Simulate IoT Towards the Cloud-to-Thing Continuum Paradigm for Task Scheduling Assessments

José A. Barriga, José M. Chaves-González, Arturo Barriga, Pablo Alonso, and Pedro J. Clemente
University of Extremadura, Quercus Software Engineering Group (http://quercusseg.unex.es), Spain

ABSTRACT Aiming to optimise the performance of the computing layers of Internet of Things (IoT) systems, one of the most widespread techniques is the well-known task scheduling. Not only to develop but also to put task scheduling techniques in production, they have to be tested. Nevertheless, IoT systems are complex scenarios with high technological heterogeneity. Thus, testing task scheduling methods in the IoT context involves an investment of money, time and effort in acquiring devices, their configuration, deployment, etc. To avoid this, the system can be simulated and tests can be conducted through these simulations. Moreover, the underlying technical complexity of IoT systems can be reduced by increasing the abstraction level from which these systems are designed. Simulators based on model-driven development can help both to test and tackle the technological complexity of IoT systems. In this paper, a Domain-Specific Language based on SimulateIoT is proposed for the design, code generation and simulation of IoT systems for the assessment of task scheduling methods. Simulations include the generation and offloading of workflow-based tasks, the components required to handle these tasks, as well as the required resources to integrate the users' task scheduling methods in the simulations. All this, while providing an infrastructure based on the cloud-to-thing continuum paradigm on which to deploy and test these task scheduling environments, i.e., simulations can include the mist, edge, fog and cloud layers and the federation between them. In addition, a case study focused on an Industrial IoT (IIoT) system is illustrated to show the applicability of the proposed simulator.

KEYWORDS IoT, Model-driven development, Simulation, Task scheduling, Cloud-to-thing continuum.

1. Introduction

The Internet of Things (IoT) is being exploited in several areas such as smart-cities, home environments, agriculture, industry, intelligent buildings, etc.(Siow et al. 2018). In this regard, IoT applications can be very different from each other and therefore have different requirements and needs such as specific Quality of Service (QoS) (Samann et al. 2021) or Service-Level-Agreement (SLA) (Girs et al. 2020).

In order to satisfy these requirements, cloud computing emerged. Thus, supporting the rapid growth of users and applications, and providing them with elastic services such as Infrastructure-as-a-Service (IaaS), Platforms-as-a-Service (PaaS), and Software-as-a-Service (SaaS) with minimum resource consumption (Qian et al. 2009; Rashid & Chaturvedi 2019). However, due to the rapid growth of the IoT and the increasing demand for a better QoS, fog computing emerged (H. & V. 2021). Nearer of the edge/mist computing, although with fewer computing resources than the cloud (H. & V. 2021), this computing layer is able to provide better QoS to specific IoT applications and users such as IoV (Internet of Vehicles) (Yu et al. 2018) or IIoT (Industrial Internet of Things) delay-sensitive applications (Aazam et al. 2018). Thus, different computing layers (cloud, fog, edge and mist) coexist in the IoT providing different services to the system, from the cloud layer to the end devices (mist/IoT layer). Furthermore, the nodes that form each of these layers can be federated, acting as a single entity instead of isolated nodes, including nodes that belong to different computing layers conforming cloud-fog-edge

JOT reference format:
José A. Barriga, José M. Chaves-González, Arturo Barriga, Pablo Alonso, and Pedro J. Clemente. *SimulateIoT Towards the Cloud-to-Thing Continuum Paradigm for Task Scheduling Assessments*. Journal of Object Technology. Vol. 22, No. 1, 2023. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2023.22.1.a6

heterogeneous federations (Bittencourt et al. 2018; Kar et al. 2022; Mijuskovic et al. 2021). Consequently, the cloud-to-thing continuum paradigm emerges, which could be defined as the coordination of services and resources between the different computing layers mentioned above. Allowing data flow across cloud data centers, intermediary nodes like edge or fog nodes, and end-user devices, facilitating more efficient, responsive, and resilient computing solutions (Bittencourt et al. 2018).

While the IoT infrastructure was being developed and enhanced, different techniques for optimally managing their resources were also developed and proposed. One of the most widespread techniques is the well-known task scheduling (Arunarani et al. 2019a; Alizadeh et al. 2020). Task scheduling is often applied to distributed computing environments, such as IoT systems that rely on an architecture based on the above described cloud-to-thing continuum, where services are decomposed into a set of tasks which have to be processed by the computing nodes of this federation (Singh et al. 2017; Hosseinioun et al. 2022). Note that in this communication, task refers to an individual unit of work or a specific job that needs to be performed. This can include data collection, data processing, control commands, or other computational processes. In this context, task scheduling proposals aim to schedule the processing of these tasks, thus optimising the system and the use of system resources from different perspectives, e.g., there are proposals that aim at reducing the makespan (time required to process a task) (S. Gupta et al. 2022; Al-Maytami et al. 2019), optimising the system energy consumption (Ding et al. 2020; Sandhu et al. 2021), the cost of task processing (Shu et al. 2021; Gazori et al. 2020), etc.

However, testing is required during the development stage of these proposals, besides these tests have to be isolated as otherwise, they could affect the system in production. Furthermore, novel task scheduling proposals are often compared with existing ones in order to better determine the strengths and limitations of these proposals. Consequently, this implies an investment of money, time and effort in the acquisition of devices, their configuration, deployment, etc. However, these IoT systems can be simulated, and the task scheduling proposals deployed, tested and analysed in these simulated systems, thus avoiding the aforementioned costs in device acquisition, configuration, etc. For instance, the study (Hosseinioun et al. 2022) reports that 90% of the task scheduling proposals applied to fog computing environments use simulators for the aforementioned purposes.

Note that in the context of this communication, simulation refers to the procedure of creating and deploying a digital replica of a real-world system, without having to materialise it physically. On the other hand, testing denotes the application of specific scenarios or conditions to a software component or a system. The purpose of these tests is to reproduce possible situations the system might encounter and then observe and analyse the responses of the system under these conditions. Therefore, in this communication, the intention is to facilitate the performing of these tests by means of simulations.

On the other hand, as described above, IoT systems present a high technological heterogeneity and a complex infrastructure. However, increasing the abstraction level from which the IoT systems are designed helps to tackle the underlying technological complexity. In this regard, model-driven development (MDD) can help to both reduce the IoT application time to market and tackle the technological complexity to develop IoT applications (Barriga et al. 2023).

In this regard, SimulateIoT (Barriga et al. 2021) is a simulator based on model-driven development that makes it possible to design and simulate IoT systems. The IoT systems designed with SimulateIoT can include different IoT nodes such as cloud, fog, or edge nodes and multiple computing services such as Complex Event Processing (CEP) services, publish/subscribe services or storage services. However, SimulateIoT is not able to simulate a suitable IoT infrastructure to test task scheduling proposals.

In this communication, SimulateIoT (Barriga et al. 2021) is extended towards the cloud-to-thing continuum paradigm for task scheduling assessments. Thus, the simulator proposed includes the main concepts of task scheduling (federations, tasks generation and processing, etc.) to model, generate and simulate IoT systems with the required infrastructure to support task scheduling, allowing users to deploy, test, compare and analyse their task scheduling proposals.

Note that the content described in this communication only focuses on describing new contributions or features added as part of the extension. Therefore, all the content in this communication is novel, although some references to SimulateIoT are included where necessary to describe some aspects of the new contributions.

The main work contributions are the following:

– The extension of the metamodel of SimulateIoT towards task scheduling and the cloud-to-thing continuum. This extension provides users with a metamodel that enables the design of models based on IoT systems with task scheduling capabilities. In addition, this extended metamodel enables users to model cloud-to-thing continuum infrastructures on which to deploy and test these task scheduling features.
– The extension of the model-to-text (M2T) transformations of SimulateIoT towards the task scheduling and the cloud-to-thing continuum. This extension ensures that the M2T transformations required to generate and simulate the systems modelled conform to the extended metamodel.
– The extension of the concrete syntax of SimulateIoT towards task scheduling and the cloud-to-thing continuum. This extension enables users to design, in a graphical manner, the IoT system models conform to the extended metamodel.
– A case study to validate and show the applicability of the proposal.

The rest of the paper is structured as follows. In Section 2, we give an overview of existing IoT simulation approaches centred on both low-level and high-level IoT simulation environments. Next, Section 3 gives a holistic view of the task scheduling and cloud-to-thing continuum model envisaged and the extended simulator. Next, Section 4 presents the extended simulator taking into account the design and implementation stages including the new metamodel and the graphical editor. In Section 5, the M2T transformations from the extended simulator models to code are addressed. In Section 6 the simulation outputs, possible tests and assessments are illustrated. In Section 7, a case study to show the applicability of the extended simulator is presented. Finally, Section 8 concludes the paper.

2. Related Works

A large amount of IoT simulators are available in the literature. However, only a few of them allow the simulation of IoT systems with task scheduling features. Below, those most relevant to the proposal carried out in this communication are addressed. Note that the review of these first related works is mainly focused on the task scheduling features that the simulators are able to simulate.

iFogSim (H. Gupta et al. 2017) is one of the most popular IoT simulators in literature. It is an extension of CloudSim (Calheiros et al. 2011), although it is focused on the simulation of the fog layer of the system. It is able to simulate the cloud, fog and edge layer of an IoT system, simulating hardware features, such as the CPU or memory of each device, network features such as the delay and bandwidth between devices, federation between the fog and the cloud nodes, etc. As for task scheduling, it facilitates the design of workflows using DAGs (Directed Acyclic Graphs), which are mathematical structures suitable for representing them. Moreover, iFogSim allows the simulation of the processing of tasks (those related to the above-mentioned workflows). In this way, users can design applications and specify the tasks they offload during the simulation. However, this simulator does not provide knowledge about the availability (status) of nodes, links between nodes or the tasks' waiting time, i.e. the time that a task needs to wait until its processing.

WorkflowSim (Chen & Deelman 2012) is another simulator based on CloudSim, although it is focused on simulating the workflow scheduling. In this way, this tool allows users to simulate the processing of these workflows, including task processing fails, to test several algorithms and policies (although it does not include resources focused on allowing the integration of users' proposals), and all the elements included in CloudSim. Although this tool is interesting because is mainly focused on task scheduling purposes, it was published in 2012 and is no longer maintained, so nowadays it is deprecated. Additionally, it does not support current elements related to IoT systems such as edge nodes, fog nodes, federations between nodes, etc.

YAFS (Lera et al. 2019) is a simulator whose main purpose is to simulate the deployment and execution of applications in a Cloud-fog IoT environment. In this way, users can analyse which is the best allocation of applications and resources strategies, the best network routing strategies for the offloaded data of the deployed applications and also the best scheduling strategy. In order to allow users to model the tasks that constitute an application, DDFs (Distributed Data Flows) are used, which are similar to workflows and DAGs. However, this simulator does not include some relevant data about task processing such as the time required to process a specific task or the status of the links that inter-connect each node of the simulation.

ScSF (Rodrigo et al. 2018) is a simulation tool that focuses only on task scheduling purposes. So, a positive aspect of this tool is that it is not only focused on IoT systems but focuses on any system with task scheduling needs, offering its utilities to a broad spectrum of users. But on the contrary, without offering specific concepts that could be found in an IoT system. In this way, ScSF includes a set of modules that takes as input a system model (processors) and the workflows to schedule. Then, it reports as output the scheduling of each workflow in the processor system taken as input.

These IoT simulators allow the simulation of IoT systems as well as the performance analysis of task scheduling algorithms running on top of these simulations. The most similar related work to the proposal presented in this paper is WorkflowSim. However, each simulator has its own advantages and disadvantages for specific use cases. Consequently, a thorough analysis tailored to the user's specific needs is necessary to select the most suitable simulator for their proposal.

So, in order to compare simulators several quality indicators could be taken into account such as the range of features or the types of environments it can simulate, its reliability, speed, flexibility or its learning curve. Following, several distinguishing features that set apart the proposal presented in this paper from WorkflowSim and the other simulators discussed in this section are highlighted.

1. The proposed simulator is a hybrid simulator/emulator, i.e. it generates and deploys the real architecture of the modelled IoT system (emulation), and simulates some processes related to task scheduling such as the generation of tasks, their offloading to the system or their processing. The rest of the simulators described above base their results on mathematical models. Note that, with the term mathematical model we refer to a set of algorithms that simulate the behaviour of a concrete physical device or system. However, they do not deploy real component architectures on which to simulate and test task scheduling processes. Although mathematical models have been widely and successfully used over time, relying only on mathematical models could affect the trustworthiness of the simulation and testing results due to the inherent limitations of these models in accurately reflecting reality, especially when dealing with complex systems (Sterman 2002; Saltelli & Funtowicz 2014; Fisher et al. 2019). For this reason, trustworthiness is one of the open challenges in the field of IoT simulation (Mishra et al. 2012; Gluhak et al. 2011; Chernyshev et al. 2018). However, by emulating part of the IoT system this gap can be mitigated (McGregor 2002; Erazo & Liu 2013).

2. The proposed simulator is based on the MDD, addressing the design of the simulations from a high level of abstraction. It focuses on the high-level concepts of the IoT and task scheduling systems domain and their relationships, rather than on low-level details.

3. The proposed simulator is updated to current simulation

and testing needs, e.g., it allows the federation of nodes regardless of the computing layer they belong to. Thus, allowing cloud-fog-edge federations (cloud-to-thing continuum infrastructure).

Finally, Table 1 shows a summary of the main features of each related work included in this section together with the proposed SimulateIoT extension. The meaning of each column in Table 1 is described below.

– Simulator: Name of the simulation tool or framework.
– Task Modelling: Indicates the type of task modelling supported by the simulator (e.g., DAG-based workflows).
– Failure Modelling: Specifies if failure modelling is supported or not.
– Task Optimisation: Indicates if task optimisation techniques are supported.
– Task Scheduling: Specifies if task scheduling capabilities are available.
– Task Queueing: Indicates if task queueing is supported.
– Network Delay: Describes the model or type of network delay that the simulator supports.
– Network Bandwidth Model: Specifies the model or approach used to simulate network bandwidth.
– Node Federation: Indicates if the simulator supports federations of edge, fog and/or cloud nodes.
– Underlying Simulation Model: Describes the underlying simulation model or approach used.
– Provides Integration API: Specifies if the simulator provides an integration API for external systems such as user proposals (e.g. task scheduling proposal).
– Focus: Describes the main focus or application domain of the simulator.
– Case Use Definition: Specifies the format or method for defining use cases in the simulator.

As for the notation used in Table 1, *Limited* means that the simulator provides some options and does not allow the user to integrate their own proposals. On the other hand, *User-proposal* means that the simulator provides the flexibility to incorporate user-suggested solutions or strategies. Note that these concepts, i.e., *Limited* and *User-proposal*, are used in several columns, where they mean the same but in the context of the respective column.

Finally, some findings related to Table 1 are highlighted below. Of the four simulators included, only YAFS, ScSF, and SimulateIoT allow users to integrate their custom solutions in terms of task scheduling or optimisation algorithms. The lack of this feature significantly compromises the practical utility of a simulator. Both SimulateIoT and iFogSim support the modelling and simulation across the four computing layers: mist, edge, fog, and cloud. Yet, SimulateIoT stands out as the only one that enables federation among these layers, making it the sole simulator capable of simulating the nuances of current cloud-to-thing IoT system infrastructures. In terms of network modelling, only iFogSim and SimulateIoT offer modelling of delay, bandwidth, and unidirectional links. Note that such features are key in any simulator oriented to the simulation of IoT systems with task scheduling capabilities. In this context, it should be noted that iFogSim is not specifically designed for testing task scheduling systems (thus lacking several task scheduling simulation capabilities). So, only SimulateIoT is specifically oriented towards task scheduling testing and provides the capabilities to model these networking features.

3. The Cloud-to-Thing Continuum and Task Scheduling Model, and the Resulting Simulator

This section aims to introduce the proposed simulator as well as the systems that it can simulate. Furthermore, as this work is an extension of SimulateIoT (Barriga et al. 2021), the aim of this section is also to outline the new features added as part of this extension. Thus, differentiating between what was previous work (SimulateIoT) and what is new. Later, with the aim of conveying a "big picture" of the work carried out in this communication, the extended SimulateIoT capabilities based on this model are shown by means of a generic simulation overview.

SimulateIoT and therefore the proposed simulator, are based on the MDD, which is an emerging software engineering research area that aims to develop software guided by models based on the metamodeling technique. Metamodeling is defined by four model layers (see Figure 1). Thus, a model (M1) conforms to a metamodel (M2). Moreover, a metamodel conforms to a metametamodel (M3) which is reflexive (Atkinson & Kuhne 2003). So, a metamodel defines the domain concepts and relationships in a specific domain in order to model partial reality. A model (M1) defines a concrete system that conforms to a metamodel. Then, from these models, it is possible to generate totally or partially the application code (M0 - code) by M2T transformations (Sendall & Kozaczynski 2003). Thus, high-level definitions (models) can be mapped by M2T transformations to specific technologies (target technology). Consequently, the software code can be generated for a specific technological platform, improving technological independence and decreasing error proneness.

Therefore, in order to extend SimulateIoT towards the cloud-to-thing continuum paradigm and to the task scheduling, it is required to work in these metamodelling layers. Specifically, it is required to extend: 1) The metamodel or abstract syntax (M2), 2) The graphical concrete syntax, the element that makes it possible to graphically design models (M1) from the metamodel (M2) and 3) model-to-text transformations, the element that carries out the code generation (M0) from models (M1).

Nevertheless, prior to extending SimulateIoT (Barriga et al. 2021) towards the cloud-to-thing continuum and task scheduling, it is crucial to identify the main concepts of these systems. This characterisation results in a conceptual model, forming the backbone of this study. Aiming to introduce the work carried out in this communication, this model is subsequently introduced. Moreover, an outline of the enhanced SimulateIoT, the resulting simulator after extending it towards this model, is also provided.

Table 1 Feature summary of related works and presented proposal.

