# An energy saving perspective for distributed environments : Deployment, scheduling and simulation with multidimensional entities for Software and Hardware

Hernan Humberto Alvarez Valera


# An energy saving perspective for distributed environments: Deployment, scheduling and simulation with multidimensional entities for Software and Hardware

**Hernan Humberto ALVAREZ VALERA**

Thèse de l'Université de Pau et Des Pays de l'Adour

## Jury de Thèse:

**Jean Marc MENAUD,** IMT Atlantique, Professeur, Rapporteur
**Romain ROUVOY**, Université de Lille, Professeur, Rapporteur
**Olivier BARAIS**, Université Rennes 1, Professeur, Examinateur
**Patricia STOLF**, Université Toulouse Jean-Jaurès, MC HdR, Examinatrice
**Marc DALMAU**, Université de Pau et des pays de l'Adour (UPPA), MC HdR, co-directeur de thèse
**Philippe ROOSE**, Université de Pau et des pays de l'Adour (UPPA), MC HdR, co-directeur de thèse

**Soutenance Prévue:** 27 juin 2022

# Abstract

Nowadays, strong economic growth and extreme weather conditions increased global electricity demand by more than 6% in 2021 after the COVID pandemic. The fast recovery regarding this demand rapidly increased electricity consumption. Even though renewable sources present a significant growth, electricity production from both coal and gas sources has reached a historical level.

On the other hand, the consumption of energy by the digital technology sector depends on its growth and its degree of energy efficiency. On this matter, although devices at all deployment levels are energy efficient today, their massive use means that global energy consumption continues to grow.

All these data show the need to use the energy of these devices wisely. For that reason, this thesis work addresses the dynamic (re)deployment of software components (containers or virtual machines) and their data to save energy. To this extent, we designed and developed intelligent distributed scheduling algorithms to decrease global power consumption while preserving the applications' quality of service.

Such algorithms execute migrations and duplications procedures considering the natural relation between hardware components' load/features and power consumption. For that, they implement a novel manner of decentralized negotiations based on a distributed middleware we created (Kaligreen) and multidimensional data structures.

To operate and assess the algorithms above, appropriate tools regarding hardware and software solutions are essential. Here, our choice was to develop our own simulation tool called: **PISCO**.

**PISCO** is a versatile and straightforward simulator that allows users to concentrate only on their scheduling strategies. It enables network topologies to be abstracted as data structures whose elements are devices indexed by one or more criteria. Additionally, it mimics the execution of microservices by allocating resources according to various scheduling heuristics.

We have used **PISCO** to implement, run and test our scheduling algorithms.

# Acknowledgments

*Tiempo: Origen de todos los tesoros que aparenta dejarse dominar cuando se presta en silencio a unos cuantos. Éste siempre es finito, por lo que se le debe valorar especialmente cuando es ajeno.*

En este documento habitan conocimientos encontrados y generados durante los últimos tres años. Me gustaría que quien emprendiese su lectura, comience por enterarse un poco acerca de quienes permitieron su creación.

**Marc Dalmau y Philippe Roose**, mis asesores, son personas excepcionales cuyas capacidades profesionales se demuestran solas en sus alumnos, sus publicaciones, sus libros y sus clases. Sin embargo, hay algo mucho más especial: Para enseñar, ellos SIEMPRE acompañan con cariño y respeto a sus estudiantes, aún en los momentos más álgidos de tribulación.

Yo no fui la excepción. En lo profesional, Philippe, además de enseñarme muchísimas cosas sobre arquitectura del software o green computing, siempre se preocupó porque esté capacitado y conquiste éxitos profesionales. Marc, quien entiende de verdad la computación (desde el funcionamiento del hardware, pasando por el comportamiento del compilador y terminando en la ejecución esperada) y es un maestro en abstracción, pasó muchísimas horas conmigo para revisar mis códigos, analizar mis ideas y concretar contribuciones.

Philippe y Marc son dos profesores que me regalaron parte de su tiempo para formarme con cariño, dos amigos que, con mucha paciencia (bastante), me entendieron y apoyaron como ser humano y finalmente, dos maestros que por sobre todo buscaron mi bienestar. Ambos para mi son un lujo y tienen mi respeto, admiración y amistad eterna.

Yo soy **Beto** (diminutivo estilado de "Humberto"), un informático que nació y se formó en el Perú. Vine a "la France" (hermoso país con el que me identifico también) teniendo la oportunidad de hacer mis estudios de postgrado en los temas que me gustan (Sistemas operativos, sistemas distribuidos, entre otros). Todo lo que soy y he podido hacer de bueno se lo debo a las personas que me edificaron a lo largo de mi vida. Aquí, quiero agradecer y dedicar este documento a **Ruth Valera Calderón**, mi mamá, constructora de mi niñez, mi amiga en la adversidad, la persona que siempre antepone mi bienestar al suyo…la persona a la que le debo todo. A **Lucía Valera**, mi tía, que junto con mi madre cuidó de mí y nunca permitió que me faltase nada. Gracias a ella y su pasión, entendí lo que significa "valorar". Hoy, ella es mi

