Microservices Scheduling Model Over Heterogeneous Cloud-Edge Environments As Support for IoT Applications

Ion-Dorinel Filip, Florin Pop, *Senior Member, IEEE*, Cristina Serbanescu, and Chang Choi, *Senior Member, IEEE*

***Abstract*—Motivated by the high-interest in increasing the utilization of nongeneral purpose devices in reaching computational objectives with a reduced cost, we propose a new model for scheduling microservices over heterogeneous cloud-edge environments. Our model uses a particular mathematical formulation for describing an architecture that includes heterogeneous machines that can handle different microservices. Since any new model asks for an early risk-analysis of the solution, we improved the CloudSim simulation framework to be suitable for an experiment that includes that kind of systems. In this paper, we discuss two examples of real-life utilizations of our proposed scheduling architecture. For an objective appreciation of the first example, we also include some experimental results based on the developed simulation tool. As a result of our interpretation of the experimental results we find out that some very simple scheduling algorithms may outperform some others in given situations that are frequently present in cloud-edge environments when we are using a microservice-oriented approach.**

***Index Terms*—Cloud computing, edge computing, energy efficiency, heterogeneous systems, microservice scheduling.**

I. Introduction

INTRODUCING technology in multiple fields of activity, our nowadays applications present a large variety and the underlaying systems become more and more complex. In fact, the growth of complexity is given by the requirements of having services that are always available on any kind of device (including smart phones and tablets), without the need of having a specialized device for each possible task. For supporting this great complexity, our current systems are more powerful, heterogeneous, and reliable. Many solutions already exist, but they are quite expensive and the main objective is reducing the cost of their utilization, considering both maintenance and efficient usage [1], [2]. Many of those resources are not conventional anymore and they can be found spread all around us. Therefore, sometimes a good load-balancing means to establish a great mobility of data by transferring it from the point that stores the data to a point that is more available for processing the task. The goal of reducing the communication cost leads to avoiding the need of mobility by processing the data exactly where it is [3].

An important issue in having multiple spread resources and using very heterogeneous system is reaching *isolation* which means that each independent task could benefit of its own dedicated environment by containing itself inside a more complex/open system. Such a protected runtime environment helps us reach a better security and trust level between the user-agent and the processing unit and emulates some of the benefits of the predictability (e.g., dedicated resources and known execution time) which are usually present inside of a homogeneous system along heterogeneous architectures.

Even though having almost any service available on any machine is solved by SDN, we cannot ignore that using them for all of our purposes may be inefficient.This introduces two more issues, *scalability* and *provisioning*. Solving the issue of *scalability* means to adapt the amount of deployed resources to the current workload. Provisioning is taking care of resource preparation (containing) and management (indexing). That last property mainly assess the costs behind the application of SDN that helps us use each resource for multiple proposes by preparing them for each task.

Power consumption is very important in considering the maintenance costs for a datacenter since it impose both cost of the consumed energy and cost of cooling the environment. If we consider the usage of embedded devices, this issue is even more critical since their autonomy could be affected by a misguided load-balancing.

In a nanodatacenter (NaDa), when the power limits are critical (enough energy may not be available for the processing), this problem is often solved by offloading tasks to a another processing unit. That remote processing unit can be a traditional one (hosted in a datacenter) or another NaDa device. Having such a complex processing, we aim to encapsulate the smallest meaningful function to a microservice (FaaS) [4]. By

Manuscript received December 18, 2017; accepted January 9, 2018. Date of publication January 12, 2018; date of current version August 9, 2018. This work was supported in part by the MONROE—Toff Project (H2020) under Grant 644399, in part by the NETIO-ForestMon under Grant 53/05.09.2016, SMIS2014+105976, in part by the SPERO under Grant PN-III-P2-2.1-SOL-2016-03-0046, 3Sol/2017, in part by the ROBIN under Grant PN-III-P1-1.2-PCCDI-2017-0734, and in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2017R1A6A1A03015496). *(Corresponding author: Florin Pop.)*

I.-D. Filip is with the Computer Science Department, University Politehnica of Bucharest, 061071 Bucharest, Romania.

F. Pop is with the Computer Science Department, University Politehnica of Bucharest, 061071 Bucharest, Romania, and also with the National Institute for Research and Development in Informatics, Bucharest, Romania (e-mail: florin.pop@cs.pub.ro).

C. Serbanescu is with the Department of Mathematical Methods and Models, University Politehnica of Bucharest, 061071 Bucharest, Romania.

C. Choi is with the IT Research Institute, Chosun University, Gwangju 61452, South Korea.

Digital Object Identifier 10.1109/JIOT.2018.2792940

doing that, we obtain an atomic task that is really easy to scale, contain, and isolate. The execution of a well-known small function can be considered more predictable from the point of view of resource utilization (including the communication ones) in different areas [5]–[7].

In this context, we propose a solution that considers a finite catalog of primitive microservices that could be used for the composition of tasks. For this purpose, we design a hybrid scheduling algorithm that considers a previous analysis of each task, makes an assess of available resources and matches them in a more appropriate way, aiming for a balanced distribution of tasks. Another optimized criteria is the cost of the execution. The impact of our solution is shown in two real-life scenarios.

The first scenario refers to processing an audio stream for the detection of unauthorized interventions. That would consist both in processing a big amount of data (audio stream—a signal for human presence detection) and on-demand raw data storage and delivery. The second scenario refers to solving the problem of scheduling urgent tasks in a Internet of Things (IoT)/smart city environment. We consider this problem with a special assumption of using microservices that can be executed faster by some involved devices. This kind of processing (with limited power resources) introduces the concept of *energy-aware processing* that differentiates itself from *energy efficient processing* by the fact it does not aim to reach the most power efficient configuration, but a performance optimizing one while taking into consideration the power-related costs [8]. Previous results show that choosing the right scheduling algorithm could provide a big overall performance boost for both real-time and batch-processing systems [9]. Many NaDa deployments are currently available, but underused by both business and home users arguing that there are not enough models for a good usage of those resources. We contribute to both a scientific goal of describing a new model and an economical goal by emphasizing the importance of improving the usage of less-noticed resources.

The main contribution of this paper is the design and implementation of a model for hybrid cloud scheduling. We discuss both scheduling and cloud-edge/NaDa considerations around a microservice oriented platform for IoT energy efficient scenarios. We design our solution to improve the usage of less-generic devices in computational proposes. We evaluate the impact on a few given real-time and batch-processing scenarios using simulated and theoretical results.