Simulator	Task Modelisation	Failure Modelling	Task Optimisation	Task Scheduling	Task Queueing	Network Delay	Network Bandwidth Model	Node Federation	Underlying Simulation Model	Provides Integration API	Focus	Case Use Definition
IFogSim	DAG-based DDFs	Not modelled	Not modelled	Not modelled	Not modelled	Yes	Unidirectional links (different delay depending on origin and destination)	Supports Fog and Cloud federations	Mathematical model	No	IoT systems across all layers (Mist, Edge, Fog, and Cloud)	Graphical Interface
WorkflowSim	DAG-based Workflows	Yes, several types	Yes, limited	Yes, limited	Yes	Yes	Not modelled	Not modelled	Mathematical model	No	Cloud	API
YAFS	DDFs	Yes, network failures model	Not modelled	Yes, user proposal	Yes	Yes	Bi-directional links (same delay regardless of origin or destination)	Supports Fog and Cloud federations	Mathematical model	Yes	IoT systems at Fog and Cloud layers	JSON
ScSF	Workflows	Not modelled	Yes, user proposal	Yes, user proposal	Yes	Not modelled	Not modelled	Not modelled	Mathematical model	No	Processing Environments	API
SimulateIoT	DAG-based Workflows	Yes, manually induced	Yes, user proposal	Yes, user proposal	Yes	Yes	Unidirectional links (different delay depending on origin and destination)	Supports heterogeneous federations across Edge, Fog, and Cloud	Mathematical model and emulation layer (Infrastructure)	Yes	IoT systems across all layers (Mist, Edge, Fog, and Cloud)	Graphical Interface

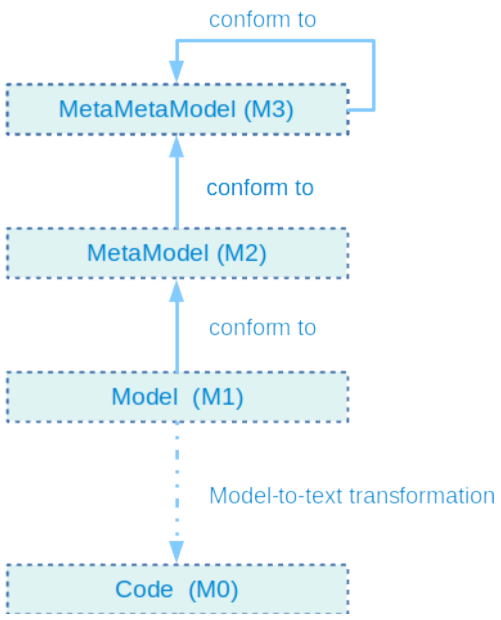


Figure 1 The four layers of metamodeling. In SimulateIoT (Barriga et al. 2021): a) M3 is Ecore, b) M2 is SimulateIoT Metamodel c) M1 is a model conforms to SimulateIoT Metamodel and d) Code is generated using the M2T transformations defined in SimulateIoT approach.

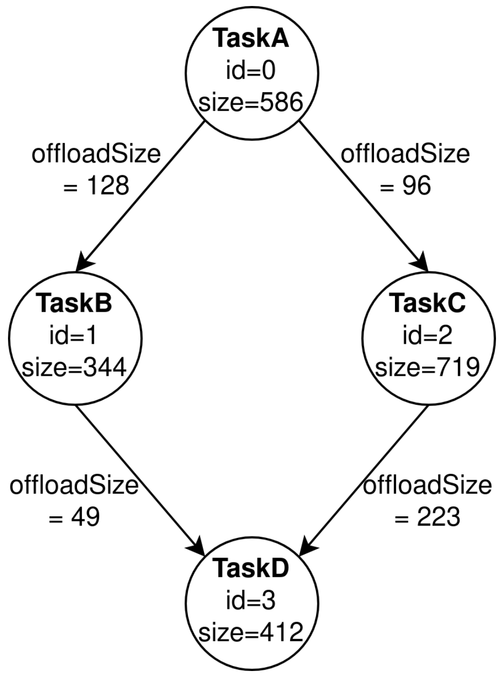


Figure 2 Graphical representation of a workflow.

3.1. The Proposed Cloud-to-Thing Continuum and Task Scheduling Model

This section addresses the conceptual model on which the SimulateIoT extension is based. It provides an introduction to the key concepts that constitute this model: `Task`, `Task App`, `Task Node`, `Networking Node`, `Task Processor` and `Task Scheduler`.

3.1.1. Task Task scheduling systems are based on tasks, i.e. in this kind of environment, there are nodes that generate, offload (to the rest of the system), schedule and process tasks. In this communication, tasks are included by means of workflows (Wu et al. 2015; Arunarani et al. 2019b), as is common in literature (Yao et al. 2021; Asghari et al. 2021; NoorianTalouki et al. 2022; Ahmad et al. 2021). So, users can define tasks by means of workflows that the nodes designed for these purposes will generate, offload, schedule or process.

Figure 2 shows a graphical representation of a workflow. This workflow represents four tasks, `Task A`, `Task B`, `Task C` and `Task D`. In this workflow, each node (circle) represents a task and each edge represents the dependency between these tasks.

Concerning task dependency, a dependent task cannot be processed until all other predecessor tasks have been processed, e.g., in the workflow shown in Figure 2, `Task B` and `Task C` cannot be processed until `Task A` has been processed. On the other hand, `Task D` cannot be processed until `Task B` and `Task C` have been processed. This is because once the tasks are processed, they can generate results that may be required by subsequent tasks in the processing pipeline. These results are the basis of the dependency between tasks, represented as relationships (edges) between nodes (tasks) in workflows (see Figure 2).

In terms of the attributes of a task, in this communication, each task has a name (e.g. `Task A`), an `id` and a `size` (bytes). Besides, each edge has a source task, a target task and a `offloadSize` which represent the size (bytes) of the data that have to be transmitted from the source task, once processed, to the target task, i.e. from the node that processes the source task to the node that processes the target task. Note that the `offloadSize` attribute represents the offload size (bytes) of the processing results of a task.

3.1.2. Task App In task scheduling scenarios, it is common to deploy applications that provide services at the fog and cloud layers. These applications are the ones that generate and offload tasks (the services they provide are decomposed into tasks or sets of tasks, i.e. workflows). In this model, these applications are included by means of the concept `Task App`, which can be deployed in the different nodes of the fog and cloud layers.

3.1.3. Task Node Traditionally, the edge and mist layers have had a different role than the fog and cloud layers. While the fog and cloud layers have had a role of providing computing resources or services to the end devices of the system (mist and edge layers), the mist and edge layers were constrained in terms of hardware and were limited to consuming these resources and services.

However, there are currently some end devices that do not face the aforementioned hardware constraints (such as mobile phones, personal computers, etc.). Therefore, their use does not have to be limited to consuming the services and resources of the fog and cloud layers but can help these layers in the provision of these services and resources to the rest of the system. Besides, in some situations it provides a better QoS than fog and cloud layers, since these devices are in the mist or edge layer itself and therefore close to the rest of the end devices. So, they are able to provide better latency, request-response time, etc. This new paradigm is called cloud-edge computing (Pan & McElhannon 2018).

In this context, the `Task Node` is designed to include in the task scheduling model this new paradigm of federation between computing layers. Thus, `Task Nodes` are conceived as nodes that belong to the edge and mist layers of the system but can be federated with cloud and fog nodes, thus providing task execution services to `Task Apps` and being able to process the tasks they generate.

On the other hand, as could be edge or mist devices that generate tasks requiring their processing, as `Task Apps`, `Task Nodes` are also designed to generate and offload tasks.

3.1.4. Networking Node

Task scheduling is frequently implemented in environments that operate on a cloud-to-thing continuum infrastructure. In this kind of system, nodes can be federated, acting as a single entity rather than isolated nodes. Federations are addressed in this model by means of `Links`, i.e. the connections between the different nodes that belong to a federation.

The `Networking Node` is the component responsible for managing these links. This model envisages a `Networking Node` for each federated (linked) node. Thus, each `Networking Node` handles the links whose source is the node where the `Networking Node` is integrated. Moreover, links are designed as unidirectional. Consequently, two links are needed to allow two components to interact with each other.

On the other hand, SimulateIoT is a hybrid simulator/emulator of IoT systems. So, SimulateIoT simulations have to be deployed over a real (or virtualised) network. Thus, without the need to simulate the latency and bandwidth of links. However, as aspects such as delay and bandwidth among nodes are critical aspects in task scheduling systems (Jamil et al. 2022), users could require, for testing purposes, specific latency and bandwidth between nodes. Configuring the network where simulations will be deployed could be a tedious, error-prone and costly task. Thus, this model also envisages the possibility to specify the delay and bandwidth of the aforementioned links, being the `Networking Node` the component which will ensure that these networking aspects are met during simulation.

Finally, note that this model envisages fully connected federations, i.e. all nodes belonging to a federation are connected to each other.

3.1.5. Task Processor

The `Task Processor` is the component that performs the processing of tasks. So, the `Task Processor node` is integrated at the deployment (of the simulation) stage into those edge (*Task nodes*), fog and cloud nodes that are modelled by the user to provide task processing services to the rest of the system.

Note that to suitably simulate the processing of tasks, this model envisages the possibility of assigning hardware resources to each node with processing capabilities, i.e. to the `Task Processors`. Thus, users can model aspects related to the hardware of each node, such as their CPU or RAM.

3.1.6. Task Scheduler

This component is the most relevant since it is where the simulator will integrate users' task scheduling proposals, as described in the following sections. Thus, the `Task Scheduler` is the node that receives and schedules (by means of the users' task scheduling proposal), the tasks offloaded to the system.

Since task scheduling algorithms can use several data sources and data types as input to perform their schedules (Bansal et al. 2022), this component is designed to provide users' proposals with resources to request data from several nodes of the simulation. Note that users' proposals can interact with these resources and therefore with the rest of the simulation by means of a REST API (which belongs to the aforementioned provided resources). Thus, the integration of users' proposals with the rest of the simulation is simplified.

This API allows users' proposals to perform four main requests: 1) request the offloaded tasks for their scheduling, 2) request data such as the current delay or available bandwidth between each node (links status), 3) request data related to the hardware usage (CPU, RAM, etc.) of each node, and 4) request the return of the scheduled tasks (once scheduled) to the system for their processing.

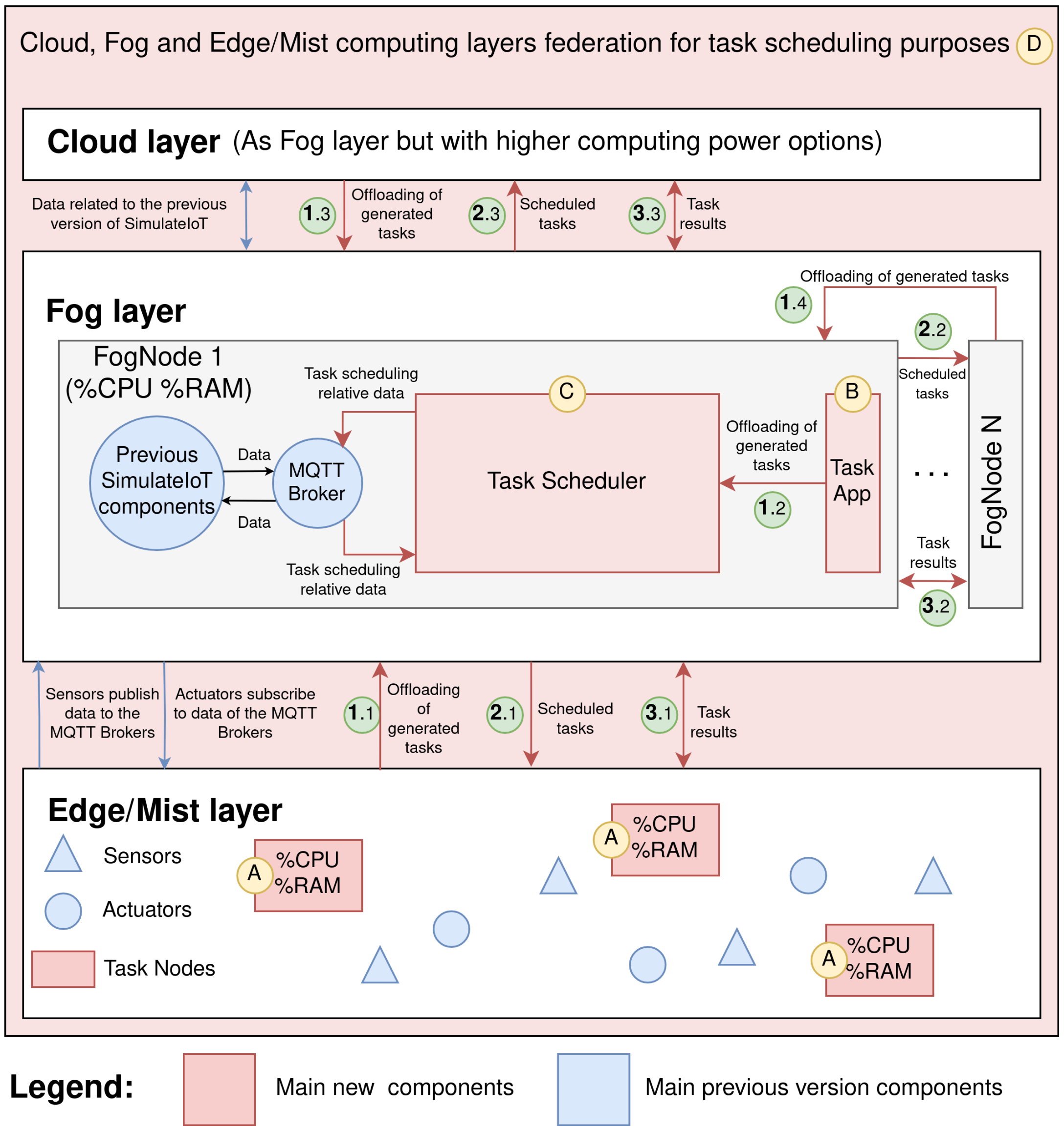
Note that in this model, these requests are limited to the nodes that belong to the same federation, i.e. those nodes that belong to different federations can not interact. On the other hand, this model envisages one `Task Scheduler` per federation. However, the `Task Scheduler` of each federation can integrate a different user scheduling proposal.

Thus, by extending SimulateIoT towards this task scheduling model, users will be able to deploy, analyse, compare, etc. their task scheduling proposals on a simulated IoT system without the need for investment in device acquisition, configuration and deployment. In Section 3.2 that follows, an overview of the resulting simulator after extending it towards this model is provided.

3.2. Overview of the Resulting Simulator

Figure 3, provides a representation of a generic simulation deployment using the extended version of SimulateIoT. This figure also distinguishes between the components developed in this work (marked in red) and those inherited from the original SimulateIoT (marked in blue). The significant enhancements, those related to the concepts belonging to the model defined in Section 3, are explained below in the context of Figure 3. Note that to facilitate referencing the concepts depicted in Figure 3 within the text, the corresponding labels included in the figure will be used (e.g. $\textcircled{1}$).

Firstly, tasks are generated and offloaded to the system by the `Task Nodes` $\textcircled{A}$ and the `Task Apps` $\textcircled{B}$. The offloading of these tasks is represented in Figure 3 by $\textcircled{1.1}$ offloading performed by `Task Nodes` to the fog layer; $\textcircled{1.2}$ and $\textcircled{1.3}$ offloading performed by `Task Apps` deployed on fog nodes to the fog layer; $\textcircled{1.4}$ offloading performed by `Task Apps` deployed on



Cloud, Fog and Edge/Mist computing layers federation for task scheduling purposes D
Cloud layer (As Fog layer but with higher computing power options)
Data related to the previous version of SimulateIoT
1.3 Offloading of generated tasks
2.3 Scheduled tasks
3.3 Task results
Fog layer
Offloading of generated tasks
1.4
2.2
FogNode 1 (%CPU %RAM)
Task scheduling relative data
C
Task Scheduler
Offloading of generated tasks
B
Task App
1.2
Scheduled tasks
. . .
FogNode N
Task results
3.2
Previous SimulateIoT components
Data
MQTT Broker
Data
Task scheduling relative data
Sensors publish data to the MQTT Brokers
Actuators subscribe to data of the MQTT Brokers
1.1 Offloading of generated tasks
2.1 Scheduled tasks
3.1 Task results
Edge/Mist layer
Sensors
Actuators
Task Nodes
A %CPU %RAM
A %CPU %RAM
A %CPU %RAM
Legend:
Main new components
Main previous version components

Figure 3 A generic simulation generated by using M2T transformations from a model defined with the proposed simulator.

cloud nodes to the fog layer.

Note that in Figure 3 all the tasks are offloaded to the fog layer as `FogNode 1` is the node where the `Task Scheduler` Ⓒ is deployed.

Once tasks have been offloaded, they are scheduled by the *Task Scheduler* Ⓒ and sent back to the system for their processing (2.1), (2.2) and (2.3).

Then, the `Task Processor` of each fog, cloud and `Task Node`, which is not directly represented in Figure 3, carries out the processing of each task and returns the result to the node that is waiting for it due to the dependency between tasks (3.1), (3.2) and (3.3).

Note that during the simulation, the generation, offloading, scheduling, and processing of each task by each node occur concurrently and adhere to the user-defined design (model).

Finally, it should be highlighted that the behaviour of each node and the overall simulation exhibit a higher level of complexity compared to what is depicted in this overview. For the sake of clarity, there are interactions and concepts that, as in the case of the `Task Processor`, are not directly depicted in Figure 3. For instance, the `Networking Node` or the interactions between the `Task Scheduler` and the nodes belonging to its federation Ⓓ to gather/share their status (available CPU and RAM).

Further details are addressed in following Sections 4 and 5.

4. Extensions of Metamodel and Concrete Syntax

The proposed simulator, as an MDD approach, is composed of three main elements: 1) metamodel or abstract syntax, 2) graphical concrete syntax and 3) model-to-text transformations. This section describes the proposed simulator metamodel and concrete syntax.

4.1. Metamodel Extensions

A Metamodel captures the concepts and relationships in a specific domain in order to model partially reality (Selic 2003). Then, it is possible to design models conforming to this Metamodel. These models can be used to generate the total or partial application code. Thus, the software code could be generated for a specific technological platform, improving its technological independence and decreasing error proneness.

