to demonstrate the extensibility capability of our proposal, to verify availability and response time, we model such requirements as extension of SoaML4IoT..

# References

[1] Alfieri, A. et al. 2007. Maximizing system lifetime in wireless sensor networks. *European Journal of Operational Research*. 181, 1 (2007), 390–402. DOI:https://doi.org/10.1016/j.ejor.2006.05.037.

[2] Banks, J. et al. 2009. *Discrete-Event System Simulation*. Prentice Hall.

[3] Basha, E.A. et al. 2008. Model-based monitoring for early warning flood detection. *Proceedings of the 6th ACM conference on Embedded network sensor systems - SenSys '08* (New York, New York, USA, 2008), 295.

[4] Bass, L. et al. 2012. *Software Architecture in Practice*. Addison-Wesley.

[5] Bassi, A. et al. eds. 2013. *Enabling things to talk: Designing IoT solutions with the IoT Architectural Reference Model*. Springer Berlin Heidelberg.

[6] Brambilla, M. et al. 2012. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers.

[7] Buyya, R. and Dastjerdi, A.V. eds. 2016. *Internet of Things: Principles and Paradigms*. Elsevier.

[8] CoAP - RFC 7252 Constrained Application Protocol: 2016. *http://coap.technology/*. Accessed: 2018-01-01.

[9] Dietrich, I. and Dressler, F. 2009. On the lifetime of wireless sensor networks. *ACM Transactions on Sensor Networks*. 5, 1 (Feb. 2009), 1–39. DOI:https://doi.org/10.1145/1464420.1464425.

[10] Erl, T. 2007. *SOA Principles of Service Design*. Prentice Hall.

[11] Gubbi, J. et al. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*. 29, 7 (2013), 1645–1660. DOI:https://doi.org/10.1016/j.future.2013.01.010.

[12] Halgamuge, M.N. et al. 2009. An Estimation of Sensor Energy Consumption. *Progress In Electromagnetics Research B*. 12, (2009), 259–295. DOI:https://doi.org/10.2528/PIERB08122303.

[13] Harbinger Systems 2016. Webinar IoT Cloud Platforms and Middleware for Rapid Application Development.

[14] IEEE 802.15 WPAN$^{TM}$ Task Group 4e (TG4e): *http://www.ieee802.org/15/pub/TG4e.html*. Accessed: 2017-07-24.

[15] IEEE P2413 Working Group: 2015. *http://grouper.ieee.org/groups/2413/*. Accessed: 2015-08-03.

[16] ISO/IEC/IEEE 2011. *ISO/IEC/IEEE 42010:2011 - Systems and software engineering -- Architecture description*.

[17] Maltempo Veneto: violento temporale e grandinata a Padova, allagamenti: 2017. *http://www.meteoweb.eu/2017/09/maltempo-veneto-violento-temporale-e-grandinata-a-padova-allagamenti/959536/#Dr3DvKhrYhEAejSD.99*. Accessed: 2017-06-10.

[18] Microsoft Azure IoT Reference Architecture: *https://azure.microsoft.com/en-au/updates/microsoft-azure-iot-reference-architecture-available/*. Accessed: 2018-02-01.

[19] Montenegro, G. et al. 2007. Transmission of IPv6 Packets over IEEE 802.15.4 Networks

- RFC4944. IETF.

[20] OMG 2008. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification*.

[21] Ouni, S. and Ayoub, Z.T. 2013. Predicting Communication Delay and Energy Consumption for IEEE 802.15.4/zigbee Wireless Sensor Networks. *International Journal of Computer Networks & Communications*. 5, (2013).

[22] Overview of Internet of Things: *https://cloud.google.com/solutions/iot-overview*. Accessed: 2018-06-26.

[23] Perera, C. et al. 2014. Sensing As a Service Model for Smart Cities Supported by Internet of Things. *Trans. Emerg. Telecommun. Technol.* 25, 1 (2014), 81–93. DOI:https://doi.org/10.1002/ett.2704.