This paper is structured as follows. Section II presents the related work and highlights some already discussed solutions for our problem. In Section III, we introduce our model and describe the characteristics and limits for the considered architecture. In Section IV, we present two use-cases that prove the impact of our solution on a real-life economical context. Section V presents the simulation framework. In Section VI, we present the experimental results. The last section includes our conclusions and propose future work and improvements.

II. Related Work

The proposed solution uses a microservice oriented architecture. That is a recent concept and many may consider that splitting each existing application on microservices is an unjustified overhead, but there are many papers that show that a microservice-based software developing culture may also improve the time to market, covering a wide area of solutions for IoT and big data domains [10], [11].

Running each task in a dedicated environment was considered costly, because of the needed provisioning time (that was in order of minutes). This problem is no longer actual, since containers-oriented technologies are available, being able to use microservices and visualization technologies together [12].

For most of the applications, the distribution in time of the workload is highly unequal. The need to cover the peak-load of each application stands for the fact that our datacenters are prone to over-provision, while many other computational-enabled devices are underused, in special when they are placed outside of a datacenter. NaDa can be implemented on servers hosted on a network of set-top-boxes administrated by an Internet service provider (ISP) [13]. The communication model is peer-to-peer, but all the interconnected devices are coordinated and managed by the ISP that also coordinate the installation and configuration of them. In the evaluation of this model multiple aspects like service proximity, self-scalability, and energy efficiency are used. The proposed scheduling optimization method consists in a new content partitioning method called "hot-warm-cold." That new method clusters the content by user interest level and decides where they will be stored. They applied this model to evaluate the energy consumption for a video on demand platform. The results show an energy-saving of 15%–30% for the most pessimistic scenario and up to 60% for the most optimistic.

Another similar solution proposes an architecture for solving the problem of live video streaming hosting (based on ISP-managed set-top-boxes), but it considers other objectives in the evaluation of the proposed models, including reducing the delay and the optimization of the throughput. They offer multiple evaluations over a big number of scheduling and content placing algorithms [14].

A more complete compendium of problems regarding the greening of heterogeneous datacenters take into account many aspects of current cloud implementations and propose a solution for greening the computational environment using a cloud broker over multiple datacenters [15]. A problem that takes into account greening 7.8 billions tons of CO_2 should concern almost any entity in the environment (cloud providers, software developers, and users) and many aspects in application design and cloud scheduling are still open for optimization.

Van den Bossche *et al.* [16] considered the problem of load-balancing tasks between two types of resources (public and private) if one of them is infinite, but more expensive than the other. The main objective is to find the cheapest offloading for the private.

Defining a job as a implementation-independent entity may lead some to argue about an application related limit of scalability for each component. The properties of microservices that connect those limits back to the workload are presented in [17]. Let us consider a data-flow that consists of two steps. If each of these steps is independent and the second depends on the result of the first, then an user-agent will not be able

to define the second job before the finish time of the first one which means that a incompatible request would be excluded from any valid work load.

The asymptotic behavior of different scheduling policies and algorithms is presented in [18] by proposing a hybrid algorithm that switches from a simple policy to another based on the workload-size. The threshold of the switch is mathematically determined and it proves that certain scheduling policies are more suitable for high-load while some simple algorithms may outperform others in certain conditions.

The security of a very popular solution for running microservices, called Docker, and offers an overview on how it impacts the security of deployed application is presented in [19].

Regarding the subject of finding the real network parameters across multiple heterogeneous communication networks, a solution presented in [20] aims to find an ideal sample size for the empirical determination of the real network throughput. They consider the problem of transferring a file over multiple networks with different bandwidth-delay product characteristics and determine that a sampling size of 10%–23% of the file-size is good for the estimation of transfer throughputs. This result is very important for most of nowadays cloud implementations that involve best-effort communication networks.

III. Proposed Model

A. Formal Description of the Model

Our model has two types of entities: 1) processing elements (PEs) and 2) jobs. The minimal abstraction of a microservice consist in a function and a given amount of atomic steps that should be done for its execution. We describe a set of microservices as $M = \{ m_i \mid m_i = (m_{\text{id}}, m_{\text{length}}) \}$, where m_{id} is a identification of the microservice and m_{length} is the amount of instructions/atomic steps that it takes to execute the microservice. Considering that we know the number of steps that a machine can execute in a given period of time, that can be used to compute the processing time needed to complete the microservice on a given machine.

Having our current atomic workload definition, we can define the two types of entities that are present in our problem. A PE can be described as a device that is able to process a specific subset of microservices/functions with a given performance. A set of processing units can be described as $P = \{ p_i \mid p_i = C_i \}$, where each C_i represents the set of capabilities that is associated with the ith PE and $\mathbf{P}$ is a finite set with a given cardinality.

Each capability in the C_i set states that the ith PE can process a given microservice (c_{id}) with a given speed (c_{speed}) and it can be formally described as a two-sized tuple: $C_i = (c_{\text{id}}, c_{\text{speed}})$, where c_{id} is a natural number that identifies a given microservice and c_{speed} is the amount of instructions/atomic steps that the ith PE can process in a unit of time when executing the m_{id} microservice, where m_{id} is the same as c_{id}.

A job can be described as a finite amount of instructions that should be executed over a given input data to provide another amount of output. In our definition, we use a sequence of microservices for the representation of the processing part. Since our problem considers an online and real-time environment, two other time related parameters are included. For our propose, a set of jobs can be described as $J = \{ j \mid j = (t_s, D_{\text{in}}, D_{\text{out}}, F, t_d) \}$, where t_s is the time-stamp of job submission/the very first moment of time when the job is defined/known; D_{in} is the amount of data the job needs to load before starting the processing; D_{out} is the amount of data the job generates as a result of the processing; and t_d is the deadline for the given job. This is expressed as a maximal wall-time between job submission (t_s) and job completion event; F is a array of microservices. Considering that we know the number of steps that a machine can execute in a given period of time, we can compute the processing time needed to complete the job on a given machine.

For an I/O aware model we have to introduce a few more elements that specify where input data is stored, where output has to be placed and what are the costs related to transferring each jobs from a PE to another.

$\mathbf{B}$ (bandwidth) and $\mathbf{L}$ (latency) are $\overline{P} \times \overline{P}$ matrices in which each element on the row i and column j represents: 1) the amount of data that can be transferred from the ith PE to the jth PE in a unit of time (for the matrix $\mathbf{B}$) and 2) the delay which is considered for a communication from the ith PE to the jth PE, expressed in units of time (for matrix $\mathbf{L}$).

For a simplification of the operations that specify different machine-related parameters of the jobs, we define a special consideration over the diagonal elements of those two arrays (that specifies the cost of transferring a job from a PE to itself). We consider that this kind of transfer should be instantaneous, so diagonal elements ($b_{i,i}$ and $l_{i,i}$) have the following values: $b_{i,i} = \infty$ and $l_{i,i} = 0$.