The SimulateIoT metamodel (Barriga et al. 2021) gathers the core concepts and relationships related to the IoT domain, including elements such as sensors, actuators, edge nodes, fog nodes, cloud nodes, databases, complex-event processing services, data definition, topics, message brokers, etc. However, it has not enough expressiveness to simulate IoT systems with task scheduling capabilities and with a cloud-to-thing continuum infrastructure. Therefore, the metamodel of the proposed simulator is an extension of the SimulateIoT metamodel with enough expressiveness to define these kinds of IoT systems (entities and services described in Section 3.1).

Figure 4 shows an excerpt of the proposed simulator metamodel. Note that the new classes and relationships included are numbered and highlighted in blue colour. Besides, note that Figure 11 (Appendix A) shows the complete metamodel, with the elements relating to the extension carried out also highlighted in blue.

This section does not aim to describe how these components work internally, which is addressed in Section 5, where M2T transformations are addressed. Finally, note that in this section, to better describe the elements of the metamodel, the numerical labels shown in Figure 4 are used below as references in the text. These references are used by means of the expression `[class name]` ⓧ, where x is the label associated with the `[class]` in Figure 4.

In order to extend the SimulateIoT metamodel towards task scheduling and the cloud-to-thing continuum, first of all, the task generation related components, i.e. the Task Node and the Task App, have been included in the metamodel by means of the classes `TaskNode` ③ and `TaskApp` ② respectively. The workflow concept has also been added by means of the `Workflow` class ① and related to the two previously mentioned classes with the aim of allowing the user to model which workflow will be generated by each modelled Task Node and Task App during the simulation. Note that each workflow will be generated every period of time, which can be specified by the `generation_period` (seconds) attribute.

To allow the user to model the different workflows that will be generated, in addition to the `Workflow` class, the `Task` $\textcircled{1.1}$ and `Edge` $\textcircled{1.2}$ classes have also been included. Thus, the `Task` class represents a workflow node and the `Edge` class represents the dependency between these tasks. Note that in these classes the user can also model the size of each task (`size` attribute) and the size of the processing results of each task (`offload_size` attribute).

On the other hand, the metamodel is extended from the `Hardware_specification` ⑤ class to allow the user to model in a more detailed manner the hardware resources of each modelled node. To this end, this class is related to the `CPU` $\textcircled{5.1}$ and `RAM` $\textcircled{5.2}$ classes, also included as part of this extension to allow the user to model these hardware aspects for each node.

Finally, the metamodel is extended with the `Federation` ④ class, which allows the user to federate the different nodes of the modelled system. To this end, federations are composed of `Links` $\textcircled{4.1}$, a class that allows modelling the characteristics of the different links between the federated nodes. Particularly, the classes `Delay_specification` $\textcircled{4.2}$ and `Bandwidth_specification` $\textcircled{4.3}$ allow the user to model these characteristics for each link.

Extending the metamodel of SimulateIoT with these classes and relationships, the concepts related to the task scheduling model defined in 3.1 are gathered by the metamodel. Therefore, the required expressiveness to model IoT systems with task scheduling capabilities is achieved.

4.2. Graphical Concrete Syntax and Validator Extensions

Model-driven development allows designing models conforming to a metamodel by means of concrete syntax. Concrete syntax refers to the specific notation used to depict instances of a model. The concrete syntax can visually or textually represent the abstract notions and relationships that are defined within the metamodel. It could be defined as graphical concrete syntax (e.g. using GMF, Eugenia or Sirius) or textual concrete syntax (e.g. using xText). In our approach the Eugenia tool (Kolovos et al. 2015) is used and it makes it possible to generate a graphical concrete syntax conforming to a metamodel.

A graphical syntax is often more intuitive and easier to comprehend at a glance, especially when dealing with complex systems. It provides a high-level overview of a system, which facilitates an understanding of the structure and relationships within the system. However, graphical syntax can become more difficult to manage in terms of automated processing.

In contrast, the textual syntax can be seamlessly processed and integrated with other systems. However, a textual syntax can be more difficult to understand and navigate, especially for individuals who are not familiar with the specific language or notation. It may require more time to interpret and does not provide a visual overview of the system.

Although, both concrete syntax could be developed to define models conform to the metamodel proposed.However, in view of the above, given the complexity of the concepts to be represented, the models to be designed (IoT systems) and the nature of the resulting tool (a simulator), it has been considered that a graphical syntax is more appropriate than a textual one. In addition, the graphical concrete syntax generated for the proposed simulator metamodel is an extension of the graphical concrete syntax defined in SimulateIoT, which is based on Eclipse GMF (Graphical Modeling Framework) and EMF (Eclipse Modeling Tools). Figure 5 shows an excerpt from this graphical editor.

It helps users to improve their productivity allowing not only defining models conforming to the proposed simulator metamodel but also their validation. In this respect, metamodels can be extended with constraints based on the Object Constraint Language (OCL) (OMG 2012). OCL is a declarative language developed by the Object Management Group (OMG) for describing rules applicable to the models designed by users conforming

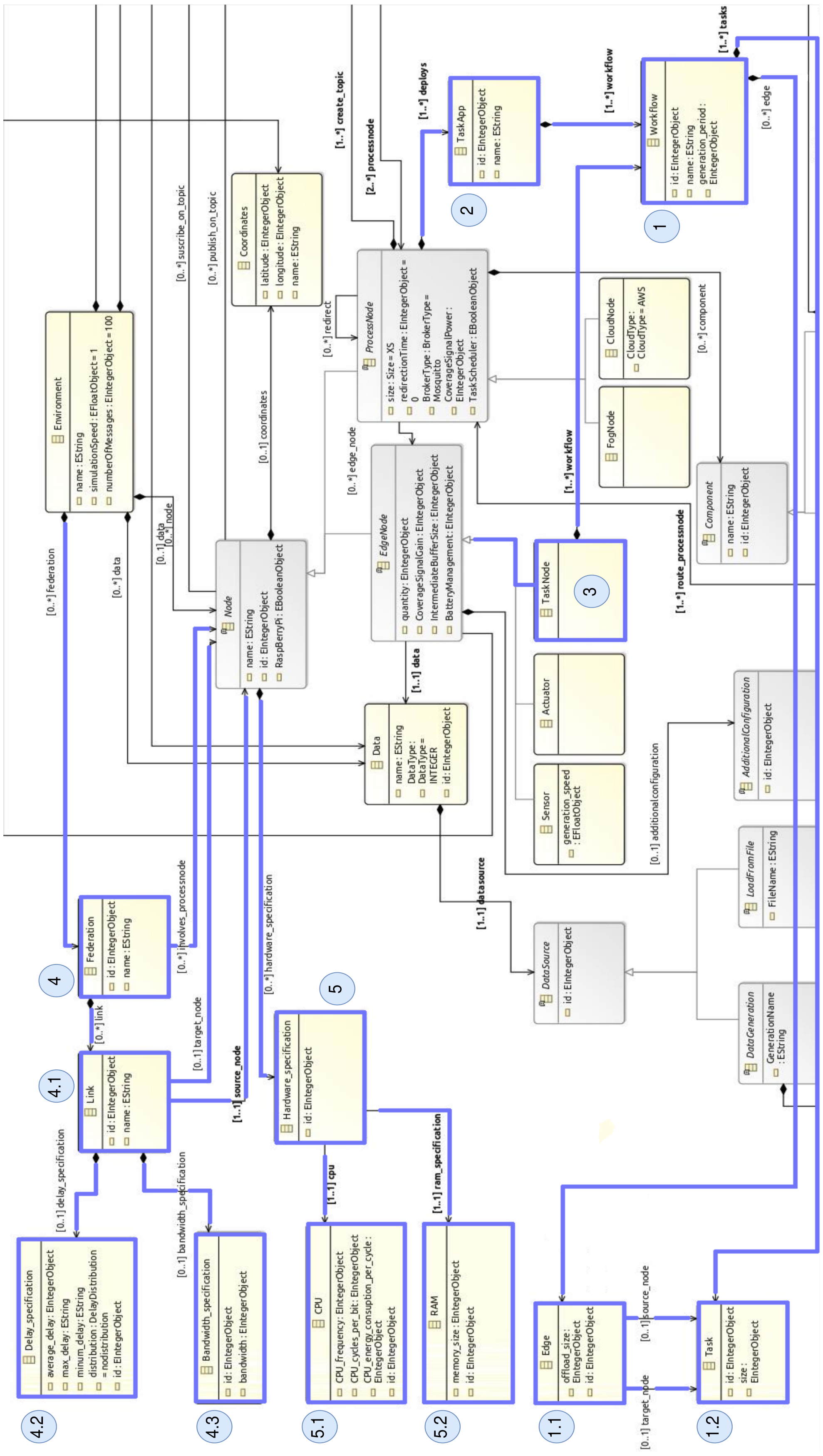


Figure 4 Excerpt of the proposed simulator Metamodel focusing on the task scheduling concepts. The complete metamodel can be found in Appendix A in Figure 11.

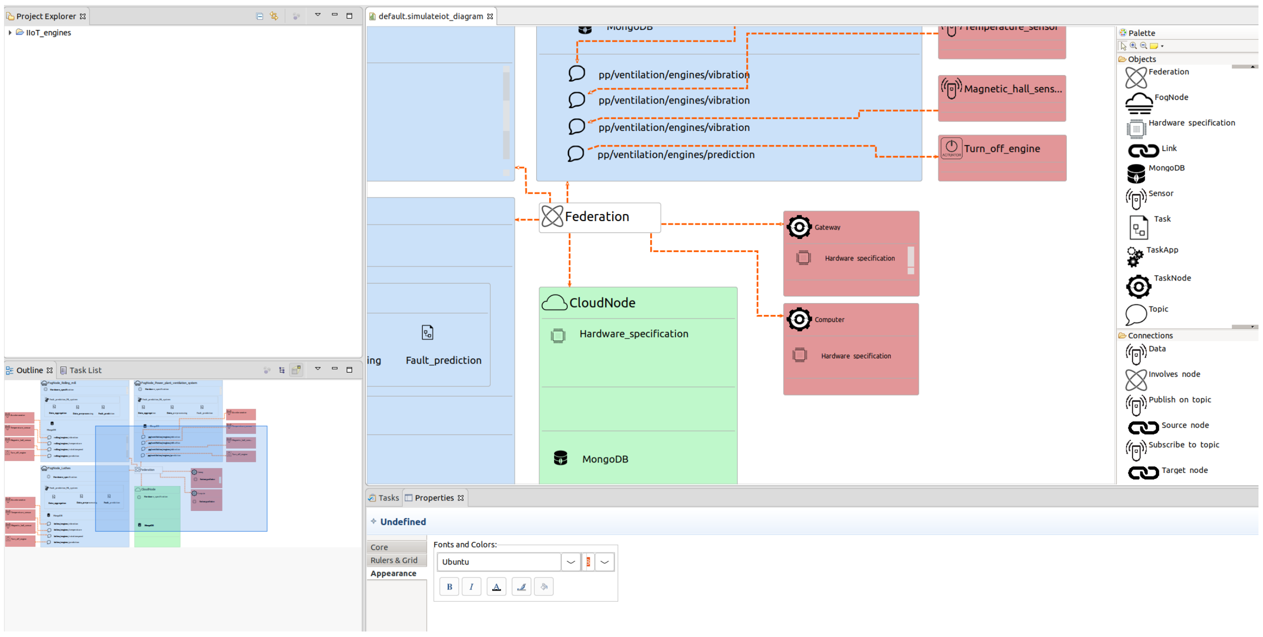


Figure 5 Graphical editor based on Eclipse to graphically design models conforming to the metamodel proposed for the simulator.

to a metamodel. As an example, some of the OCL restrictions defined are shown below.

```
context Task
invariant UniqueId: Task.allInstances()->forAll(
    t1, t2 | t1 <> t2 implies t1.id <> t2.id)
```

This OCL constraint ensures that each task has a unique ID.

```
context Delay_specification
invariant MaxDelayGreaterThanMinDelay: self.
    max_delay >= self.minum_delay
```

This invariant ensures that the maximum delay (max_delay) is always greater than the minimum delay (minum_delay) in each Delay_specification instance.

```
context Bandwidth_specification
invariant ValidBandwidth: self.bandwidth <> null
    and self.bandwidth > 0
```

This invariant ensures that the bandwidth specification is not null and is greater than 0.

To sum up, the graphical concrete syntax developed offers a suitable way to model and validate the IoT environment by using the high-level concepts defined in the SimulateIoT metamodel (Figure 4).

5. Extensions of M2T Transformations

Once the models have been defined and validated conforming to the proposed simulator metamodel (example of a model in Figure 10), a M2T transformation defined using Acceleo (Obeo 2012) can generate the IoT system modelled. This section describes the main extensions included in the M2T transformations of SimulateIoT in order to generate IoT systems simulations with task scheduling capabilities.

For the sake of clarity, this section is divided into the domain-specific concepts identified in Section 3.1 (as in Section 4.1). In this way, each subsection includes the contributions that make it possible to generate the code of each component related to these task scheduling concepts. However, there are no sections dedicated to the `Networking Node` and the `Task Processor`. This is because these components are subcomponents of other elements, such as the Task Node. As a result, they are addressed in the sections pertaining to these primary components.

Acronym	**Meaning**
TApp	Task App
NN	Networking Node
TN	Task Node
TP	Task Processor
TS/TSN	Task Scheduler Node
SSA	System Status Agent

Table 2 Acronyms used in figures of Section 5.

Note that the descriptions of the components, from a high level of abstraction, are already covered in Section 3 and 4.1. Therefore, the current section omits this high-level description and exclusively delves into the inner workings of the components, offering a low-level perspective.

Finally, note that Table 2 summarises the acronyms used in several of the figures included in this section.

5.1. Task

Section 4.1 describes the extensions carried out to make it possible to model the tasks that will be generated, offloaded and processed by the IoT system. However, the task concept does not become a concrete component or service but is integrated as part of the logic of the rest of the components, e.g. in the `Task App` or the `Task Node`, which need the logic to generate, offload, receive or process tasks. Therefore, the transformations related to this concept are addressed below, in the sections that explore the M2T transformations of the components that are related to tasks, such as the aforementioned.

However, since tasks are managed by these components using the JSON notation, below are the different JSON codes that can be exchanged between components during a simulation.

To illustrate how tasks are generated by the `Task App` or the `Task Node` (components that can generate tasks), Listing 1 shows an example of the JSON code related to the set of tasks that constitute the workflow shown in Figure 2. This JSON code example includes all the necessary fields to represent this workflow, such as the *id* and *name* of the workflow, the node and the component that generate it (field *"generatedBy"*), the tasks that constitute the workflow (field *"nodes"*), the edges (field *"edges"*) and all the data related to these elements. Note that Appendix B, Listing 5, shows the Acceleo code related to the M2T transformations of these tasks.

```
{
  "workflow": {
    "id": 0,
    "name": "exampleWorkflow",
    "generatedBy": {
      "nodeId": "fogA0",
      "generatorId": "taskAppA0",
      "generationId": "0"
    },
    "nodes": [{
        "task": {
          "name": "TaskA",
          "id": "0",
          "size": "586"
        }
      },{
        "task": {
          "name": "TaskB",
          "id": "1",
          "size": "344"
        }
      },{
        "task": {
          "name": "TaskC",
          "id": "2",
          "size": "719"
        }
      },{
        "task": {
          "name": "TaskD",
          "id": "3",
          "size": "412"
        }
      }
        ],
    "edges": [{
        "edge": {
          "id": "0",
          "sourceTaskId": "0",
          "targetTaskId": "1",
          "offloadSize": "128"
        }
            },{
        "edge": {
          "id": "1",
          "sourceTaskId": "0",
          "targetTaskId": "2",
          "offloadSize": "96"
        }
      },{
        "edge": {
          "id": "2",
          "sourceTaskId": "1",
          "targetTaskId": "3",
          "offloadSize": "49"
        }
      },{
        "edge": {
          "id": "3",
          "sourceTaskId": "2",
          "targetTaskId": "3",
          "offloadSize": "223"
        }
      }]}}
```

Listing 1 JSON code to represent a workflow. Specifically, the workflow illustrated in Figure 2.

On the other hand, workflows are sent to the `Task Scheduler` for scheduling purposes. In this regard, the `Task Scheduler` decomposes the workflow to schedule the processing of its tasks. Listing 2 shows the JSON code related to the scheduling of `Task C`, which belongs to the workflow shown in Figure 2 and in Listing 1.

Among the fields of this JSON code, there are data related to the task itself (*id, name, size, offloadSize*), data related to the node that generated it (*generatedBy*), as well as data related to the scheduling of the task (*schedulingData*). Data relating to the scheduling of the task includes the node that has to process it (*processAt*) and at what time its processing has to be performed (*processAt*). Furthermore, since the task scheduling model (Section 3.1) envisages dependency between tasks, this JSON code also includes to which node the processing results have to be sent (*resultsTo*) and also whether the task depends on the processing results of another task (*waitForResults*).

```
{
  "scheduledTask": {
    "id": "2",
    "name": "TaskC",
    "size": "719",
    "offloadSize": "49",
    "generatedBy": {
      "nodeId": "fogA0",
      "generatorId": "taskAppA0",
      "generationId": "0"
    },
    "schedulingData": {
      "processIn": "TaskNodeA",
      "processAt": "2023-1-25 11:29:52",
      "resultsTo": "fogC",
      "waitForResults": {
        "taskId": "1",
        "taskName": "TaskA"
      }
    }}}
```

Listing 2 JSON code related to the scheduling of TaskC of the workflow illustrated in Figure 2.

Once a node performs the processing of a task, it includes in its JSON code some fields related to the results of its processing. Listing 3 shows the JSON code fields added to `Task C` after its processing. Among these fields, there is the time at the task reached the `Task Processor` (*arrivedToTaskProcessorAt*), the time at the processing of the task started (*processingStartedAt*) and the time at the processing of the task finished (*processingFinishedAt*). In short, it includes data that could be useful for the user in the analysis stage.

```
{
        .

    "processingResults": {
        "arrivedToTaskProcessorAt": "2023-1-25 11:28:38"
        "processingStartedAt": "2023-1-25 11:29:54",
        "processingFinishedAt": "2023-1-25 11:29:57"
    }
        .
}
```

Listing 3 JSON code excerpt that shows the fields related to the processing results of TaskC of the workflow illustrated in Figure 2.

As discussed above, since the task scheduling model (Section 3.1) envisages dependency between tasks, once a task is processed, the processing results are sent to the node that is waiting for them, i.e. the node that has to process a task that depends on these results.