[24] Reussner, R.H. et al. 2016. *Modeling and Simulating Software Architectures -- The Palladio Approach*. The MIT Press.

[25] Rozanski, N. and Woods, E. 2011. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*.

[26] Saraswat, J. et al. 2012. Techniques to Enhance Lifetime of Wireless Sensor Networks: A Survey. *Global Journal of Computer Science And Technology Network, Web & Security*. 12, 14 (2012).

[27] Taivalsaari, A. and Mikkonen, T. 2017. A Roadmap to the Programmable World: Software Challenges in the IoT Era. *IEEE Software*. 34, 1 (Jan. 2017), 72–80. DOI:https://doi.org/10.1109/MS.2017.26.

[28] Tatibouët, J. et al. 2014. Formalizing execution semantics of UML profiles with fUML models. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 8767, (2014), 133–148. DOI:https://doi.org/10.1007/978-3-319-11653-2_9.

[29] The Constrained Application Protocol (CoAP) - RFC 7252: 2014. *https://tools.ietf.org/html/rfc7252*.

[30] Winter, T. et al. 2012. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks - RFC6550. IETF.

[31] Zanella, A. et al. 2014. Internet of Things for Smart Cities. *IEEE Internet of Things Journal*. 1, 1 (Feb. 2014), 22–32. DOI:https://doi.org/10.1109/JIOT.2014.2306328.

# Appendix A – SoaML4IoT Textual Semantics

This appendix provides the textual semantics of SoaML4IoT.

### A.1 IoT System

A cyber-physical solution composed of platforms, applications, networks, and devices.

**Extensions**

- None

**Generalizations**

- None

**Associations**

- attributes: Attribute [*]: list of attributes.

### A.2 Platform

A computer node.

**Extensions**

- Cloud
- Fog
- Device

**Generalizations**

- None

**Associations**

- attributes: Attribute [*]: list of attributes.
- networks: Network [1..*]: list of network connections by through the cloud node receive and transmit data.

### A.3 Cloud

Platforms located in the cloud provide a set of resources (i.e., processing, storage), which can be provisioned and paid on demand.

**Extensions**

- None

**Generalizations**

- Platform

**Associations**

- None

### A.4 Fog

Platforms that act as near- devices computing/storage resources or as bridges to connect the devices to the cloud.

**Extensions**

- None

**Generalizations**

- Platform

**Associations**

- None

**A.5 Device**

A platform attached to physical entities providing them sensing, actuating, computing, and communication capabilities.

**Extensions**

- None

**Generalizations**

- Platform
- Provider

**Associations**

- attachedTo: Physical Entity [0..1]: the physical entity that the device is attached to.

**A.6 Physical Entity**

Anything of the real world, since objects and cars to animals and human beings.

**Extensions**

- None

**Generalizations**

- None

**Associations**

- location: Location [0..1]: the geographic location.

**A.7 Location**

Geographic location.

**Extensions**

- None

**Generalizations**

- None

**Associations**

- None

**A.8 Provider**

The role of a participant that provides services.

**Extensions**

- Device

**Generalizations**

- Participant

**Associations**

- capability: Capability[1..*]: the capabilities performed by the provider participant.
- service: Service [1..*]: the services provided by the provider participant.

### A.9 Participant

Entities that provide or use services.

**Extensions**

- Consumer
- Provider

**Generalizations**

- None

**Associations**

- None

### A.10 Consumer

The role of a participant that consumes services.

**Extensions**

- Component

**Generalizations**

- Participant

**Associations**

- invokes: Request[1..*]: the requests performed by the consumer participant.

### A.11 Request

The invocation to a service.

**Extensions**

- None

**Generalizations**

- None

**Associations**

- service: Service[1..*]: the service that is invoked.

### A.12 Service

Value delivered to another through well-defined interface, and available to a community.