ángel de la guarda. A mi papá, **Humberto Valera**, que desde siempre y hasta hoy (con 103 años) sigue siendo maestro de disciplina y gratitud. A **Sara Calderón**, por haberme permitido estar vivo hoy. A mi tío **Martín Valera**, que custodia mi casa y que trabaja muchísimo para que todos en ella estén bien. A **Tania Barrientos**, mi eterno amor que en la eternidad vive, estela diaria de mi mente, maestra que me enseñó que el amor genuino y espiritual existe. A **Dennis Barrios y Raquel Patinio**, maestros y amados amigos de la UCSP. A **Regina Ticona**, que me inició en el camino a Francia y que siempre me ayuda. A **Julio Labady**, mi mejor amigo, con quien he podido esclarecer todas mis dudas existenciales. A **Karito Machaca**, por haberme acompañado en momentos de depresión y ser inspiración de fortaleza en la adversidad. A **Jorge Andrés Larracoechea**, que además de ayudarme con parte de la implementación de la tesis, es un gran amigo. A **Amaia** y **Martita**, que me brindaron lindos momentos de amistad y siempre me apoyaron desde que llegué a Francia. A **José Manuel Negrete**, que siempre me ayudó en situaciones de emergencia. A amigos queridísimos como **Patrick Etcheverry**, compañero de largas tardes de trabajo (¡club de los bobitos!). A **Martin Walton**, gran amigo por quien mejoré mi nivel de inglés…En fin a todos mis grandes amigos a los que quiero mucho y agradezco. Si alguien no encontrase su nombre aquí, ha de saber que lo guardo en el corazón y mis oraciones.

…Gracias **Dios mío** por TODO y perdón por tantos errores. Tú, creador del tiempo, el cielo y la tierra, permíteme nunca olvidar que: **Al atardecer, seremos evaluados en el amor …**

*Oh María concebida sin pecado, ruega por nosotros que recurrimos a vos.*

# Contents

# List of Publications

## International Journals


[1] Hernan Humberto Alvarez-Valera, Philippe Roose, Marc Dalmau, Christina Herzog, and Kyle Respicio. **KaliGreen: A distributed Scheduler for Energy Saving**. *Procedia Computer Science* 141 (2018). The 9th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2018) / The 8th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2018) / Affiliated Workshops, 223–230. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2018.10.172. URL: https://www.sciencedirect.com/science/article/pii/S1877050918318222.


## International Conference Proceedings


[2] Hernan Humberto Alvarez Valera, Marc Dalmau, Philippe Roose, and Christina Herzog. **The Architecture of Kaligreen V2: A Middleware Aware of Hardware Opportunities to Save Energy**. In: *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. 2019, 79–86. DOI: 10.1109/IOTSMS48152.2019.8939237.

[3] Hernan Humberto Alvarez Valera, Marc Dalmau, Philippe Roose, Jorge Larracoechea, and Christina Herzog. **An Energy Saving Approach: Understanding Microservices as Multidimensional Entities in P2p Networks**. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. SAC '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, 69–78. ISBN: 9781450381048. DOI: 10.1145/3412841.3441888. URL: https://doi.org/10.1145/3412841.3441888.

[4] Hernan Humberto Alvarez Valera, Marc Dalmau, Philippe Roose, Jorge Larracoechea, and Christina Herzog. **DRACeo: A smart simulator to deploy energy saving methods in microservices based networks**. In: *2020 IEEE 29th International Conference on Enabling Technologies:*

*Infrastructure for Collaborative Enterprises (WETICE)*. 2020, 94–99. DOI: 10.1109/WETICE49692.2020.00026.

[5] Hernan Humberto Alvarez Valera, Marc Dalmau, Philippe Roose, Jorge Larracoechea, and Christina Herzog. **(DEMO) PISCO: A smart simulator to deploy energy saving methods in microservices based networks**. In: *2022 International Conference on Intelligent Environments (IE)*. 2022.

## Patents and Licenses

[6] Philippe Roose, Marc Dalmau, and Hernan Humberto Alvarez Valera. "PISCO: A smart simulator to deploy energy saving methods in a services/microservices based network." **Dépôt logiciel : IDDN.FR.001.300050.000.S.P.2021.000.10700** (Fance). 2021.

[7] Philippe Roose, Marc Dalmau, and Hernan Humberto Alvarez Valera. "Système et procédé de planification de traitement de programme." **Patent N°2107199** (Fance). 2021.

## National Conferences and Workshops

[8] Hernan Humberto Alvarez Valera, Marc Dalmau, Philippe Roose, Christina Herzog, and Jorge Andres Larracoechea Gonzalez. **A smart simulator to deploy energy saving methods in a services/microservices based network**. *COMPAS 2020*. Lyon, 2020.

[9] Hernan Humberto Alvarez Valera, Marc Dalmau, Philippe Roose, Christina Herzog, and Jorge Larracoechea. **DRACeo: A smart simulator to deploy energy saving methods in a services/microservices based network**. *NEGIS – Next Generation Information Systems: Modeling, Monitoring, and Management in Cloud and Fog Computing at Caise'2020*. Grenoble, 2020.

# CHAPTER 1

# INTRODUCTION

# 1 Introduction

Nowadays, strong economic growth and extreme weather conditions increased global electricity demand by more than 6% in 2021 after the COVID pandemic[99]. The fast recovery regarding this demand rapidly increased electricity consumption. Even though renewable sources present a significant growth, electricity production from both coal and gas sources has reached historic level[98].

Some difficulties concerning this subject have been pointed out. For instance, *MD Staff* argued that the emissions from electricity need to decline by 55% by 2030 to meet Zero-Emissions by 2050. However, in the absence of major policy action, those emissions are set to remain around the same level for the next three years(2022-2025). Among these significant changes, the world needs massive transformations for the electricity sector to fulfill its critical role in decarbonizing the broader energy system[165].