Thus, when a node receives processing results, it includes them in the JSON code of the task that is waiting for them. In this way, the exit (last) task of a workflow will include the processing results of the rest tasks of the workflow. Note that this JSON code is the processing result that is returned to the node that generated and offloaded the workflow to the system for its processing (`Task App` or `Task Node`).

Listing 4 shows the JSON code fields related to the processing results of the predecessors of `Task C` (in this example, `Task A`). The data included in this case are the received results related to the processing of the task together with its *id* and *name*. Note that, if a task includes results related to the processing of another task (as is the case in this example where Task C contains the results of Task A) when offloading its processing results, it also includes the results of its predecessors. So, in this example, `Task C` will offload its processing results together with the processing results of `Task A`.

```
{
        .

    "predecessorsProcessingResults": [{
        "task": {
            "id": "0",
            "name": "TaskA",
            "processedIn": "FogA"
            "arrivedToTaskProcessorAt": "2023-1-25 11:26:
18",
            "processingStartedAt": "2023-1-25 11:28:34",
            "processingFinishedAt": "2023-1-25 11:28:47"
        }
    }]

        .
}
```

Listing 4 JSON excerpt that shows the fields related to the processing results of the predecessor tasks of Task C of the workflow illustrated in Figure 2.

5.2. Task App

Figure 6 shows a generic `Task App` node Ⓑ (represented by a red box) and its main component (element within the red box), the `Workflow Generator` Ⓒ. Figure 6 also shows how the `Task App` is deployed on a fog/cloud node and the interactions that its component could perform with other artefacts of the fog/cloud node and with the rest of the IoT system. Below, the `Task App` node is illustrated by describing its component and its interactions with the rest of the system.

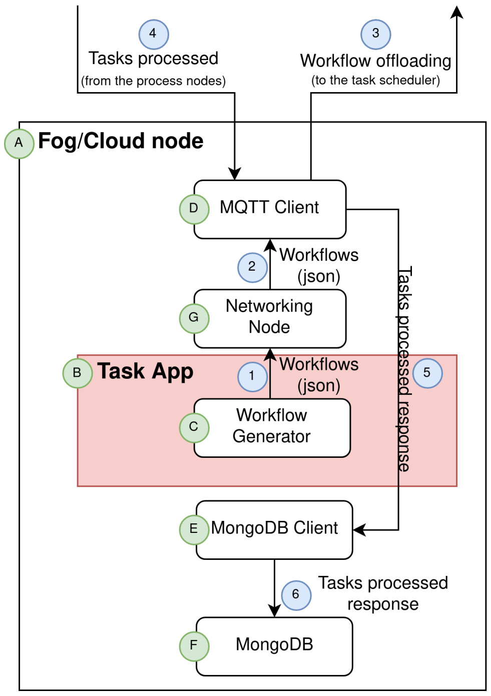


Figure 6 Task App component.

`Workflow Generator` Ⓒ As described in Section 3.1.2, the `Task App` can generate tasks by means of workflows and offload them to the rest of the system. The `Workflow Generator` is the component of the `Task App` whose aim is to generate and offload ① the workflows modelled by the users in the modelling of the system stage.

The `Task App` is not composed of additional components. However, to provide a comprehensive understanding of its role and impact within the entire system, a description of the components associated with the `Task App` is presented below.

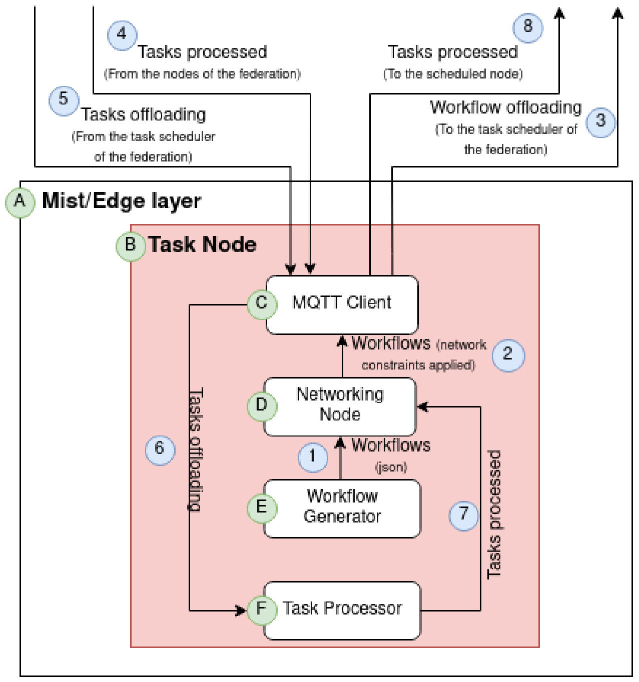
`Networking Node` Ⓖ This component simulates the network aspects related to the *bandwidth* and *delay* modelled by users for each `Link`, i.e. the unidirectional connection between two federated nodes. So, as the workflows generated have to be offloaded to the system through a `Link`, the `Networking Node` applies to these workflows the *bandwidth* and *delay* constraints modelled for the `Link` through which they have to be offloaded. Note that the `Networking Node` is described in more depth in Appendix C.

`MQTT Client` Ⓓ The MQTT Client is the component that allows publish/subscribe communication between the components of the system through the MQTT protocol. In this context, it is implied in the offloading of the generated workflows to the rest of the system (interactions ② and interactions ③) and in the reception of the processing results of the tasks (interactions ④ and ⑤).

`MongoDB` Ⓕ and `MongoDB Client` Ⓔ To provide data storage services to the IoT systems simulations, seamlessly to the user, a MongoDB database Ⓕ is deployed on each modelled fog/cloud node during the deployment stage of the simulation. In the same way, a `MongoDB Client` Ⓔ is also deployed to carry out the needed interaction between this database and the task scheduling components. In this context, these two components are employed to store the processing results of the tasks (interactions ④, ⑤ and ⑥).

5.3. Task Node

Figure 7 shows a generic `Task Node` Ⓑ (represented by a red box) and all its components (elements within the red box) and their interactions. The main components of the `Task Node` are the `Workflow Generator` Ⓔ, the `Task Processor` Ⓕ, the `Networking Node` Ⓓ and the `MQTT Client` Ⓒ. Besides, Figure 6 also shows how the `Task Node` is deployed as part of the mist/edge layer and the interactions that its components could perform among them and with other artefacts of the system. Note that in this case, components such as the `Networking Node` or the `MQTT Client` are part of the `Task Node`. In contrast to the `Task App`, the `Task Node` is not deployed on a fog/cloud node (which provides with a `MQTT Client`, etc. to the `Task App`), the `Task Node` is a device itself belonging to the edge layer. Below, the `Task Node` is illustrated by describing each of its components and their interactions with the rest of the system.



4 Tasks processed (From the nodes of the federation)
8 Tasks processed (To the scheduled node)
5 Tasks offloading (From the task scheduler of the federation)
Workflow offloading (To the task scheduler of the federation) 3
A Mist/Edge layer
B Task Node
C MQTT Client
Workflows (network constraints applied) 2
D Networking Node
1 Workflows (json)
E Workflow Generator
F Task Processor
6 Tasks offloading
7 Tasks processed

Figure 7 Task Node components.

`Workflow Generator` Ⓔ, `Networking Node` Ⓓ and `MQTT Client` Ⓒ As the `Task App`, the `Task Node` also generates and offloads workflows to the rest of the system. In this context, the behaviour of these components (interactions ①, ②, ③ and ④) is the same as the behaviour already explained in the `Task App` section (Section 5.2).

`Task Processor` Ⓕ `Task Nodes` Ⓑ belong to the edge/mist layer, however, they can be part of the computing power of a federation, thus being able to process tasks. The `Task Processor` is the component of the `Task Node` that performs the processing of tasks. In this way, the `Task Scheduler` could schedule workflows and assign the processing of its tasks to a `Task Node` (interaction ⑤). As these scheduled tasks are offloaded via MQTT protocol, the first to receive them is the `MQTT Client` of the `Task Node` (interaction ⑤). Next, the `MQTT Client` forwards these tasks to the `Task Processor` (interaction ⑥), which performs their processing. Once processed, the `Task Processor` returns the processing results of these tasks to the `MQTT Client`, which sends them to their target nodes, i.e. the nodes waiting for the results of the processed tasks (specified by the field *resultsTo* of the JSON code illustrated in Listing 2). Note that as outgoing data, network constraints are also applied in this context. This flow of data is represented by the interactions ⑦, ② and ⑧. Finally, note that the behaviour of the `Task Processor` is described in more depth in Appendix D.

5.4. Task Scheduler

Figure 8 shows a generic `Task Scheduler` Ⓑ (represented by a red box), all its components (elements within the red box) and the interaction between them. The main components of the `Task Scheduler` are the `Workflow Buffer` Ⓓ, the `System Status Agent` Ⓖ, the `Task Scheduler API` Ⓔ and the `Task Scheduling Proposal` Ⓕ. Besides, Figure 8 also shows how the `Task Scheduler` is deployed on a fog or cloud node Ⓐ and the interactions that these components could perform with other artefacts of the fog/cloud node and with the rest of the IoT system. Below, the `Task Scheduler` is illustrated by describing each of its components and their interactions.

`Workflow Buffer` Ⓑ The `Task Apps` and the `Task Nodes` generate and offload workflows to the system. The `Task Scheduler` Ⓑ is the component that first receives these workflows ① with the aim of scheduling the processing of these tasks. In this context, the `Workflow Buffer` holds these incoming workflows ②. Hence, when the `Task Scheduler Proposal` Ⓕ is ready to schedule the processing of a workflow, it retrieves it from this buffer by means of the `Task Scheduler API` (interactions ③, ④, ⑤ and ⑫).

`System Status Agent` Ⓓ The `System Status Agent` aims to gather the status of both the nodes that belong to the same federation as the `Task Scheduler` and the network of the federation. In this context, first, the `Task Scheduling Proposal` requests this data to the `System Status Agent` by means of the `Task Scheduler API` ① (interactions ③ and ⑥). Then, the `System Status Agent` component requests the status of the federation network (`Links` that connect the nodes of the federation) to the `Network Status Reporter` component of each `Networking Node` (see Appendix C, Figure 12). Moreover, the `System Status Agent` also requests to the `Task Processor Status Reporter` component (see Appendix D, Figure 13) the status related to the hardware usage of each `Task Processor` (i.e. `Task Nodes` and other fog/cloud nodes) that belong to the same federation that the

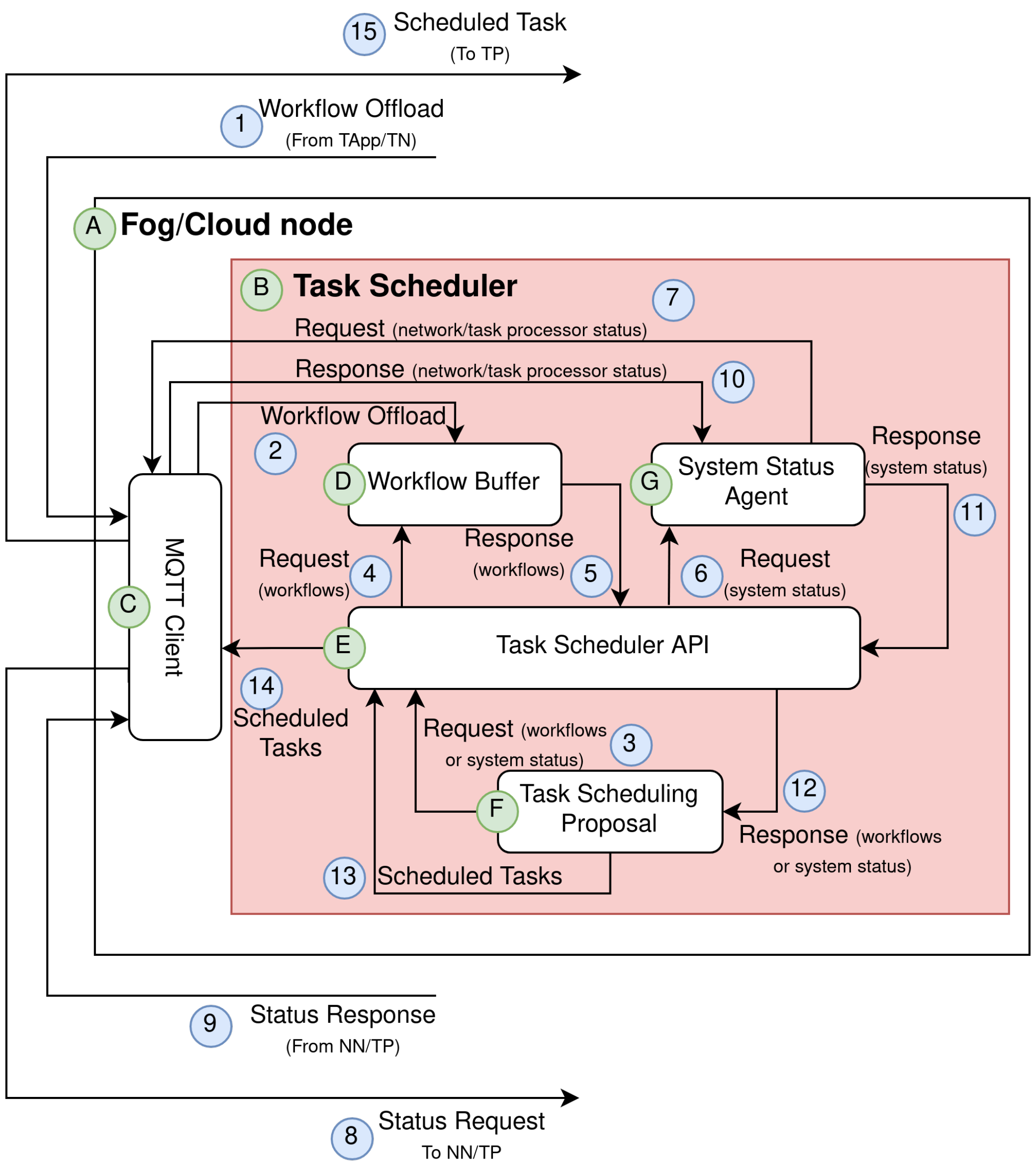

15 Scheduled Task (To TP)
1 Workflow Offload (From TApp/TN)
A Fog/Cloud node
B Task Scheduler
7 Request (network/task processor status)
10 Response (network/task processor status)
2 Workflow Offload
D Workflow Buffer
G System Status Agent
Response (system status)
11
C MQTT Client
Request (workflows) 4
Response (workflows) 5
6 Request (system status)
E Task Scheduler API
14 Scheduled Tasks
Request (workflows or system status) 3
F Task Scheduling Proposal
12 Response (workflows or system status)
13 Scheduled Tasks
9 Status Response (From NN/TP)
8 Status Request To NN/TP

Figure 8 Task Scheduler components.

`Task Scheduler`. These two aforementioned requests are represented by the interactions ⑦ and ⑧. When the status data is received ⑨, the `System Status Agent` provides this data to the `Task Scheduling Proposal` ⑪ and ⑫. Thus, the `Task Scheduling Proposal` can use this data as input for task scheduling purposes.

`Task Scheduler API` Ⓔ The `Task Scheduling Proposal` is the task scheduling approach that the user can deploy into the simulation for testing and analysis purposes. The `Task Scheduler API` has been conceived as a middleware service to integrate the `Task Scheduling Proposal` with the simulated system. Thus, the `Task Scheduler API` is able to 1) gather key data about the status of the simulated system and provide it to the `Task Scheduling Proposal`, 2) provide the `Task Scheduling Proposal` with the workflows that have been offloaded to the `Task Scheduler` Ⓑ and 3) return the tasks of a workflow scheduled to the system for their processing. Note that the request/response interactions related to 1) and 2) (③, ④, ⑤, ⑥, ⑪, ⑫) have been already addressed through the explanation of the `Workflow Buffer` Ⓓ and the `System Status Agent` Ⓖ. Hence, with the offloaded workflows and the status of each node and link of the federation, the `Task Scheduling Proposal` can perform the schedule of each task. Once the `Task Scheduling proposal` has finished the schedule, returns the tasks scheduled to the system for their processing, also by means of the `Task Scheduler API` (interaction ⑬, ⑭ and ⑮).

`Task Scheduling Proposal` Ⓕ The `Task Scheduling Proposal` implements the task scheduling approach that the user can deploy into the simulation for testing and analysis purposes. The aim of the `Task Scheduling Proposal` is to schedule the offloaded workflows. The `Task Scheduling Proposal` only interacts with the `Task Scheduler API` Ⓔ. This is because the `Task Scheduling Proposals` is developed by the user, and will not necessarily have been developed to be tested in the simulator proposed in this communication. Therefore, to facilitate their integration with the simulator, the `Task Scheduler API` Ⓔ is used as an interoperability layer. By means of a series of requests, the `Task Scheduling Proposal` can interact with the simulated system receiving the offloaded workflows, gathering data related to the status of the system to carry out the schedule of each task and return them scheduled to the system for their processing. Note that all these interactions have been described above.

5.5. Deployment Infrastructure

The deployment of simulations is carried out through container orchestration. Thus, the system components are wrapped in Docker (Merkel 2014) containers as shown in Figure 9, and subsequently deployed in different clusters (orchestration). Regarding the wrapping of the components, Figure 9 uses the Docker logo within the represented boxes to indicate that said component or set of components is wrapped in a Docker container.

On the other hand, the default supported orchestrator is Docker Swarm, although since the deployment is carried out using `Docker-Compose`, it is possible to perform the deployment on Kubernetes (*Kubernetes Documentation* 2023) by automatically generating the deployment files for this orchestrator using the Kompose tool (*Kompose* 2023).

As for communication between containers, it is carried out through DNS (name + id of the component, specified in the modelling stage). Communication through DNS is supported by the overlay network that is generated to support the deployment.

Finally, orchestration takes into account the fog and cloud nodes modelled as part of the system. In this way, a "cluster" is generated for each fog or cloud node modelled. Subsequently, all the Docker containers related to the fog or cloud node are deployed on this "cluster", as well as those components belonging to the mist or edge layer that interact with it (e.g. Task Nodes).

Note that all these specifications are low-level specifications that, although not previously mentioned, are part of the simulator, in this case of the files generated through M2T transformations to perform the simulation deployment.

