EDICT: simulation of edge interactions across IoT-enhanced environments

Houssam Hajj Hassan, Georgios Bouloukakis, Ajay Kattepur, Denis Conan, Djamel Belaïd

EDICT: Simulation of Edge Interactions across IoT-enhanced Environments

Houssam Hajj Hassan[1], Georgios Bouloukakis[1], Ajay Kattepur[2], Denis Conan[1], Djamel Belaïd[1]
[1]*Télécom SudParis, Institut Polytechnique de Paris, France*
{houssam.hajj_hassan, georgios.bouloukakis, denis.conan, djamel.belaid}@telecom-sudparis.eu
[2]*Ericsson AI Research, India*, ajay.kattepur@ericsson.com

***Abstract*—This paper presents EDICT, a tool for simulating Edge interactions in IoT-enhanced environments. Recently, ML and AI-based techniques have gained prominence to solve IoT-related challenges. However, such models require large and diverse datasets to perform well. Finding real-world datasets that capture the performance of IoT systems is a challenging task due to the cost of deploying devices and instrumenting environments, as well as privacy/security concerns. This task becomes more challenging when datasets for specific situations (e.g., overloaded system, emergency scenarios) are needed. EDICT enables IoT systems designers to evaluate the performance of their IoT systems at design time. EDICT is capable of generating performance metrics datasets for specific instances of IoT-enhanced environments under different configuration parameters. To support runtime adaptation of smart environments, EDICT enables rapid performance prediction using ML techniques.**

***Index Terms*—Simulation, Smart Environments, QoS, Data Exchange**

I. Introduction

With the advent of Internet of Things (IoT) devices and supporting technologies, spaces (e.g. buildings, homes) are becoming smarter and interconnected. Edge-based infrastructures of today's sensorized environments include: IoT devices to sense physical phenomena or receive actuation commands; software components to distribute the processing of raw data and provide semantically enriched data; message brokers to exchange data; Edge servers as hosting machines; and the networking infrastructure. IoT applications operating over such Edge-based infrastructures provide services to improve people's daily activities, life quality, and public safety: for example, IoT applications for building utility optimization, air/noise monitoring, enforcing emergency & rescue procedures. Such applications are composed of different Quality of Service (QoS) requirements such as end-to-end latency bounds, throughput, and tolerated message loss rates. IoT systems designers have to tune the data exchange infrastructure to ensure that the QoS requirements of deployed applications are satisfied. For instance, designers must ensure sufficient network and processing resources for timely data exchange between devices and applications, and at the same time proactively address dynamic situations that may require additional resources. Currently, designing such a distributed IoT system is a manual "by-experience" process that is error-prone and time consuming.

Commercial and open-source simulation tools [1]–[4] have been developed to facilitate such tasks prior to system deployment. They enable simulating some aspects of IoT systems and provide system performance insights in terms of energy-efficiency, battery consumption, etc. Network emulators [5]–[7] can be used to emulate networking events and evaluate the performance of networking infrastructures. These simulators usually provide graphical/command line interfaces, or scripts to create a virtual representation of the IoT system and run simulations. IoT designers have to spend considerable effort in learning how to use a simulator for representing an IoT system. In addition, it is often hard to define the desired output (e.g., performance measurements and energy consumption). Finally, creating variations of an IoT system that represents multiple situations, often requires complex simulation deployments. Hence, there is a need for a simulation tool enabling IoT designers to quickly simulate IoT systems designed based on standard IoT representations and data exchange architectures.

This paper presents EDICT, a simulation tool for evaluating the performance of Edge interactions in distributed IoT-enhanced environments. EDICT abstracts the hardware and network implementation details, as well as the application-layer interactions of an IoT system as a *queueing network* [8], [9] (also called *generic QoS model*). First, EDICT leverages the standard NGSI-LD protocol specification [10] to represent systems deployed in IoT-enhanced environments. Second, Edge interactions are represented based on the IoT-suitable publish/subscribe interaction paradigm [11]. These are given as input to the EDICT generic QoS model to be instantiated for simulating and evaluating the performance of Edge interactions. Multiple QoS model instances can be used to represent different situations (e.g., number of devices/applications), or QoS configuration parameters (e.g., applying priorities and resource allocation policies) can be applied to provide designers with a performance metrics dataset that can be used for design time system tuning. In addition, to support runtime adaptation of IoT systems, EDICT provides designers with a QoS prediction component that rapidly generates performance metrics when changes in the IoT system occur.

The key contributions of this paper are:

- A standard representation of IoT data exchange in smart environments using an extended NGSI-LD model and a publish/subscribe architecture (§IV).
- A simulation tool for generating multiple, customizable datasets containing performance metrics of Edge interactions in distributed IoT-enhanced environments (§V).

- A QoS prediction mechanism for rapid runtime adaptation (§VI).

The rest of the article is organized as follows. In §II, we compare EDICT against existing IoT simulators. An overview of the EDICT architecture is presented in §III. We then present in §IV details related to the representation of IoT-enhanced environments. In §V we present the generic QoS Model used to model the performance of IoT systems. §VI describes how IoT systems designers can use the EDICT prototype for design time system tuning and runtime adaptation. We conclude the paper with a look towards future extensions of EDICT in §VII.

II. Related Work

This section presents an overview of IoT simulators. We present in Table I a comparative summary of such tools.

To allow the evaluation of IoT systems prior to deployment, the research community has developed simulation tools representing IoT system deployments. Tools such as iFogSim [12] and IoTSim-Edge [15] provide simulation environments that evaluate the effects of resource allocation and provisioning techniques for improving the performance of IoT systems in terms of end-to-end latency, energy consumption, and network congestion. IoTSim [13] allows the simulation and performance evaluation of big data analytics, such as MapReduce, in cloud environments. iFogSim [12], IoTSim-Edge [15], and IoTSim [13] are built on top of CloudSim [18], a toolkit that allows modeling and simulating large-scale Cloud computing environments, such as data centers. iFogSim, IoTSim-Edge and IoTSim follow a similar approach for the allocation of cloud related-resources needed for IoT applications. Kaala [16] aims to bridge the gap between simulators and real-world resources by integrating the simulated devices with real-world Cloud services such as AWS and Google IoT cloud services. DPWSim [4] focuses on service-oriented IoT applications and relies on the OASIS DPWS standard to provide developers with a simulator that allows them to prototype their applications prior to deployment. Commercial tools, such as IoTIFY [1] and CupCarbon [2], are able to simulate IoT interactions within a smart city environment.

In addition, there are simulators that focus on simulating events at the network layer. These tools model communication protocols and packet-based interactions. For example, OMNeT++ [7] and QualNet [5] provide environments for simulating network interactions in distributed and large-scale systems, and allow to evaluate the performance of network protocols. NS-3 [6] is a discrete-event network simulator that allows users to evaluate the performance of packet data networks. SenseSim [17], on the other hand, does not focus on wireless communication issues but provides a tool for simulating WSN and IoT systems under changing environments.

While the above tools are powerful enough for simulating IoT systems, we identify the following challenges:

IoT-layer Abstractions. Even though network emulators such as NS-3 [6], OMNeT++ [7], and QualNet [5] can be used to simulate the network aspects of IoT systems, they require configuring the network infrastructure of the IoT system. This is not always convenient, especially when IoT designers need to only simulate application-level events. EDICT abstracts the hardware and network implementation details and helps users focus on application-layer interactions in Edge environments.

Usability. IoT designers often have to learn and get used to the specifics of the simulator they choose to use. Tools like CupCarbon [2] require users to learn a new scripting language and write scripts to program sensor nodes. This requires spending considerable time figuring out how to use the tool to get the desired output. In contrast, EDICT users only need to provide the standard NGSI-LD description of their system, as shown in § III. EDICT then automatically simulates Edge interactions in the provided environment and provides simulation results as a performance metrics dataset.

