YAFS: A Simulator for IoT Scenarios in fog Computing

ISAAC LERA , CARLOS GUERRERO , AND CARLOS JUIZ , (Senior Member, IEEE)

Department of Mathematics and Computer Science, University of the Balearic Islands, 07122 Palma, Spain

Corresponding author: Isaac Lera (isaac.lera@uib.es)

ABSTRACT Fog computing is a paradigm that extends the cloud to intermediate network devices with computational and storage capacities. This allows the execution of applications closer to edge devices and end-users by allocating services in those intermediate devices. The placement of those services has an influence on the performance of the fog architecture. We propose a fog computing simulator for analyzing the design and deployment of applications through customized and dynamical strategies. We model the relationships among deployed applications, network connections, and infrastructure characteristics through complex network theory, enabling the integration of topological measures in dynamic and customizable strategies, such as the placement of application modules, workload location, and path routing and scheduling of services. We present a comparative analysis of the efficiency and the convergence of results of our simulator with the most referenced one, iFogSim. To highlight the YAFS functionalities, we model three scenarios that, to the best of our knowledge, cannot be implemented with current fog simulators: dynamic allocation of new application modules, dynamic failures of network nodes, and user mobility along with the topology.

INDEX TERMS Complex networks, fog computing, Internet of Things, simulator.

I. INTRODUCTION

Cisco coined the term "fog computing" as an extension of cloud computing, placing computer services closer to the users [1]–[4]. Approximately speaking, some network devices, called fog nodes, perform computational tasks or data storage functions in the same way as cloud entities. This novel application placement has some advantages, such as the reduction of latency time, a lower network bandwidth utilization, a reduction in the cloud costs, and an increase in the reliability and fault tolerance through the geographical distribution of devices.

Another related concept similar to fog computing is edge computing [5]–[7]. The small difference between them lies in the localization of fog nodes. In edge computing, the nodes are at the edge of the network, near the users. In the context of the Internet of Things (IoT), the role of fog computing is to leverage functionalities such as on-demand scalability, real-time interaction, better security and privacy management,

The associate editor coordinating the review of this manuscript and approving it for publication was Weiguo Xia.

battery power savings, streamlining of communications, and rapid service delivery, among others [1], [6], [8]–[11]. Figure 1 shows the topology of a network with typical en-routing entities in comparison to network nodes with computational and storage capabilities in the fog computing model. The requests are generated from the *endpoints / things* layer and are routed to services deployed in upper-layer devices. These services can be allocated in several intermediate nodes or in the cloud. The placement problem of software resources for fog computing or edge computing is an NP-hard problem that consists of the selection of the optimal network entity to deploy a user application. There are some constraints and optimization factors that influence this problem, such as the user location, hardware and software features of the network entities, link characteristics (e.g., propagation, utilization, and bandwidth), user requirements, application decomposition (e.g., containers, microservices, and serverless functions), QoS, energy, and cost, among others.

These factors affect the dynamical evolution of the user movement, link failures, network congestion, and application popularity, among other aspects. One way to evaluate
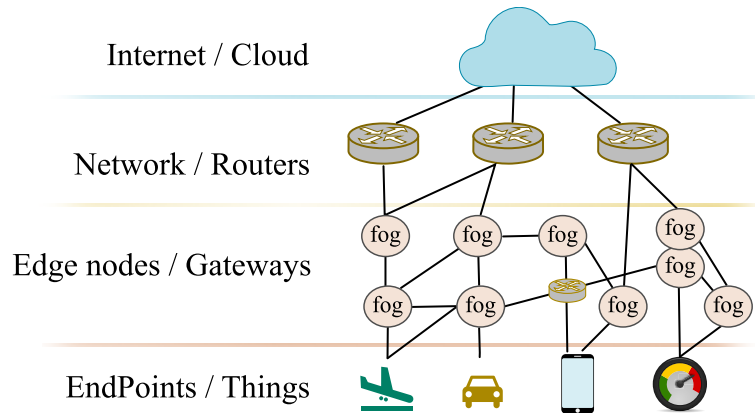
FIGURE 1. Relationships in a fog computing model.

placement solutions is through simulation. Simulators are enabling tools for modeling, analyzing and evaluating the diversity of policies and configurations.

In this paper, we present a discrete-event simulator focused on, but not restricted to, fog environments called YAFS (Yet Another Fog Simulator). YAFS is designed to analyze the design of applications and incorporates strategies for placement, scheduling and routing. We compare the YAFS characteristics with that of the iFogSim [12] simulator as a reference. iFogSim is the most widely used fog simulator, an extension of the well-known CloudSim [13] simulator. YAFS includes more functionalities than current simulators for modeling IoT scenarios. We highlight the following points:

I) *Network*: a graph or network vision allows the modeling of the communication links among machines, end-devices and users. We can obtain useful measures for our custom strategies by analyzing non-trivial topological features of the graph, such as vulnerable regions to random failures, user community distribution, efficient routing policies, and so on. Complex network theory [14]–[16] embraces all these studies that are applied to model real systems: computer networks, technological networks, brain networks and social networks. In the literature, complex networks are used to model relationships of data centers [17], virtual machine allocation [18], fog colonies [19], load balancing mechanisms [20], etc. We can take advantage of extensively tested implementations to use them along the custom simulator policies, for example, we can import CAIDA [21] and BRITE [22] topologies. In our implementation, any network entity or link can be created or removed during the simulation following custom temporal distributions and with custom tags to allow an efficient and flexible definition of scenarios.

II) *Workload sources*: each workload source represents the connection of a user or an IoT sensor or actuator that demands a service. Each source is associated with a network entity and generates requests according to a custom distribution. The workload sources can be created, changed or removed dynamically enabling the modeling of the user movements in an ecosystem.

III) Customized *Placement, scheduling and routing algorithms*: these algorithms are defined by the user. The placement algorithm is invoked in the initialization and runs along with the execution according to a personalized distribution. The routing algorithm chooses the path that connects the transmitter and the receptor, and the scheduling algorithm chooses the application that runs the task. The scheduling and routing algorithms are defined in the same manner since the path selection depends on which application is chosen. The existence of the scheduling algorithm allows choosing between different modules in case of scaling policies. By default, the simulator includes both implementations: a static placement and the selection of the minimum path between two entities where the application is deployed.

IV) *Custom processes*: personalized functions can be invoked at runtime to provide flexible implementations of real events such as the movement of the workload sources, generation of network failures, and specific data collection using third-applications such as Grafana [23] (an open platform for analytics and monitoring of computer infrastructures and services).

V) *Post-simulation data analysis*: YAFS performs automated CSV-based logging of two types of events: workload generation and computation, and link transmissions. The results are both analyzed post-simulation and provided to the user. This generates less overhead, avoids repeating the simulation to re-analyze other indicators, and enables the shareability of raw results. YAFS includes functions to obtain metrics such as network utilization, response time, network delay, and waiting time, etc. Using CSV dumps, user can implement complex analyses using R, Python or another language.

VI) *JSON-based scenario definition*: YAFS supports the importation of the scenario definition from JSON-format files. It enables the use of third-party tools that generate scenarios in a common JSON format; in addition, non-expert developers can use basic functionalities of the simulator. Thus, it is more user-friendly and a better starting point to understand the simulator. However, YAFS API enables more

modeling possibilities than the JSON-based syntax such as for example, the modeling of dynamic scenarios.

YAFS is developed in Python following the style guide PEP8 [24]. It is available under MIT license in a code repository[1] with detailed documentation, a tutorial and several examples.

To sum up, the contribution of this paper is the design of a highly customizable simulator and the design of JSON-based files for analyzing mobile IoT scenarios under the fog and edge computing paradigms.

This paper is organized as follows: Section 2 describes the state the art of the fog and cloud computing simulators. Section 3 includes the justification of the use of complex networks to model the infrastructure network. Section 4 describes the design and some details of the implementation of each component of our proposal. Section 5 includes three cases studies (involving allocation of modules, failure behavior of devices, and movement of users) and presents a comparative study of results with iFogSim in terms of performance and convergence.

II. RELATED WORK

Our solution is focused in the design of fog computing simulators. There are different simulators for several types of distributed environments such as cloud, grid, and fog edge. We know of four specific simulators regarding the topic of fog computing: FogTorch [25], EmuFog [26], Edge-CloudSim [27], FogNetSim++ [28] and iFogSim [12]. We first analyze some common features of these simulators and then provide more specific features of all of them individually.