$\mathbf{S}$ (source) and $\mathbf{D}$ (destination) arrays are two $\overline{J}$-sized arrays in each ith element is storing: 1) the identification of the PE that initially stores the input data for the ith job (for array $\mathbf{S}$) and 2) the identification of the PE that should receive/store the output for the ith job (for array $\mathbf{D}$).

Please note that each symbol has an unique appearance in the given mathematical notations. In any following equation, we consider a single set of $\{ J, P, B, L, S, D \}$ parameters that describe a complete model and we use the notation $Symbol^{(i)}$ to refer the ith element of the given set/array. This notation also applies recursively. For example $C^{(i,k)}$ is used for the reference to the kth capability of the ith job in the J set.

For any reference to the element on the ith row and jth column of a bi-dimensional array, we use one of the following notations: 1) $H^{(i,j)}$ or 2) $h_{i,j}$.

Let us define an operator that states if a given job $j_i \in J$ can be scheduled on a given PE $p_j \in P$

$$\text{Comp}(j_i, p_j) = \forall m_{\text{id}} \in F^{(i)}, \exists c_{\text{id}} \in C^{(j)} \text{ as } m_{\text{id}} = c_{\text{id}}. \quad (1)$$

Now we can define an arbitrary scheduling for a job set J over a PEs set P as the following:

$$S = \{ s_i | s_i = (j_k, p_l), j_k \in J, p_l \in P \text{ and } \text{Comp}(j_k, p_l) \text{ is true} \} \quad (2)$$

where each $j_k \in J$ appears once and only once.

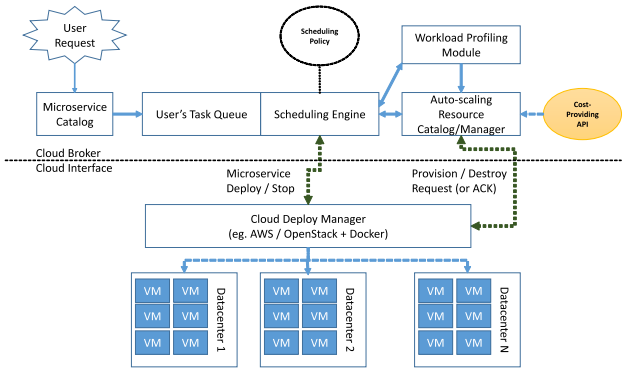

Fig. 1. Generic real-time scheduling system.

B. Architecture

Task → Resource allocation is the most important part of a scheduling system, but some other components are also important.

Fig. 1 describes the architecture behind our proposed model. We split this complex schema in two big parts: 1) *cloud broker* and 2) *cloud interface*. On the top part of the schema, we find the modules that compose the *cloud broker*. Those modules receive the jobs from the user-agent then apply needed scheduling algorithms and policies to take both scaling and scheduling decisions.

Cloud interface is an abstraction of the system that implements the actions asked by the cloud broker, called *cloud deploy manager*. In this second part, we also include the resources that consist in datacenters. Some of them are conventional cloud datacenters and others are just subsets of NaDa/embedded devices.

In our architecture, the user-agents ask for the execution of a job by looking up the *microservice catalog* and defining a job as a composition of given primitives. The defined job is immediately added to the user's task queue. From this point, the scheduling engine becomes aware of the jobs and it decides, based on some scheduling policies and algorithms (that also combine workload profiling and resource availability and cost meters), when and where the job should be executed. The *schedule engine* represents the main link between the cloud broker and the cloud deploy interface. Another important connection between cloud broker and cloud interface comes from the *auto-scaling deploy manager* that keeps track of available resources and decides the provisioning level. It is the component that devises the provision to cloud deploy manager. In our architecture there is no action flow to cancel a microservice. That comes from the assumption that each microservice represents a minimal/almost-atomic job and it cannot be preempted or divided.

A *datacenter* represents a resource availability zone, a subset of available resources that comes with a given cost table. Also. each datacenter assures or negotiates with the cloud deploy manager a service level agreement (SLA). We consider that SLA matches the user requirements and that is why we do not have an exception flow in job execution.

A virtual machine (VM) represents a unit of resource allocation and serves as an isolation tool by offering a specific environment for the job execution.

C. Architecture Components

In this section, we widely describe each module in our architecture. The order in which we choose to discuss them comes from the flow that is described in the previous section.

The *user-request* represents the action of adding a job. Each request is defined as a composition of microservices that should be executed for a given input to obtain a result. We consider that jobs are online (each of them gets reviled at a specific time) and real-time (deadline-constrained) tasks.

A microservice represents an atomic execution unit of a job. We can describe it as simple function that should be applied on a given input.

The *microservice catalog* contains a list of microservices that can be used for the composition of jobs. Each microservice definition includes a function, an input and output schema and a execution profiling for it (including required CPU, I/O, bandwidth, and memory). This execution profiling set should be usable in the way to determinate time and resources that are needed to execute the microservice on a specific machine that is located in a datacenter/NaDa. A execution profiling set is described by the elements that are included in our mathematical model (in Section III-A).

The *user task queue* represents a buffer of job definitions that are served to the scheduling engine. For concurrency matters, we consider that there is a queue for each user, but the scheduling engine is free to consider them as a single collection. Please notice that each of those collections is dynamically defined by the considered workload and the scheduling engine should be aware of any change. That is why the *user task queue* and the *scheduling engine* come as a strong-connected tuple of modules.

The *scheduling engine* is one of the most important modules in our architecture. The main responsibility of it is to apply some given scheduling policy in the way to provide a set of *Job* → *Machine* allocation to satisfy our scheduling problem. As shown, this module communicate the output of the scheduling algorithm to the cloud deploy manager, but it also changes information with workload profiling module and auto-scaling resource manager for both optimization and operational matters. The dataset of changed information may include parts of both scheduling algorithm input and output.

The *workload profiling module* is a software component that analyzes the characteristics of the workload and provide some optimization hints/techniques to both scheduling engine and auto-scaling resource manager. This module is very important in our architecture since it enables us to take most of the predictability behind a microservice oriented system.

The *scheduling policy* is a pluggable part of the cloud broker that defines the set of scheduling policies and algorithms that is used for problem solving.

Since we consider a very heterogeneous environment, that includes specialized processing units, an important step that comes before applying the scheduling solution and it is also handled by the scheduling engine is matching the job to a subset of machines that could be used for running a particular job. This match can be a simple selection algorithm and it can be made disregarding any cost-related matters.