On the other hand, the consumption of energy by the digital technology sector depends on its growth and its degree of energy efficiency. On this matter, a division regarding three categories is highlighted[208]. First, personal devices (phones, computers, tablets) correspond to the primary source of energy use, with approximate calculations between 38%[224] and 50%[7]. Next are data centers (servers, data, computing units) with consumption values of 1.1-1.4%[97]. Finally, communication networks (wireless, data transport).

The afore-mentioned subdivisions had an annual increase surpassing the growth of worldwide electricity consumption during the period comprehending from 2007 to 2012: Personal devices 10%, data centers 5%, and communication networks 4%[224].

All these data show the need to use the energy of these devices wisely. For that reason, this thesis work addresses the dynamic (re)deployment of software components (containers or virtual machines) and their data to save energy. To this extent, we designed and developed intelligent distributed scheduling algorithms to decrease global power consumption while preserving the applications' quality of service.

Such algorithms execute migrations and duplications procedures considering the natural relation between hardware components' load/features and power consumption. For that, they implement a novel manner of decentralized negotiations

based on a distributed middleware we created (**Kaligreen**) and multidimensional data structures.

In order to operate and assess the algorithms above, appropriate tools regarding hardware and software solutions are essential. For example, infrastructures customized with specially designed middlewares (Kalimucho[51], Kubernetes[136], etc.), measurement devices (wattmeters or uninterruptible power supply (UPS)), among others.

Because these technologies were unavailable to us and the proven effectiveness of power formulas (explanation in chapter 2), we introduced our own simulation tool called: **PISCO**.

Regarding the formulas, we have studied, adapted existing approaches, and proposed (if concerned) our own mathematical models to describe the energy consumption of essential hardware components (**CPU**, **RAM**, **NIC**, and **storage** device). For the **CPU** and the **RAM**, we have selected and adapted already tested formulas to detail their workload, power state, and power consumption. Then, we have proposed and tested our own energy consumption models for the **storage** device and the **NIC**. It is relevant to indicate that each model's parameters' values can be collected from different **GNU/Linux** interfaces (such as APIs, specialized programs, and others).

**PISCO** allows the (re)deployment of software components (executing processes) as well as their connections in the network's nodes. Additionally, it calculates hardware components' energy consumption based on their workload over time extent. This load can be evaluated taking into account as many hardware components as defined.

The strength of our simulator consists of its versatility and simplicity. It enables network topologies to be abstracted as data structures whose elements are devices indexed by one or more criteria. Additionally, it mimics the execution of microservices by allocating resources according to various scheduling heuristics. For that reason, we have used our simulator to implement, run and test our scheduling algorithms.

## 1.1 Thesis Methodology

This thesis work seeks to create and execute distributed tools and algorithms to save energy while maintaining the notion of QoS. For this, its chapters describe our proposal of a way forward toward intelligent distributed scheduling.

The first step on this path is to have energy information usable by any distributed middleware and simulation tools. Chapter 2 studies this problem. In it, we provide

energy formulas for four important hardware components (CPU, RAM, NIC, and storage) to describe the consumption of running applications in heterogeneous devices.

Our next step is to understand how distributed approaches work. For this, Chapter 3 shows a different way of categorizing research works based on an algorithmic procedure. The strategy considers 1) important input variables such as hardware components load and devices positions, 2) scheduling operations as migrations or duplications, and 3) optimized variables such as QoS and energy consumption. Moreover, we study this flow in (de)centralized environments at different deployment levels, such as cloud, grid, or edge. That makes it easy to understand the scope of new scheduling proposals.

Then, having explored state of the art in distributed environments, Chapter 4 describes our scheduling strategies. We have designed a distributed middleware called **Kaligreen**. It is aware of the software and hardware components of the devices on which it runs. Furthermore, it implements special communication methods that enable centralized and decentralized strategies in any network topology. **Kaligreen** includes a default algorithm that performs neighborhood microservices exchanges after negotiation and microservices-filtering processes.

Using kaligreen, we have designed, implemented, and tested a fully decentralized algorithm inspired by multidimensional data structures, representing one of this thesis's major contributions. That considers the devices as nodes in a space with as many edges as the characteristics of said nodes. In this space, each device can find peers to negotiate microservice migrations or duplications at an optimal computational cost.

Next, Chapter 5 shows the design and implementation of **PISCO**, the simulator we created and use to deploy and evaluate our scheduling algorithms. The importance of our tool is to allow its users to focus solely on their scheduling heuristics. For this, **PISCO** implements an overlay as any data structure (graphs, trees, etc.), devices and software as dynamic objects, and middlewares as threads to execute any scheduling policies.

Finally, the thesis finishes by showing our methods' experiments and results in chapter 6, and its conclussions and future work in section 7.

Regarding the structure of each chapter, each begins by defining its introduction and context, next explaining the methodology that follows, showing its content, and finally arguing its conclusions and contributions.

# CHAPTER 2

# PROFILING APPS' POWER CONSUMPTION

# 2 Profiling applications' power consumption

## 2.1 Introduction

This research work studies how to dynamically (re)deploy software components (containers or virtual machines) and their data to save energy. To do this, we designed and developed intelligent distributed scheduling algorithms that **reduce global power consumption, maintaining the applications' quality of service**. They operate (migrations and duplications procedures) considering the natural relation between hardware components' load/features and power consumption.