We classify the simulators considering the following criteria, which, from our point of view, are essential for realistic modeling of fog scenarios. I) The first criterion involves the structure of the topology. The topology allows us to represent the infrastructure of the network. EdgeCloudSim and IFogSim use a tree-based structure. In contrast, FogTorch, EmuFog, FogNetSim++ and YAFS use a graph structure. In addition, YAFS supports the definition of subgraphs within a topology. Subgraphs can be used to represent isolated regions or fog colonies [19]. The generation of topologies is a complex and hard task due to the number of elements and connections. There are definitions of topologies such as BRITE and CAIDA topologies. These formats are supported by EmuFog and YAFS but not by the other simulators. II) The second criterion is related to the coding of the scenario. Existing fog simulators include an API where the characteristics of the scenario are defined; YAFS also supports the definition of the scenario through JSON-based files. The topology can also be defined using this syntax. FogNetSim++ has a graphical interface. III) The third criterion involves the characteristics of the results. If complex scenarios are designed with customizable policies, it is necessary to record all the events of the simulator in files. Thus, users

can perform complex analysis of these records to find specific indicators. EdgeCloudSim, FogNetSim++ and YAFS record these data to allow post-simulation analysis. IV) The fourth criterion involves the capability to perform changes in the fog scenario during the simulation. The modeling of realistic scenarios must include changes in different strategies. EdgeCloudSim and FogNetSim++ support changes representing the movement of users in the infrastructure. Only YAFS supports dynamic scenarios in the next strategies: placement, path routing, service orchestration and workload or user movement. V) Finally, the last criterion is the programming language. Most of them are implemented in Java, FogNetSim++ in C++, and YAFS uses Python.

We summarize these five comparative criteria in Table 1. The nomenclature used in the *Policy* column represents the policy that can be customized: *A* - stands for allocation or placement; *R* - stands for path routing, *O* - stands for service orchestration, and *W* - stands for workload or user movement. *Dynamic Policies* column indicates the previous strategies that the simulator can execute dynamically during execution.

After a global analysis of the simulators, we also present a more specific description for each of them independently.

FogTorch [25] uses Monte Carlo simulations to determine the best allocation for an application through QoS indicators such as latency, bandwidth, cost, and response time. This simulator addresses the application allocation problem. Our approach simulates the whole ecosystem only where the allocation is one of the available inputs of the simulation. In other words, FogTorch optimizes the deployment of applications under QoS restrictions, and YAFS integrates this optimized allocation values to obtain simulated metrics. Brogi *et al.* defined an application as a set of triplets of software components and interactions among components with a QoS profile. They used Monte Carlo simulations to compute the eligible deployments of software components. They also presented a fire alarm IoT application as a case study with three components: a fire manager (an actuator to extinguish the fire), a database system, and a machine learning engine. The IoT infrastructure was based on three fog nodes, two cloud entities and nine network links among them.

EmuFog [26] is a set of scripts to transform a set of initial configurations (network topology and placement criteria) into the input of the MaxiNet [29] simulator. It uses a graph representation to define the network topology. The authors implemented some functionalities to simplify the process of selection of fog nodes in regards to the topological features of the graph. Our simulator also implements this process. We delegate this type of computational processes in a complex network library to obtain topological features that the user can integrate into the topology. EmuFog application representation comes from Dockers,[2] a container platform that encapsulates an application in a stand-alone package. The evaluation uses three graph types (the Albert and Barabasi model [30], and real-world topologies from CAIDA [21] and

---

TABLE 1. Comparative table of fog simulators.

Simulator	Coding	Topology structure	Topology definition	Result traceability	Policy	Dynamic policies	Language
FogTorch	API	Graph	API	No	[A]	-	Java
EmuFog	API	Graph	API, Graph-formats	No	[A]	-	Java
EdgeCloudSim	API	Tree	API	Yes	[A,W]	[W]	Java
FogNetSim++	API	Graph	API	Yes	[O,W]	[O,W]	C++
iFogSim	API	Tree	API	No	[A]	-	Java
YAFS	API, JSON	Graph	API, JSON, Graph-formats	Yes	[A,R,O,W]	[A,R,O,W]	Python

from the BRITE tool [22]) for representing the network. The authors analyzed the edge-nodes and the most suitable placements in the evaluation. From our point of view, however, the type of application used and the relationships among containers is not clear.

EdgeCloudSim [27] is simulator based on CloudSim [13], which is one of the most referenced simulators in the field of cloud computing. Sonmez *et al.* introduced functionalities such as mobility models, network link models and edge server models to represent more realistic scenarios. Thus, new additional results were provided such as the LAN delay, number of failed tasks due to mobility and the average number of mobile clients in a specific location. They presented a scenario with three configurations: one tier, two tiers, and two tiers with an edge orchestrator. The edge orchestrator entity controls the selection of the tier in each possible task execution. This simulator incorporates new functionalities relative to the original but is restricted in the taxonomy definition and how the mobility is defined. The type of results is also limited to the CloudSim version.

FogNetSim++ [28] is an extension of OMNet ++. It inherits some functionalities of the original simulator: a graphical environment, traceability of the results at the end of the simulation, implementation of different network protocols, etc. The modeling of the ecosystem is based on four types of entities: sensors, mobile devices, fog nodes and broker nodes. It is possible to implement dynamic management of the nodes, different types of movement on the nodes and scheduling policy. The planning policy is carried out by the broker that allows the distribution of petitions among the nodes. The services of the nodes are modeled as M/M/1 or M/M/c queue. In YAFS, the service of a node follows an M/M/1 policy, but a node can have multiples deployed services. The movement of entities is based on user implementations such as changing the links of the network, or the location of the users or the characteristics of the nodes as geo-location attributes.

iFogSim [12] is a CloudSim extension that supports the management of edge-network entities and the evaluation of allocation policies. The infrastructure is defined by a set of

entities: *fog devices* (or fog nodes), *sensors*, *tuples* (such as a network link) and *actuators*. The application is modeled as a directed graph with *modules* (representing computational resources), *edges* (a data dependency between application modules), and *loops* (defining a sequence of *edges* that should be monitored along the simulation to compute the response time.

In the article, the authors present two placement strategies that we describe in detail in the evaluation section: *cloud-only placement* and *edge-ward placement*. They introduce the simulator with two case studies: a latency-sensitive online game (namely, the EGG Tractor Beam game) and intelligent surveillance through distributed camera networks. Based on the iFogSim simulator, we use the application model in our simulator, introducing new improvements in the API, and we compare our results using the first case study and the two placement strategies as explained in the article.

III. YAFS ARCHITECTURE
YAFS uses a generic library for the generation of discrete-event simulation scenarios called Simpy.[3] Simpy is a robust and stable discrete-event simulator (DES) implementation that contains functions for the definition of processes (active components) and shared resources (such as network links and queues). It performs the execution of the simulation in three modes: as fast as possible, in real time, or manually stepping through events. It can also halt the simulation in case of lack of interaction or with a fixed step size. YAFS relies in the functionalities of Simpy to control the atomic processes behind a fog domain: the transmission of workloads among network links, the computation of processes in fog nodes, and others issues that we describe below.

YAFS is defined by six main classes: *core, topology, selection, placement, population*, and *application*. Figure 2 shows the relationships among them. *Core* class integrates the rest of the fog scenario definitions and manages the simulation execution controlling the cycle of life of processes, including the customized policies: selection, placement, population
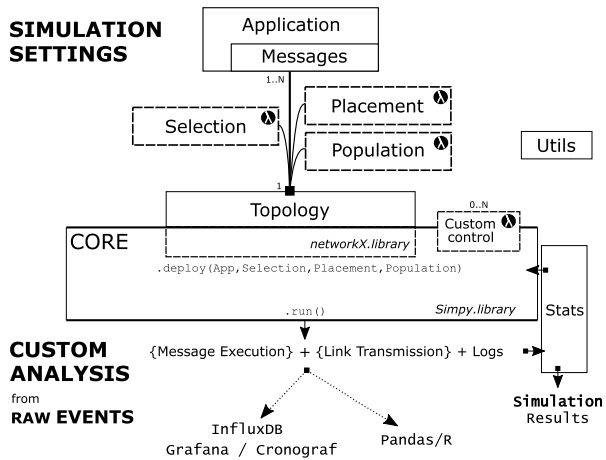
---

[3]https://simpy.readthedocs.io

FIGURE 2. The YAFS architecture is defined by six main classes: Topology, Core, Application, Selection, Placement, and Population.

and personalized controls. The main element of the *Core* class is the *Topology*. The topology structure is accessed by other classes through the control of the *Core* class. Simulation processes such as *Selection*, *Placement*, *Population* are integrated into the *Core* class to provide orchestration and selection of processes, allocation of software modules in the entities of the structure; and allocation and characterization of workloads, respectively. As shown in Fig. 2, the classes with a white lambda on a black circle symbol can interact dynamically along the simulation execution. *Core* class gathers events (such as message transmission and message execution) and stores them in a raw format. *Stats* and *Metrics* classes implement several functions to compute common measures such as the average response time, link latency, and resource utilization.