System Reconfiguration. Simulating the same environment with multiple configuration parameters usually requires creating a new simulation with new instances of IoT devices and applications. This can be time consuming especially when IoT designers need to evaluate the performance of their systems in different situations by applying multiple configuration parameters (to identify which configuration is best-suited for their needs). EDICT provides simulation results per subscription and per application category for multiple configuration parameters of the IoT system in one iteration.

Automating System Tuning. Support for automated system tuning is not straightforward when using the simulation results of existing tools. Such tools provide results visually ([12], [15], [14], [13]) or provide them in files that require further processing ([6], [7]). As shown in §V-B and §VI-C, EDICT provides a CSV-based performance metrics dataset that can easily be integrated as part of an automated approach for system tuning. This is especially useful given the recent advances in ML and AI techniques to enable autonomous IoT systems and to solve IoT-related challenges.

Support for Runtime Adaptation. Existing simulation tools do not provide support for runtime adaptation of IoT systems. At rutime, if changes occur in the Edge infrastructure, IoT designers have to re-run the simulations; this process is time consuming and cannot be applied at runtime. A distinguishing feature of EDICT is that it provides a *QoS prediction* component that can be integrated as part of a runtime adaptation approach for timely readaptation of IoT systems.

III. EDICT Overview

EDICT leverages NGSI-LD models and queueing networks to represent and simulate IoT data exchange in smart environments. We present next a motivating example and an overview of EDICT's architecture.

Motivation. Edge-based infrastructures of today's sensorized spaces (homes, buildings, etc.) include: (i) *IoT devices* to sense physical phenomena or receive actuation commands; (ii) *virtual sensors* that process one or more flows of IoT raw data and provide new, possibly high-level, data (e.g., occupancy of spaces); (iii) *messaging systems* to enable decoupling between IoT devices and applications/services; (iv) *Edge servers* as hosting nodes; and (v) the *networking infrastructure*.

Tool	IoT Layer Abstraction	Scope	Input	Automated System Tuning	Simulation Domain	Support for Readaptation
DPWSim [4]	Fog/Cloud	Deployment of web services on IoT devices	GUI	Through graphs	Generic	No
iFogSim [12]	Fog/Edge	Resource management / application scheduling	GUI / API calls / JSON configuration file	Through graphs	Generic	No
IoTSim [13]	Cloud	Big data processes	API calls	Through graphs	Generic	No
IoTNetSim [14]	Cloud/Edge/Network	Simulation of IoT services	API calls	Through graphs	Generic	No
IoTSim-Edge [15]	Edge	Application composition / mobility modeling / resource provisioning	GUI / JSON Configuration	Through graphs	Healthcare/Buildings/ Transportation/ Manufacturing	No
Kaala [16]	Cloud	Interaction with real-world cloud services	Configuration file	N/A	Generic	No
SenseSim [17]	Network	Simulation of network interactions in WSN	GUI	N/A	Generic	No
IoTIFY [1]	Cloud	Large-scale IoT deployment	GUI / API calls	Through graphs / API	Smart City	No
CupCarbon [2]	Network	Evaluation of distributed algorithms / creation of environmental scenarios	GUI	Through CSV files	Smart City	No
OMNeT++ [7]	Network	Discrete event simulation	NED files / C++ modules sources	Through TXT files	Generic	No
QualNet [5]	Network	Heterogeneous networks / distributed applications	GUI / Configuration files	Through STAT files	Generic	No
NS-3 [6]	Network	Discrete network event simulation	API calls	Through PCAP / ASCII files	Generic	No
EDICT	Edge interactions over pub/sub	Simulation of Edge interactions in IoT environments using Queueing Networks	GUI / NGSI-LD models	Through CSV files	Generic	Yes

TABLE I: Comparison of Existing IoT Simulation Tools

IoT applications operating over such infrastructures provide a variety of services, such as energy monitoring, audio and video conferencing, and emergency response services. These applications have to specify QoS requirements to be met—such as energy consumption, accuracy, and end-to-end latency.

We classify IoT applications into *categories* that each define QoS parameters to be satisfied. In particular, we identify five application categories: (i) *real-time (RT)*, e.g. a robot delivering mails in a university campus; (ii) *streaming (ST)*, e.g. a video surveillance application; (iii) *transactional (TS)*, e.g. a meeting-room reservation tool; (iv) *IoT analytics (AN)*, e.g. an occupancy-based application using WiFi connectivity data; and (v) *emergency response (EM)*, e.g. a fire detection application. Currently, IoT system designers have to manually use different system configurations for different situations to satisfy QoS requirements of the deployed applications. This task becomes more challenging when we consider the evolution of smart environments over time due to adding/removing IoT devices and applications. Thus, evaluating the performance of IoT systems under all possible situations through manual deployment and testing is virtually impossible.

EDICT introduces an IoT simulation tool that enables systems designers to quickly setup and simulate the performance of IoT systems deployed in smart environments under different configurations without the need of real-world deployments. The output of the simulation is a performance metrics dataset that can be used for system tuning or performance predictions by relying on basic ML techniques.

EDICT Architecture. Fig. 1 shows the high-level architecture of EDICT. To represent characteristics of smart environments, we leverage the *Next Generation Service Interfaces-Linked Data* (NGSI-LD) [10] specification. In particular, EDICT uses existing data models for the standard representation of smart environments as NGSI-LD *entities* that are further enhanced with IoT aspects based on our proposed application categories and QoS requirements. More details about the proposed NGSI-LD representation are presented in §IV-A. The *IoT-enhanced NGSI-Model* can be instantiated to represent the interactions of smart environments to be simulated. NGSI-LD instances of smart environments and their interactions are represented using JSON-LD notation, a JSON-based serialization format.

To simulate interactions between IoT-enhanced NGSI-LD entities, we rely on Queueing Networks [9]. In particular,

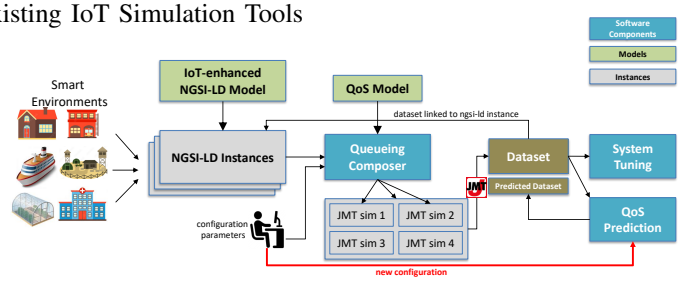

Fig. 1: EDICT architecture

we define a *generic QoS Model* that consists of series of queues to model IoT interactions in smart environments. Such queueing networks can be composed, on-the-fly, depending on the corresponding smart environment instance and the ongoing situation (e.g., applications activated for emergency response)—more details in § V. Finally, the EDICT end-user can configure simulations by applying parameters such as priorities, dropping rates, routing policies, network allocation policies, that can improve the end-to-end performance (i.e., latency and throughput) of IoT interactions. Depending on the given NGSI-LD instance and the configuration parameters, the *Queueing Composer* instantiates the composed code artifacts that, when executed, simulate end-to-end IoT interactions (from the IoT sensor to the IoT application) using the open-source queueing simulator *Java Modeling Tools* (JMT) [19].

We define a *JMT simulation model* as an instance of the *generic QoS Model* that represents a specific situation of a smart environment with specific configuration parameters. Different *JMT simulation models* represent IoT interactions of different situations in smart environments. In addition, a number of additional *JMT simulation models* may be generated due to the applied configuration parameters. Executing the composed *JMT simulation models* results in the creation of a dataset of performance metrics that includes end-to-end latency, throughput, energy consumption, etc. The *QoS prediction* component uses this dataset to train ML models and predict performance metrics without running time-consuming JMT simulations. Such component can be used at runtime, alongside a real IoT system that may require to be adapted to new situations (e.g., emergency response). IoT designers can rely on the generated dataset for IoT system tuning at design time and ensure efficient runtime behavior.