In order to execute and evaluate these algorithms, we needed to use special hardware and software tools. For example, infrastructures configured with specialized middlewares (Kalimucho[51], Kubernetes[15], etc.), measurement devices (wattmeters or uninterruptible power supply (**UPS**)), among others.

However, since we did not have these technologies available and the effectiveness of power formulas has already been demonstrated[53][38] (even taking into account the hardware heterogeneity), we decided to implement and use our own simulation tool.

***PISCO*** is a highly dynamic simulator (explained in detail in chapter 4) for distributed scheduling. First, it allowed us to deploy heterogeneous nodes in any network topology. Secondly, it served us to define different hardware components for each physical node. Next, it found the component's power/energy consumption from their load and energy mathematical models. Finally, our tool executed and evaluated our distributed scheduling approaches.

From the software point of view, ***PISCO*** allows the (re)deployment of software components (for us, running processes) and their connections in the network's nodes. It also calculates said components' energy consumption based on their workload over time. This load can be analyzed considering as many hardware components as defined.

In this research work, we have chosen four components to energetically profile any running process (application, microservice, virtual machine, etc.): The **CPU** (Central Processing Unit), the **RAM** (Random access memory), the **network card**, and the **storage device**. We based this decision mainly 1) on the fact that for these components, it is possible to obtain the necessary profile values from **OS** (operating system) interfaces and 2) on the existence of works that analyze these components

to outline the energy consumption of a distributed system node. For example, while some works understand that the most important devices to analyze global power consumption are the **CPU** and **RAM**[120][14][155][16][1], others also take into account other components such as the hard drive, the ethernet card, or even the video card[16][153][238].

This chapter describes our distributed scheduling algorithms' input criteria. In the following paragraphs, we study each of these four components' importance, operation, and features. Moreover, we explain the mathematical models that relate the workload produced by a process identifiable by a **PID** to the respective power consumption.

## 2.2 Methodology

This chapter contains four main sections explaining the four components we study: **CPU**, **RAM**, **Network Card**, and **Storage device**. Each section explains first the technical hardware details regarding how the component uses power according to a given workload. Then, it describes how several authors study and optimize the components' power consumption. Here we have grouped the approaches into four levels: 1) low-level configuration, 2) software design and development, 3) launching applications, and 4) applications already running. This categorization allows us to identify the techniques and mathematical models compatible with distributed energy savings strategies. If a proposal is possible with the values obtained from **OS** interfaces, we use them for our algorithm implementation. Otherwise, we develop and use our own formulas.

It is important to say that we have made the differentiation of points 3 and 4 for two reasons. On the one hand, the applications usually contain different startup and post-start execution routines. On the other hand, many profiling tools need to launch the application to analyze it. This fact creates an undesirable dependency relation for production environments but interesting results for building a pre-execution profile.

We present the summary section at the end of the chapter, highlighting our contributions and procedures.

## 2.3 The CPU's energy consumption

The **CPU** (Central Processing Unit) is the hardware component that executes the instructions described in a program through logical/arithmetic operations (among

others). Depending on the **CPU's** structural characteristics, these instructions can be executed at different frequencies, using different parallelism and prediction criteria, and finally, using different amounts of energy.

Compared to the other components studied in this chapter, the **CPU** has the greatest structural variations (even within the same brand) due to technological advancement and the plurality of existing devices. Furthermore, each processor could operate at different voltages or activate different power-related capabilities for energy-saving purposes.

In order to understand how these mechanisms consume energy, it is necessary to study some basic aspects of the processor's operation, which we will describe in the following section.

### 2.3.1 Background

From an electronics point of view, the **CPU** comprises millions of silicon transistors that act as switches useful for generating bits from electricity. With these transistors, each processor can execute the instructions described in its **ISA** (Instruction Set Architecture) using several **cores**. On the one hand, each **core** has processing units such as a set of **ALUs** (Arithmetic Logic Unit) and **FPUs** (Float Point Unit). Moreover, they have other internal units such as one **MMU** (memory management unit), one **RS** (Register Set), a **CU** (Control Unit), and a set of **cache memories**. These last usually have three categories: 1) **L1**, which is the closest (and therefore more quickly accessible) to each core. 2) **L2**, which each pair of cores share, and 3) **L3**, which all cores can access[69][231][85].

Each time a compiler processes an application source code, it generates a set of instructions specific to the **CPU's ISA**. Then, employing internal or external (operating system ones) selection algorithms, one or more **CPU's** cores process these instructions. If the **CPU** implements **hyperthreading** technology (two data and instruction streams per core), the selection algorithms will also use the core's threads to improve parallel execution.

When executing an application, each **core** performs the concerning operations using its **CU** to synchronously utilize its necessary internal units. This work is performed at a **frequency** determined by the **clock's speed**, which is dependent on the **core's voltage**. Therefore, the more frequency, the more operations in a given time a core can process, and the more energy it uses[107].

However, the frequency and voltage are also directly related to the heat produced, which can damage the chip's internals. For this reason, each **CPU** has a **TDP** (Thermal Design Power) expressed in Watts, which defines the maximum power with which a processor can work safely for an indeterminate time[111].

It is important to mention that the **TDP** may be exceeded for a short amount of time to increase processing capacity in certain circumstances. That is known as **Overclocking**[107].