Even though we represented a single scheduling policy in our graphical representation, multiple policies, and algorithms could be used at different moments of time. This choice can be influenced by both hybrid implementation and hints coming from the workload profiling module.

The *auto-scaling resource catalog/manager* is the module that keeps track of provisioned resources/machines and assures a minimal provision level for solving our scheduling problem (if it is possible). This component deals with the aspect of scalability that is critical in a real-time (deadline-constrained) environment. He uses a *cost providing API* to adapt in order to minimize the cost for the provisioned resources (for example if the required resources are available in different availability zones with different costs).

The most important part inside of the cloud interface is the *cloud deploy manager* that assures the deploy and execution of our tasks on given resources. Two important issues (*provisioning* and *containing*) are handled by this module that also handles the communication between the cloud broker and the job execution units. Some examples of technologies that could be found behind such a cloud manager are included on the schema. They can be divided in provisioning (e.g., AWS or OpenStack) and containing responsible (like Docker). The relation between those two kinds of technology consists in a resource provisioning versus abstraction/preparation linkage.

A *datacenter* is as a subset of resources that can be reached/used with a given cost (per resource unit). Such a datacenter is composed of multiple machines that can be used to run the jobs. We consider that our datacenters are fault-tolerant systems and that the penalty behind this tolerance is already included in resource related costs. Inside of a datacenter, we find multiple hosts, that provide a set of VMs. Such a VM represents an abstraction of resource allocation and it offers the isolation of job execution. Each machine capabilities includes the ability to execute a certain set of microservices. Those VMs are managed by the cloud manager and they are deployed or destroyed by the request of the auto-scaling module. The limits of this provisioning are contained in each characteristics.

D. Proposed Algorithms

In this section, we describe three scheduling algorithms that are used to extract our experimental results. One of the most popular policy for online job scheduling is using a first come first served (FCFS) scheduling algorithm over a round-Rubin (modulo-circular) machine allocation. For our purpose, the round Robin allocation policy should also take care of the fact that we cannot schedule a job on a PE that does not support all needed microservices. Having two or more partitions of PEs that can be used for some subsets of jobs, comes with an extension of the policy that should specify how we find the next suitable PE. In our implementation skip all the incompatible PEs, then resume the circular allocation policy, with the next index (see Algorithm 1).

We expect that such a simple algorithm to perform very well on a homogeneous system and we use it as a baseline for assessing another algorithm that is more likely to outperform in heterogenous environments.

Algorithm 1 Microservice Aware Round Robin Algorithm

```
Define a circular order of Processing Units
for all jobs j ∈ J do
    schedule j → i such that i is the next PE in the circular
    order that can process all microservices in j
end for
```

Algorithm 2 Classical Min–Min Algorithm (C-MM)

```
while J̄ > 0 do
  for all j_i ∈ J do
    for all p_j ∈ P do
      Compute ET_ij as the estimated execution time of the
      j_i on p_j;
      Compute r_j as the amount of time until the p_j will
      be waiting for a new job;
      Find a pair (i, j) s.t. ET_ij + r_j is minimum;
    end for
  end for
  schedule j → i;
  remove j_i from J;
end while
```

Algorithm 3 Multiheuristic Min–Min Algorithm (MH-MM)

```
while J̄ > 0 do
  for all j_i ∈ J do
    for all p_j ∈ P do
      Compute ET_ij as the estimated execution time of the
      j_i on p_j
      Compute r_j as the amount of time until the p_j will
      be waiting for a new job
      find a pair (i,j) s.t. S_ij = ET_ij + r_j is minimum OR
      find (i_2, j_2) s.t. abs(S_{i_2 j_2} − S_ij) < ϵ AND Micro(p_{j_2})
      < Micro(p_j)
    end for
  end for
  schedule j → i
  remove j_i from J
end while
```

As we described in Section IV-B, the optimization of many real-time processes leads to matching each task to the PE that can finish the job earliest (see Algorithm 2).

We expect that algorithm to perform better when considering an heterogeneous workload. We notice the pseudo-code does not take into consideration incompatible pairs, due to microservice incompatibility. We assume that $E_{i,j}$ will be infinite for a job-machine pair in which the machine cannot execute the job.

For taking into account the different capabilities of processing units, we propose Section III-D that makes an alternative choice for almost-equal values of the minimized sum. More specific, it tunes Section III-D such that if two processing units can offer almost-equal costs for a specific job, the one with a smaller number of capabilities to be chosen (see Algorithm 3).

Please note that the algorithm schedules a single job on each execution of the outer-most loop so we should be scheduling

one of $j \to i$ or $j_2 \to i_2$. The right interpretation in this case is keeping the last found pair, based on a greedy approach.

IV. Examples of Real-Life Utilizations

A. Implementation of Array of Sensors for Reporting Unauthorized Human Intervention in Forest

In this section, we describe the utilization of the proposed model and architecture for the implementation of a system that could be used for the detection of unauthorized human intervention in a wide area (e.g., a forest).

Solving this problem for an in-house environment is simple: we install an array of microphones all over the area and connect them to a central processing unit that decomposes each audio stream (Fourier analysis [21]) in order to detect the presence of a specific subset of components (in our case, anthropological sounds). As an energy aware solution we propose a NaDa system based on two types of entities.

1) Sensors that are low-power single board computers (SBCs) that are able to record and analyze audio streams. In the proposed architecture, they are not connected to the Internet, but they communicate with some NaDa concentrators through low range connection.
2) NaDa concentrators are local NaDa cloud brokers, that also implement the communication between the NaDa devices and the Internet. For offloading proposes, those devices also include signal processing capabilities.

In this system, each sensor generates audio samples of a given (Δt) before applying FFT Analysis. Since it would be very inefficient to send each answer to a *NaDa concentrator*, they store a given number of (p) results before sending a batch to the gateway. On the other side, the *NaDa concentrator* waits for a new dataset every $p\Delta t$ units of time and these are all the FFT processing tasks in $[(\tau-1)p\Delta t, p\Delta t]$ time interval which should be done by $p\tau\Delta t$.

The audio processing tasks can be described as

$$J_{\mathcal{F}} = \bigcup_{i=1}^{n}\bigcup_{\tau=1}^{t}\{(\tau\Delta t, D_{\text{in}}, D_{\text{out}}, (m_{\mathcal{F}}, s_{\mathcal{F}}), (1+p-\tau \bmod p)\Delta t)\} \tag{3}$$

where **n** number of sensors in the system; Δt length (in units of time) of an audio sample; **p** number of results in a single reporting batch; D_{in} size of a generated file size; D_{out} size of a single report; $m_{\mathcal{F}}$ identification of the FFT microservice; and $s_{\mathcal{F}}$ amount of instructions/atomic steps that it takes to analyze the audio sample.