IV. IoT System Representation

To build a tool for simulating IoT interactions at the Edge of any smart environment, it is essential to represent: (i) the environment and its IoT capabilities; and (ii) the interactions between IoT entities by relying on a generic architecture. Related generic models are presented in this section.

A. IoT Domain Model

We use Next Generation Service Interfaces-Linked Data (NGSI-LD) [10] as a flexible and dynamic basis for the modeling of smart environments. In short, NGSI-LD is a data model for the standard representation of smart environments, and it provides an API for publishing, querying, and subscribing to context data. NGSI-LD has the unique characteristic of deriving its information model from property graphs, its meta model on the basis of RDF/RDFS/OWL [20]–[22], and partially on the basis of JSON-LD. Its purpose is to facilitate open exchange and sharing of structured data amongst various parties. While other data models exist to describe smart environments (e.g., BOT [23], SAREF [24]), we use NGSI-LD entities that can be easily imported into triple stores (i.e., RDF) [25] because they are serialized in JSON-LD [26].

NGSI-LD is utilized in Smart Cities, Smart Industry, and Smart Agriculture, as well as the IoT, Cyber-Physical Systems, Systems of Systems, and Digital Twins more generally. However, existing models are focused on describing the functional semantics of IoT systems and not QoS semantics such as requirements of applications and categories. In addition, entities that represent observations are missing, as well as entities for the processing of raw data. We extend the NGSI-LD representation of an IoT system by introducing Entities and Relationships as shown in Fig. 2. First, an existing `Device` entity is extended by adding attributes and relationships with new entities to represent the general function of IoT devices. In addition, we create a new `SmartEnvironment` entity that can either extend existing NGSI-LD models (e.g. Building, Museum) or define a new environment. IoT devices deployed in the smart environment can be either sensors that sense physical characteristics and generate `Observations` or `ActuationCommands`, or actuators that receive `ActuationCommands`. An `Observation` represents a quantitative measurement in a specific physical space. For example, a temperature sensor deployed in `room324` would generate the observation `"temperature in room324"`. Thus, an application can easily receive the necessary data by specifying the type of data needed along with the name/id of the space.

We also create a new Entity type `VirtualSensor`, which represents software components that process raw IoT data to produce high-level measurements. Unlike physical sensors, virtual sensors take as input one or more `Observations`, process the received data, and output a new type of `Observation`. Finally, to represent different categories and QoS semantics of applications (see § III), we classify them into `ApplicationCategories` with each

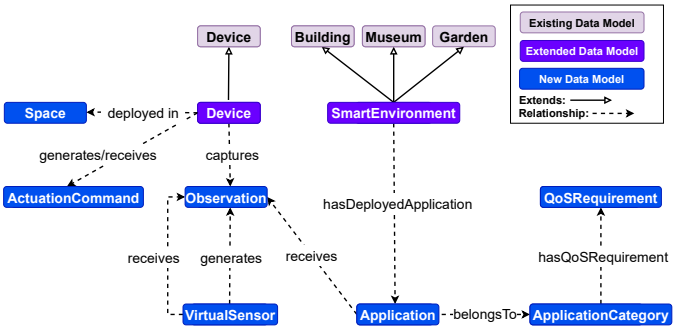

Fig. 2: Representing IoT-enhanced Environments using the NGSI-LD Data Model

category specifying `QoSRequirements` (e.g., end-to-end latency, energy consumption, and drop rates) that have to be met. The NGSI-LD model presented in Fig. 2 can be instantiated to describe an IoT system in any smart environment. We show how to use the NGSI-LD model to define an instance of an IoT system in §VI-A and §VIII. Our extended NGSI-LD data models along with examples of smart environments can be found at https://github.com/SAMSGBLab/edict--datamodels.

B. IoT Data Exchange Model

Existing IoT deployments consist of devices employing network access protocols (e.g., ZigBee and Z-Wave) for data collection in IoT gateways (or IoT agents in NGSI-LD), as well as application-layer protocols (e.g., CoAP, MQTT, ZeroMQ, and AMQP) that forward data from IoT gateways (or directly from devices) to message brokers (e.g., EMQx, Mosquitto, and RabbitMQ) for data processing and dissemination to IoT applications [27]. Data exchanged between IoT devices, applications and message brokers, are based on the publish/subscribe interaction paradigm [11].

EDICT relies on the publish/subscribe paradigm to represent IoT interactions at the Edge of smart environments, where IoT devices and applications interact via a message broker. We develop a generic QoS model (§V-A) that abstracts the underlying network, different application categories, and application instances that interact via the message broker with the IoT devices. The broker manages all the traffic at the Edge and for this reason message handling strategies are applied at the broker. As depicted in Fig. 3, ***IoT devices*** act as publishers that produce data related to the environment sensed; these data are encapsulated into messages that are tagged with a ***topic*** name for routing. Messages of a topic can be captured from both IoT devices and virtual sensors, while topics are characterized from the observation type (e.g., temperature data), the space that a device is deployed (e.g., room 2065), the average message size and the message frequency. Such information can be found in the NGSI-LD instance describing a smart environment. Virtual sensors, actuators, and applications *subscribe* to receive relevant messages using topic-based subscription filters. Virtual sensors receive input from one or more sources of messages, process the received messages (e.g., using AI-based algorithms), and generate output messages that are sent back

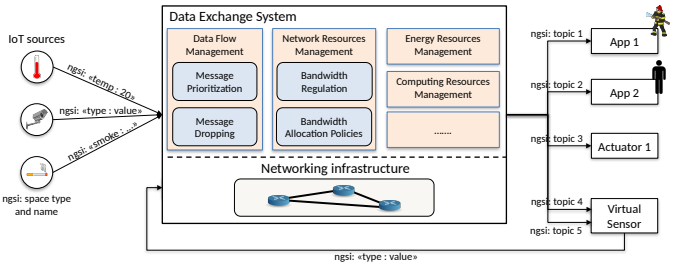

Fig. 3: Data Exchange Model

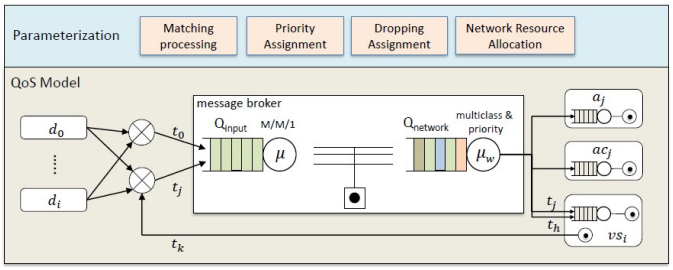

Fig. 4: Queueing Representation of an IoT Data Exchange System

to the message broker [28]. Note that such processed messages can be received by applications or other virtual sensors.

IoT applications can be characterized by heterogeneous QoS requirements and they can be grouped into application categories (RT, ST, TS, AN, EM), as specified in § III. A message broker system is responsible for forwarding data from IoT devices to the corresponding application recipients. IoT systems designers can configure the data exchange infrastructure according to their needs. For instance, they can assign priorities to some topics via a *Data Flow Management* component, or/and specify a network resources allocation policy (e.g., max-min policy [29]) via a *Network Resource Management* component. Thus, IoT systems designers can use different configurations to tune the system and ensure the desired QoS performance at runtime.

V. QoS Modeling for IoT Simulation

This section presents a generic (Quality of Service) model that can be used to model the performance of IoT systems. Given a specific smart environment, a queueing network can be created using the EDICT composer that is presented next.