On the other hand, executing a program's instructions requires correct memory handling. To be executed, each of them must first be retrieved from **RAM** or the **CPU's caches** in order to be placed into the **CPU's instruction registers**. This process is known as **"fetch."** Then, through the **"decode"** process, the **CPU** discovers each instruction type. Finally, in a process called **"execute,"** the instruction's data is placed in the **registers** of the corresponding execution unit, such as the **ALU** for arithmetic, logical, and branch operations or the **MMU** for memory retrieval operations.

Depending on the type of instruction, there are very complex operations and strategies to be carried out. For instance, if the required instruction's data is not in the **core's** registers, more clock cycles are required to query the **RAM**[210]. To face that, the **cache controller** will try to predict (using special predictive algorithms) and **fetch** additional to-request data. On the other hand, statements such as **"for"** or **"if"** may involve executing a sequence of instructions different from the current one. These instructions are known as **branch operations**. As the mentioned statements' immediate results are unknown, **branch operations** can involve unnecessary processing instructions that use indeterminately different **core** processing units. This phenomenon makes parallelism difficult and the execution efficiency of an application not optimal. To address this, using complex structures called **branch predictors**, the **CPU** tries to predict whether a branch should be taken or not with about 95% accuracy on modern processors[85][21].

As it is easy to see, studying the **CPU's** power consumption, including parallelism and prediction criteria particular to each model, is an extremely complex task (more than for other components[53]). Each **CPU** model could also implement different forms of energy saving, such as voltage reduction or turning off certain parts of a particular **core**. However, several studies have managed to relate workloads with the energy consumption they produce in the **CPU**.

The next section describes these approaches.

### 2.3.2 Finding and optimizing the CPU's energy consumption

From the point of view of hardware evolution, **CPU** manufacturers have made processors work at high processing frequencies, using power more and more efficiently. For that, they have implemented two main techniques that seek to adapt the amount of energy invested with the workload type and the chip's temperature.

- The **DVFS** (Dynamic Voltage Frequency Scaling) enables the operating system to reach a certain **P-State** (Performance State). It lowers the **CPU's** frequency and voltage under certain circumstances to save energy at the cost of processing capacity[144][145][233][114].
- The **C-States** allow each **CPU's core** to turn off some of its parts as requested by the **OSPM** (Operating System Directed Power Management). These states establish the core's idle state-level defining the following relation: *The deeper the **C-state** is, the more energy is saved, but the higher is the exit latency to the active state*[105]. In figure 2.1, Intel shows the **C-states** of some of its processors[115] and the relation just mentioned. It is important to note that in order to obtain the best energy-saving results, all **(v)cores** that share resources such as caches should be affected by the same **C-state**.

Other works propose improvements in the design of the processing units to save energy. For example, some approaches take better advantage of the **turbo boost** in order to avoid **cache misses** and **branch mispredictions**[133]. For their part, other scientists intelligently use the **power gates** to turn off certain processor parts according to the workloads' characteristics[141][151].

We can use physical sensors, mathematical models, or system-level and processor simulators to discover the effectiveness of these approaches[18]. Although there are several creations of this type, we do not mention them as they are beyond the scope of this work.

Some other approaches seek to save energy from a higher level of abstraction by tuning the operating system scheduler. They propose techniques such as 1) modifying the run-queues to halt the **CPU** when power consumption reaches a certain threshold[16], 2) grouping tasks of the same type (characteristics, number of cycles in the **CPU** internals, etc.) for their execution to trigger the **DVFS** (using, for example, **GNU/LINUX ACPI/CPUFREQ**) or the **C-States**[120][41][30], 3) enabling the scheduler to predict the energy-impact of its decisions[130], or 4) enabling **CPU** and **GPU** to work as a single unit[123][211].

For these approaches, the power/energy consumption can be estimated eighter from 1) mathematical models that can, among others, take into account the number of events/cycles that a thread produces in the **CPU's** internals (**ALU**, **FPU**, or **caches**)[16][120], 2) from the battery usage in a certain period[41], or 3) from chip-specific measurement tools[123].

At the next level of abstraction, from the point of view of applications development and execution, some papers relate power consumption to source code procedures. This relation enable developers to perform different energy-aware coding strategies[91][42][172][2]. As in the other approaches, we can use APIs, physical