Another type of jobs contains those that should be done by the *cloud concentrators*. Those are p times less frequent then FFT jobs and have the following mathematical formulation:

$$J_{\text{send}} = \bigcup_{i=1}^{m}\bigcup_{\tau=0}^{t/p}\{(p\tau\Delta t, D_{\text{in}}, D_{\text{out}}, (m_{\text{send}}, s_{\text{send}}), p\Delta t)\} \tag{4}$$

where **p**, **t**, and Δt same meaning as in (3); D_{in} size of the input data; D_{out} size of output data; m_{send} identification of the microservice that sends data to the cloud; s_{send} amount of instructions/atomic steps that it takes to complete the job; and **m** number of senders.

Equation (3) is only suitable for modeling the problem in an ideal environment. For a more realistic formulation we should consider both job completion delays and errors in the approximation of the microservice length ($s_{\mathcal{F}}$). That leads us the a new equation

$$J_{\mathcal{F}} = \bigcup_{i=1}^{n}\bigcup_{\tau=1}^{t}\{((1+\epsilon_1)\tau\Delta t, D_{\text{in}}, D_{\text{out}}, (m_{\mathcal{F}}, (1+\epsilon_2)s_{\mathcal{F}}), (1+p-\tau \bmod p)\Delta t)\} \tag{5}$$

where ϵ_1 and ϵ_2 are expressions of the error/heterogeneity. Such an extension of (4) is also necessary and leads to similar changes. We use ϵ_2 and ϵ_3 to refer to the parameters that are corespondent to ϵ_1 and ϵ_2 in (5). The total job set (J) is the union of the sets in (3) and (4): $J = J_{\mathcal{F}} \cup J_{\text{send}}$.

Each entity in this system can be described as a processing unit. We also define the set of processing units (P) as an union of two other sets (one for each kind of entity in our system)

$$P_{\text{sensors}} = \bigcup_{i=1}^{n}\{P_s | P_s^i = [(m_{\mathcal{F}}, s_{\mathcal{F}_1})]\} \tag{6}$$

$$P_{\text{coordinators}} = \bigcup_{i=1}^{m}\{P_c | P_c = [(m_{\mathcal{F}}, s_{\mathcal{F}_2}), (m_{\text{send}}, s_{\text{send}_2})]\} \tag{7}$$

where n number of sensors; m number of coordinators; $m_{\mathcal{F}}$ and m_{send} keep the meaning from (3) and (4); and $s_{\mathcal{F}_1}$, $s_{\mathcal{F}_2}$, and s_{send} are machine-related parameters that specify the speed of the PEs.

B. Scheduling Critical Processes in Smart Cities

Smart city is one of the most interesting usage of the IoT [22] concept that brings together a wide variety of computation-enabled devices (such as SBC sensors, smart TVs, and networking devices).

A system of such big complexity includes a wide spectrum of real-time processes, each having different associated levels of delay tolerance, based on their characteristics [23].

This kind of processing, that considers the problem of limited power resources, introduces the concept of *energy aware processing* that differentiate from *energy efficient processing* by the fact it do not aim to reach the most power efficient configuration, but a performance optimizing one while keeping the consideration of the power-related costs.

We can consider a task that is solvable by a minimum number of two machines in 20 min with the smallest cost, but our user needs the results in less than 60 s. In an energy efficient approach, the user would receive his results in 20 min, using the ideal power-related configuration. An energy aware system would take care of both energy and performance-related problems by finding the cheapest configuration that can execute the task in less than a minute.

We describe the solution for finding the fastest scheduling for an emergency level task that should be executed as soon as possible disregarding any energy-related consideration. We consider this solution for jobs described as a single microservice and that can be executed faster by some devices in the system (e.g., a smart TV can be really fast

in solving a task that uses video-processing algorithms, while a Firewall is specialized in analyzing TCP dumps for intrusion detection). Considering that any device becomes immediately available for this kind of tasks, the problems resumes to finding the device that is the fastest in processing our microservice. The function that finds the fastest PEs for executing a given microservice can be described as: $Find(m_{\text{id}}) = \{p_j \in P \mid m_{\text{speed}}^{(j,\text{id})} \textit{ is maximum}\}$, where m_{id} is the microservice describing the job and $m_{\text{speed}}^{(j,\text{id})}$ is the speed of running the microservice identified by id on the jth processing unit. This does not take into account any data transfer, coming from the assumption that no critical job should use a best-effort network.

V. Implementation Details

Considering the complexity of modeling the interaction between all the entities that should be included in the simulation, we decided to use a well-known framework as a starting point.

CloudSim is a cloud simulation framework maintained by a group within the School of Computing and Information Systems at the University of Melbourne, Melbourne, VIC, Australia [24].

1) CloudSim Simulation Work Flow: It takes nine steps to create a simple CloudSim simulation. We describe those steps by the following sequence of actions: Initialization of CloudSim package, create datacenter(s), create VM(s), create broker(s), create cloudlet(s), submit cloudlet(s) to broker(s), stop simulation, and print results. The first step represents the initialization of the simulation environment. At this step we specify the number of users and the time model that should be used. Steps 2–4 instantiate right models to describe our simulation environment. Steps 5 and 6 are defining the workload that each broker will handle.

2) CloudSim Networking Simulation Model: Modeling network topologies to connect simulated cloud computing entities is important since each latency on message passing can change the entire flow of the simulation. In CloudSim there are no specific models available to simulate network entities, like routers and switches, but it uses a conceptual network abstraction. In this model, each network interaction is simulated as a delay on passing a given message. This abstraction is presented to the user as a BRITE topology [25] that takes into account the bandwidth and delay of each directed network link, but gets simulated only as a message passing delay [24].

3) Experiment Description Serialization: As shown in (5), our simulation consists in a family of concrete scenarios that depends on some expressions of job execution predictability/workload heterogeneity. In this situation, taking advantage of the reproducibility inside simulated environments ask for a serialization of each concrete scenario. In this way, we implemented our own serialization toolkit that includes the following.

1) A Python script that can generate and serialize a simulation scenario.
2) A JSON schema for each type of described set of entities.
3) Java source code for importing a serialized experiment into the simulation.

The test generator is written as a Python package implementing a few classes for containing each simulation entity and a single entry-point that can be used for generating a JSON file that represents the serialization of an experiment. The experiments are mapped to the description in Section IV-A.