6. Simulation Outputs, Tests and Assessments

The primary motivation for the simulation and testing of an IoT system is to derive valuable insights, enabling the assessment of its behaviour or the optimisation of its performance. Consequently, the advantages that can be drawn from an IoT simulator rely on the variety and depth of tests and evaluations it supports. The SimulateIoT extension carried out in this communication facilitates several options for conducting tests and evaluations from a task scheduling perspective. The most significant tests and assessments that can be performed are listed below.

– Task distribution: How, in general, the task scheduling proposal distributes tasks among devices.

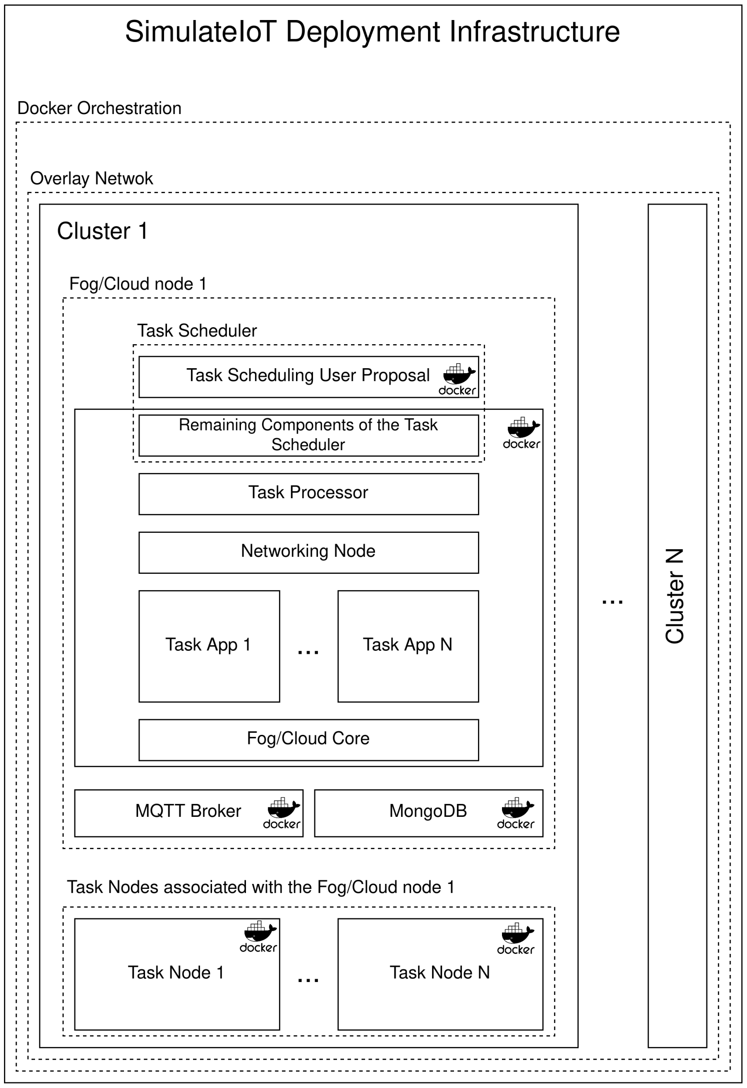


Figure 9 Deployment infrastructure of the extended version of SimulateIoT.

– Overload avoidance: The task scheduling proposal's ability to prevent any single device from becoming overloaded with tasks.
– Dynamic load balancing: How the task scheduling proposal can adjust task distribution as the load changes.
– Task prioritisation: If included in the policies of the task scheduling proposal, how it balances the load while considering task priorities.
– Response time load balancing: If included in the policies of the task scheduling proposal, how it can maintain uniform response times by effectively balancing the load.
– Queue length: The task scheduling proposal's ability to maintain balanced queue lengths across all devices.
– Task rebalancing: How effectively the task scheduling proposal can rebalance tasks when new devices join or existing devices leave the system.
– Network load variation: How the task scheduling proposal balances the load under varying network conditions.

- Workflow response time: The elapsed time from when a workflow is generated by a device, processed by the system, to the delivery of the processing results back to the device. This can also be applied in the context of tasks.
- Workflow processing rate: The number of workflows processed by a device over a specific period of time. This can also be applied in the context of tasks.
- Workflow processing throughput: The maximum number of workflows a device can process in a given time frame. This can also be applied in the context of tasks.
- Hardware consumption: How the system or a device uses its resources (CPU, memory, etc.) during the whole simulation or at a specific time.
- Resource allocation: Whether the resource allocation strategy followed during the design stage of the system was appropriate.
- Bottleneck identification: Finding points in the system where bottlenecks occur that could limit the system's performance.
- Workflow response time, processing rate and throughput scalability: Evolution of these parameters as the workload on the system is increased or reduced.
- Device scalability: How the system handles an increasing number of IoT devices to check if the system can maintain performance as the network grows.
- Network traffic scalability: How the system performs under different levels of network traffic.
- Fault tolerance: How the system behaves, in general, when faced with hardware or software faults. Note that this proposal does not include a specific model to induce failures during simulations. However, a simple script can stop (and redeploy later if required) the components of the simulated system (Docker containers). Thus, thanks to the components notifying the status of each node to the `Task Scheduler`, it can notice if a device is available or not (lacked response when stopped) and take it into account.
- Fault recovery: How the system handles recovery processes in the event of a failure, ensuring no task or data loss.
- Fault tolerance load balancing: The task scheduling proposal's ability to redistribute tasks when a device fails or becomes unavailable.
- Power consumption: The power consumed by the IoT system during a simulation or in a specific period of time. Note that a model related to the power consumption of devices has not been included in this work, however, it can be assessed from the hardware consumption.
- Workflow-specific energy consumption: The amount of energy consumed for each specific workflow.
- Device-specific energy consumption: The energy consumption of individual devices under different task loads.

As can be seen, the extension developed for SimulateIoT in this study provides users with a wide spectrum of testing and analysis opportunities for their task scheduling proposals. Thus, providing users with a holistic understanding of their IoT systems design and their task scheduling proposals performance.

7. Case Study: An IIoT System Applied to the Steel Industry for Predictive Maintenance

In this section, a study case focused on the IIoT applied to the steel industry for predictive maintenance is illustrated.

7.1. Motivation

Task scheduling techniques can be applied to any IoT system to optimise the processing of their tasks (Potluri & Rao 2020). However, their application is particularly appealing in the so-called critical IoT systems, i.e. IoT systems on which the safety of users depends and IoT systems on which specific response times, fault tolerance, etc. have to be met in order to perform suitably. Some of these critical IoT systems could be those focused on healthcare, traffic safety and control (IoV) or industry (IIoT) (Andersson et al. 2016).

In industry, optimal maintenance of production equipment and facilities is one of the keys to global competitiveness and survival (Zhao et al. 2022). Over time, different maintenance strategies, such as corrective and preventive maintenance, have been developed and applied (Lie & Chun 1986; Hao et al. 2010). Nowadays, with the possibility of continuous monitoring of equipment and facilities provided by the IoT and machine learning, a new maintenance strategy is being developed and implemented, predictive maintenance (Çınar et al. 2020; Cheng et al. 2020). This type of maintenance is based on predicting equipment failures or breakdowns, allowing maintenance work to be carried out before the equipment suffers further damage and causes a more negative impact on production (Carvalho et al. 2019).

In this context, electric motors are one of the most widely used tools in industry (Cakir et al. 2021). Their applications are varied, primarily including blowers, turbines, pumps, compressors, alternators, rolling mills, movers, etc. Thus, for the reasons outlined above, proper maintenance of these engines is crucial for companies to be competitive.

Given the need for predictive maintenance of electric motors, as well as the suitability of the application of IoT and task scheduling to achieve this purpose, it has been considered appealing to show the application of the proposed simulator in this context. Thus, this case study illustrates how the proposed simulator can assist in the design, development and implementation of an IoT system for monitoring and predicting the failure of electric motors in a steel company.

7.2. Overview

The use case presented in this Section is based on those works referenced in the above Section 7.1. The aim of the modelled system (using the metamodel presented in this communication) shown in Figure 10 is to provide a steel company with the capability to perform predictive maintenance on their electric engines. On the other hand, the aim of the use case is to test whether the system identifies and stops faulty engines below a time threshold. For example, if at any point during the entire simulation, the system takes no more than 10 seconds (hypothetical time threshold) from the moment an engine starts malfunctioning until the system identifies this situation and stops the engine. Note that these time thresholds can not be specified as part of the system as it is not supported by the DSL proposed. Instead, the user is who has to model the system and later, analyse the logs of the simulation to check the behaviour of the system. Continuing with the proposed example, check whether the time threshold is met during the entire simulation.

For this purpose, a set of sensors has been included in the edge layer of the system (red-coloured components) in order to continuously monitor each electric engine. Two `Task Nodes` have also been added to the edge layer, a gateway, which carries out the aggregation of the publishing data for each sensor (note that this is a task included in the simulation by means of a workflow), and a computer, which is used only to provide support for the processing of the tasks to be carried out in the system. In addition, an actuator has also been included in the edge layer to stop the operation of those engines whose failure has been predicted.

As for the fog layer, the modelled system includes several fog nodes that provide the different deployed edge nodes with topics for subscribing or publishing their data. Besides, these fog nodes also perform two tasks in the system, the pre-processing of the received data (aggregated by the gateway) and the failure prediction of each monitored engine (from this pre-processed data). Note that these two tasks have been also included in the simulation by means of workflows.

Furthermore, a cloud node has been added to the system. This cloud node aims to provide additional hardware resources to the system if needed.

Finally, note that the simulation related to this use case has been deployed on a personal computer with the following specifications: Model: MSI GP63 Leopard 8RE; CPU: Intel Core i7-8750H; Graphics: GeForce GTX 1060 with 6GB GDDR5; RAM Memory: 16GB DDR4-2400.

7.3. Model Definition

Figure 10 shows an excerpt of the IIoT system model. For the purpose of explaining this model, it includes numerical references for each node, which are referenced below when describing each component of the case study. Besides, for the sake of clarity, note that the description of the case study is divided into three parts: 1) edge layer (red nodes), 2) fog layer (blue nodes), and 3) cloud layer (green nodes).

7.3.1. Edge Layer The edge layer of the IIoT system modelled for this case study is comprised of three kinds of devices: sensors (S.X), actuators (A.X) and `Task Nodes` (T.X). The sensors included are accelerometers (S.1, S.4 and S.7), thermometers (S.2, S.5 and S.8) and (magnetic) Hall sensors (S.3, S.6 and S.9). The accelerometers gather vibration data, i.e. the vibration that the bearings of the engine produce, the thermometers gather the temperature of the engine, and 3) the (magnetic) Hall sensors collect data related to the rotational speed of the engines. Note that these sensors and the data they collect are often used to predict failures in electric engines (Cakir et al. 2021).

These sensors collect this data and publish it on the topics To.X provided by the fog nodes F.X for further use. Accelerometers publish their data in the topic *x/.../engines/vibration*, thermometers in the topic *x/.../engines/temperature* and (magnetic) Halls sensors in the topic *x/.../engines/rotationspeed*. Note that the data gathered and published by these sensors has been modelled by the user with the expressiveness already provided by the previous version of SimulateIoT.

The actuators included A.X aim to stop the operation of those engines whose failure has been predicted. Thus, they subscribe to the topic *x/.../engines/prediction*, where the nodes that carry out the prediction (fog F.X and Cloud C.X nodes) of the failure of the engines publish their predictions. In this way, when an engine failure prediction is published in this topic, the actuators receive it and stop the operation of the engine.

Finally, two `Task Nodes` Tn.X have been modelled. On the one hand, a gateway Tn.1, which has a `Hardware_specification` H.5 where the CPU and RAM of the device have been modelled. This device has been modelled to take advantage of its hardware for task scheduling purposes. Moreover, this `Task Node` performs a task, the data aggregation T.7 of the data published by the modelled sensors. In this way, when the data reach the topics and the fog nodes, it is already aggregated. Note that this task has been modelled by means of a `Workflow` in the *properties* of this component. On the other hand, a personal computer has been added as a `Task Node` of the system, being part of the edge layer and providing the rest of the system with more computing power. This `Task Node` have also a `Hardware_specification` H.6.

Note that each sensor, actuator and `Task Node` have an attribute named `quantity` (included in the previous version of SimulateIoT) where the user can specify the quantity of each node. In this case study, the `quantity` attribute is ten for sensors, three for actuators, three for the gateway `Task Node` and one for the computer `Task Node`.

7.3.2. Fog Layer The fog layer of the IIoT system modelled for this case study is comprised of three fog nodes, `FogNode_rolling_mill` F.1, `FogNode_power_plant_ventilation_system` F.2 and `FogNode_lathes` F.3. The `FogNode_rolling_mill` is the fog node deployed near the rolling mill, the `FogNode_power_plant_ventilation_system` is the fog node deployed near the ventilation system of the power plant of the company, and the `FogNode_lathes` represents the fog node deployed near the lathes of the company. The aim of these fog nodes is to provide services to the rest of the system.

In this regard, the `FogNode_rolling_mill` provides topics To.1 to subscribe or publish data to those sensors installed on the engines of the `FogNode_rolling_mill` of the company. Besides, this node carries out two tasks, 1) the data pre-processing T.1 of the received data (already aggregated by the gateway `Task Node`) to send this data in a suitable way to the machine learning model of the system, which use it as input to make predictions, and 2) The fault prediction T.2 of the engines of the

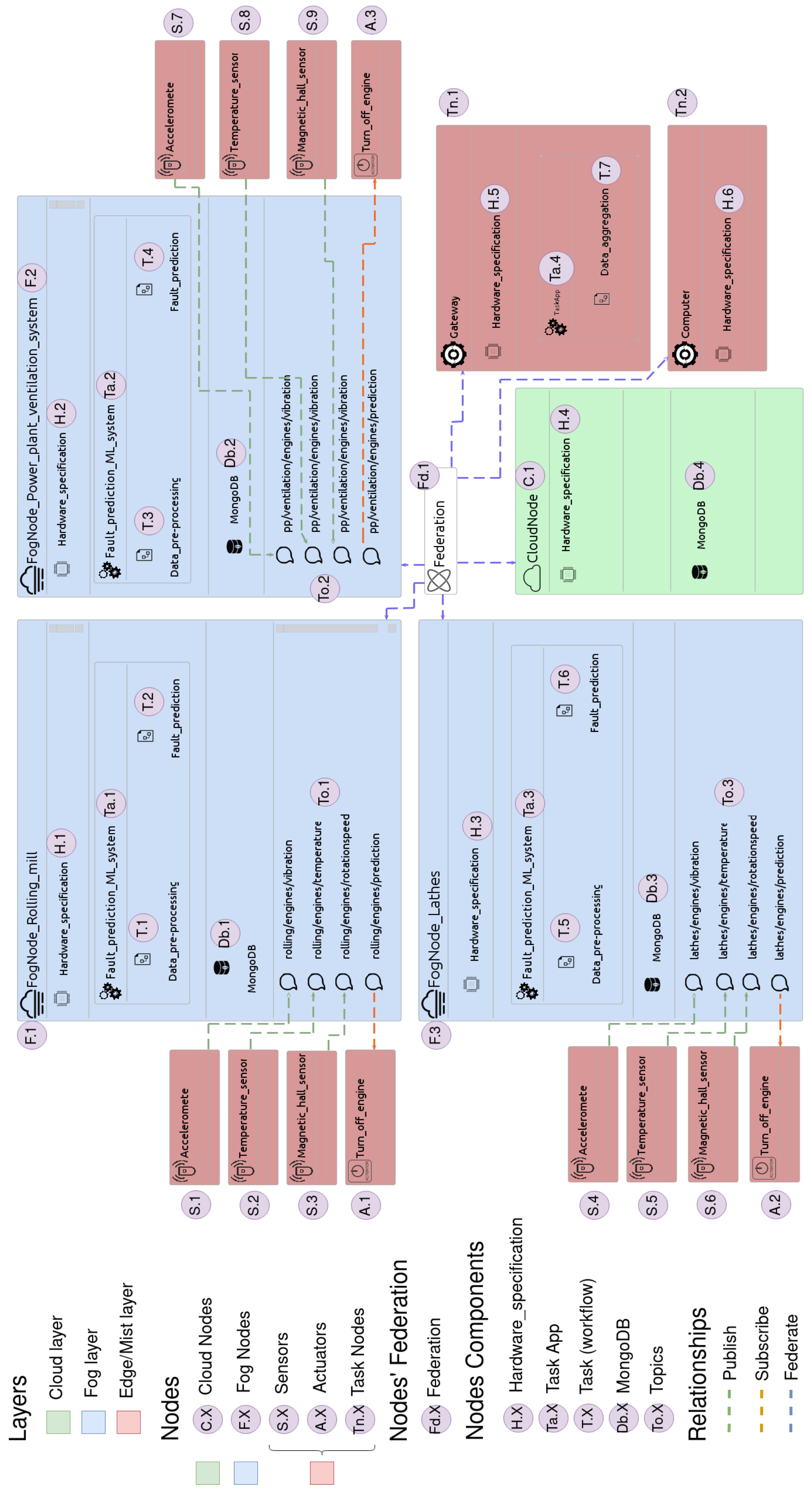


Figure 10 Model conforms to the proposed simulator metamodel. An IIoT system applied to the steel industry for predictive maintenance of electric engines.

rolling mill. These predictions are later published in the topic *x/.../engines/prediction* where the actuators responsible to stop the engines are subscribed. Note that these tasks have been modelled by means of `Workflows` in the *properties* of these components. This fog node has also a `hardware_specification` H.1 and a mongo database Db.1.

The other two fog nodes, the `FogNode_power_plant_ventilation_system` and the `FogNode_lathes` have a similar configuration to the `FogNode_rolling_mill`. The differences are 1) The `Workflows` modelled in the tasks defined in each of them are different 2) Each fog node represents the implementation of the fault prediction system in the electric engines of the different equipment and facilities of the company.

7.3.3. Cloud Layer The cloud layer has been included with one aim, to provide hardware resources to the rest of the system. Thus, one cloud node has been modelled C.1. In this regard, this cloud node has a `hardware_specification` greater than the `hardware_specification` of the fog nodes, and a Mongo database Db.4.