In the following sections, we examine the topology and entity modeling, the application model, the internal structure of the DES processes and the generation of the results. API documentation and a tutorial are available on-line,[4] providing further detail.

A. TOPOLOGY AND ENTITY MODELLING
All network intermediate entities are modeled as a set of nodes of a graph interconnected via network links, i.e., the edges. Thus, network elements, fog nodes, clusters or data centers are nodes and the network links are edges. This graph-based perspective allows the applicability of complex network theory. We integrate the NetworkX [31] library in our simulator. NetworkX is a well-known python library with several algorithms to compute measures such as degree, centrality, clustering, assortativity, communities, communicability, flows, isomorphism, and similarity. In addition, this library can import CAIDA [21] and BRITE [22] topologies, and it supports graph formats such as JSON, GML, GEXF, Pickle, GraphML, and Pajek.

---

[4]https://yafs.readthedocs.io/en/latest/

```
1  {
2    "id": 120, "RAM": 1, "IPT": 530,
3    "POWERmin": 574,
4    "POWERmax": 646
5    "coordinate": {"lat":39.30, "long":3.34}
6  },
7  {
8    "id": 12, "RAM": 10, "IPT": 100
9  },
```

Listing 1. Definition of two examples of fog nodes using a JSON-based representation.

The mandatory attributes to define a fog node are an identifier (ID), the number of instructions performed per unit of time (IPT) and the memory capacity (RAM). Users can include other attributes to define the topology entities such as: storage capacity, power, network adapter, and so on. In listing 1, we include two JSON-based example definitions of nodes: one with a range of power consumption and a coordinate value and one that contains only the mandatory attributes. YAFS supports a flexible definition of entities in the same scenario. Using customized attributes, we can represent logical relationships such as virtualization, containers, microservices, and serverless functions using nodes and vertices.

The definition of link attributes is similar. A network link has two mandatory attributes: bandwidth (BW) and link propagation (PR), and the user can include more attributes.

Finally, a simulation contains a unique *topology* class. As mentioned, this class is a graph-based representation where a determined number of applications and their corresponding policies are deployed. That is, each application has a unique policy of allocation of resources (*placement*), a policy of selection and orchestration of services (*selection*), and a variation of the workload (*population*). Furthermore, we can deploy *customized controls* that dynamically interact with the application and the simulation variables (such as failure generation or to improve the computational capacity of a node).

B. APPLICATION MODEL
The application model is the same than in iFogSim [12] and is based on a distributed data flow (DDF) [32]. An application is defined by modules that run services and messages (or dependencies) among modules. Thus, a DDF is represented with a directed acyclic graph where nodes are modules that perform one action on the incoming data and edges denote interoperability between modules. This application representation enables the partitioning and scaling of an application, which is useful for real program models such as microservices [33] and serverless [34] paradigms.

We adapted the application definition with regard to the iFogSim approach to complying with our design principles: independence of the results and ease of export and reuse functions. This fact is reflected in each definition phase of an application: modules, dependencies, messages and results. The mandatory attributes of a message are *instructions*
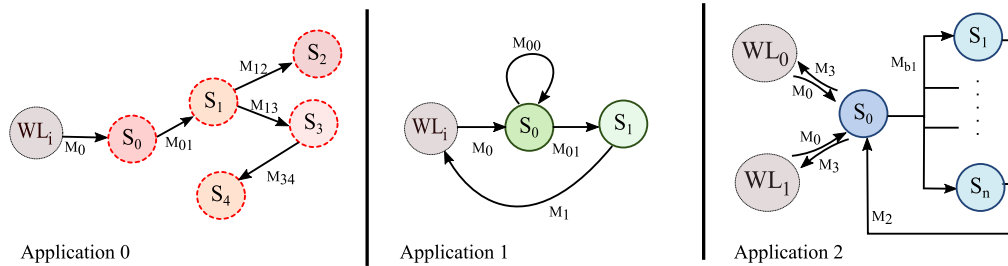
FIGURE 3. Three application types with a simple message passing, a loop message and a broadcast message, respectively.

and *bytes*. The instructions affect the service time, and the bytes affect the transmission time.

In YAFS, all types of modules are defined with the same methods. iFogSim authors use the term *dependency* to represent the relationship between modules, and these modules do not start the execution until they receive a message; instead, we use the term *message*. These messages can be used for other applications that have the same modules. The transfer of messages indicates how to transform a type of input message into another output message. In YAFS, all transfers are defined, including the generation of messages in sensors or the reception in actuators and the generation within modules (periodic messages). The decision to transmit a message within a module is also implemented, with two methods: fractional selectivity and broadcasting. The latter allows message transmission to all replicated modules. Finally, in YAFS, the response time is obtained independently of the declaration of *loops* (an internal control of iFogSim for monitoring tasks of sequence of dependencies between modules of an application), i.e., in iFogSim, if a loop is not declared before the simulation, the execution times of a sequence of dependencies between modules cannot be measured.

To understand the differences between iFogSim and YAFS, we implement the application used in the first case study of iFogSim: the EGG Tractor Game (Listing 2). The game consists of three modules: *client*, *concentration calculator* and *coordinator*, performing processing of the messages generated in the *EGG sensor*; some results are visualized in the *Display* actuator. The modules are defined in lines 9-11; modules that will be workload sources or simple sensors are defined as *sources* and *sinks*. They are necessary only to define the application. The messages are defined in lines 11-13. The following attributes are required: name, module source, module destination, instructions, and bytes. Finally, the remaining lines define the transmissions. This is where we define how a message is transformed into another and how a message is sent between modules (through a distribution, a selection or a broadcast process). The placements of workloads (source entities) are defined in the population policy.

Our implementation includes additional types of applications that can be modeled and their interactions with the workload generators, or users, with regard to other simulators.

```
1  a = Application(name="EGG_GAME")
2  a.set_modules([
3  {"EGG":{"Type":Application.TYPE_SOURCE}},
4  {"Display": {"Type": Application.TYPE_SINK}},
5  {"Client": {"RAM": 10, "Type": Application.TYPE_MODULE
        }},
6  {"Calculator": {"RAM": 10, "Type": Application.
        TYPE_MODULE}},
7  {"Coordinator": {"RAM": 10, "Type": Application.
        TYPE_MODULE}}
8  ])
9  m_egg = Message("M.EGG", "EGG", "Client", instructions
        =2000*10^6, bytes=500)
10 m_sensor = Message("M.Sensor", "Client", "Calculator",
        instructions=3500*10^6, bytes=500)
11 m_player_game_state = Message("M.Player_Game_State", "
        Calculator", "Coordinator", instructions
        =1000*10^6, bytes=1000)
12 # ...
13 a.add_source_messages(m_egg)
14 dDistribution = deterministicDistribution(name="
        Deterministic", time=100)
15 a.add_service_source("Calculator", dDistribution,
        m_player_game_state)
16 a.add_service_source("Coordinator", dDistribution,
        m_global_game_state)
17 a.add_service_module("Client", m_egg, m_sensor,
        fractional_selectivity, threshold=0.9)
18 a.add_service_module("Client", m_concentration,
        m_self_state_update, fractional_selectivity,
        threshold=1.0)
19 # ...
```

Listing 2. Definition of EGG Game application presented in iFogSim [12] using our API.