Many parameters (including simulation time, number of sensors/senders and the values of each microservice-related parameter) are given by static variables included of the class. The JSON schema consists of three parts: 1) job descriptions; 2) machine descriptions; and 3) simulation options.

4) Java Model Improvements: As we stated before, the CloudSim toolkit of models is not enough to describe each aspect of our simulation and we had to extend in many ways to fit our needs. CloudSim 3.0.3 offers no support for containers or microservices. We extended the definition of a cloudlet as a user-defined *DynamicCloudlet* that includes all the properties of a standard CloudLet, but also includes a list of supported microservice. That list is used by the cloud broker in matching suitable machines for our tasks. On the other side, each VM has to specify a set of supported microservices and microservice-related capabilities and we extended it to a Java class called *MicroserviceVm*.

5) Algorithm Implementation: We implemented all algorithms presented in Section III-D. Each algorithm is described as a pluggable part of the a DynamicWorkloadBroker, that is an improved CloudBroker class. Each of these algorithms also has to take account of our microservice-related restrictions.

6) Adding Online-Job Support: The method for submitting cloudlets in CloudSim environment is only available before the start of the simulation, witch does not fit our online-jobs. In order to achieve our simulation goals, we implemented a brand-new cloud broker (called DynamicWorkloadBroker) that can receive new cloudlets in any moment of the simulation. Moreover, we implemented a message-passing flow that allows us to collectively threat each set of simultaneous sent cloudlets.

7) Extracting Simulation Results: Our simulation results consist in two categories of disposables: 1) reports and 2) plots. For our propose, we extended the default set of metrics and exported some of them as CSV files. Plots are generated using GNU Plot.

VI. Experimental Results

A. Experimental Setup and Configuration

In this chapter, we study the behavior of the system that we described in Section IV-A, based on a simulated environment that we implemented as an extension of CloudSim. We make this simulation in order to asses the proposed algorithms against different metrics. For this propose, we use a simplified model of the system that is described in this section. CloudSim uses a combination of parallel processes and algebraic equations to simulate a contiguous time environment, but the discretization of our simulation scenario is almost trivial by using the notations in (5)–(7).

We choose a particular scenario that includes a number of 100 sensors connected to five NaDa concentrators. The size

TABLE I
SIMULATION PARAMETERS

Parameter	Value	Interpretation
n	100	Number of sensors
m	5	Number of senders
p	5	Number of samples in a transmission
Δt	1 second	Each audio sample has a unit of time (only important for plotting)
$s_{\mathcal{F}}$	$\Delta t s_{\mathcal{F}_1}$	A sensor process a FFT job in Δt time
s_{send}	$0.8 \cdot s_{s_{send_1}}$	Each sender can process a send job in 80% of a transmission period

TABLE II
SIMULATED WORKLOAD DESCRIPTION

Parameter	Value	Interpretation
ϵ_1	0	No delays in recoding part of the scenario
ϵ_2	random between - 0.2 and 0.2	Small heterogeneity of FFT work load
ϵ_3	0	Fixed periods of data transmission
ϵ_4	random between - 0.2 and 0.2	Small heterogeneity of FFT work load

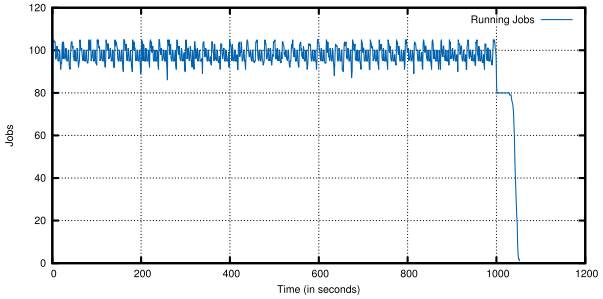

Fig. 2. Time plot of running jobs for FCFS algorithm.

of our simulation is arbitrary chosen, but the ratio between the number of sensors and the number of concentrators comes from our observations around the complexity of designing such embedded devices that usually connect a maximum number of 20 sensors/actuators to an SBC.

We choose the base size of an FFT task ($S_{\mathcal{F}}$) such that each sensor to be powerful enough to complete a single task in a period of Δt. We make that assumption based on the fact that the array of sensors should be energy efficient so that they implement a dynamic voltage scaling approach.

Regarding the parameters of each NaDa concentrator we choose that each of them is p times more powerful than a sensor and each NaDa concentrator can finish a task in about 80% of $p\Delta t$. This assumption is the best available guess based on our experience in using different cloud infrastructures. A more fine grained choice is included in our future work, by making an objective analysis on a real-life workload.

We choose the sending period to be five times the period of recording and analyzing an audio sample. That would place our transmission sample at 20% of the dataset which is very likely to be optimal for network transmission as shown in [20]. As already described, our simulation includes two types of jobs and two types of PEs. The first type of PE is only able to process the first type of microservices, while the other is more general. Table I specifies some simulation parameters while some others are specified in Table II.

In our framework the simulation time is seen contiguous, but the events are driven by each Δt interval that describes a step of simulation from the point of view of the CloudBroker. Such a step is complete when a new set of jobs is processed by the CloudBroker and it devises a scheduling for all available jobs. Special Δt moments are those that are also $p\Delta t$, since they introduce both types of jobs into the CloudBroker's queue.

1) Machines, Buffers, and Jobs: Our goal is simulating a set of scheduling algorithms for splitting the jobs between available VMs. Of course, the execution of jobs is instantaneous, so each machine has to manage its own buffers. In our experiment, each PE has an unlimited buffer for job reception and each processing unit applies an FCFS algorithm for its already assigned jobs. We also assume that each processing unit can execute a single job at a moment. Merging those two assumptions comes to each received task being placed in a simple queue and processed when the designated PE is free. No workload related I/O is included in our simulation.

B. Performance Analysis

1) Simulation Performance: Since most real-time systems tend to change their behavior after a certain amount of time, we have to analyze the behavior of our system for a very long period of time. That is not a problem since we can take advantage of the simulated time. For example, simulating the FCFS algorithm for an experiment that includes *105 machines* and *simulates 101.000* jobs takes about *30 s* to execute on a mid-level notebook.

2) Fitness of Scheduling Algorithms: Two different aspects are critical in every real-time energy aware system. For the one we described we can identify a full-time routine (with no idles) and that is why our primary energy efficiency-related goal is finding a as much as possible balanced workload assignment over all the available systems. In traditional real-time systems, the most used scheduling algorithm is FCFS. This algorithm is best suited for homogeneous systems. Using the other two algorithms we aim to improve the throughput of the system, by taking advantages of the heterogeneous of the system (that takes in the fact that we can use NaDa concentrators in the way to offload our array of sensors). We analyze load-balancing level and deadline aware matters for each of those three algorithms.