7.3.4. Cloud-Fog-Edge Federation Although the cloud-fog-edge `Federation` Fd.1 is not a layer, it allows each cloud, fog and edge node modelled to operate as a single entity instead of isolated nodes. This component has been modelled federating each fog, cloud and `Task Node` modelled. Thus, all the nodes with computing power (a `hardware_specification`) can cooperate for task scheduling purposes. Note that in the properties of this component, the features of each `Link` that interconnects the nodes belonging to the `Federation` (`bandwidth` and `delay`) have been modelled.

Finally, note that although in this case study has not been modelled, additional nodes could be defined to build other federations.

7.4. M2T Transformations

Once the model has been defined, the M2T transformations are applied with the following goals:

- i) to generate Java, Python, NodeJs, etc. code that wraps each device behaviour.
- ii) to generate configuration code to deploy all the generated services, such as the message brokers necessary, including the *topic* configurations defined, the gateway configurations, etc.
- iii) to generate the code and deployment configuration files of the architecture that supports task scheduling (`Task Apps`, `Task Nodes`, `Networking Nodes`, `Task Processors` and the `Task Schedulers`).
- iv) to generate the code and deployment configuration of the users' task scheduling proposal and their integration with the simulation.
- v) to generate the configuration files and scripts necessary to deploy the databases and stream processors defined; and finally, to generate the code necessary to query the databases where the data will be stored.
- vi) to generate for each cloud, fog and edge node a *Docker* container which can be deployed throughout a network of nodes using *Docker Swarm*.

Consequently, each edge node, fog node and cloud node and their related components are generated following the software architecture defined in Section 5 where the M2T transformations have been defined.

Finally, note that to generate a part of the code in a target language/infrastructure different from the one supported, users will need to make the following efforts: 1) Understand the metamodel or the concepts related to the component/s they wish to update; 2) Understand the M2T transformations related to the component/s to be modified; 3) Develop the code from scratch in their target language; 4) Integrate it into Acceleo; 5) Conduct sufficient tests and trials to confirm the successful update of the component/s.

7.5. Simulation Analysis

The benefits that can be obtained from a simulator come from the outputs, data, etc. from the simulations and tests performed. In this section, a set of simulations and different tests based on the model described above are carried out, illustrating the possibilities and benefits provided by the proposed simulator. To carry out these simulations and tests, the HEFT algorithm (Topcuoglu et al. 2002), one of the most widely used and extended algorithms in the field of task scheduling, on which some recent algorithms are based (Ojha et al. 2020; Divyaprabha et al. 2018; Faragardi et al. 2020), has been integrated into the simulator. Consequently, the `Task Scheduler` applies the aforementioned task scheduling algorithm (HEFT) to process the workflows during simulations. Note that the M2T transformations only need the task scheduling proposal (in this case the HEFT algorithm) to be on the same path as the generated components (by the M2T transformations) in order to automatically integrate it into the simulation.

As for the test, first is desired to determine the average and maximum time that the modelled IIoT system needs to predict the failure of an engine. This involves the time it requires to aggregate and pre-process the data related to each engine and the time it requires to carry out the prediction (failure or not).

For this purpose, the simulation was run for 120 seconds. Once completed, the average time reported to predict the failure of an engine (any engine) is 4.391 seconds, and the maximum time is 6.384 seconds. The maximum time was related to the engines of the rolling mill. This was expected since the workflows related to failure prediction for this kind of engine are more complex (higher amount of bytes to process and to offload) than for the others.

At this point, the user has to determine whether this response time satisfies his performance needs i.e., if the response time is lower than expected, the user could reduce the hardware of the designed system, thus saving costs. If, on the other hand, the response time exceeds the estimated time to avoid severe engine damage, the system has to be upgraded, either by software or hardware.

In this use case, it has been determined to carry out a software upgrade. The HEFT algorithm includes an insertion policy, i.e., the idle slots of each processor (time in which the processor is not used between processing each task) can be used to process tasks. However, in the previous test, a modified HEFT algorithm was used in which the use of idle slots had been restricted. The simulation is then re-deployed with the HEFT algorithm and its insertion policy enabled.

For this purpose, again, the simulation was run for 120 seconds. Once completed, the average time reported to predict the failure of an engine (any engine) is 3.637 seconds, and the maximum time 6.314 seconds. Although the average execution time has improved (4.391 to 3.637), the maximum execution time has remained the same. This is because, due to the number of nodes in the federation, their hardware configuration and the workflows related to the tasks to be processed in the system, at a specific time, the insertion policies cannot be applied as there are no idle slots available.

Thus, in order to reduce the maximum processing time for the failure prediction of the engines of the rolling mill, this software improvement is not enough, so it is determined to double the computational capacity of the fog nodes. After this improvement, the maximum processing time has been reduced to 3.259 seconds, about half.

Analysing why the processing of worst-case tasks is reduced a half, the simulator logs reported that the worst case (maximum execution time) occurs at an instant when the tasks (related to the worse case) are processed by fog nodes. Specifically for the fog node deployed near the rolling mills. Thus, by doubling its computing power, the processing of these tasks is reduced by about half.

These are some of the tests that the proposed simulator allows to carry out. These tests allow users to analyse the performance of their IoT systems and re-design them until reaching an optimal configuration that satisfies their performance requirements In this case study, users could have tested the impact of other adjustments, such as the modification of the system architecture (adding or subtracting nodes), modifying the workflows of each task, the features of the links that inter-connects each node to the federation, etc.

8. Conclusions and Future Work

Model-driven development (MDD) offers an effective solution for dealing with the technological complexity of domains where diverse technologies are used. The key of MDD lies in its emphasis on the creation of abstractions of the application domain using the four-layer metamodel architecture. This architecture facilitates a structured approach towards system design.

Once these models are established, we can proceed to the model-to-text (M2T) transformation stage, where the developed models are transformed into executable code specific to the technology in use. This approach effectively mitigates the risks and challenges of manual coding, enhancing productivity and reducing the margin for error.

This paper proposes an extension of the SimulateIoT domain-specific language (DSL) towards IoT simulation in the context of task scheduling and the cloud-to-thing continuum paradigm. This DSL extension aids users in conceptually framing their task scheduling proposals within their IoT system designs based on the cloud-to-thing continuum paradigm. By using this language, users can propose, simulate, and evaluate various IoT system designs and task scheduling solutions, thereby working towards a system that fulfils their specific requirements, such as Quality of Service (QoS) or Service Level Agreements (SLAs).

The system design's components can include a variety of elements including cloud, fog, edge, and mist nodes. These nodes can be federated to create an integrated system. Additionally, devices and applications that generate and offload workflow-based tasks can be incorporated, along with the necessary architecture for processing these tasks. This broad range of possibilities allows users to model realistic IoT systems where task scheduling plays a pivotal role.

Finally, once the IoT system is modelled and simulated, the simulation's outputs can be gathered, analysed, and leveraged for the purpose of system refinement and optimisation. This iterative process of design, simulation, deployment, and analysis serves as a feedback loop, enabling continuous improvement of the system in line with the user's evolving needs and technological advancements. This process, founded on the principles of the MDD, ensures a systematic, rigorous approach to designing and test task scheduling proposals and complex IoT systems based on the cloud-to-thing continuum paradigm.

Regarding the limitations of our extension, there are several points to consider. While we offer valuable logs pertaining to the energy consumption of the simulated IoT system, the extension does not encompass a comprehensive model that directly indicates the system's energy usage. Given the emphasis on energy efficiency in the current climate change scenario and the rising trend of energy-awareness task scheduling algorithms (green IoT) (Ghafari et al. 2022), this absence is a notable limitation of our tool.

Additionally, although our extension provides logs concerning the hardware consumption of each simulated component, potentially allowing for cost inference for deploying these components on private clouds like AWS or Google Cloud, it does not furnish a full-fledged pricing model. As many contemporary task scheduling algorithms consider the costs associated with IoT system deployments (Shu et al. 2021; Yuan et al. 2020), this omission is another significant limitation.

Moreover, since SimulateIoT emulates the infrastructure of modelled IoT systems rather than merely simulating it, the hardware demands for running a simulation are higher than for simulators reliant solely on mathematical models. This increases the hardware requirements to run simulations, which can pose challenges for scalability, especially when simulating large IoT systems.

Lastly, while SimulateIoT is designed to simplify the process of integrating and testing user task scheduling proposals, manual effort is still required to carry out this integration.

As for future work, there are several extensions that could be interesting to develop:

- Mobility: The proposed simulator does not support device mobility. Currently, some works in the literature focus on the study of the federation of an edge layer composed

of mobile devices, the mobile edge computing paradigm (Mao et al. 2017; Maray & Shuja 2022). In this computing paradigm, mobile nodes belonging to the edge layer can leave or join the federation. This dynamic property of the edge layer requires an architecture to support it and the corresponding software to handle it. Thus, the inclusion of this paradigm in the simulator could be an advantage for work that focuses on the development of task scheduling techniques for this kind of system (Ma et al. 2021; Wang et al. 2021).
– Energy consumption: Currently, many task scheduling proposals focus on the sustainable development of the IoT, thus prioritising energy consumption optimisation over the makespan of the tasks to be processed (Ghafari et al. 2022). Introducing the concept of devices' batteries or system energy consumption to the proposed simulator could help those users who require test their task scheduling proposals for energy optimisation.
– Cost of use: Given that several cloud platforms offer their services for a certain price and also that energy has a monetary cost, in literature there are several task scheduling proposals focused on optimising the use of system resources (Shu et al. 2021; Yuan et al. 2020). Integrating the concept of resources cost or the model of *pay-as-you-use* could be interesting to allow these users to test their proposals.
– A textual concrete syntax will be developed in order to facilitate the modelling of this kind of system by using a textual notation.
– Taking into account that currently users have to manage the QoS and SLAs manually, it could be interesting to consider an extension of the proposed metamodel to define QoS and SLAs. Thus, facilitating users the modelling and handling of such concepts for each model.

Acknowledgments

This work was funded by project TED2021-129194B-I00 funded by MCIN/ AEI/ 10.13039/501100011033 and for European Union NextGenerationEU/ PRTR; the Government of Extremadura, Council for Economy, Science and Digital Agenda under the grant GR21133 and the projects IB20058, and by the European Regional Development Fund (ERDF).

Proyecto TED2021-129194B-I00 financiado por:

MINISTERIO DE CIENCIA E INNOVACIÓN

Financiado por la Unión Europea NextGenerationEU

Plan de Recuperación, Transformación y Resiliencia

AGENCIA ESTATAL DE INVESTIGACIÓN

JUNTA DE EXTREMADURA

Consejería de Economía, Ciencia y Agenda Digital

References

Aazam, M., Zeadally, S., & Harras, K. A. (2018). Deploying fog computing in industrial internet of things and industry 4.0. *IEEE Transactions on Industrial Informatics*, *14*(10), 4674-4682. doi: 10.1109/TII.2018.2855198

Ahmad, Z., Jehangiri, A. I., Ala'anzy, M. A., Othman, M., Latip, R., Zaman, S. K. U., & Umar, A. I. (2021). Scientific workflows management and scheduling in cloud computing: taxonomy, prospects, and challenges. *IEEE Access*, *9*, 53491–53508.

Alizadeh, M. R., Khajehvand, V., Rahmani, A. M., & Akbari, E. (2020). Task scheduling approaches in fog computing: A systematic review. *International Journal of Communication Systems*, *33*(16), e4583.

Al-Maytami, B. A., Fan, P., Hussain, A., Baker, T., & Liatsis, P. (2019). A task scheduling algorithm with improved makespan based on prediction of tasks computation time algorithm for cloud computing. *IEEE Access*, *7*, 160916–160926.

Andersson, M. A., Özçelikkale, A., Johansson, M., Engström, U., Vorobiev, A., & Stake, J. (2016). Feasibility of ambient rf energy harvesting for self-sustainable m2m communications using transparent and flexible graphene antennas. *IEEE Access*, *4*, 5850-5857. doi: 10.1109/ACCESS.2016.2604078

Arunarani, A., Manjula, D., & Sugumaran, V. (2019a). Task scheduling techniques in cloud computing: A literature survey. *Future Generation Computer Systems*, *91*, 407-415. Retrieved from https://www.sciencedirect.com/science/article/pii/S0167739X17321519 doi: https://doi.org/10.1016/j.future.2018.09.014

Arunarani, A., Manjula, D., & Sugumaran, V. (2019b). Task scheduling techniques in cloud computing: A literature survey. *Future Generation Computer Systems*, *91*, 407–415.

Asghari, A., Sohrabi, M. K., & Yaghmaee, F. (2021). Task scheduling, resource provisioning, and load balancing on scientific workflows using parallel sarsa reinforcement learning agents and genetic algorithm. *The Journal of Supercomputing*, *77*, 2800–2828.

Atkinson, C., & Kuhne, T. (2003). Model-driven development: a metamodeling foundation. *IEEE software*, *20*(5), 36–41.

Bansal, S., Aggarwal, H., & Aggarwal, M. (2022). A systematic review of task scheduling approaches in fog computing. *Transactions on Emerging Telecommunications Technologies*, e4523.

Barriga, J. A., Clemente, P. J., Pérez-Toledano, M. A., Jurado-Málaga, E., & Hernández, J. (2023). Design, code generation and simulation of iot environments with mobility devices by using model-driven development: Simulateiot-mobile. *Pervasive and Mobile Computing*, *89*, 101751. Retrieved from https://www.sciencedirect.com/science/article/pii/S1574119223000093 doi: https://doi.org/10.1016/j.pmcj.2023.101751

Barriga, J. A., Clemente, P. J., Sosa-Sánchez, E., & Prieto, A. E. (2021). Simulateiot: Domain specific language to design, code generation and execute iot simulation environments. *IEEE Access*, *9*, 92531-92552. doi: 10.1109/ACCESS.2021.3092528

Bittencourt, L., Immich, R., Sakellariou, R., Fonseca, N., Madeira, E., Curado, M., … Rana, O. (2018). The internet of things, fog and cloud continuum: Integration and challenges. *Internet of Things*, *3*, 134–155.

Cakir, M., Guvenc, M. A., & Mistikoglu, S. (2021). The experimental application of popular machine learning algorithms on predictive maintenance and the design of iiot based condition monitoring system. *Computers & Industrial Engineering*, *151*, 106948. Retrieved from https://www.sciencedirect.com/science/article/pii/S0360835220306252 doi: https://doi.org/10.1016/j.cie.2020.106948

Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A., & Buyya, R. (2011). Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, *41*(1), 23–50.

Carvalho, T. P., Soares, F. A., Vita, R., Francisco, R. d. P., Basto, J. P., & Alcalá, S. G. (2019). A systematic literature review of machine learning methods applied to predictive maintenance. *Computers & Industrial Engineering*, *137*, 106024.

Chen, W., & Deelman, E. (2012). Workflowsim: A toolkit for simulating scientific workflows in distributed environments. In *2012 ieee 8th international conference on e-science* (p. 1-8). doi: 10.1109/eScience.2012.6404430

Cheng, J. C., Chen, W., Chen, K., & Wang, Q. (2020). Data-driven predictive maintenance planning framework for mep components based on bim and iot using machine learning algorithms. *Automation in Construction*, *112*, 103087.

Chernyshev, M., Baig, Z., Bello, O., & Zeadally, S. (2018). Internet of things (iot): Research, simulators, and testbeds. *IEEE Internet of Things Journal*, *5*(3), 1637-1647. doi: 10.1109/JIOT.2017.2786639

Çınar, Z. M., Abdussalam Nuhu, A., Zeeshan, Q., Korhan, O., Asmael, M., & Safaei, B. (2020). Machine learning in predictive maintenance towards sustainable smart manufacturing in industry 4.0. *Sustainability*, *12*(19), 8211.

Ding, D., Fan, X., Zhao, Y., Kang, K., Yin, Q., & Zeng, J. (2020). Q-learning based dynamic task scheduling for energy-efficient cloud computing. *Future Generation Computer Systems*, *108*, 361–371.

Divyaprabha, M., Priyadharshni, V., & Kalpana, V. (2018). Modified heft algorithm for workflow scheduling in cloud computing environment. In *2018 second international conference on inventive communication and computational technologies (icicct)* (p. 812-815). doi: 10.1109/ICICCT.2018.8473237

Erazo, M. A., & Liu, J. (2013). Leveraging symbiotic relationship between simulation and emulation for scalable network experimentation. In *Proceedings of the 1st acm sigsim conference on principles of advanced discrete simulation* (pp. 79–90).

Faragardi, H. R., Saleh Sedghpour, M. R., Fazliahmadi, S., Fahringer, T., & Rasouli, N. (2020). Grp-heft: A budget-constrained resource provisioning scheme for workflow scheduling in iaas clouds. *IEEE Transactions on Parallel and Distributed Systems*, *31*(6), 1239-1254. doi: 10.1109/TPDS.2019.2961098

Fisher, A., Rudin, C., & Dominici, F. (2019). All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously. *J. Mach. Learn. Res.*, *20*(177), 1–81.

Gazori, P., Rahbari, D., & Nickray, M. (2020). Saving time and cost on the scheduling of fog-based iot applications using deep reinforcement learning approach. *Future Generation Computer Systems*, *110*, 1098–1115.

Ghafari, R., Kabutarkhani, F. H., & Mansouri, N. (2022). Task scheduling algorithms for energy optimization in cloud environment: a comprehensive review. *Cluster Computing*, *25*(2), 1035–1093.

Girs, S., Sentilles, S., Asadollah, S. A., Ashjaei, M., & Mubeen, S. (2020). A systematic literature study on definition and modeling of service-level agreements for cloud services in iot. *IEEE Access*, *8*, 134498–134513.

Gluhak, A., Krco, S., Nati, M., Pfisterer, D., Mitton, N., & Razafindralambo, T. (2011). A survey on facilities for experimental internet of things research. *IEEE Communications Magazine*, *49*(11), 58-67. doi: 10.1109/MCOM.2011.6069710

Gupta, H., Vahid Dastjerdi, A., Ghosh, S. K., & Buyya, R. (2017). ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, *47*(9), 1275–1296.

Gupta, S., Iyer, S., Agarwal, G., Manoharan, P., Algarni, A. D., Aldehim, G., & Raahemifar, K. (2022). Efficient prioritization and processor selection schemes for heft algorithm: A makespan optimizer for task scheduling in cloud environment. *Electronics*, *11*(16), 2557.