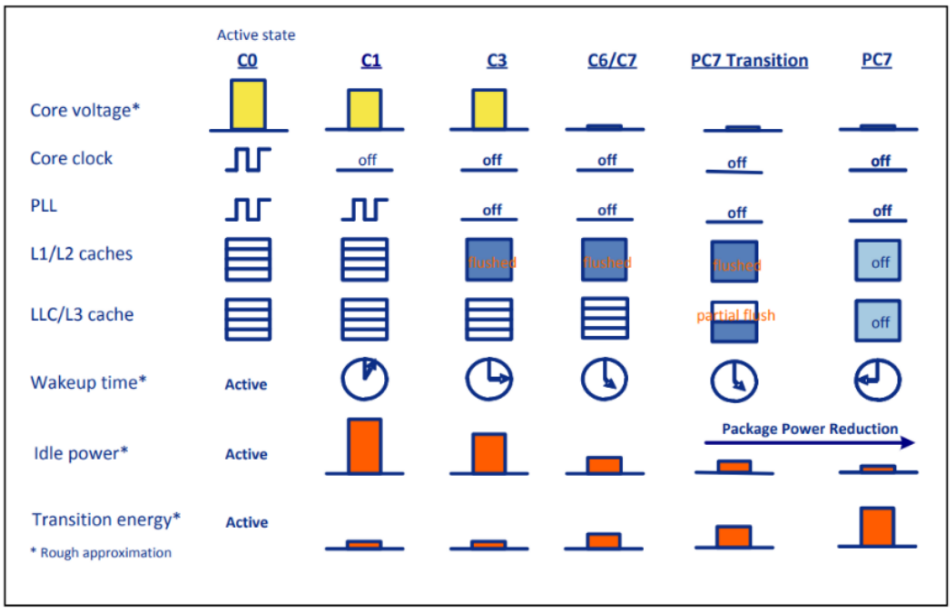


**Figure 2.1:** Core's C-States[105].

sensors, or measurement applications (e.g., powertutor[186]) to corroborate the results of these techniques.

Once the application has been developed, there are tools that can execute it by counting the number of events/operations that it generates in each of the **CPU's** internals[110][180][230][53] (i.e., integer, float point operations, etc.). These tools then infer energy consumption using mathematical models and physical measurement devices[120].

On the other hand, when the application is already running and identified by its **PID**, it is possible to relate the frequency, voltage, or operations type that it needs/performs from/in the **CPU** with the corresponding energy amount[106][202][172].

We summarize all the methods described so far in the table 2.1.

Since we optimize the distributed-applications scheduling in this thesis, the next section deeply describes the approaches mentioned in the last two paragraphs.

### 2.3.3 Proposal to measure the CPU's power consumption from the perspective of distributed algorithms

As Table 2.1 shows, most approaches analyze and improve the **CPU's** energy consumption using hardware and operating system features. Nevertheless, as we study software components' distributed scheduling, the next section describes the

| Techniques to measure CPU power/energy consumption | | | | |
|---|---|---|---|---|
| **Approach** | **Stage of:** | | | |
| | Hardware manufacturing/ low-level configuration | Software Design and development | Launching applications | Applications already running |
| Direct measurement instruments. | X | X | X | X |
| System-level and processors simulators. | X | | | |
| Relate source code with energy consumption. | | X | | |
| Relate CPU's operations (obtained from dmidecode, perf, intel processor counter monitor; etc.) with energy consumption. | | X | X | X |
| Relate battery usage with CPU's energy consumption. | | | X | X |
| Relate CPU's frequency with energy consumption. | | | X | X |
| **Techniques to optimize CPU power/energy consumption** | | | | |
| Improve the C-States and the DVFS. | X | | | |
| Improve the turbo boost usage. | X | | | X |
| Grouping tasks for scheduling improvement. | X | | | |
| Improve run-queues management. | X | | | |
| Energy-aware scheduling. | X | | | |
| GPU-CPU based scheduling. | X | | | |
| Optimize development methods. | | X | | |
| Perform cloud-based scheduling operations. | | | X | X |

**Table 2.1:** Approaches to measure and optimize CPU energy consumption

best methods for analyzing the **CPU** considering two moments: When a component is executed and when it is already running and identifiable by its **PID**.

### Analyzing a process from the moment it is executed and when it is already running

Currently, there are several tools (programs and **APIs**) compatible with certain **OSs** that allow estimating the energy consumption of running applications[183][112][53]. The problem with some of them is that either they do not offer all the models they use openly (at least in their documentation), or their results have been discussed by other scientific works[124][172][38].

For these two reasons, we use the model offered by **PowerAPI**, but also consider the fan's power consumption. In one of its versions presented by Adel Noureddine et al.[172], the model relates the total current **CPU** power consumption to the percentage of its capacity an application uses in a given time.

$$\sum_{i=0}^{nCores} E_{CPU_i}^{PID}(t) = \frac{0.7 \times TDP}{f_{TDP} \times V_{TDP}^2} \times f_i \times V_i^2 \times \frac{U_{CPU_i}^{PID}}{U_{CPU_i}}(t) + fans_s(t), where \tag{2.1}$$

$$fans_s = J \forall K(f_i > f_x > f_j)$$

In the formula 2.1, the first factor refers to the **CPU's capacitance**, which is not always found in datasheets, nor is it easy to obtain through operating system interfaces. For that reasons, the authors deduce it from the processor's **TDP** and its respective voltage and frequency (the same model, but in terms of the **capacitance** and **TDP's** corresponding values). They do that also considering an already proven surplus proportional to 0.7. Then, the second and third factors refer to the current frequency ($f$) and the current voltage ($V$), respectively. The last factor represents the ratio between the **CPU** time for a process represented by its **PID** ($U_{CPU_i}^{PID}$), and the time the **CPU** is active for all the processes ($U_{CPU_i}$). Finally, we added the $fans$ variable which represents the **CPU's** fan energy consumption. That assumes a certain speed ($s$) when a core's frequency $f_x$ reaches a value between $f_i$ and $f_j$.

In the context of our algorithms, we use this formula for each **core** $i$, considering multithreading situations. We can obtain its values using **GNU/Linux** interfaces (applications) such as **turbostat**[22] for the **TDP** (***MSR_PKG_POWER_INFO*** interface) and idle statistics, **msr-tools**[108] (***rdmsr 0x198 -u –bitfield 47:32)/8192***)

for core voltage[1], **perf**[180](***perf stat –per-core –pid***) or **ps**[181](***ps -p PID/PSR***) for processes' per-core **CPU** usage, and **lm-sensors**[229] for the **CPU's** fan information.