3) Workload Description: We consider the analysis the behavior of the system in Section IV-A. As we described in (5), this scenario includes a bounded level of unpredictability that comes from the heterogeneity of the workload.

C. Analysis of Experimental Results

We obtain our reference experiment for a period of 1000 s of simulation. That workload includes 101.000 jobs, 100 sensors, and five concentrators. According to the split of tasks, 1000 are send jobs and they sum up a number of 10 451 867 MIPS with an average (per Job) of 103.48.

For a visual appreciation of the load that each algorithm produces, we plot the number of jobs that is running over time. Please refer to Figs. 2–4. The plots show the following.

1) None of those algorithm devises a scheduling that includes more then 105 running jobs at a certain moment

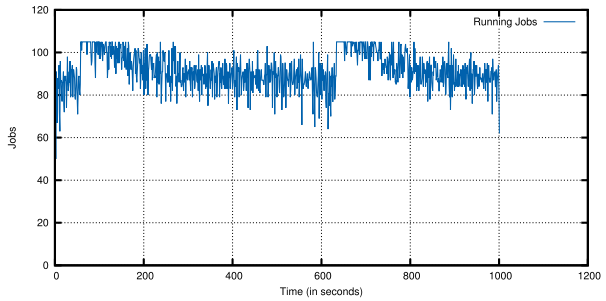

Fig. 3. Time plot of running jobs for C-MM algorithm.

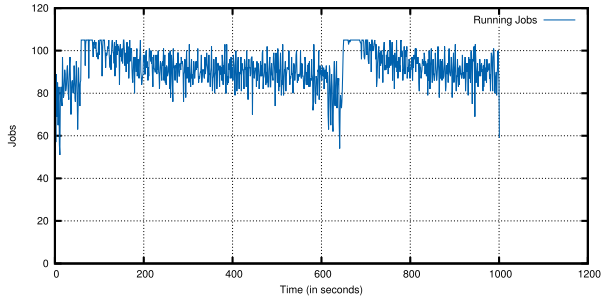

Fig. 4. Time plot of running jobs for our new version of min min.

TABLE III
LOAD BALANCING OF JOBS

N	Number of Jobs		101000	
P	Number of PEs		105	
M	Number of Jobs per PE in a perfect balancing (N/P)		$\sim$ 961,9	
x_i	No. of jobs run by i-th PE		Not a scalar	
	Algorithm	FCFS	C-MM	MH-MM
$\sum$	Standard Deviation of $x_i - M$	78,41	570,08	486,12
$1 - \sum /M$	**Load Balancing**	**0,91**	**0,40**	**0,49**

TABLE IV
VALUES OF JOB RELATED METRICS

	FCFS	C-MM	MH-MM
Completion Time	1053,13	1001,73	1001,69
Completion Time (FFT tasks)	996,46	998,76	998,92
Average Queue Waiting Time (FFT tasks)	17,27	0,86	0,84
Average Queue Waiting Time (SEND tasks)	0,29	2,51	2,56
Average Queue Waiting Time	17,1	0,88	0,86
Average Slack Time	4,10	4,74	4,74

of simulation. That limitation comes from the fact that we never run more then a job on a PE.

2) Every algorithm produces an almost maximum load for the biggest part of the simulation.
3) FCFS algorithm produces the best fit of maximum number of running jobs for most of the simulation time.
4) We can notice that for FCFS there is a tail on the last part of the plot. That comes from a pretty simple reason: scheduling each job at its release time does not take account of the heavier ones that may take longer. The "tail" comes from send jobs that takes longer then others.

We also notice that the graph for our new version of min min implementation has a slightly different pattern. In Table III, we present an evaluation of jobs load balancing based on the standard deviation of number of tasks/processing units (Σ). We

TABLE V
APPRECIATION OF JOB RELATED METRICS

	C-MM	MH-MM
Completion Time	-5.00%	-5.00%
Completion Time (FFT tasks)	0.23%	0.24%
Average Queue Waiting Time (FFT tasks)	-95.02%	-95.13%
Average Queue Waiting Time (SEND tasks)	+765.5%	+782.75%
Average Queue Waiting Time	-94,85%	-94,87%
Average Slack Time	14.47%	14.47%

notice that FCFS outperforms in terms of fitness of the most-equal load-balancing of the number of jobs. That is strong metric for homogeneous systems, but that may not determine the biggest throughput on a homogeneous system. For assessing this metric, we propose a comparison between some other statical values included in Table IV. Table V presents the information in Table IV using the FCFS algorithm as a baseline. Having all those metrics available, we can conclude our performance analysis by the following observations.

1) FCFS best fits the requirements of a homogeneous real-time system while a more heterogeneous environment may take great advantages from using a greedy (and also time-space polynomial) algorithm for task scheduling.
2) In our scenario, FCFS works very well due to the fact that all resources are properly scaled.
3) Increasing the matching of a subset of tasks to a subset of suitable machines (a formal description of our microservice-related binding) may optimize the load-balancing of a real-time/deadline-constrained system.
4) MM algorithms gives a better average queue waiting time which means a smaller job delay. We notice that, for send tasks, that metric is seven times bigger, but in evaluating that fact we should also consider that send jobs are five times less frequent and more heavy.

VII. CONCLUSION

In this paper, we made a summary of problems that are related to the utilization of microservices in heterogeneous systems and proposed an architecture for scheduling tasks using this kind of systems. The novelty of this new model comes by the microservice-oriented approach that enables the usage of less-general devices in computational proposes and extracts most of the predictability behind representing each task as a small and already-profiled function. In connection with this architecture we also developed a mathematical model for the description and formal analysis of those systems.

For a more practical evaluation of this kind of systems we realized an extension of an academical simulation tool called CloudSim and compared the performance of different scheduling algorithms while validating one of the proposed use-cases by some usability-related measured metrics.

As a future work we are going to: develop a more reusable toolkit for this kind of simulations by defining a better API for each of the new developed functions; extend the usage of our mathematical model for other scenarios and goal functions; create, simulate and evaluate some scenarios with more complex communication/processing patterns that may fit our scheduling model. In the IoT context, this paper has an added

value in giving an overview of how we can design and implement an architecture that helps us reach a better utilization of nongeneric devices for different computational purposes. It also improves the current knowledge base with a new mathematical model of describing this kind of systems that are spread all around us and many times underused.

REFERENCES

[1] G. Skourletopoulos, C. X. Mavromoustakis, G. Mastorakis, J. M. Batalla, and J. N. Sahalos, "An evaluation of cloud-based mobile services with limited capacity: A linear approach," *Soft Comput.*, vol. 21, no. 16, pp. 4523–4530, 2017.