H., S., & V., N. (2021). A review on fog computing: Architecture, fog with iot, algorithms and research challenges. *ICT Express*, *7*(2), 162-176. Retrieved from https://www.sciencedirect.com/science/article/pii/S2405959521000606 doi: https://doi.org/10.1016/j.icte.2021.05.004

Hao, Q., Xue, Y., Shen, W., Jones, B., & Zhu, J. (2010). A decision support system for integrating corrective maintenance, preventive maintenance, and condition-based maintenance. In *Construction research congress 2010: Innovation for reshaping construction practice* (pp. 470–479).

Hosseinioun, P., Kheirabadi, M., Kamel Tabbakh, S. R., & Ghaemi, R. (2022). atask scheduling approaches in fog computing: A survey. *Transactions on Emerging Telecommunications Technologies*, *33*(3), e3792. Retrieved from https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.3792 (e3792 ETT-19-0285.R1) doi: https://doi.org/10.1002/ett.3792

Jamil, B., Ijaz, H., Shojafar, M., Munir, K., & Buyya, R. (2022). Resource allocation and task scheduling in fog computing and internet of everything environments: A taxonomy, review, and future directions. *ACM Computing Surveys (CSUR)*.

Kar, B., Yahya, W., Lin, Y.-D., & Ali, A. (2022). A survey on offloading in federated cloud-edge-fog systems with traditional optimization and machine learning. *arXiv preprint arXiv:2202.10628*.

Kolovos, D. S., García-Domínguez, A., Rose, L. M., & Paige, R. F. (2015). Eugenia: towards disciplined and automated development of GMF-based graphical model editors. *Software & Systems Modeling*, 1–27.

Kompose. (2023). https://kompose.io/. ([Online; accessed 09-Oct-2023])

Kubernetes Documentation. (2023). https://kubernetes.io/docs/home/. ([Online; accessed 09-Oct-2023])

Lera, I., Guerrero, C., & Juiz, C. (2019). Yafs: A simulator for iot scenarios in fog computing. *IEEE Access*, *7*, 91745-91758. doi: 10.1109/ACCESS.2019.2927895

Lie, C. H., & Chun, Y. H. (1986). An algorithm for preventive maintenance policy. *IEEE Transactions on Reliability*, *35*(1), 71-75. doi: 10.1109/TR.1986.4335352

Ma, X., Zhou, A., Zhang, S., Li, Q., Liu, A. X., & Wang, S. (2021). Dynamic task scheduling in cloud-assisted mobile edge computing. *IEEE Transactions on Mobile Computing*.

Mao, Y., You, C., Zhang, J., Huang, K., & Letaief, K. B. (2017). A survey on mobile edge computing: The communication perspective. *IEEE communications surveys & tutorials*, *19*(4), 2322–2358.

Maray, M., & Shuja, J. (2022). Computation offloading in mobile cloud computing and mobile edge computing: survey, taxonomy, and open issues. *Mobile Information Systems*, *2022*.

McGregor, I. (2002). The relationship between simulation and emulation. In *Proceedings of the winter simulation conference* (Vol. 2, p. 1683-1688 vol.2). doi: 10.1109/WSC.2002.1166451

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, *2014*(239), 2.

Mijuskovic, A., Chiumento, A., Bemthuis, R., Aldea, A., & Havinga, P. (2021). Resource management techniques for cloud/fog and edge computing: An evaluation framework and classification. *Sensors*, *21*(5). Retrieved from https://www.mdpi.com/1424-8220/21/5/1832 doi: 10.3390/s21051832

Mishra, S., Mishra, S., Kayal, A., & Chudi, S. R. (2012). Simulation in wireless sensor networks. *International Journal of Electronics Communication and Computer Technology (IJECCT)*, *2*(4), 176.

NoorianTalouki, R., Shirvani, M. H., & Motameni, H. (2022). A heuristic-based task scheduling algorithm for scientific workflows in heterogeneous cloud computing platforms. *Journal of King Saud University-Computer and Information Sciences*, *34*(8), 4902–4913.

Obeo. (2012). *Acceleo project http://www.acceleo.org.*

Ojha, S. K., Rai, H., & Nazarov, A. (2020). Enhanced modified heft algorithm for task scheduling in cloud environment. In *2020 2nd international conference on advances in computing, communication control and networking (icacccn)* (p. 866-870). doi: 10.1109/ICACCCN51052.2020.9362975

OMG. (2012, January). *OMG Object Constraint Language (OCL), Version 2.3.1.* Retrieved from http://www.omg.org/spec/OCL/2.3.1/

Pan, J., & McElhannon, J. (2018). Future edge cloud and edge computing for internet of things applications. *IEEE Internet of Things Journal*, *5*(1), 439-449. doi: 10.1109/JIOT.2017.2767608

Potluri, S., & Rao, K. S. (2020). Optimization model for qos based task scheduling in cloud computing environment. *Indonesian Journal of Electrical Engineering and Computer Science*, *18*(2), 1081–1088.

Qian, L., Luo, Z., Du, Y., & Guo, L. (2009). Cloud computing: An overview. In *Cloud computing: First international conference, cloudcom 2009, beijing, china, december 1-4, 2009. proceedings 1* (pp. 626–631).

Rashid, A., & Chaturvedi, A. (2019). Cloud computing characteristics and services: a brief review. *International Journal of Computer Sciences and Engineering*, *7*(2), 421–426.

Rodrigo, G. P., Elmroth, E., Östberg, P.-O., & Ramakrishnan, L. (2018). Scsf: A scheduling simulation framework. In D. Klusáček, W. Cirne, & N. Desai (Eds.), *Job scheduling strategies for parallel processing* (pp. 152–173). Cham: Springer International Publishing.

Saltelli, A., & Funtowicz, S. (2014). When all models are wrong. *Issues in Science and Technology*, *30*(2), 79–85.

Samann, F. E. F., Zeebaree, S. R., & Askar, S. (2021). Iot provisioning qos based on cloud and fog computing. *Journal of Applied Science and Technology Trends*, *2*(01), 29–40.

Sandhu, M. M., Khalifa, S., Jurdak, R., & Portmann, M. (2021). Task scheduling for energy-harvesting-based iot: A survey and critical analysis. *IEEE Internet of Things Journal*, *8*(18), 13825–13848.

Selic, B. (2003). The pragmatics of model-driven development. *IEEE software*, *20*(5), 19–25.

Sendall, S., & Kozaczynski, W. (2003). Model transformation: The heart and soul of model-driven software development. *IEEE software*, *20*(5), 42–45.

Shu, W., Cai, K., & Xiong, N. N. (2021). Research on strong agile response task scheduling optimization enhancement with optimal resource usage in green cloud computing. *Future Generation Computer Systems*, *124*, 12–20.

Singh, P., Dutta, M., & Aggarwal, N. (2017, Jul 01). A review of task scheduling based on meta-heuristics approach in cloud computing. *Knowledge and Information Systems*, *52*(1), 1-51. Retrieved from https://doi.org/10.1007/s10115-017-1044-2 doi: 10.1007/s10115-017-1044-2

Siow, E., Tiropanis, T., & Hall, W. (2018). Analytics for the internet of things: A survey. *ACM Computing Surveys (CSUR)*, *51*(4), 74.

Sterman, J. D. (2002). All models are wrong: reflections on becoming a systems scientist. *System Dynamics Review: The Journal of the System Dynamics Society*, *18*(4), 501–531.

Topcuoglu, H., Hariri, S., & Wu, M.-Y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, *13*(3), 260–274.

Wang, W., Lu, B., Li, Y., Wei, W., Li, J., Mumtaz, S., & Guizani, M. (2021). Task scheduling game optimization for mobile edge computing. In *Icc 2021-ieee international conference on communications* (pp. 1–6).

Wu, F., Wu, Q., & Tan, Y. (2015). Workflow scheduling in cloud: a survey. *The Journal of Supercomputing*, *71*(9), 3373–3418.

Yao, F., Pu, C., & Zhang, Z. (2021). Task duplication-based scheduling algorithm for budget-constrained workflows in cloud computing. *IEEE Access*, *9*, 37262–37272.

Yu, C., Lin, B., Guo, P., Zhang, W., Li, S., & He, R. (2018).

Deployment and dimensioning of fog computing-based internet of vehicle infrastructure for autonomous driving. *IEEE Internet of Things Journal*, *6*(1), 149–160.

Yuan, H., Liu, H., Bi, J., & Zhou, M. (2020). Revenue and energy cost-optimized biobjective task scheduling for green cloud data centers. *IEEE Transactions on Automation Science and Engineering*, *18*(2), 817–830.

Zhao, J., Gao, C., & Tang, T. (2022). A review of sustainable maintenance strategies for single component and multicomponent equipment. *Sustainability*, *14*(5). Retrieved from https://www.mdpi.com/2071-1050/14/5/2992 doi: 10.3390/su14052992

About the authors

José A. Barriga obtained his BSc degree in 2018 at the University of Extremadura and his MSc degree in 2019 at the International University of La Rioja. Currently, he is a PhD student at the University of Extremadura. He has been working for five years in the areas of IoT systems simulation, model-driven development applied to the IoT and machine learning applied to agriculture. You can contact the author at jose@unex.es.

José M. Cháves-González Jose M. Cháves-González is an Associate Professor of the Computer Science Department at the University of Extremadura (Spain). He received his BSc in Computer Science from the University of Extremadura in 2005 and a PhD in Computer Science in 2011. His research activity focuses on problem solving in the field of bioinformatics, the design and development of evolutionary and bio-inspired algorithms, multi-objective optimisation and the optimisation of algorithms using parallelism techniques. You can contact the author at jm@unex.es.

Arturo Barriga is a junior researcher in the Quercus Software Engineering Group at the University of Extremadura. He obtained his BSc degree from the University of Extremadura, Spain, 2022. Currently, he is a MSc student at the International University of La Rioja. His research focuses on the fields of digital twins and machine learning applied to agriculture. You can contact the author at arturobc@unex.es.

Pablo Alonso is a junior researcher in the Quercus Software Engineering Group at the University of Extremadura. He obtained his BSc degree in computer science from the University of Extremadura, Spain, in 2022. Currently, his research focuses on the fields of digital twins and simulation of Internet of Things (IoT) environments. You can contact the author at pabloap@unex.es.

Pedro J. Clemente is an Associate Professor of the Computer Science Department at the University of Extremadura (Spain). He received his BSc in Computer Science from the University of Extremadura in 1998 and a PhD in Computer Science in 2007. He has published numerous peer-reviewed papers in international journals, workshops, and conferences. His research interests include component-based software development, aspect orientation, service-oriented architectures, business process modelling, and model-driven development. He is involved in several research projects. He has participated in many workshops and conferences as speaker and member of the program committees. You can contact the author at pjclemente@unex.es

.

A. Appendix: The complete metamodel of the proposed simulator

This Section shows in Figure 11 the complete metamodel of the proposed simulator. This metamodel is composed of the SimulateIoT metamodel and the extension carried out (highlighted in blue). The description of the classes and relationships that are not part of the extension (and that have not been addressed in this article), can be found in the article (Barriga et al. 2021) Section IV, subsection A.

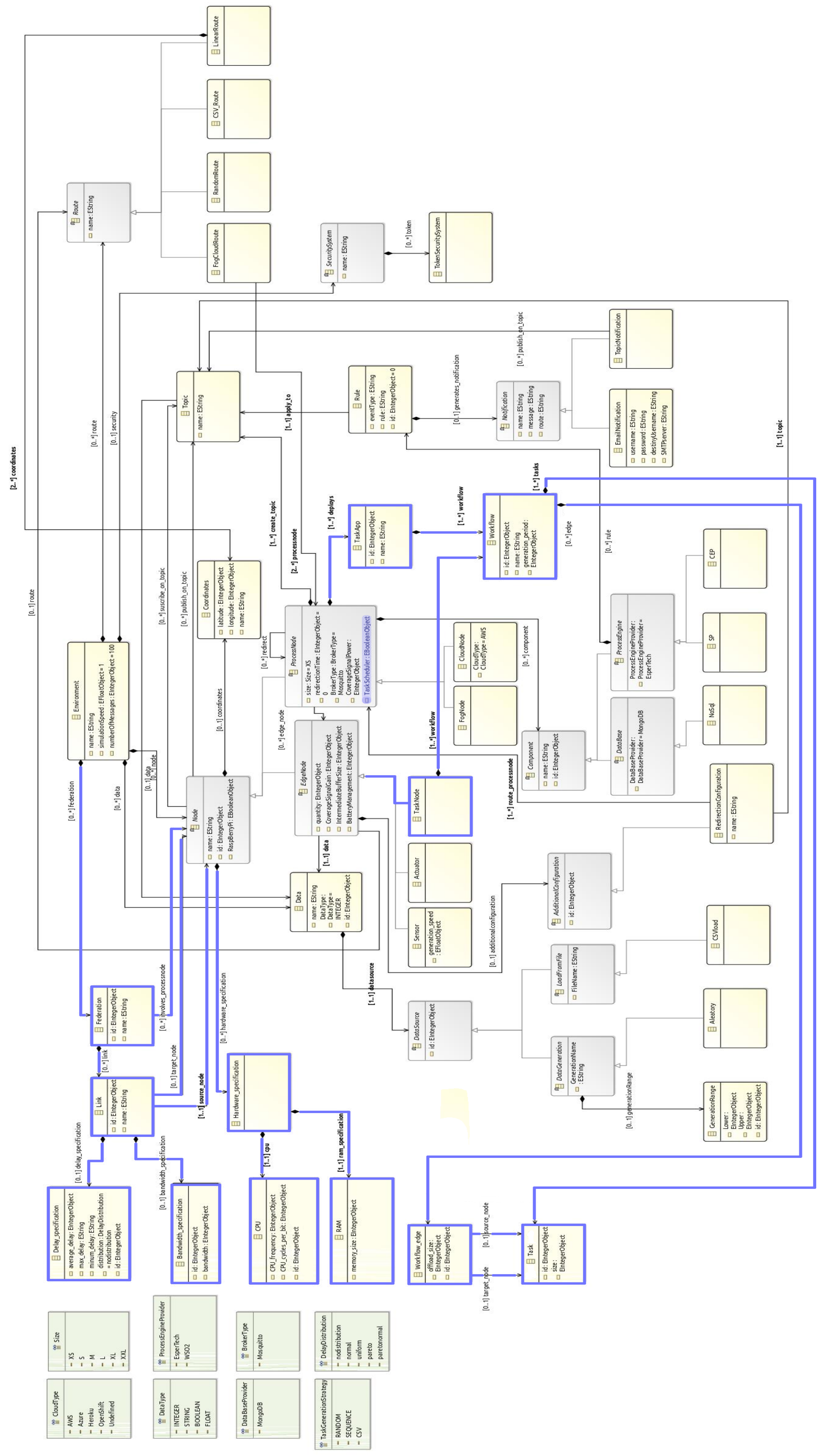


Figure 11 Complete SimulateIoT metamodel with the extension concepts and relationships included.

B. Appendix: Acceleo M2T example

```
[template public generateWorkflow(anEnvironment : Environment)]
[comment @main/]
[for (tnode : TaskNode | anEnvironment.node->filter(EdgeNode)->filter(TaskNode))]
  [for (workflow : Workflow | tnode.workflow)]
    [file ('/' + tnode.name + tnode.id + '/src/main/resources/wflows/wflow' + workflow.id + '.json', false, 'UTF-8')]

{
    "workflow": {
        "id": "[workflow.id/]",
        "name": "[workflow.name/]",
        "generatedBy": {
            "nodeId": "TaskNode[tnode.name + tnode.id/]",
            "generatorId": "[tnode.name + tnode.id/]",
            "generationId": "" // This ID acts as a counter within the component responsible for generating workflows,
 and is added during simulations.
        },
        "nodes":
            [for (task : Task | workflow.task) before('[') separator(', ') after(']')]
            {
                "task": {
                    "name": "[task.name/]",
                    "id": "[task.id/]",
                    "size": "[task.size/]"
                }
            }
      [/for]
    [if (workflow.edge->size()>0)]
    "edges":
    [/if]
    [for (edge : Edge | workflow.edge) before('[') separator(', ') after(']')]
            {
                "edge": {
                    "id": "[edge.id/]",
                    "sourceTaskId": "[edge.source.id/]",
                    "targetTaskId": {
            [for (target : Task | edge.target) separator(', ')]
            "target[i/]" : "[edge.id/]"
            [/for]
          },
                    "offloadSize": "[edge.offload_size/]"
                }
            }
        [/for]
    }
}
    [/file]
  [/for]
[/for]
[/template]
```

Listing 5 M2T developed in Acceleo for the generation of the workflows that the Task Nodes will offload to the system during the simulation.

C. Networking Node

Figure 12 shows a generic `Networking Node` Ⓑ (represented by a red box), all its components (elements within the red box) and the interaction between them. The main components of the `Networking Node` are the `Delay Controller` Ⓒ, the `Synthetic Delay Generator` Ⓓ, the `Bandwidth Controller` Ⓔ and the `Network Status Reporter` Ⓖ. Besides, Figure 12 also shows how the `Networking Node` is deployed on an edge (`Task Node`), fog or cloud node Ⓐ and the interactions that these components could perform with other artefacts of the edge/fog/cloud node and with the rest of the IoT system. Below, the `Networking Node` is illustrated by describing each of its components and their interactions.

`Delay Controller` Ⓒ The `Delay Controller` aims to apply the delay of the `Links` that connect the nodes belonging to a federation. The delay is applied from the source node to the target node, i.e. the `Delay Controller` applies the delay constraints modelled to the outgoing traffic. Note that, as there is one `Networking Node` per node belonging to a federation, the `Delay Controller` applies the delay modelled to those `Links` whose source node is the edge/fog/cloud node where it is deployed.

In this regard, tasks are transmitted by means of workflows in JSON format (Listing 1). This JSON code has among its fields the target node of the workflow (Listing 1, field *generatedBy*), i.e. where the workflow has to be sent. In this way, when the `Delay Controller` receives an outgoing workflow (interaction ①), it is able to identify the `Link` through which the workflow has to be sent. This also applies to the outgoing tasks processed in the node where the `Networking Node` is integrated (interaction ②).