Figure 3 shows three application types. Each application is set up by software modules represented with circles. *Application 0* presents a hierarchical structure where the messages (identified with M_{ij}) trigger other messages. In the example, post-execution of M_{01} triggers M_{12} and M_{13}. In *application 1*, we can observe a self-message and an interaction with the user from other module. In the last case, *application 2*, there is a broadcasting message (M_{b1}) that reaches all S_1-module deployments. Each S_1-module returns a message that is addressed by S_0-module and finally returns a response to a specific user. This last feature enables the return of responses to the initial claimant.

We include a supplementary PDF document which contains several examples of applications. In one of them, we define an application based on the structure of *application 0* (Fig. 3) and the correspondent scenario (population policy, allocation policy and infrastructure) using a specific JSON-based syntax. The use of JSON files provides a common framework for the definition of Fog scenarios where

```
1   delayActivation = deterministicDistributionStartPoint
        (3000,300,name="Deterministic")
2   periodicActivation = deterministicDistribution(name="
        Deterministic", time=100)

3
4   popA = Statical(name="StaticalPop")
5   popA.set_sink_control({"id": a_id_fog_device, "number":
        2, "module": appA.get_sink_modules()})
6   popA.set_src_control({"number": 1, "message": appA.
        get_message("M.Action"), "distribution":
        periodicActivation})

7
8   top20Devices = [''array_ids_fog_devices'']
9   popB = Evolution(top20Devices, name="DynamicPop",
        activation_dist=delayActivation)
10  popB.set_sink_control({"model": "actuator-device","
        number":2,"module":appB.get_sink_modules()})
11  popB.set_src_control({"number": 1, "message": appB.
        get_message("M.Action"), "distribution":
        periodicActivation})
```

Listing 3. Declaration of two population policies: one static (*popA*) and the other dynamic (*popB*).

non-expert developers can easily design experiments. This document is available at http://ieeexplore.ieee.org.

C. DYNAMIC POLICIES

The *Selection*, *Placement* and *Population* classes dynamically generate the events in the scenario. The first class chooses the entity that performs the execution of application modules; hence, it routes the workload. The *Placement* class determines the allocation of each application module. The *Population* class allocates the workload generators in the network entities. These classes possess two main interfaces: an initialization function and a function invoked according to a customized temporal distribution. The initialization function prepares the allocation of modules and workloads on topology entities.

To illustrate these type of processes, we describe the population definition where we map workload generators in the entities of the infrastructure. This procedure requires three steps. First, we need to choose the type of message that is generated from the workload sources. The messages are defined in the *Application* class, and they are requested to perform the execution of application modules. Second, we have to define the temporal distribution. Finally, we have to associate how many of these generators we wish to have in the nodes. We have included an example of population criteria in Listing 3). Lines 1 and 2 define two temporal distributions. The first one starts at 3000 time units, and from that point in time, it triggers an activation every 300 time units. The second one triggers an activation every 10 time units. Line 4 generates an instance of a predefined extended class of *Population*. In *application A*, there are two types of modules: workload sources and workload sinks (similar to sensors and actuators). Thus, Lines 5 and 6, through JSON-based syntax, define the allocation of each sink module (in this case, we incorporate all sink modules, and we duplicate the number in the same entity), and the allocation of each workload source with a distribution and a type of message.

We can extend the *Population* class to model more complex scenarios (Lines 8-11). In this sense, the new instance (Line 9)

```
1   class Evolution(Population):
2     def __init__(self,listIDEntities,**kwargs):
3       #initialisation of internal variables...
4       super(Evolution, self).__init__(**kwargs)

5
6     def initial_allocation(self, sim, app_name):
7       #dealing assignments...
8       sim.deploy_sink(app_name, node=fog_device, module=
           module)

9
10    def run(self, sim):
11      #dealing assignments: msg, distribution and app_name.
12      id = ... # listIDEntities.next
13      idsrc = sim.deploy_source(app_name, id_node=id, msg
           =..., distribution=...)
```

Listing 4. Structure of a population class with three mandatory functions: *init*, *initial_allocation* and *run*.

of an extended *Population* operation follows the distribution defined in Line 1. This DES process starts generating workload sources at a certain time point, after which it is activated every 300 time units. In each activation, it generates a new workload source with the characteristics defined in Line 11, and it is assigned in each entity defined in the array (*top20Devices*). Sink modules are generated in the initialization phase. A simplified version of this *Evolution* class is shown in Listing 4). In these types of processes, there is a mandatory function called *initial_allocation* and, optionally, a function called *run* that is invoked dynamically according to the distribution. Internally, functions defined in the *Core* class are used for our modeling, such as *deploy_source* or *deploy_sink*. Note that the *sim* variable is the instance of the simulation. We can control the topology and the rest of the DES processes, together with the simulation execution. This variable is created in the *Core* class. The rest of the classes (selection, placement and customized processes) present a similar structure and behavior.

D. RESULTS

There are two types of events recorded (namely, task executions and network transmissions), but users can record specific metrics with customized DES processes. The results are stored in two CSV files.

When a node performs the work associated with a message, the simulator records the following attributes: *id, type, app, module, message, DES.src, DES.dst, TOPO.src, TOPO.dst, module.src, service, time_in, time_out, time_emit, time_reception*. Specifically, ***id*** is an incremental integer value that remains constant during message propagation or transformation in other messages of the application. This approach allows controlling when an application partitioned in modules ends the execution of its complete service. The attribute ***type*** identifies the type of module (computational or sink). The attribute ***app*** identifies the application (the name attribute). ***module*** identifies the application module (the name attribute) that performs the service. ***message*** identifies the message (the name attribute). ***DES.src*** and ***DES.dst*** are the identifiers of the DES processes that send and receive the message, respectively. ***TOPO.src*** and ***TOPO.dst*** are the identifiers of the topology entities where the modules are
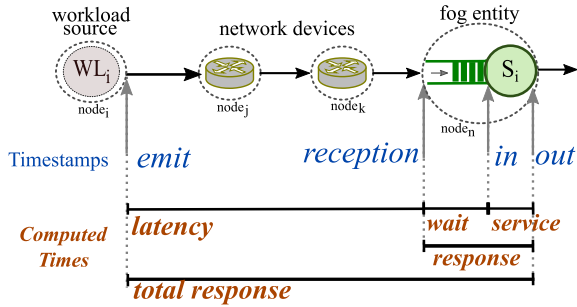
FIGURE 4. Timestamps generated by the discrete-event simulator and post-computed times in the fog node computation of a message.

```
1   id,type,app,module,message,DES.src,DES.dst,TOPO.src,
        TOPO.dst,module.src,service\\
2   ,time_in,time_out,time_emit,time_reception
3   1,COMP_M,EGG_GAME,Client,M.EGG,0,258,4,3,EGG
        ,1.9994,104.005,105.9994,100.0,104.0005
4   2,COMP_M,EGG_GAME,Client,M.EGG,2,259,7,6,EGG
        ,1.9994,104.005,105.9994,100.0,104.0005
5   3,COMP_M,EGG_GAME,Client,M.EGG,4,260,10,9,EGG
        ,1.9994,104.005,105.9994,100.0,104.0005
6   ...
```

Listing 5. Fog node computations recorded in a CSV file for subsequent analysis of the results.

deployed. *module.src* identifies the application module that sends the message. The *service* attribute can have a None value (if the message record comes from a workload source) or a numerical value corresponding to the service time.

Figure 4 shows the four timestamps involved in the transmission of a message from the source to the destination entity where the software module performs the action. The next timestamps are relative to the simulation time because it is a discrete-event simulator. The label *time_emit* is the value that represents the emission time of a message in a module source. The label *time_reception* represents the recorded time when a message arrives to the destination module. When the message arrives, it is enqueued; finally, we record the entry and the exit of the service (*time_emit* and *time_out*, respectively). The service time is the division of *instructions* (message attribute) between *instructions per time* (entity attribute). These times are used to compute useful measures such as the latency, waiting time, response time and total response time (see computed times in Fig. 4). We show a sample of those records in Listing 5. Using the timestamps entries of the first requests (Listing 5), $time_{in}$ is 104.005, $time_{out}$ is 105.9994, $time_{emit}$ is 100.0, and $time_{reception}$ is 104.0005; we compute the latency as 104.0005-100.0, the waiting time as 104.005-104.005, the service time as 105.9994-104.005, the response as 105.9994-104.005, and the total response as 105.9994-100.00.