It is important to mention that **CPU** undervolting or **C-states** can formally be triggered from the **OS** internal calls (***mwait, htl, IA*_PERF_CTL, etc.***). However, some unofficial tools allow manual procedures[147].

In the context of our work, we indirectly consider the **DFVS** and **C-States** from the non-linear nature of the formula 2.1.

### 2.3.4 Important considerations and future work

Although the model we use is quite accurate to measure the applications' energy consumption, we know that new processors are more complicated to study. We will consider **C-states** and **P-States** more deeply in future works, modeling them per level and residency.

In addition, we also know that we must evaluate special widely-used **CPU** characteristics such as ***overclocking***. To do that, in future works, we may improve the model we described in the previous section. However, we can also use other approaches, such as the last provided by **powerAPI**[38]. It dynamically learns **CPU** power models by exploring the space of hardware performance counters.

On the other hand, we obtained the formula's values using **GNU/Linux** interfaces. That allows us to prove its viability and obtain the input numbers for our simulated environment. However, we will extend our approach by considering different **OS** and tools in future works.

## 2.4 The RAM's energy consumption

As it is known, the **RAM** (Random Access Memory) is the hardware element that the **CPU** actively employs to read and write data (programs, operations, operations results, etc.) in a program execution context. Here, each time the **CPU** needs to read an n-bit word, it first checks to see if that word is in its **cache memories**, spending an approximate time of two clock cycles. If so (**cache hit**), the **CPU** will continue the program's execution with the information found. Otherwise (**cache miss**), that component will have to search the word in the **RAM** and swap it with the old **cache's** value, spending more time and energy[210].

The operating systems use **RAM** to run applications for efficiency reasons. Permanent storage devices, such as hard drives, do not achieve enough data transfer

1 https://askubuntu.com/questions/876286/how-to-monitor-the-vcore-voltage

speed for the **CPU** to operate at its optimum frequency. For example, while a solid-state hard drive on the market today offers a transfer rate of $2000MB/s$ for reading operations and $1700MB/s$ for writing operations[44], a current **RAM** can transfer data at $35200MB/s$ for both operations[47]. For that reason, when an application is launched, or a file is opened, the operating system first copies the concerning data from the storage device to the **RAM**.

All these processes lead several **RAM's** electric parts to work and spend non-negligible amounts of energy. The following section deeply explains the necessary **RAM** internals operation to understand its power consumption modeling.

### 2.4.1 Background

From an internal point of view, each **RAM** type can manage data using different electric devices. For example, if it is an **SRAM** (static random access memory), it stores 1 bit using a six transistor memory cell. Otherwise, in the case of **DRAM** (dynamic random access memory), it uses a transistor and capacitor pair to do that[45].

In the current hardware industry, **SRAMs** are usually employed as **CPU** caches and **DRAMs** as user-replaceable memory modules[45]. For this reason, this section focuses on understanding how a **DRAM** stick works.

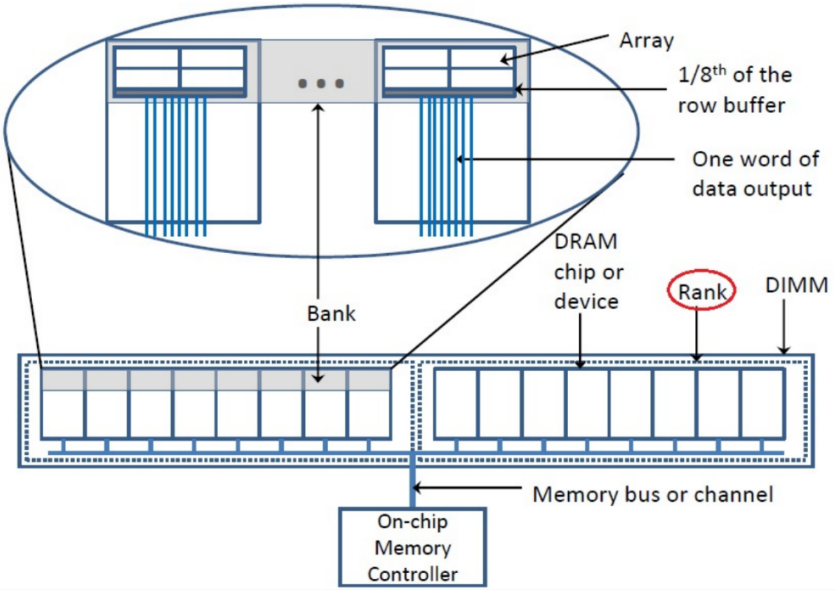


**Figure 2.2:** Memory module structure[216].

As Figure 2.2 shows, internally, a **DRAM module** can be seen as a hierarchically structured hardware component[31]. Within it, the most basic element is the **memory cell**. As aforementioned, to store **1 bit of data**, each **memory cell** uses its internal **transistor** as a switch and its **capacitor** as short-time storage of electric charge. If the **capacitor** contains an electric charge, it represents a binary number **1**; otherwise, the binary represented is **0**[64]. The next element is the **DRAM BANK**, which contains a set of **memory cell matrices**. That set usually has an 8-bit size to simultaneously process a byte per **bank**. Then, a set of **memory banks** (typically 4, 8, or 16) makes up a single **memory CHIP**. Finally, the **memory chips** that can be accessed simultaneously by the same **memory channel** (the link between the **CPU** memory controller and **RAM** modules) compose a **memory RANK**[160].