A. IoT Data Exchange Formal Model

This section presents the formal model of data exchange systems following the model presented in §IV-B. EDICT leverages Open Queueing Networks [9] to dynamically compose queueing networks (i.e., QoS models) that evaluate the performance of diverse data exchange systems. As depicted in Fig 4, each IoT device $d_i \in D = \{d_i : i \in [1..|D|]\}$[1] (e.g., smoke detector) acts as a publisher and publishes data to one or more topics $t_j \in T$. Note that a topic represents a mapping based on an observation type and a space (e.g., "smoke in room312"). IoT devices may also publish actuation commands (e.g., water sprinklers activation commands), which are aimed to be received by actuators. Let T_{d_i} be the set of topics that device d_i publishes to. We represent each topic t_j as a class switch x_{t_j} that receives incoming messages from devices D and tags them with the appropriate topic.

Topics can be characterized by an average message size G_{t_j} and the rate λ_{t_j} at which messages are produced. Note that λ_{t_j} is based on a probability distribution (e.g. Normal or Exponential) or by relying on real-world traces of existing smart environments (more details in § VI). A virtual sensor $vs_i \in VS$ subscribes to receive data from a set of topics, denoted by T_{vs_i}. Let τ_{vs_i} be the average processing time of a virtual sensor. A queue q_{vs_i} with a service rate of $\mu_{vs_i} = \frac{1}{\tau_{vs_i}}$ represents data processing by the virtual sensor. vs_i then publishes the processed high-level data (e.g., occupancy) to the message broker using a topic $t_k \in T$.

At the message broker side, an input queue Q_{in} (of type G/G/1) accepts the publications of devices and virtual sensors. Its service rate μ_{in} represents the subscription matching process that is applied to identify data recipients based on the available subscriptions. Let $K_{Q_{in}}$ be the queue's maximum capacity that represents the data exchange system's overall capacity. To model the networking infrastructure between the message broker and the system's subscribers (i.e., applications/actuators/virtual sensors), we use a multiclass queue Q_{net}. The service rate μ_{net} of Q_{net} represents the available network resources W_{DX} between the data exchange system and all subscribers. Then, μ_{net} is calculated based on W_{DX} and the average message size G_{t_j} of the corresponding topic. Note that Q_{net} can act as a priority queue, and prioritize messages matching topics based on assigned priorities by the EDICT user. Also, EDICT offers the ability to choose a network allocation policy π.

The default policy π_{all} uses all available network resources to transmit all messages (typical allocation policy used at the NIC of an Edge server). On the other hand, middleware-based approaches [29], [30] offer to application designers different policies for allocating resources. The π_{shared} policy can be used to equally share resources between all application categories. The π_{topics} policy allocates resources to each category based on the arrival rate of topics: application categories that receive more data are allocated more resources. Finally, the $\pi_{max\text{-}min}$ policy allocates resources that guarantee fairness among all subscribers. Therefore, IoT designers can choose a policy π from $\Pi = \{\pi_{all}, \pi_{shared}, \pi_{topics}, \pi_{max\text{-}min}\}$. Because some IoT applications can be loss-tolerant and several middleware-based approaches support message dropping (e.g., JMS, MQTT, CoAP, AMQP [11]), we model message losses by routing a percentage of messages ω_{net} matching a topic to a sink node. Currently, EDICT models message dropping only at the middleware layer and assumes no message losses due to the network coverage. However, the QoS model can be extended to represent message losses due to network coverage (e.g., modeling coverage impacts on wireless protocols).

[1]The subscript of a notation represents the index of an element in the corresponding set. For the remainder of this paper, we omit the explanation of subscripts to simplify our approach.

At the application side, let $app_i \in A$ be the application table with attributes $\{cat, qos, topics\}$, where $cat \in \{AN, RT, ST, TS, EM\}$ represents the category of the application, qos is a tuple representing the QoS requirements of the applications, and $topics$ is the set of topics T_{app_i} that app_i has subscribed to. Each app_i is modelled as a queue q_{app_i} that receives messages from Q_{net} according to the application's subscriptions. A subscription is defined as the tuple $r_{ij} = (app_i, t_j) \in R$. We consider that qos = (δ_{max}, θ_{min}, ω_{max}), where δ_{max} represents the application's maximum tolerated end-to-end latency, θ_{min} represents the minimum required throughput, and ω_{max} represents the application's tolerated drop rate. Note that $app_i.qos$ can be extended to include more QoS requirements. Similarly to applications, each actuator $ac_i \in AC$ is modelled as a queue q_{ac_i} and subscribes to one or more topics $t_{ac_i} \in T$.

B. IoT-aware Queueing Composition

The generic model defined in §V-A provides all the basic elements to compose a queueing network, as well as the necessary configuration parameters used by application designers (network policies, priorities, etc.) to parameterize the composed queueing network. For each situation $s \in S$ and configuration parameters $p \in P$, the function $f(s, p)$ instantiates the QoS model to compose the queueing network representing the environment under situation s and configuration parameters p. Depending on the IoT environment and deployed devices/applications, EDICT provides the basic queueing network, defined as a ***default network***, which models the default performance of the IoT data exchange system. Then, EDICT offers different configuration parameters (priorities, dropping, network policies) to model the system's performance.

To enable such dynamic queueing network composition, we leverage the JMT (Java Modelling Tools) simulator [19], [31]. JMT is an open-source suite of applications that offer a comprehensive framework for system modelling with analytical and simulation techniques, and performance evaluation. While JMT JSIMgraph provides a graphical user interface to design queueing models, we use JMT's API to dynamically compose and run the simulations. As depicted in Fig. 5, we design a *Queueing Composer* that instantiates the corresponding JMT queueing representation of an IoT environment's situation and parameterization. The composer takes as input (i) the NGSI-LD instance of an IoT-enhanced environment (see §IV-A), and (ii) the generic QoS model (see §V-A). Through calls to the JMT Library, the composer then creates the queueing network that represents the IoT-enhanced environment.

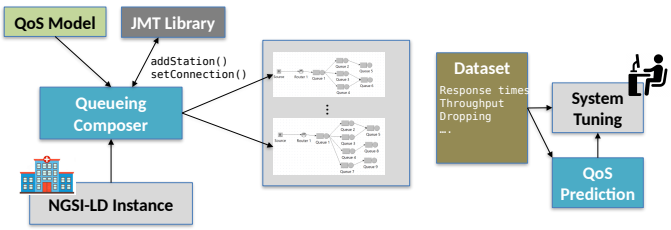

Fig. 5: Queueing Network Composition Process

We implement a Queueing Network Composer to generate the JMT queueing network that represents a specific instance of an IoT-enhanced environment. The *Queueing Composer* parses the JSON-LD files that represent the smart environment to retrieve the following IoT information: IoT devices, virtual sensors, actuators, and applications deployed; application categories and their QoS requirements; and data exchange configuration parameters (e.g., available network resources). Then, using the generic QoS model presented in § V-A, it composes the queueing network using the JMT library. The library contains functions that enable creating a JMT-based queueing model, such as adding queueing stations, creating connections between stations, and setting service times for queues. For example, the code of Listing 1 is used to create and connect the input and network queues of the QoS model.

```
1  public static void composeQueueingNetwork() {
2    CommonModel jmtModel = new CommonModel();
3    inQueue = model.addStation("inQueue",CommonConstants.STATION_TYPE_SERVER);
4    netQueue = model.addStation("netQueue",CommonConstants.STATION_TYPE_SERVER);
5    model.setStationQueueStrategy(netQueue,model.
           STATION_QUEUE_STRATEGY_NON_PREEMPTIVE_PRIORITY);
6    for (Application app : applications)
7      model.setConnected(netQueue, app.getQueue(), true);
8    for (Topic topic : topics) {
9      model.setRoutingStrategy(netQueue,topic.getClass(),probRouting);
10     model.setServiceTimeDistribution(netQueue,topic.getClass()),netResources);}}
```

Listing 1: Queueing Composer code snippet

The EDICT composer first instantiates a *JMT common model* (see Line 2) that represents an instance of a queueing network. This holds all the elements of the queueing model, runs the simulation, and saves its results. Then, the `addStation` function adds sources, queues, forks and joins, and sinks: e.g. the input and network queues at Lines 3–4. A queue's discipline can be updated using the `setStationQueueStrategy`: e.g. the network queue is defined as a priority queue at Line 5. After adding all the components, connections between stations are created by calling the `setConnection`: e.g. connections between the network queue and application queues at Lines 6–7. Data flowing from IoT sources to the data recipients are configured by setting the appropriate routing strategies using `setRoutingStrategy`: e.g. data routed based on the topic class at Lines 8–9. Finally, the `setServiceTimeDistribution` is used to set the service time: e.g. based on network resources between the network queue and applications at Line 10.