[2] K. Karolewicz, A. Beben, J. M. Batalla, G. Mastorakis, and C. X. Mavromoustakis, "On efficient data storage service for IoT," *Int. J. Netw. Manag.*, vol. 27, no. 3, 2017, Art. no. e1932.

[3] C. X. Mavromoustakis, G. Mastorakis, and J. M. Batalla, *Internet of Things (IoT) in 5G Mobile Technologies*, vol. 8. Cham, Switzerland: Springer, 2016.

[4] M. Gribaudo, M. Iacono, and D. Manini, "Performance evaluation of replication policies in microservice based architectures," in *Proc. 9th Int. Workshop Pract. Appl. Stochastic Model.*, 2017.

[5] E. K. Markakis *et al.*, "EXEGESIS: Extreme edge resource harvesting for a virtualized fog environment," *IEEE Commun. Mag.*, vol. 55, no. 7, pp. 173–179, Jul. 2017.

[6] J. M. Batalla, G. Mastorakis, C. X. Mavromoustakis, and J. Zurek, "On cohabitating networking technologies with common wireless access for home automation system purposes," *IEEE Wireless Commun.*, vol. 23, no. 5, pp. 76–83, Oct. 2016.

[7] A. Bourdena, C. Mavromoustakis, G. Mastorakis, J. Rodrigues, and C. Dobre, "Using socio-spatial context in mobile cloud offload process for energy conservation in wireless devices," *IEEE Trans. Cloud Comput*, to be published.

[8] N. Bessis, S. Sotiriadis, F. Pop, and V. Cristea, "Optimizing the energy efficiency of message exchanging for service distribution in interoperable infrastructures," in *Proc. IEEE 4th Int. Conf. Intell. Netw. Collaborative Syst.*, 2012, pp. 105–112.

[9] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz, "Analysis and lessons from a publicly available Google cluster trace," EECS Dept., Univ. California at Berkeley, Berkeley, CA, USA, Rep. UCB/EECS-2010-95, vol. 94, 2010.

[10] I. Vakintis, S. Panagiotakis, G. Mastorakis, and C. X. Mavromoustakis, "Evaluation of a Web crowd-sensing IoT ecosystem providing big data analysis," in *Resource Management for Big Data Platforms*. Cham, Switzerland: Springer, 2016, pp. 461–488.

[11] J. M. Batalla, C. X. Mavromoustakis, G. Mastorakis, and K. Sienkiewicz, "On the track of 5G radio access network for IoT wireless spectrum sharing in device positioning applications," in *Internet of Things (IoT) in 5G Mobile Technologies*. Cham, Switzerland: Springer, 2016, pp. 25–35.

[12] M. Fazio *et al.*, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 81–88, Sep./Oct. 2016.

[13] V. Valancius, N. Laoutaris, L. Massoulié, C. Diot, and P. Rodriguez, "Greening the Internet with nano data centers," in *Proc. 5th Int. Conf. Emerg. Netw. Exp. Technol.*, 2009, pp. 37–48.

[14] J. He, A. Chaintreau, and C. Diot, "A performance evaluation of scalable live video streaming with nano data centers," *Comput. Netw.*, vol. 53, no. 2, pp. 153–167, 2009.

[15] S. K. Garg and R. Buyya, "Green cloud computing and environmental sustainability," *Harnessing Green IT: Principles and Practices*. Chichester, U.K.: Wiley, 2012, pp. 315–340.

[16] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove, "Cost-optimal scheduling in hybrid IaaS clouds for deadline constrained workloads," in *Proc. IEEE 3rd Int. Conf. Cloud Comput. (CLOUD)*, 2010, pp. 228–235.

[17] N. Dragoni *et al.*, "Microservices: How to make your application scale," in *Perspectives of System Informatics—PSI 2017* (LNCS 10742), A. Petrenko and A. Voronkov, Eds. Cham, Switzerland: Springer, 2018, pp. 95–104.

[18] A. Sfrent and F. Pop, "Asymptotic scheduling for many task computing in big data platforms," *Inf. Sci.*, vol. 319, pp. 71–91, Oct. 2015.

[19] T. Bui, *Analysis of Docker Security*, document T-110.5291, Aalto Univ. Seminar Netw. Security, Espoo, Finland, pp. 1–7, 2014.

[20] E. Yildirim, J. Kim, and T. Kosar, "Modeling throughput sampling size for a cloud-hosted data scheduling and optimization service," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1795–1807, 2013.

[21] X. Serra and J. Smith, "Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition," *Comput. Music J.*, vol. 14, no. 4, pp. 12–24, 1990. [Online]. Available: http://www.jstor.org/stable/3680788

[22] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, 2013.

[23] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of Things for smart cities," *IEEE Internet Things J.*, vol. 1, no. 1, pp. 22–32, Feb. 2014.

[24] R. N. 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. Exp.*, vol. 41, no. 1, pp. 23–50, 2011.

[25] A. Medina, A. Lakhina, I. Matta, and J. Byers, "BRITE: An approach to universal topology generation," in *Proc. IEEE 9th Int. Symp. Model. Anal. Simulat. Comput. Telecommun. Syst.*, 2001, pp. 346–353.

Ion-Dorinel Filip received the Engineering degree in computer science from the University Politehnica of Bucharest (after defending his diploma project on a subject related to cloud-edge scheduling), Bucharest, Romania, in 2017, where he is currently pursuing the M.Sc. degree in parallel and distributed computer systems.

Florin Pop (S'06–M'08–SM'17) received the Engineering, M.Sc., and Ph.D. degrees in computer science from the University Politehnica of Bucharest, Bucharest, Romania, in 2003, 2004, and 2008, respectively.

His current research interests include scheduling and resource management, multicriteria optimization methods, grid middleware tools and applications development, prediction methods, and self-organizing systems.

Cristina Serbanescu is an Assistant Professor (Lecturer) with the Department of Mathematical Methods and Models, University Politehnica of Bucharest, Bucharest, Romania. Her current research interests include hidden Markov chain and their applicability, and statistics and probability theory applied in computer science.

Chang Choi (M'16–SM'17) received the B.S., M.S., and Ph.D. degrees in computer engineering from Chosun University, Gwangju, South Korea, in 2005, 2007, and 2012, respectively.

He is currently a Research Professor with Chosun University. His current research interests include intelligent information processing, semantic Web, smart Internet of Things systems, and intelligent system security.

Dr. Choi was a recipient of the Academic Awards of the Graduate School, Chosun University, in 2012.