In a network transmission, YAFS records the following attributes: *id, type, src, dst, app, latency, message, ctime, size, buffer*. Here, *id, type, app,* and *message* take the same values as mentioned previously. *src* and *dst* are the identifiers of the topology entities that send and receive the messages. *ctime* is the simulation time when the action is performed. *size* is the size of the message. Finally, *buffer* is an integer value that

```
1   id,type,src,dst,app,latency,message,ctime,size,buffer
2   1,LINK,4,3,EGG_GAME,4.000005,M.EGG,100,500,0
3   2,LINK,7,6,EGG_GAME,4.000005,M.EGG,100,500,1
4   3,LINK,10,9,EGG_GAME,4.000005,M.EGG,100,500,2
5   ...
```

Listing 6. Network link transmissions are recorded in another CSV file.

```
1   simulation_time = 100000
2   time_loops = [["M.EGG", "M.Sensor", "M.Concentration"]]
3   s = Stats(defaultPath=path+"Results")
4   s.showResults(simulation_time, time_loops=time_loops)
5
6   s.df["date"]=s.df.time_in.astype('datetime64[s]')
7   s.df.index = s.df.date
8   print s.df.resample('300s').agg(dict(time_latency='mean
        '))
```

Listing 7. Analysis of the simulation results.

represents the number of messages in the whole network that are waiting for a link service. Consequently, a link can send only one message at a time, and messages have to wait for unused slots. This value is an indicator of network saturation and is updated in each record. In the case shown in Fig. 4, the transmission of the message from the workload source to the fog node generates three network transmission records. We show a sample of these types of records in Listing 6.

We have implemented some common methods (in the *Stats* class) to obtain more complex measures using the Pandas library [35]. Pandas is an open source library with several data analysis tools. To illustrate this data analysis, we include the next example in Listing 7. The first approach follows the idea of *sequences* defined in iFogSim. From a sequence of messages (line 2), the *showResults* function provides the same results as iFogSim (line 4). In addition, we can perform more complex analysis. For example, we can compute the average latency each 300 units of time. To obtain these values, we use the Pandas time series functionalities to sample the records in that time period and to apply the average function on latency values (lines 6-8), where `df` (a *dataframe*) contains the CSV data.

IV. EVALUATION

In the first section, we compare YAFS and iFogSim simulators in terms of performance and results using an application case defined in iFogSim [12]. In the second section, we analyze the convergence of both simulators using the same experiments.

It is important to note that the results are not equal between both simulators, although we try to use similar settings. The definition of attributes is different in both simulators. These cases are the following: I) iFogSim uses the measure of MIPS in its computational devices. YAFS uses IPT (*instructions per time*. Both measures are related but some MIPS possible values in iFogSim are not clear. II) In iFogSim, the attributes of a link are included in the fog node using terms such as *upBW* and *downBW*, and there is another latency value in the connection between modules (i.e., *eegSensor.setLatency(6.0)*). In our case, the BW is defined in the link and has the same value in both directions. In addition, we define the
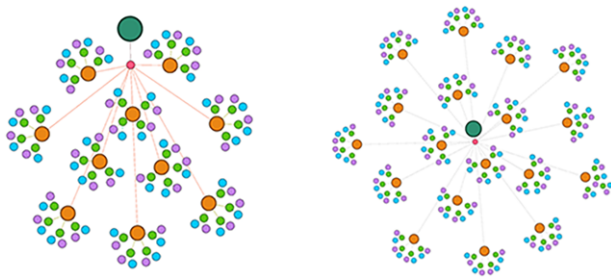
FIGURE 5. Network topologies with 10 fog devices (left figure) and 18 fog nodes (right figure). The large green node represents the cloud entity, pink nodes are the proxies, orange nodes are the gateways or fog devices, and small green nodes are client devices with one sensor (purple) and one actuator (blue).

FIGURE 6. Execution time comparison between *cloud* and *edge* policies with a different number of gateways: 4, 8, 12 and 16.

FIGURE 7. Number of messages enqueue (waiting) by network saturation using YAFS with *cloud policy* in increments of fog devices (4, 8, 12 and 16).

propagation time, which is not included in iFogSim III) In iFogSim, a message has attributes such as *tupleCPULength* and *tupleNwLength*, corresponding to the number of millions of instructions and bytes, respectively. In any case, temporal distributions are the same in the experiments, and we try to use similar values in the previously described attributes.

A. COMPARISON WITH IFOGSIM

We use the first case study presented in the iFogSim paper (namely, the EGG Tractor Beam game) for the comparison between both simulators. This application consists of 3 modules: *client*, *concentration*, and *coordinator*, and the experiment deploys the modules in a hierarchical three-based topology with a cloud entity that is linked to a gateway where all fog devices are connected. The network can be scaled from the gateway device generating several subgroups. Figure 5 represents an example of two topologies with 10 and 18 gateway subgroups.

We analyze two different placement strategies: a *cloud-only placement* (cloud policy) where all modules are deployed in the cloud entity and an *edgeward placement* (edge policy) where the modules are deployed in fog devices (orange nodes in Fig. 5). Both strategies are explained in the iFogSim paper. From the simulation, we analyze the following data: execution time and response time. In addition, we vary the number of fog nodes: 4, 8, 12 and 16. The simulation is executed in a machine with 8 i7-cores running at 3.745 GHz with 8 GB RAM. Because of the stable convergence of both simulators, as described in Section 4.2, we performed each experiment only once.

Figure 6 shows the execution time in both policies with regard to the increment of fog nodes. Blue lines are the results of iFogSim and green lines, YAFS. Circle marks correspond to *cloud policy* and star marks to *edge policy*. At first glance, the behavior of both simulators is quite similar, but we can appreciate some differences: I) In the *cloud policy*, a greater number of transmissions must be made since all messages go through more network links to compose the cloud entity. This volume of traffic possibly generate a saturation issue in the network that affects the iFogSim runtime; II) *edge policy* generates more application modules; there are more
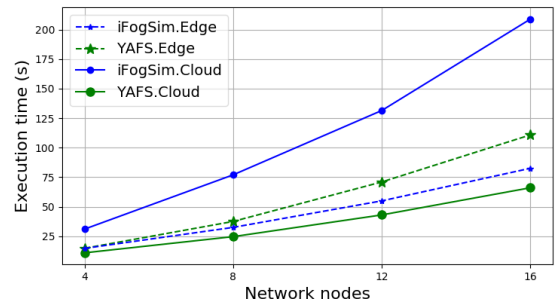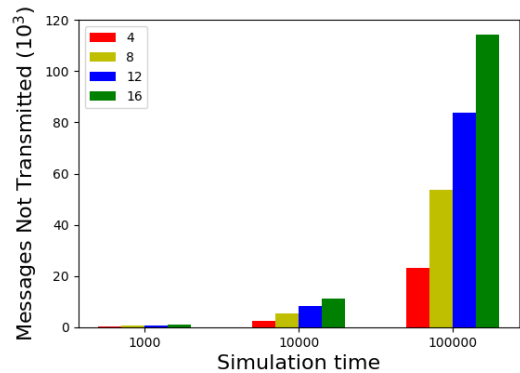
DES processes to control each module, and this fact slightly affects the YAFS runtime. An increment of the simulation runtime is reasonable as more modules are controlled; however, the saturation of the simulated system not should affect the simulator itself.

The network is saturated with the parameterization of the *cloud policy* experiment. The saturation is greater when there are more network devices and is proportional between different gateway subgroups. In Fig. 7, we represent the total number of messages waiting for the service in each level of fog nodes using YAFS. iFogSim does not provide this measure.

Another comparison is the latency time of the application. In this experiment, the latency is calculated as the sum of transmissions among the three modules, and the response times is included: the EGG sensor ↔ client module, client ↔ coordinator module and coordinator ↔ client module. A substantial difference between the simulators is the need to indicate the sequence of messages in iFogSim before simulation execution. In YAFS, this step is not necessary; those latencies are calculated post-simulation.

With the *edge policy*, the clients and fog devices entities always have the same network distance, and the response time has to be constant and independent of the number of fog nodes. Table 2 shows that the convergence of the latency time is better in YAFS. Note that the seed of the random numbers is always the same in each increment of gateways

in both simulators. This seed changes only with the number of messages and the simulation time.