Once a *default* queueing network is created, the EDICT user has the option to specify the performance metrics to be saved, such as the end-to-end latency and throughput for flows matching subscriptions, the utilization of the system, and the data loss. EDICT also offers the option to configure the system by setting priorities and drop rates, and by specifying the available network resources and the resource allocation policy. Having different configurations of the data exchange system helps in creating a richer dataset that includes performance metrics for each of the configurations of the system.

Performance Metrics Dataset Generation. The JMT composed models are provided as XML-based files that contain the topology of the queueing network and the simulation parameters. Using the JMT library, EDICT simulates the

queueing network and appends the results to the same XML files. The EDICT *dataset generator* parses the XML file and creates a dataset of performance metrics as a set of `csv` files, which are easy to read and widely used. Metrics are saved for each flow matching a subscription, and when applicable, for the whole system. For example, EDICT stores end-to-end latency per flow and the utilization of the data exchange system under all configurations specified. The format of the dataset is shown in Table II. Each row contains the metrics for a flow matching a subscription under different configurations parameters. For example, if the developer chooses to simulate an IoT system using the default network allocation policy as well as the max-min policy, a subscription flow r_{ij} would have two entries in the dataset that represent its metrics under these two policies. Note that EDICT datasets are generated to contain features that impact the performance of IoT systems, based on the QoS model presented in §V-A. These features represent the configuration parameters that are applied to the data exchange system , i.e., priorities, drop rates, network allocation policy, and available network resources. Generating the dataset in this format makes applying ML models easier, as it is easy to extract the needed features for prediction. In addition, this approach can be extended to include features for other QoS models (e.g., energy consumption) to support potential extensions of EDICT (§V-C).

The dataset generated by EDICT can be leveraged by automated approaches for system tuning. For example, the PlanIoT [32] framework relies on EDICT-related performance metrics datasets to automatically manage IoT data flows using AI planning. EDICT datasets can also be used for training ML models for performance prediction purposes. This is especially useful when dynamic changes are happening in a smart environment (e.g., subscriber churn).

C. EDICT's Extensibility

A unique characteristic of EDICT is the generic QoS model and different application classes that can be instantiated for different IoT systems. EDICT 's extensibility is enhanced due to the use of queueing networks. Such fundamental QoS modeling methodologies provide accurate performance metrics without deploying and benchmarking the actual software and hardware entities of IoT systems. The generic queueing model of EDICT can be extended using different types of queues to represent the parameters that impact the performance of IoT systems. Currently, EDICT supports message prioritization, assigning drop rates to data flows, and setting up buffer capacities for the message broker. However, the generic queueing model can be easily extended to model wireless communication channel error rates [33], hardware infrastructure and computing resources. This section describes some of the extensions that can be applied to EDICT.

Modeling the network infrastructure. To avoid having complex queueing networks modeling all layers of IoT systems, EDICT provides metrics related to the performance of IoT systems at the middleware layer. However, hierarchical queueing models [34] can be used to model the performance of IoT systems at different layers (middleware, network, etc.). Using such models, we can represent message losses due to network coverage by considering that applications connect to a wireless access protocol. Similarly, such a specific network, along with message losses, maximum bit rate, and other network parameters can be modeled using a queueing network dedicated for modeling the network infrastructure.

Device properties. Extensions can also be provided to model hardware and computing resources of devices. For instance, queues with an intermittent available server [11] can be used to represent the IoT devices' sleep patterns. Computing resources of devices can be taken into account by appropriately setting the service rate of queues that model devices.

Performance metrics. EDICT can be extended to provide a wider range of metrics that give IoT developers insights into the performance of their system. For example, the energy consumption of data flows can be measure using Energy Packet Networks [35].

VI. EDICT Evaluation

This section presents the evaluation of EDICT by (i) demonstrating how the EDICT prototype can be used to represent IoT infrastructure components and generate a performance metrics dataset (§VI-A); (ii) creating an experimental setup of an IoT-enhanced building (§VI-B); (iii) showing how EDICT enables design-time system tuning (§VI-C); and (iv) showing how EDICT's QoS prediction component enables rapid adaptation when changes happen to the Edge infrastructure (§VI-D). The EDICT source code can be found on https://github.com/SAMSGBLab/edict.

A. Using the EDICT Prototype

As a first step, IoT designers need to define the components of their IoT system infrastructure. For this purpose, EDICT provides a Graphical User Interface[2] to add, edit, and delete such components. The devices interface (Fig. 8) allows designers to add new devices, and edit and delete existing ones. For each device, designers have the option to specify the size of messages and the probability distribution based on which messages are generated by the device, the frequency at which messages are generated, and the observations that the device captures. Next, the applications interface (Fig. 9) window allows adding new applications and editing their properties (the application category they belong to, the observations they receive, and the rate at which they process messages). This is where designers also have the option to define specific priorities for applications. In a similar fashion, designers can define properties of application categories, QoS requirements, and observations in dedicated interfaces.

Once all IoT system components are defined, EDICT generates the JSON-LD files[3] corresponding to these components according to the NGSI-LD model presented in IV-A. Listing 2 shows how an IoT device is defined in JSON-LD format.

[2]Screenshots showing EDICT's GUI are available in §VIII-A

[3]The JSON-LD notations for defining all the IoT system components are available in §VIII-B

subscriptionFlow	category	priorityAN	priorityRT	priorityTS	priorityVS	droppingAN	droppingRT	droppingTS	droppingVS	networkPolicy	networkResources	ressponseTime	throughput	...
amazonecho/app14	TS	0	0	0	0	0	0	0	0	default	650	1.694455	...	...
intrusion/app12	RT	0	0	0	0	5	0	0	2	default	650	1.232349	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

TABLE II: Output Dataset Format

Each device has a `name` and is identified by a unique URN (`id`). A device sends messages with a `messageSize` at a specific `publishFrequency`. In addition, we consider that devices generate messages based on a probability distribution (`dataDistribution`). A device captures one or more observation; this is represented in the `capturesObservation` *relationship*.

```
1 "id": "urn:ngsi-ld:edict:Device:dcbd9ee7-...",
2 "type": "Device",
3 "name": "device 1",
4 "publishFrequency": 150,
5 "messageSize": 2000,
6 "dataDistribution": "exponential",
7 "capturesObservation": ["urn:ngsi-ld:edict:Observation:85
      c38830-...", ... ],
8 ...
```

Listing 2: IoT device JSON-LD definition

Similarly, the definition of applications deployed in the smart space is shown in Listing 3. Each application belongs to an `applicationCategory` and is assigned a `priority`. Note that a lower integer represents a higher priority. Applications receive one or more observation, which are defined as a list in the `receivesObservation` field. In addition, applications process the received messages at a specific rate (`processingRate`), and following an exponential probability distribution (`processingDistribution`). Each application deployed belongs to a specific category (Listing 4), and each category defines QoS requirements in terms of maximum response time (in seconds), minimum throughput (in Kbps), and maximum drop rate (Listing 5). Note that IoT designers that already have a representation of their IoT system in the JSON-LD notation may readily upload their files to EDICT.