Thus, with this information, the `Delay Controller` request to the `Synthetic Delay Generator` Ⓓ the current delay of the `Link` (interaction ③). Note that the delay is generated synthetically following user modelling. Once the response is received (interaction ④), the `Delay Controller` holds tasks during the received delay, thus simulating it. Finally, when the delay has been simulated, the traffic is forwarded to the `Bandwidth Controller` Ⓔ (interaction ⑤).

`Synthetic Delay Generator` Ⓓ Users can model the delay of each `Link` (average, minimum, maximum, etc.) that connects each node in a federation. Thus, the aim of the `Synthetic Delay Generator` is to generate the delay of each `Link` during simulation.

Thus, the `Synthetic Delay Generator` interacts with the `Delay Controller` Ⓒ and with the `Network Status Reporter` Ⓖ of the same `Networking Node`. In this way, when any of these components need to know the delay of a specific `Link`, they request it to the `Synthetic Delay Generator` (interaction ③ and ⑩). Then, the `Synthetic Delay Generator` responds to them with the delay of the `Link` requested (interaction ④ and ⑪).

`Bandwidth Controller` Ⓔ The `Bandwidth Controller` aims to apply the bandwidth constraints of the `Links` that connect the nodes belonging to a federation. Thus, the bandwidth is applied from source to target, i.e. the `Bandwidth Controller` applies the bandwidth constraints modelled to the outgoing traffic. Note that, as in the case of delay, the `Bandwidth Controller` applies the bandwidth modelled from source, to target.

Thus, the `Bandwidth Controller` receives traffic (workflows or results related to a processed task) from the `Delay Controller` (interaction ⑤). If traffic t_i is received and no traffic is being transmitted, the `Bandwidth Controller` holds the traffic t_i for the time resulting from applying the mathematical Expression 1. Thus, simulating the time that the traffic would have needed to be transmitted in a real environment.

$$TT_{t_i} = \frac{TS_{t_i}}{LB_{t_i}} \tag{1}$$

Where TT_{t_i} is the transmission time (seconds) required to send a traffic t_i of a specific size TS_{t_i} (bytes) through a `Link` with a bandwidth LB_{t_i} (bytes/seconds).

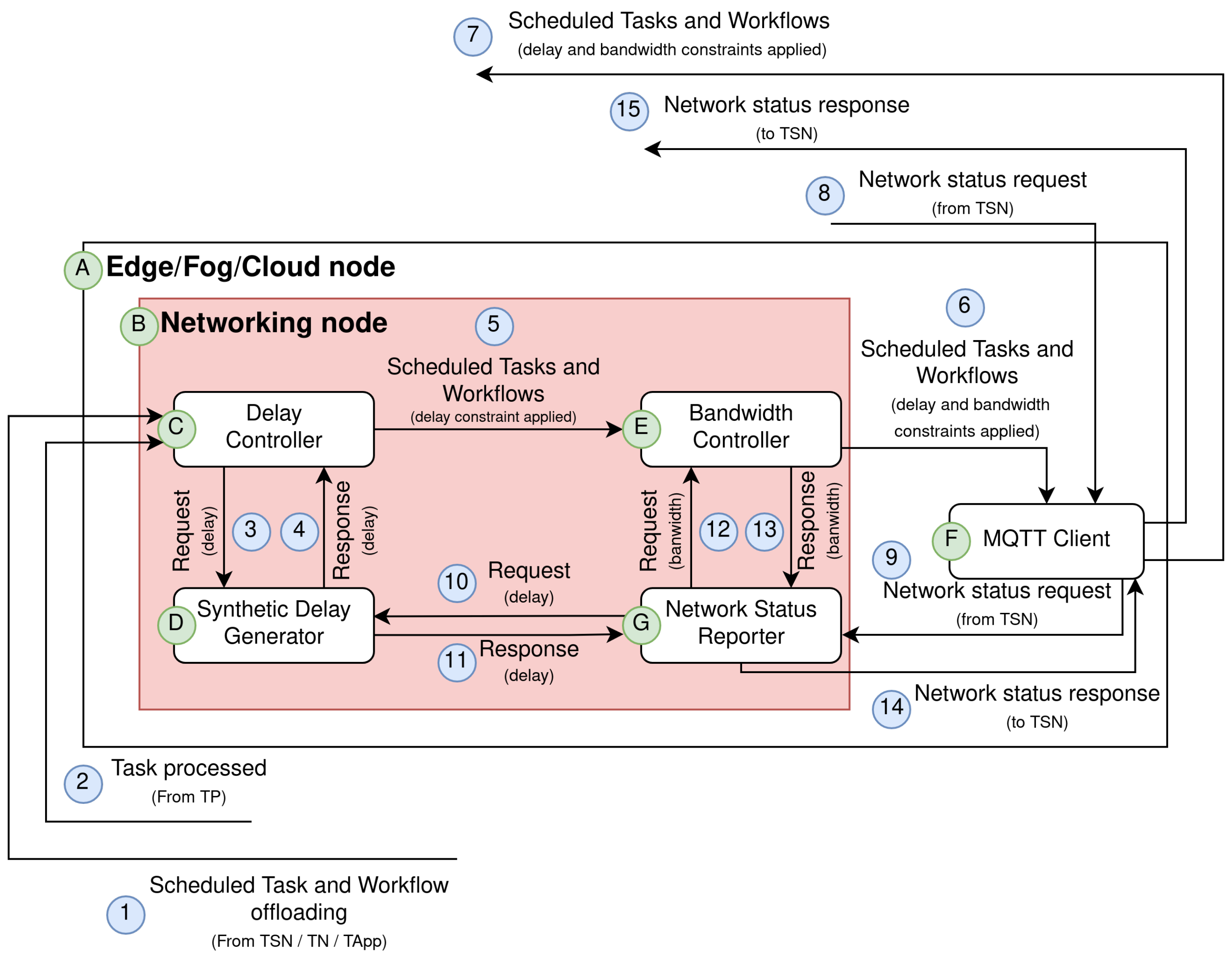
On the other hand, if a traffic t_n arrives at the `Bandwidth Controller`, but there are $n-1$ workflows or processed tasks (traffic) being transmitted or pending to be transmitted through the same `Link` over which the traffic t_n has to be transmitted, traffic t_n is queued in a FIFO (First In First Out) traffic queue. So, in this case, the transmission time of the traffic t_n can be determined by the Expression 2. Thus, simulating the time that the traffic t_n would have needed to be transmitted in a real environment.

$$TT_{t_n} = \left(\sum_{i=1}^{n} \frac{TS_{t_i}}{LB_{t_i}} \right) + \frac{RT_{t_0}}{LB_{t_i}} \tag{2}$$

Where a) TT_{t_n} is the transmission time required to send a traffic t_n with a size of TS_{t_n} bytes through a `Link` with a bandwidth of LB_{t_i} bytes, b) over which a workflow or processed task (traffic) t_0 is being transmitted and RT_{t_0} bytes of this traffic remain to be transmitted (when t_n arrives), and c) a set of $n-1$ workflows or processed tasks (traffic) are pending to be transmitted (queued before t_n).

Thus, once the delay and bandwidth constraints are applied, the traffic is sent to the `MQTT Client` (interaction ⑥), which forwards it to its target node (interaction ⑦). Finally, note that for the sake of clarity, interactions ⑫ and ⑬ are described below as part of the `Network Status Reporter` Ⓖ description.

`Network Status Reporter` Ⓖ The `Task Scheduler` of a federation could need to request the status of the `Links` (delay and bandwidth use) of the federation. Thus, it can use this data as input to perform the scheduling of the offloaded tasks. In this regard, the `Network Status Reporter` of each node belonging to a federation is the node that receives this request and responds to it with the current delay and available bandwidth of each `Link`. Note that, as there is one `Networking Node` per node belonging to a federation, the `Network Status Reporter` responds with the delay and bandwidth of those



7 Scheduled Tasks and Workflows (delay and bandwidth constraints applied)
15 Network status response (to TSN)
8 Network status request (from TSN)
A Edge/Fog/Cloud node
B Networking node
5 Scheduled Tasks and Workflows (delay constraint applied)
6 Scheduled Tasks and Workflows (delay and bandwidth constraints applied)
C Delay Controller
E Bandwidth Controller
F MQTT Client
Request (delay)
3
4
Response (delay)
Request (banwidth)
12
13
Response (banwidth)
9 Network status request (from TSN)
10 Request (delay)
D Synthetic Delay Generator
G Network Status Reporter
11 Response (delay)
14 Network status response (to TSN)
2 Task processed (From TP)
1 Scheduled Task and Workflow offloading (From TSN / TN / TApp)

Figure 12 Networking Node component.

Links whose source node is the edge/fog/cloud node where it is deployed.

Since the Task Scheduler carries out these requests through the MQTT protocol, the MQTT Client Ⓕ is the first to receive them (interaction ⑧). Then, the MQTT Client forwards these requests to the Network Status Reporter. Following, the Network Status Reporter requests the current delay of each Link to the Synthetic Delay Generator (interactions ⑩ and ⑪) and the current use of bandwidth of each Link to the Bandwidth Controller Ⓔ (interactions ⑫ and ⑬). Once the Network Status Reporter has gathered all the requested data, it sends this data to the MQTT Client (interaction ⑭), which finally forwards the data to the Task Scheduler.

D. Task Processor

Figure 13 shows a generic Task Processor Ⓑ (represented by a red box), all its components (elements within the red box) and the interaction between them. The main components of the Task Processor are the Task Manager Ⓓ, the Task Performer Ⓔ and the Task Processor Status Reporter Ⓕ. Moreover, Figure 13 also shows how the Task Processor is deployed on an edge (Task Node), fog or cloud node Ⓐ and the interactions that these components could perform with other artefacts of the edge/fog/cloud node and with the rest of the IoT system. Below, the Task Processor is illustrated by describing each of its components and their interactions.

Task Manager Ⓓ The Task Manager aims to ensure that the schedule performed by the Task Scheduler of a federation is followed by the Task Processors that belong to the federation. Note that, as there is one Task Processor per node (with computing power) belonging to a federation, the Task Manager ensures that the schedule performed by the Task Scheduler is followed by the edge/fog/cloud node Ⓐ where it is deployed.

Thus, when tasks reach the Task Processors, as the tasks

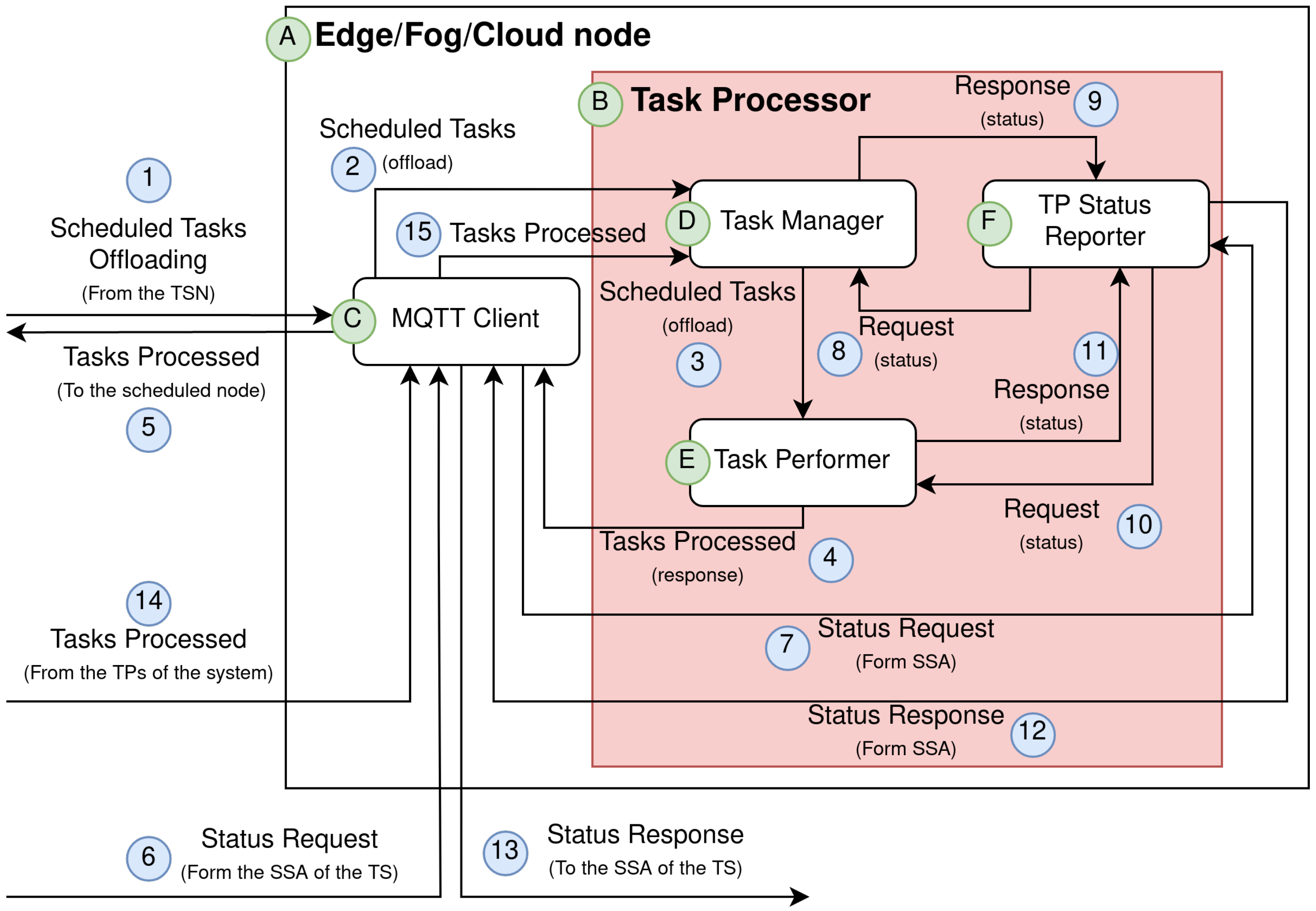

A Edge/Fog/Cloud node
B Task Processor
C MQTT Client
D Task Manager
E Task Performer
F TP Status Reporter
1 Scheduled Tasks Offloading (From the TSN)
2 Scheduled Tasks (offload)
3 Scheduled Tasks (offload)
4 Tasks Processed (response)
5 Tasks Processed (To the scheduled node)
6 Status Request (Form the SSA of the TS)
7 Status Request (Form SSA)
8 Request (status)
9 Response (status)
10 Request (status)
11 Response (status)
12 Status Response (Form SSA)
13 Status Response (To the SSA of the TS)
14 Tasks Processed (From the TPs of the system)
15 Tasks Processed

Figure 13 Task Proccessor components.

are sent through the system using the MQTT protocol, the first component that they reach is the `MQTT Client` Ⓒ of the computing nodes (interaction ①). Later, the `MQTT Client` forwards these tasks to the `TaskManager` (interaction ②).

The `Task Manager` Ⓓ receives, stores (buffer) and sends these tasks (following the schedule performed by the `Task Scheduler`) to the `Task Performer` Ⓔ (interaction ③), which performs their processing.

The `Task Manager` also handles the interdependency among tasks. Consequently, 1) the `Task Manager` holds dependent tasks until the arrival of the processing results of the tasks on which these tasks depend, 2) when these results arrive, the `Task Manager` includes in the task the processing results of the tasks on which it depended 3) Finally, following the schedule the `Task Manager` sends the task to the `Task Performer` for its processing (interaction ③).

Finally, note that for the sake of clarity, interactions ⑧ and ⑨ are described below, in the section reserved for the `Task Processor Status Reporter` Ⓕ.

`Task Performer` Ⓔ The `Task Performer` is the component of the `Task Processor` Ⓑ which performs the processing of the tasks. The `Task Performer` simulates the processing of the tasks holding them the time that would be required for their processing in a real environment. For this purpose, the `Task Performer` applies the expression 3.

$$PT_{t_i} = \frac{TS_{t_i}}{CF_{c_i}/CCB_{c_i}} \tag{3}$$

Where PT_{t_i} is the time (seconds) required to process the task t_i which has a size of TS_{t_i} bits on a CPU c_i with CCB_{p_i} cycles per bit (i.e the cycles that the CPU needs to process a bit) and a frequency of CF_{c_i} (i.e. cycles that the CPU can perform per second). Note that the parameters of the CPU are the CPU attributes that the user can specify to model the CPU of each node.

Once a task is processed, as transmitted in JSON, the `Task Performer` includes in it fields data such as the timestamp related to the start of the processing of the task and the timestamp related to the end of the processing of the task.

Then, the `Task Performer` sends the processed task to the `MQTT Client` (interaction ④), which forwards the processed task to their next target node. For the sake of clarity, interactions ⑩ and ⑪ are described below in the section reserved for the `Task Processor (TP) Status Report` Ⓕ.

`Task Processor Status Reporter` Ⓕ The `Task Scheduler` of the federation could need to know the status of the task processing, i.e. CPU use of each `Task Processor` Ⓑ and the tasks pending of processing. So, the `Task Processor Status Reporter` has the same aim that the `Network Status Reporter` of the `Networking Node` (Section C), although in the context of the `Task Processor`. Thus, in this case, the data that is reported is related to the use of the CPU and RAM of the `Task Processor` (by the `Task Performer` Ⓔ) and the status of the `Task Manager` Ⓓ (`Tasks pending to be processed`). Thus, the `Task Scheduler` can use this data as input to perform the scheduling of the offloaded tasks.

The `Task Scheduler` sends these requests through the MQTT protocol, so the `MQTT Client` Ⓒ is the first to receive them (interaction ⑥). Then, the `MQTT Client` forwards these requests to the `Task Processor Status Reporter` (interaction ⑦). Once the `Task Processor Status Reporter` receives a request, it requests the tasks pending to be processed (their size, estimated queue time, etc.) to the `Task Manager` Ⓓ (interaction ⑧), and the status of the use of the CPU and RAM to the `Task Performer` Ⓔ, (interaction ⑩).

When these components receive the requests from the `Task Processor Status Reporter`, they respond to it with the requested data (interactions ⑨ and ⑪). So, once the `Task Processor Status Reporter` gathers all the CPU, RAM and task processing related data, it forwards this data to the `MQTT Client` (interaction ⑫), which sends it to the `Task Scheduler` (interaction ⑬).