With the *cloud policy*, the coordinators are allocated in the cloud entity. Subsequently, more messages are transmitted across the network and must pass through the same link. In both cases, the latency presents an exponential trend (Table 2). The parameterization avoids the network saturation in the iFogSim execution with 4 fog nodes.

B. CONVERGENCE

We analyze the convergence of YAFS using the same example of iFogSim as in Section 4.1. In this case, we use the *edge policy* since it is a stable configuration of the system. We run both simulators 50 times with a simulation time of 10,000 units in each fog node configuration (4, 8, 12 and 16) to compute the latency time.

In this experiment, two factors change the precision of the latency time: the simulation time and the number of fog nodes. The first factor exhibits coherent behavior in simulation experiments but in this case is constant. The second factor, the number of fog nodes, affects the number of transmitted messages; then, it statistically increases the number of samples. We can expect a reduction in the variance in each fog node increment. In Table 3, we include the numerical values (mean, variance, minimum, and maximum) of each simulator. We observe the reduction of the variance in each experiment. The divergence between the different ranges may be due to differences in the configuration of the experiment in each simulator, but YAFS is slightly more stable.

V. THREE COMPLEX SCENARIOS

In a second experimental phase, we highlight selected YAFS features, and we implement three dynamic IoT scenarios: allocation of new modules, failures on the infrastructure, and user mobility.

The first step is the definition of the network infrastructure (or topology). To illustrate the use of complex networks, we use the Graph Stream Generator library [36] to create a Euclidean random graph [37]. This topology is the same for all three experiments (with a size of 400 nodes and 2242 edges) where the links have the same propagation speed (1 time unit) and fog nodes can serve an unlimited number of modules. We choose this type of graph since such graphs represent social relationships among individuals and have a high connection degree.

The application consists of two modules: *senders* and *receivers*, and it has only one type of message. In this way, complex data analysis is avoided in the experiment. Initially, we randomly allocated 100 *senders* in the topology, and the number of *receivers* depends on each case study. Each *sender* generates a message each 10 time units, and the service time of the *receiver* is 0.0. Thus, the response time is equal to the latency time. To ensure accurate replication of the experiments, the seed of the random number generator is the same for all the experiments.
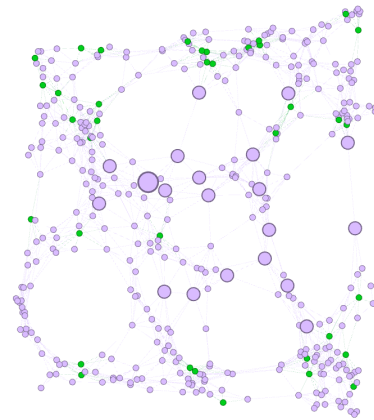
(a) A Euclidean random topology where the senders are allocated on green nodes and the size of the nodes represents the betweenness centrality.
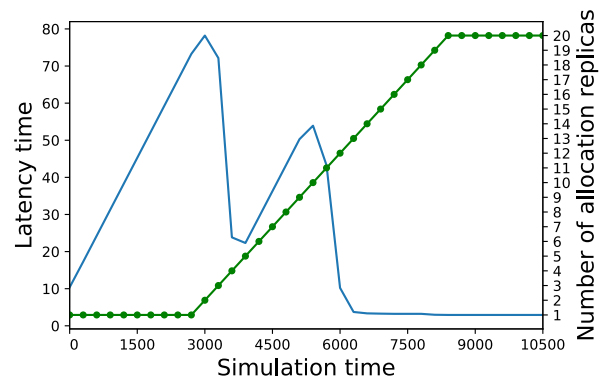
(b) Evolution of the latency time (blue line) relative to the number of allocated fog nodes (green dotted line).

FIGURE 8. First scenario: Dynamic allocation of modules.

In these three experiments, the results (latency times) are average values from the simple sequence between a *sender* and a *receiver*. The computation is similar to lines 6-8 from Listing 7, i.e., the value is the average aggregation of a set of values of a time period. The selection policy is based on the minimal path distance between a *sender* and a *receiver*.

We allocate the *receiver* modules selecting the nodes with the biggest betweenness centrality of a graph. The betweenness centrality is a measure of the relative importance of a node in the path between other pairs of nodes. A way of computing the betweenness centrality of a node is counting the number of shortest paths that travels through it [38]. Nodes with high betweenness centrality values facilitate or inhibit the communication between other nodes in the network. The goodness-of-fit evaluation of this measure as an indicator to select a network device such as a fog node is not part of this study, but some analyses have been performed in previous studies [39]. All three experiments and results are available in the code repository in the example folder .[5]

---

[5] https://github.com/acsicuib/YAFS/tree/master/src/examples

TABLE 2. Latency time with two different policies: *Cloud* and *edge* varying the number of fog nodes and the simulation time.

	Simulator	YAFS		iFogSim	
	Simulation Time	10^3	10^5	10^3	10^5
	Cloud Policy				
fog nodes	4	725.7662	59778.5152	225.5632	225.5609
	8	847.6964	78614.0355	440.5626	28074.4655
	12	1100.2678	85519.9283	536.8919	40334.4660
	16	459.7284	89119.8638	569.4413	44597.4717
	Edge Policy				
fog nodes	4	11.2744	11.2696	30.5536	31.6587
	8	11.2691	11.2667	31.4244	31.6954
	12	11.2671	11.2654	30.8828	31.6785
	16	11.2661	11.2649	30.2859	31.6964

TABLE 3. Convergence of YAFS and iFogSim using 50 samples with an *edge policy* configuration and different number of fog nodes with 10,000 time units of simulation.

		YAFS				iFogSim			
		mean	var.	min.	max.	mean	var.	min.	max.
fog nodes	4	11.2697	4.646e-07	11.2683	11.2706	31.6771	0.0414	31.2096	32.0984
	8	11.2664	9.704e-07	11.2655	11.2668	31.6699	0.0203	31.3082	31.9101
	12	11.2654	4.018e-08	11.2649	11.2656	31.6564	0.0156	31.3886	32.0163
	16	11.2648	2.064e-08	11.2642	11.2650	31.6598	0.0169	31.3923	31.9032

A. DYNAMIC ALLOCATION OF MODULES

In the first scenario, we scale the number of *receiver* modules. The objective is to observe how the latency time improves as this number grows. In the initialization phase, 100 *senders* are deployed with one *receiver*, which is deployed on the node with the highest betweenness centrality. From time point 3000 of the simulation time, a new receiver is added with a period of 300 units. This process is repeated 19 times (a total number of 20 receivers are deployed). With this experimentation configuration, we observe that with 20 fog nodes, the response time tends to be stable.

Figure 8a represents the network, where green nodes contain the *senders* and the size of the nodes represents the betweenness centrality. The results of the execution are shown in Fig. 8b. The blue line is the evaluation of latency time, and the green dotted line is the number of *receivers* deployed.

We can observe that the network is saturated with only one receiver (from 0 to 3000 in simulation time) because the latency is continuously increased. From the first deployment (time point 3300), the latency is reduced due to a higher number of available receivers, and the messages are more evenly distributed across the network. From the fifth module, however, the inclusion of new receivers along the network does not introduce any improvement since previous receivers still receive the workload. Note that the selection process of receivers is based on the minimum path between a source and a destination node. Starting at the eleventh module, the allocations have an impact in the selection strategy, and the latency is again stabilized.

B. DYNAMIC FAILURES ON NETWORK DEVICES

In the second scenario, we implement a dynamic failure of nodes where the failure rate is based on an exponential distribution. The objective is to observe how the latency time worsens as this number of failures grows and consequently to show how the simulator can implement dynamic scenarios. In this experiment, we remove only the fog nodes and other network entities. *Sender* modules are not removed to ensure that the workload is the same throughout the simulation.

When a node fails, the node and its links are removed from the topology, which can affect the internal processes that the simulator handles. Thus, a new routing is computed for the messages that had a path through the failed node. If there is no other possible path, the simulator catches and records this outcome in a log. When a removed node has waiting messages to be served, the messages are discarded, and the simulator records this case.