After defining the components of their IoT system infrastructure, designers can now specify the configuration parameters of their systems, as shown in Fig. 6. Designers can specify the available network resources (i.e., the available bandwidth between the data exchange system and the applications), and the network resource allocation policy to be used. Currently, EDICT supports three network allocation policies: (i) the *default* policy, where all network resources are used to forward all data flows, (ii) the *shared* policy, where network resources are equally shared between the application categories defined, and (iii) the *max-min* policy, which shares network resources among categories based on the max-min resource allocation policy. Moreover, the configuration parameters include setting drop rates for application categories, and defining the capacity (in number of messages) of the data exchange system. Designers can then define some simulation settings: the simulation duration, an alias to be used for saving the simulations results, and a global message size to be used when running the simulation.

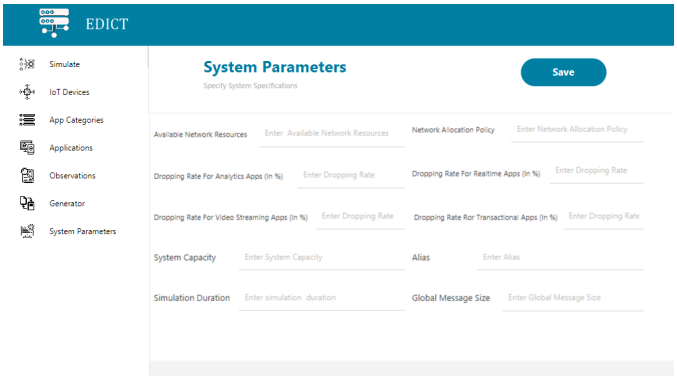

Fig. 6: EDICT Parameters

IoT-enhanced Environment Properties				*QoS Requirements*		
App. categories	**|A|**	**|R|**	W_{DX}	δ_{max}	θ_{min}	ω_{max}
AN	6	21	650 MB /s	best effort	best effort	best effort
RT	9	17		<400 ms	384 KB / s	0%
TS	6	12		<4 s	-	0%
ST	9	10		<2 s	384 KB / s	<2%
Total	30	60	650 MB/s			

TABLE III: Experimental Setup

B. Experimental Setup

We demonstrate the utility of EDICT by simulating IoT interactions in an environment that has the properties presented in Table III. We rely on the works presented in [36], [37] to set the values for the message sizes and publishing frequencies for the devices. We consider a smart office building equipped with 726 sensors/IoT devices that generate 9 types of observations and publish them to an Edge broker. 30 applications belonging to 4 application categories (AN, RT, TS, ST) subscribe to receive data from one or more observation. The total number of subscriptions is 60. We use the ETSI TS 1212 105 V15.0.0 standard [38] to compose the QoS requirements for the four application categories. However, IoT designers can use other existing standards or define their own requirements.

Using the setup displayed in Table III and the NGSI-LD model presented in §IV-A, we create the corresponding NGSI-LD instance of the smart building. The complete JSON-LD files for this setup can be found at https://github.com/SAMSGBLab/edict--datamodels.

C. Design-time System Tuning

We feed the JSON-LD files to EDICT that composes the queueing models and simulates them using JMT. EDICT then generates the performance metrics dataset that captures the Edge interactions in the smart building described above. Note that the realism of the generated dataset has been validated in our previous work [33] using a real IoT system implementation. The JMT composed network and the dataset can be found on https://github.com/SAMSGBLab/edict. As mentioned in §V-B, the generated dataset contains metrics for the performance of the IoT system under different configuration parameters.

The simulation results show an 82% utilization of the system. Fig. 7 shows the average end-to-end latency for each application category for some of the configurations: (i) the default configuration, where all network resources are used to transmit all data without setting any priorities or drop rates, (ii) using the max-min network allocation policy [29], (iii) prioritizing time-sensitive applications based on the QoS requirements (RT applications), and (iv) setting a dropping rate of 2% for loss-tolerant applications based on the QoS requirements (AN and ST applications). Running simulations with multiple configuration parameters allows IoT designers to know which configuration is best suited for their needs. For instance, based on the results of Fig. 7, IoT designers can consider that they have to prioritize flows belonging to RT applications (prioritize RT) to satisfy all applications' QoS requirements. EDICT saves IoT designers the tedious effort needed to manually test different configurations after deployment. The dataset generated by EDICT can be integrated as a part of an automated approach for system tuning.

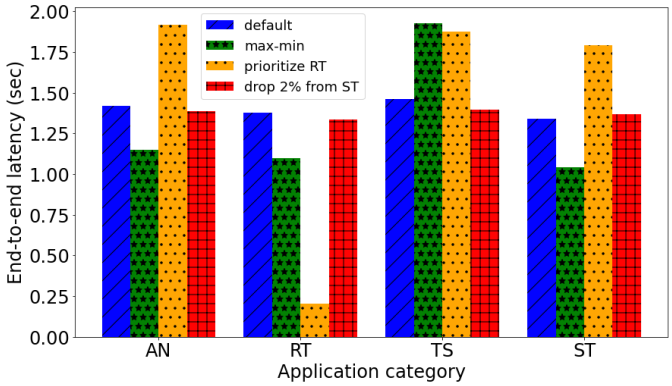

Fig. 7: End-to-End Latency per Application Category

D. Enabling Rapid Runtime Adaptation

To enable runtime adaptation in smart environments, IoT designers need to know the performance of their systems on-the-fly when changes in the Edge infrastructure occur. The time needed to run simulations is too long to allow for a rapid performance analysis of the system at runtime. For example, EDICT simulations can take up to 5 minutes to converge; this time is convenient for running design time simulations, but is too long to perform runtime adaptation. EDICT supports runtime readaptation through an ML-based QoS prediction component that quickly provides insights into the performance of the IoT system when subscriptions or configuration parameters change.

We validate EDICT's QoS prediction mechanism in two ways: (i) by considering a changing number of subscriptions, and (ii) by considering changing configuration parameters. In the remainder of this section, we consider the following configurations: prioritizing one of the four application category, prioritizing applications based on their QoS requirements (giving the highest priority to RT applications, then to ST applications, then to TS applications, then to AN applications), and applying a drop rate of 2% for loss-tolerant applications (AN and ST). In the case of a congested system, we apply more aggressive drop rates (5% and 10%).

	New subscriptions				*New configurations*			
Dataset size	**KNN**	**LR**	**DT**	**DW**	**KNN**	**LR**	**DT**	**DW**
220	0.035	0.036	0.031	0.035	0.0203	0.021	0.022	0.024
440	0.069	0.191	0.072	0.205	0.235	0.195	0.233	0.162
660	0.101	0.648	0.110	0.706	0.090	0.27	0.066	0.118
880	0.199	2.882	0.193	3.165	0.247	3.973	0.062	1.424
1100	0.195	3.487	0.204	3.875	0.346	7.975	0.202	3.027

TABLE IV: Comparison of RMSE (sec) of QoS prediction

For the sake of comparison, we test four QoS prediction models that EDICT uses to support rapid runtime adaptation: (i) the KNN algorithm, (ii) the Linear Regression algorithm, (iii) Decision Trees, and (iv) AWS's DataWig library [39]. DataWig is a deep learning-based framework to impute missing values in datasets. To predict the metrics values for new subscriptions or configurations, we consider these values as unknown, and use DataWig to impute them. We compare the RMSE and the prediction time needed for the four aforementioned models.