**Extensions**

- None

**Generalizations**

- None

**Associations**

- capability: Capability[1..*]: the capability exposed through the service
- attributes: Attribute [*]: list of attributes
- messageType: Message Type[1..*]: the data structure of messages
- specifiedIn: Service Description[1..*]: the specification of the service

### A.13 Service Description

The description of how the participants interacts to provide or use a service.

**Extensions**

- Simple Interface
- Service Contract

**Generalizations**

- None

**Associations**

- None

### A.14 Simple Interface

Specify a unidirectional service

**Extensions**

- None

**Generalizations**

- None

**Associations**

- None

### A.15 Service Contract

Specifies the roles each participant plays in the service – provider or consumer – and the choreography of the service, that is, what information is sent between the provider and consumer and in what order.

**Extensions**

- None

**Generalizations**

- None

**Associations**

- choreography: Choreography[1]: the choreography of the service contract

### A.15 Choreography

Specifies what information is sent between the provider and consumer and in what order.

**Extensions**

- None

**Generalizations**

- None

**Associations**

- None

### A.16 Capability

Functions required by stakeholders and provided by a participant through a service.

**Extensions**

- Task

**Generalizations**

- None

**Associations**

- None

### A.17 Services Architecture

Describes how participants work together for a purpose by providing and using services expressed in the service contracts.

**Extensions**

- None

**Generalizations**

- None

**Associations**

- participants: Participants: the participants of the architecture
- services: Services: the services of the architecture

### A.18 Task

A capability performed by a device.

**Extensions**

- Sense
- Actuate
- Transmit
- Receive
- Compute

**Generalizations**

- None

**Associations**

- participants: Participants: the participants of the architecture
- services: Services: the services of the architecture

### A.19 Sense

A task that aim to collect data from physical entities (e.g., car speed, building structural health) or the environment it is inserted into (e.g., room’s temperature or lighting level).

**Extensions**

- Task

**Generalizations**

- None

**Associations**

- None

### A.20 Actuate

A task that aim to affect the physical realm (e.g., turn on/off a heater, sound alarm).

**Extensions**

- Task

**Generalizations**

- None

**Associations**

- None

### A.21 Transmit

A task that aim to send data.

**Extensions**

- Task

**Generalizations**

- None

**Associations**

- None

### A.22 Receive

A task that aim to receive data.

**Extensions**

- Task

**Generalizations**

- Task

**Associations**

- None

### A.23 Compute
CPU processing capability of the device.

**Extensions**
- Task

**Generalizations**
- Task

**Associations**
- None

### A.24 Application
A set of specialized algorithms that request services and process data aiming at fulfilling its requirements.

**Extensions**
- None

**Generalizations**
- None

**Associations**
- component: Components: the components that compose the application
- requirement: Requirement[1..*]: the requirements the application aims to fulfill
- attribute: Attribute[*]: list of attributes

### A.25 Component
Software units that can be deployed within platforms.

**Extensions**
- None

**Generalizations**
- None

**Associations**
- component: Components: the components that compose the component
- platform: Platform[1]: the platform in which the component is deployed

### A.26 Requirement
A stakeholder’s desire that aims to be fulfilled by the application.

**Extensions**
- None

**Generalizations**
- None

**Associations**
- None

### A.27 Network
Set of nodes and links providing the communication path through the platforms receive and transmit data.

**Extensions**
- None

**Generalizations**
- None

**Associations**
- attribute: Attribute[*]: list of attributes

### A.28 Attribute
A property that may be useful for the architectural decision-making process.

**Extensions**
- None

**Generalizations**
- None

**Associations**
- unit: Unit of Measurement[1..*]: the unit of the property

### A.29 Unit of Measurement
Expresses the magnitude of a quantity.

**Extensions**
- None

**Generalizations**
- None

**Associations**
- expresses: Quantity Kind[1..*]: the quantity kind that the unit expresses

### A.30 Quantity Kind
A dimension or kind of quantity.

**Extensions**
- None

**Generalizations**
- None

**Associations**
- None