In the initialization of this experiment, there are 100 *senders* and 20 *actuators* deployed. All of them are allocated in the same nodes as in the previous experiment. The failures are generated from 500 time units and beyond with a mean of 100 time units. At the end of the experiment, the number of nodes available is 314, and the number of links is 1359, i.e., 86 nodes and 883 links are removed.

We represent the topology in Fig. 9a, where red colored nodes are randomly chosen to be removed during the simulation. There are five red colored nodes (fog nodes with allocated *actuators*), which will be removed.

In Fig. 9b, we represent the evolution of aggregate latency times (samples are aggregated each 100 time units), and the failures are represented with black lines or green arrows in the upper part of the graph. A black line marks the failure of a network device, and a green arrow represents the failure of a fog device. As we can observe, the latency worsens as failures occur.
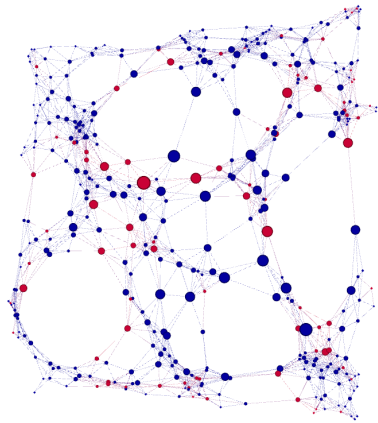
(a) Network topology where red nodes are randomly chosen to be removed.
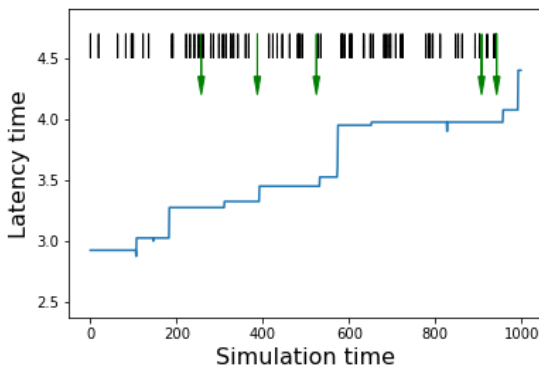

(b) Evolution of the latency with failures on fog nodes (green arrows) and failures on network nodes (black lines).

**FIGURE 9.** Second scenario: Dynamic failures on network devices.

C. DYNAMIC MOVEMENT OF MESSAGE SENDERS

Our objective with the third scenario is showing how the simulator can be used to model dynamical environments in which users or services change the location along the simulation. We model the localization of both of them (*senders* and *receivers*) according to the nodes where they are connected. In periodic steps, we approximate the *senders* by allocating them in one closer node to the *receivers*, which are statically allocated in specific nodes. The allocation is based on the shortest path between the *sender* and *receiver* nodes. Thus, in each step, all *senders* are allocated in the next node of this path, reducing, by this, the hop count (or the shortest path length). The objective is to observe how the latency time is reduced since, in each step, *receivers* and *senders* are closer and, also, to show how the simulator allows the modeling of scenarios with dynamic workloads.

In the initialization phase, there are 100 *senders* randomly allocated and 20 *receivers*. All the *receivers* are allocated in the node with the highest betweenness centrality. Every 400 time units, all the *senders* are moved to the next nearest node with regard to the *receiver* nodes. We use the shortest path function to compute the next node. As the links around the *receiver* nodes receive many requests, we reduce the generation rate of the requests (100 time units). In addition,
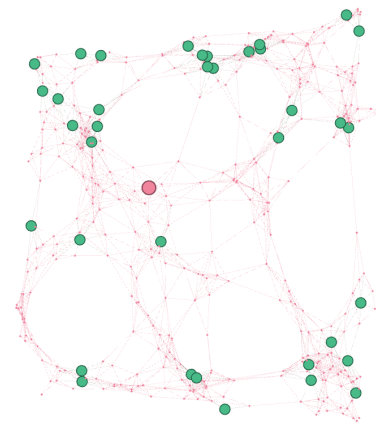
(a) Initial locations of *sender* modules (green nodes) on the topology. Receivers are in the pink node.
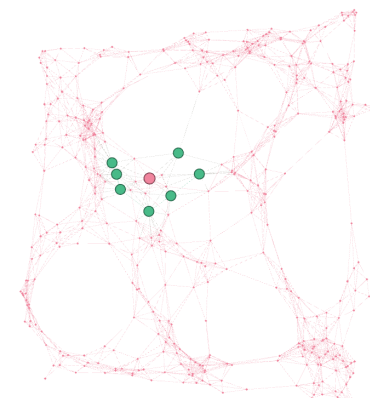
(b) Final location of *sender* modules.
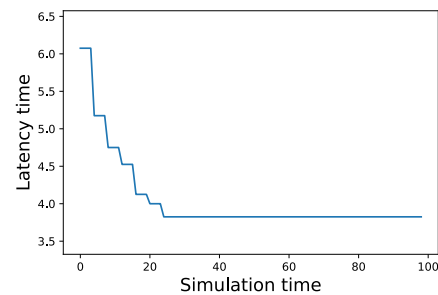
(c) Evolution of the latency time during the simulation.

FIGURE 10. Third scenario: Dynamic movement of workload sources.

the selection policy of this experiment includes a round robin scheduler to select different *receivers*.

Figure 10a and Fig. 10b represent the topology (same layout configuration than previous experiments). The green colored nodes represent the initial location of the *senders* (Fig. 10a), and at the end of the simulation (Fig. 10b). The pink colored node contains the *actuators*.

The latency decreases at each step and ultimately converges at approximately 4.5 time units (Fig. 10c). The latency is obtained from the aggregation of the time series of events every 100 time units. Most of the *senders* pass from the

node with most closeness to the *actuators* node in an average of 5 steps. We include in supplementary material a MPG4 format movie clip where we represent the movement of the senders at each step. This will be available at http://ieeexplore.ieee.org.

In summary, the customization of temporal distributions and the structure of the YAFS engine enable a direct and flexible control of any type of event inside of the DES engine. Another notable aspect of the YAFS design is that it is based on a style of open programming, maximizing the use of third-party libraries for delegating internal tasks such as the generation of topologies, visualization, or data analysis. For instance, as we show in experiments, we use complex network theory to perform several studies, and we export the topology to other graph formats for debugging and visualization.

VI. CONCLUSION AND FUTURE WORK

We present a fog computing simulator for modeling complex IoT domains. Our simulator, called YAFS, meets several design objectives: a user customized configuration of fog attributes, policies and a dynamic invocation of policies during the simulation, a definition of network topologies based on complex network theory, and the capacity to record computational and transmissions results in a CSV format. This last point makes the simulator ideal for enhancing interoperability with third-party libraries, such as Grafana, for the creation of control panels to simulate monitoring infrastructures, or Panda or R for data analysis. In addition, the infrastructure and policies definitions can be done following a predefined JSON-based format, which simplifies the use of the simulator by non-expert programmers and facilitates the integration of results of optimization algorithms for the evaluation of fog placement proposals.

Regarding the evaluation, we compare two policies (cloud and edge allocations) with iFogSim simulator. In both policies, the convergence of the results is similar. However, the YAFS runtime is slightly better than that of iFogSim. We highlight some YAFS characteristics designing three complex experiments that are not compared with iFogSim since they cannot be implemented under its API: in the first experiment, we create new fog nodes along the simulation; in the second one, we dynamically simulate failures of devices; and in the third experiment, we represent the movement of workloads in the infrastructure. The results are consistent with the expected values in each experiment.

Future work will mainly cover the development of power-aware management policies, functions for controlling the computational capacity of the resources and improvements in the nomenclature.

REFERENCES

[1] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the Internet of Things," *IEEE Access*, vol. 6, pp. 6900–6919, 2018.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. 1st Ed. MCC Workshop Mobile Cloud Comput. (MCC)*. New York, NY, USA: ACM, 2012, pp. 13–16. doi: 10.1145/2342509.2342513.

[3] O. Consortium. (2017). *Openfog Reference Architecture for fog Computing*. Accessed: Dec. 18, 2018. [Online]. Available: https://www.openfogconsortium.org/wp-content/uploads/OpenFog_Reference_Architecture_2_09_17-FINAL-1.pdf

[4] P. M. Mell and T. Grance, "SP 800-145. The NIST definition of cloud computing," Gaithersburg, MD, USA, Tech. Rep., 2011.