We start first by evaluating EDICT's predictions when new subscriptions are added. We consider different dataset sizes that have an increasing number of subscriptions. For each dataset, the number of samples is equal to the number of subscriptions times the number of configurations. For example, when simulating a system with 20 subscriptions under 11 configurations, the number of samples is 220. We test EDICT's prediction mechanism on 5 datasets. For each iteration, we train the models on the dataset generated by EDICT, and then use the models to predict the end-to-end latency for new subscriptions that are added to the IoT system. We validate the predicted values by using EDICT to simulate the IoT system with the new subscriptions. Table IV shows the RMSE for the ML models when predicting metrics for new subscriptions. We notice that KNN and Decision Trees have the best performance even when the size of the dataset increases: the maximum RMSE for KNN is 0.195 s and that of Decision Trees is 0.204 s. In contrast, the RMSE of the Linear Regression model and DataWig increases significantly as the dataset size increases. This is caused by the non-linear nature of queueing delays that cannot be captured by these models. In terms of prediction time, Decision Trees achieve the best performance they can predict metrics for new subscriptions in less than $4ms$. The linear regression model also has a low prediction time of $20ms$—albeit with a much higher error. Even though KNN's prediction time is significantly lower than DataWig's for small datasets, both models tend to have a similar prediction time of $2s$ when the dataset size increases.

Next, we test how well the four models perform when predicting the performance of the IoT system under new configuration parameters. Similarly to the approach above, we test EDICT's predictions on 5 datasets. However, instead of using the models to predict the metrics values when we add new subscriptions, we test how well the models can predict the metrics values for existing subscriptions under different configuration parameters. For instance, we train the models

on a dataset generated by EDICT that contains prioritizing AN, RT, and ST applications, then we test the models by giving them as an input a new configuration that prioritizes TS applications. The models then have to predict the metrics for the subscriptions under the given configuration. This task is challenging since the models have to predict values for configurations not seen during training. Again, as Table IV shows, KNN and Decision Trees perform better than DataWig and the Linear Regression model, with a maximum RMSE of 0.34 s and 0.233 s, respectively. We notice that the execution time for predicting metrics under new configurations is higher than that for predicting new subscriptions for all models. Decision Trees can predict metrics in $18ms$ and a Linear Regression model takes about $10ms$. As expected, KNN and DataWig have a much higher prediction time above $3s$. This is due to the lazy nature of KNN and the fact that DataWig relies on neural networks, which are complex and have a higher prediction time than other models.

VII. Conclusion and Future Work

This paper presents EDICT, a simulation tool for evaluating the performance of Edge interactions in smart environments. EDICT leverages the NGSI-LD information model to represent data exchange in smart environments. We also present a generic QoS model that can be instantiated to create queueing networks that represent the smart environments instances. These queueing networks are simulated to provide a metrics dataset that evaluates the performance of data exchange for multiple situations and configuration parameters of an IoT-enhanced environment. The output dataset can be integrated into an automated system tuning approach by IoT systems designers. To support runtime adaptation, EDICT provides a QoS prediction mechanism that allows designers to get the performance of their systems on-the-fly when changes in subscriptions or configurations occur. Our future work includes adding more performance metrics in the dataset such as the energy consumption of the devices and the needed accuracy of the data. We shall also extend EDICT to support more input types (e.g., RDF), and investigate more QoS prediction techniques to allow faster and more accurate predictions.

References

[1] http://iotify.io/, .

[2] K. Mehdi, M. Lounis, and A. Bounceur et al., "Cupcarbon: A multi-agent and discrete event wireless sensor network design and simulation tool," in *Proc. of the 7th Intl. Conf. on Simulation Tools and Techniques*, Lisbon, Portugal, Mar. 2014, pp. 126–131.

[3] S. K. Moorthy, A. Harindranath, and McManus et al., "A middleware for digital twin-enabled flying network simulations using ubsim and ub-anc," in *DCOSS*, 2022, pp. 322–327.

[4] S. Han, G. Lee, and N. Crespi et al., "DPWSim: A simulation toolkit for IoT applications using devices profile for web services," in *Proc. of the IEEE WFIoT*, 2014, pp. 544–547.

[5] S. Dinesh and G. Sonal, "Qualnet simulator," *Proc. of the Intl. Journal of Information & Computation Technology*, pp. 1349–1354, 2014.

[6] G. Riley and T. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.

[7] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Proc. of the 1st Intl. Conf. on Simulation Tools and Techniques for Communications, Networks and Systems*, 2010.

[8] D. Gross, J. Shortle, and J. Thompson et al., *Fundamentals of queueing theory, 4th Ed.* John Wiley & Sons, 2008.

[9] E. Lazowska, J. Zahorjan, and S. Graham et al., *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[10] "Context Information Management (CIM) NGSI-LD API V1.4.2," https://www.etsi.org/deliver/etsi_gs/CIM/001_099/009/01.04.02_60/gs_cim009v010402p.pdf, 04 2021.

[11] G. Bouloukakis, N. Georgantas, and A. Kattepur et al., "Timeliness Evaluation of Intermittent Mobile Connectivity over Pub/Sub Systems," in *Proc. of the 8th ACM/SPEC ICPE*, L'Aquila, Italy, 2017, pp. 275–286.

[12] H. Gupta, A. Vahid Dastjerdi, and S. Ghosh et al., "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*, vol. 47, no. 9, pp. 1275–1296, 2017.

[13] X. Zeng, S. Garg, and P. Strazdins et al., "IOTSim: A simulator for analysing IoT applications," *Elsevier Journal of Systems Architecture*, vol. 72, pp. 93–107, 2017.

[14] M. Salama, Y. Elkhatib, and G. Blair, "IoTNetSim: A modelling and simulation platform for end-to-end IoT services and networking," in *Proc. of the 12th IEEE/ACM UCC*, 2019, pp. 251–261.

[15] D. Jha, K. Alwasel, and Alshoshan et al., "IoTSim-Edge: a simulation framework for modeling the behavior of Internet of Things and edge computing environments," *Software: Practice and Experience*, vol. 50, no. 6, pp. 844–867, 2020.

[16] U. K. Dayalan, R. A. K. Fezeu, and T. J. Salo et al., "Kaala: Scalable, end-to-end, iot system simulator," in *Proceedings of the ACM SIGCOMM Workshop on Networked Sensing Systems for a Sustainable Society*, ser. NET4us '22. New York, NY, USA: ACM, 2022, p. 33–38.

[17] M. Dyk, A. Najgebauer, and D. Pierzchala, "Sensesim: An agent-based and discrete event simulator for wireless sensor networks and the internet of things," in *Proc. of the 2nd IEEE WFIoT*, 2015, pp. 345–350.

[18] R. Calheiros, R. Ranjan, and A. Beloglazov et al., "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and experience*, vol. 41, no. 1, pp. 23–50, 2011.

[19] M. Bertoli, G. Casale, and G. Serazzi, "An overview of the jmt queueing network simulator," Politecnico di Milano-DEI, Tech. Rep. TR, Tech. Rep., 2007.

[20] https://www.w3.org/RDF/, .

[21] https://www.w3.org/TR/rdf-schema/, .

[22] https://www.w3.org/OWL/, .

[23] M. H. Rasmussen, M. Lefrançois, and G. F. Schneider et al., "Bot: the building topology ontology of the w3c linked building data group," *Semantic Web*, vol. 12, no. 1, pp. 143–161, 2021.

[24] S. Appliances, "Smartm2m; smart appliances; reference ontology and onem2m mapping," *RTS/SmartM2M-103264v2, Rev*, vol. 2, no. 1, 2017.