[5] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[6] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[7] T. V. Do, N. H. Do, H. T. Nguyen, C. Rotter, A. Hegyi, and P. Hegyi, "Comparison of scheduling algorithms for multiple mobile computing edge clouds," *Simul. Model. Pract. Theory*, vol. 93, pp. 104–118, May 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1569190X18301527

[8] E. Baccarelli, P. G. V. Naranjo, M. Scarpiniti, M. Shojafar, and J. H. Abawajy, "Fog of everything: Energy-efficient networked computing architectures, research challenges, and a case study," *IEEE Access*, vol. 5, pp. 9882–9910, 2017.

[9] A. V. Dastjerdi and R. Buyya, "Fog computing: Helping the Internet of Things realize its potential," *Computer*, vol. 49, no. 8, pp. 112–116, Aug. 2016.

[10] R. Buyya and A. V. Dastjerdi, *Internet of Things: Principles and Paradigms*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2016.

[11] R. K. Naha, S. Garg, D. Georgakopoulos, P. P. Jayaraman, L. Gao, Y. Xiang, and R. Ranjan, "Fog computing: Survey of trends, architectures, requirements, and research directions," *IEEE Access*, vol. 6, pp. 47980–48009, 2018.

[12] H. Gupta, A. V. Dastjerdi, S. K. Ghosh, and R. Buyya, "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, edge and fog computing environments," *Softw., Pract. Exper.*, vol. 47, no. 9, pp. 1275–1296, 2017. doi: 10.1002/spe.2509.

[13] R. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw., Pract. Exper.*, vol. 41, no. 1, pp. 23–50, 2011. doi: 10.1002/spe.995.

[14] S. H. Strogatz, "Exploring complex networks," *Nature*, vol. 410, pp. 268–276, Mar. 2001. doi: 10.1038/35065725.

[15] M. E. J. Newman, "The structure and function of complex networks," *SIAM Rev.*, vol. 45, no. 2, pp. 167–256, 2003. doi: 10.1137/S003614450342480.

[16] B. Saha, A. Mandal, S. B. Tripathy, and D. Mukherjee, "Complex networks, communities and clustering: A survey," *CoRR*, vol. abs/1503.06277, pp. 1–15, Mar. 2015. [Online]. Available: http://arxiv.org/abs/1503.06277

[17] S. Filiposka and C. Juiz, "Complex cloud datacenters," *IERI Procedia*, vol. 7, pp. 8–14, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2212667814000227

[18] O. Biran, A. Corradi, M. Fanelli, L. Foschini, A. Nus, D. Raz, and E. Silvera, "A stable network-aware vm placement for cloud systems," in *Proc. 12th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2012, vol. 1, no. 1, pp. 498–506.

[19] I. Lera, C. Guerrero, and C. Juiz, "Availability-aware service placement policy in fog computing based on graph partitions," *IEEE Internet Things J.*, vol. 6, no. 2, pp. 3641–3651, Apr. 2019.

[20] Z. Zhang and X. Zhang, "A load balancing mechanism based on ant colony and complex network theory in open cloud computing federation," in *Proc. 2nd Int. Conf. Ind. Mechatron. Automat.*, May 2010, vol. 2, no. 1, pp. 240–243.

[21] (2018). *Center for Applied Internet Data Analysis. Macroscopic Internet-topology Data Kit (ITDK)*. Accessed: Dec. 18, 2018. [Online]. Available: https://www.caida.org/data/internet-topologydata-kit

[22] A. Medina, A. Lakhina, I. Matta, and J. Byers, "BRITE: An approach to universal topology generation," in *Proc. MASCOTS*, Aug. 2001, pp. 346–353.

[23] I. Nurgaliev, E. Karavakis, and A. Aimar, "Kibana, Grafana and zeppelin on monitoring data," Tech. Rep., Aug. 2016. doi: 10.5281/zenodo.61079.

[24] D. Goodger and B. Warsaw. (2000). *Index of Python Enhancement Proposals*. Accessed: Dec. 18, 2018. [Online]. Available: https://www.python.org/dev/peps/

[25] A. Brogi and S. Forti, "QoS-aware deployment of IoT applications through the fog," *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1185–1192, Oct. 2017.

[26] R. Mayer, L. Graser, H. Gupta, E. Saurez, and U. Ramachandran, "Emu-Fog: Extensible and scalable emulation of large-scale fog computing infrastructures," in *Proc. IEEE Fog World Congr. (FWC)*, Oct./Nov. 2017, vol. 1, no. 1, pp. 1–6.

[27] C. Sonmez, A. Ozgovde, and C. Ersoy, "Edgecloudsim: An environment for performance evaluation of edge computing systems," in *Proc. 2nd Int. Conf. Fog Mobile Edge Comput. (FMEC)*, May 2017, vol. 1, no. 1, pp. 39–44.

[28] T. Qayyum, A. W. Malik, M. A. K. Khattak, O. Khalid, and S. U. Khan, "FogNetSim++: A toolkit for modeling and simulation of distributed fog environment," *IEEE Access*, vol. 6, pp. 63570–63583, 2018.

[29] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. Zahraee, and H. Karl, "Maxinet: Distributed emulation of software-defined networks," in *Proc. Netw. Conf. (IFIP)*, Jun. 2014, pp. 1–9.

[30] A.-L. Barabási and M. Pósfai, *Network Science*. Cambridge, U.K.: Cambridge Univ. Press, 2016. [Online]. Available: http://barabasi.com/networksciencebook/

[31] A. Hagberg, P. Swart, and D. S. Chult, "Exploring network structure, dynamics, and function using NetworkX," in *Proc. 7th Python Sci. Conf.*, Jan. 2008, pp. 11–15.

[32] N. K. Giang, M. Blackstock, R. Lea, and V. C. M. Leung, "Developing iot applications in the fog: A distributed dataflow approach," in *Proc. 5th Int. Conf. Internet Things (IOT)*, Oct. 2015, vol. 1, no. 1, pp. 155–162.

[33] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*. Cham, Switzerland: Springer, 2017, pp. 195–216. doi: 10.1007/978-3-319-67425-4_12.

[34] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, *Serverless Computing: Current Trends and Open Problems*. Singapore: Springer, 2017, pp. 1–20, doi: 10.1007/978-981-10-5026-8_1.

[35] W. McKinney, "Data structures for statistical computing in python," in *Proc. 9th Python Sci. Conf.*, S. van der Walt and J. Millman, Eds., 2010, pp. 51–56.

[36] Y. Pigné, A. Dutot, F. Guinand, and D. Olivier, "GraphStream: A tool for bridging the gap between complex systems and dynamic graphs," *CoRR*, vol. abs/0803.2093, pp. 1–10, Mar. 2008. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr0803.html#abs-0803-2093

[37] P. Erdős and A. Rényi, "On the evolution of random graphs," in *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 1960, pp. 17–61.

[38] L. C. Freeman, "Centrality in social networks: Conceptual clarification," *Social Netw.*, vol. 1, no. 3, pp. 215–239, 1978. doi: 10.1016/0378-8733(78)90021-7.

[39] I. Lera, C. Guerrero, and C. Juiz, "Comparing centrality indices for network usage optimization of data placement policies in fog devices," in *Proc. 3rd Int. Conf. Fog Mobile Edge Comput. (FMEC)*, Apr. 2018, vol. 1, no. 1, pp. 115–122.

ISAAC LERA received the Ph.D. degree in computer engineering from Balearic Islands University, in 2012. He is currently an Assistant Professor of computer architecture and technology with the Computer Science Department, University of the Balearic Islands. He has authored several journals and international conferences. His research interests include semantic web, open data, system performance, educational innovation, and human mobility.

CARLOS GUERRERO received the Ph.D. degree in computer engineering from Balearic Islands University, in 2012. He is currently an Assistant Professor of computer architecture and technology with the Computer Science Department, University of the Balearic Islands. He has authored around 40 papers in different international conferences and journals. His research interests include web performance, performance optimization, resource management, web engineering, and cloud computing.

CARLOS JUIZ (M'02–SM'12) received the Ph.D. degree in computer engineering from Balearic Islands University, in 2001. He is currently a Full Professor of computer architecture and technology with the Computer Science Department, University of the Balearic Islands. He has authored around 150 papers in different international conferences and journals. His research interests include performance engineering, green IT, and IT governance.

● ● ●