[25] A. Abid, J. Lee, and F. Le Gall et al., "Toward Mapping an NGSI-LD Context Model on RDF Graph Approaches: A Comparison Study," *Sensors*, vol. 22, no. 13, 2022.

[26] https://json-ld.org/, .

[27] E. Al-Masri, K. Kalyanam, and J. Batts et al., "Investigating Messaging Protocols for the Internet of Things (IoT)," *IEEE Access*, vol. 8, pp. 94 880–94 911, Apr. 2020.

[28] D. M. Polanski and C. M. Angelopoulos, "Modelling virtual sensors for indoor environments with machine learning," in *DCOSS*, 2022, pp. 222–228.

[29] D. Pan and Y. Yang, "Max-Min Fair Bandwidth Allocation Algorithms for Packet Switches," in *Proc. of the IEEE IPDPS*, 2007, pp. 1–10.

[30] G. Bouloukakis, K. Benson, and L. Scalzotto et al., "PrioDeX: a Data Exchange Middleware for Efficient Event Prioritization in SDN-based IoT systems," *ACM TIOT*, 2021.

[31] M. Bertoli, G. Casale, and G. Serazzi, "JMT: performance engineering tools for system modeling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 10–15, 2009.

[32] H. Hajj Hassan, G. Bouloukakis, A. Kattepur, D. Conan, and D. Belaïd, "PlanIoT: A Framework for Adaptive Data Flow Management in IoT-enhanced Spaces," in *IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Melbourne, Australia, 2023.

[33] K. E. Benson, G. Bouloukakis, C. Grant, V. Issarny, S. Mehrotra, I. Moscholios, and N. Venkatasubramanian, "Firedex: a prioritized iot data exchange middleware for emergency response," in *Proceedings of the 19th International Middleware Conference*, 2018, pp. 279–292.

[34] P. Buchholz, "A class of hierarchical queueing networks and their analysis," *Queueing Systems*, vol. 15, pp. 59–80, 1994.
[35] Y. A. El Mahjoub, J.-M. Fourneau, and H. Castel-Taleb, "Energy packet networks with general service time distribution," in *MASCOTS'20*. IEEE, 2020, pp. 1–8.
[36] R. Kumar, M. Swarnkar, and G. Singal et al, "IoT Network Traffic Classification Using Machine Learning Algorithms: An Experimental Analysis," *IEEE Internet of Things Journal*, vol. 9, no. 2, pp. 989–1008, 2022.
[37] A. Sivanathan, D. Sherratt, and H. Gharakheili et al., "Characterizing and classifying IoT traffic in smart cities and campuses," in *INFOCOM Workshops*, 2017, pp. 559–564.
[38] ETSI, "Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); LTE; Services and service capabilities," 3GPP TS 22.105 V15.0.0, Jul. 2018.
[39] F. Biessmann, T. Rukat, and P. Schmidt et al., "DataWig: Missing Value Imputation for Tables." *J. Mach. Learn. Res.*, vol. 20, no. 175, pp. 1–6, 2019.

VIII. Appendix

A. EDICT's Graphical User Interface (GUI)

IoT systems designers can add, edit, and delete devices using the interface shown in Fig. 8.

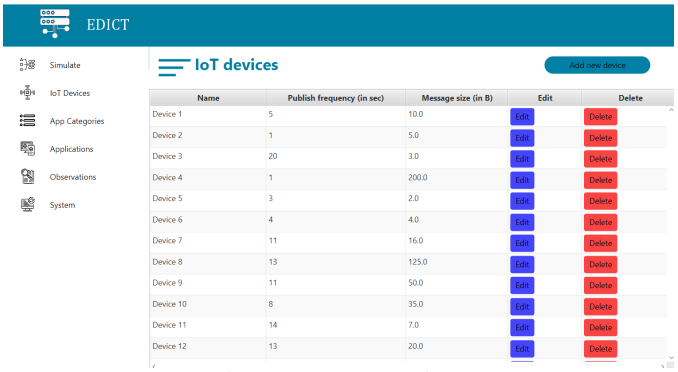

Fig. 8: EDICT Devices

Similarly, they can manage the properties of the applications deployed in the smart environment using the interface shown in Fig. 9.

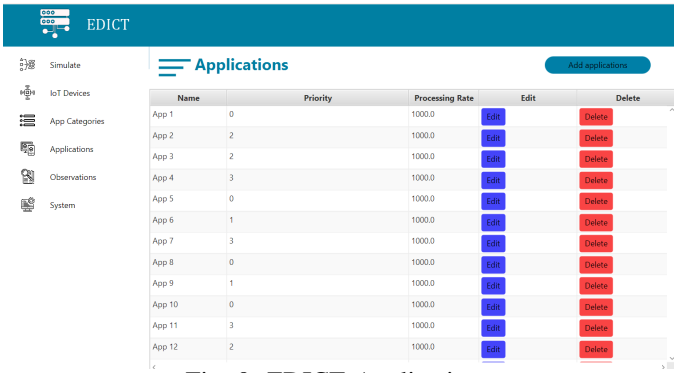

Fig. 9: EDICT Applications

Fig. 10 shows an example of the queueing network composed using JMT.

B. Defining the IoT System Components Using JSON-LD

Listing 3 shows how an application is defined using JSON-LD notation.

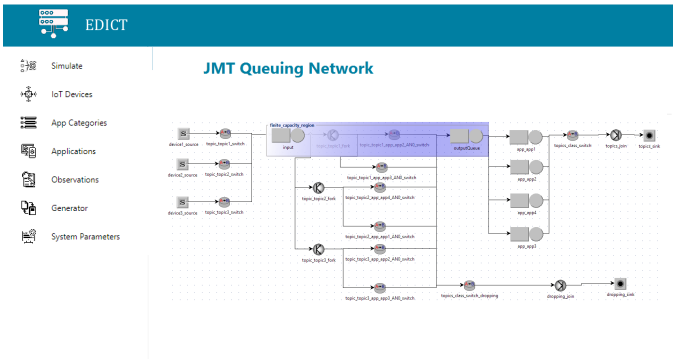

Fig. 10: JMT Composed Queueing Network

```
1 "id": "urn:ngsi-ld:edict:Application:a3b6dc85-...",
2 "type": "Application",
3 "name": "app 1",
4 "applicationCategory": "AN",
5 "priority": 0,
6 "processingRate": 1000,
7 "processingDistribution": "exponential",
8 "receivesObservation": ["urn:ngsi-ld:edict:Observation
      :01437e79-...", "urn:ngsi-ld:edict:Observation:58
      f0581d-...", ... ],
9 "@context": ["https://raw.githubusercontent.com/SAMSGBLab
      /edict--datamodels/main/context.jsonld"]
```

Listing 3: Application JSON-LD definition

Listing 4 shows how to define an application category using JSON-LD notation.

```
1 "id": "urn:ngsi-ld:edict:ApplicationCategory:b340208c
      -...",
2 "type": "ApplicationCategory",
3 "name": "analytics",
4 "@context": ["https://raw.githubusercontent.com/SAMSGBLab
      /edict--datamodels/main/context.jsonld"]
```

Listing 4: Application category JSON-LD efinition

Listing 5 shows how to define the QoS requirements of application categories using JSON-LD notation.

```
1 "id": "urn:ngsi-ld:edict:QosRequirement:9ffd2f4b-...",
2 "type": "QosRequirement",
3 "name": "realtime requirements",
4 "maxResponseTime": 0.4,
5 "minThroughput": 28.2,
6 "maxDropRate": 0.02,
7 "@context": ["https://raw.githubusercontent.com/SAMSGBLab
      /edict--datamodels/main/context.jsonld"]
```

Listing 5: QoS requirements JSON-LD definition