It is also important to mention that a **RAM** module can contain several **ranks** and **access pins** distributed on both sides (both easily visible). This architecture is known as **DIMM** (Dual Inline Memory Module).

The access speed and the energy usage of each **DIMM** depend on the number of **pins** it has, the voltage it needs to work, the frequency its clock operates, and how it executes read and write operations.

To perform the latter, current **DIMMs** use both the **clock's** rising and falling edges to trigger the transfer operation to the **data bus**[93]. This technology is known as **DDR** (Double Data Rate), which has several upgrade versions over time[48].

With the aim of reading or writing data, a **module** distributes memory addresses to each **bank** through a special bus. This process can be done in two ways: 1) Consecutively per **bank**, or 2) evenly among **banks**. The latter represents a working fashion known as ***memory interleaving mode***.

The mentioned bus, alongside the **RAS** (row address strobe) and **CAS** (column address strobe) pins, are used to find the data bits to operate. First, each data bit is stored in a ***data buffer*** after bit-line **precharging** operations (i.e., set the **bitline** to half the voltage of the entire module). Then, the mentioned elements write the data bit's row and column numbers in the ***row and column addresses buffer***. Finally, the **module** interprets the **WE** (write enable) pin's load to differentiate the operation type (read or write).

The data preservation in a **DDR-DIMM** is given in its normal working state by refresh cycles generated by the **clock**. These are synchronized with the natural **capacitors'** energy loss and the reading/writing operations described above. That is because executing these operations always discharges **capacitors**, which implies its data destruction[65].

Other **RAM's** working states aim to save energy by turning off some of its internals. They are triggered through a **per-rank** signal called **CKE** (clock en-

able)[124][155][161], depending on the **OS** mode and the time that the **RAM** is idle. For instance, the ***self-refresh*** state keeps the **pins** and **banks** working with the minimum voltage necessary to avoid the data disappearance. Here, the **refresh** operation is performed by a built-in timer instead of by the external **clock**. Other states seek to save energy by periodically deactivating the **CKE** or certain **ranks**.

As it is possible to see, each memory **module** uses energy to preserve data in each **bank** and perform read and write operations. However, the **RAM** implements several energy-saving states which are triggered depending on certain circumstances. The next section explains some approaches that seek to measure and optimize the **RAM** power consumption considering the explained elements and some software capacities.

### 2.4.2 Finding and optimizing the RAM's energy consumption

Studying the **RAM** power consumption is a complex task that can be achieved from a hardware or software perspective. Regarding the first, memory manufacturers have lowered the voltage and increased the transfer rate with which the modules work. This fact can be seen in each of the **DDR** versions. For instance, the **DDR2**, **DDR3**, and **DDR4 modules** provide an operating voltage of 3.3, 1.35, and 1.2, respectively[212]. It's important to say that these values can be easily verified with direct measurement instruments or through **OS** interfaces such as the ***dmidecode*** tool[148].

On the other hand, other scientists proposed some **RAM** structural changes. For example, the elimination of the **DRAM** chip **timing circuitry**, delegating its responsibility to the **memory controller**. This strategy decreases the idle state in the **modules'** active mode, allowing data transfers immediately after wake-up operations. As a result, this approach reduces energy per transfer by 50% with no performance impact[155]. This value can be verified using tools such as x86_64 execution-driven processor simulators.

In the literature, it is possible to find more works related to hardware improvement; but we will not detail them as they are outside the scope of this work.

From a software perspective, some **RAM** configurations can be activated from the motherboard's **firmware**. For example, the last work we cited[155] also mentions that it is possible to decrease the power consumption by disabling the delay-locked loops or **memory** interleaving in the **BIOS**. However, that implies serious performance penalties. Moreover, other approaches also use the **BIOS** options to modify the **RAM's** working voltage and frequency. Nevertheless, this strategy also has efficiency negative consequences without achieving desirable power consumption results[55].

| Techniques to measure RAM power/energy consumption | | | | |
|---|---|---|---|---|
| **Approach** | **Stage of:** | | | |
| | **Hardware manufacturing/ low level configuration** | **Software Design and development** | **Launching applications** | **Applications already running** |
| Direct measurement instruments | X | X | X | X |
| Software interfaces such as: dmidecode, perf, intel processor counter monitor, heaptrack, hotspot, etc. | | X | X | X |
| Execution-driven processor simulator | X | | | |
| Count hardware events (cache operations, float point operations, etc.) | | X | | |
| Model the consumption of operations contained in source code | | X | | |
| Model the consumption of memory based on applications workloads and/or applications memory load. | | X | X | X |
| **Techniques to optimize RAM power/energy consumption** | | | | |
| Reduce RAM voltage/frequency | X | | | |
| Reduce unnecessary idlestates | X | | | |
| Deactivate/activate memory interleaving | X | | | |
| Smart scheduling taking into account hardware events or type of operations | | X | X | X |

**Table 2.2:** Approaches to measure and optimize RAM energy consumption

On the other hand, some approaches study the **RAM** power consumption from a higher level of abstraction. They analyze the **OS** and **compilers** configurations, considering the **RAM** workload (amount of reading and writing operations) over time and its corresponding power states[161][160]. For example, several authors tune the operating system to induce some **banks** (normally by **ranks**) to spend less power. Some do that by modifying the page allocation policies[139][84]. Others consolidate the memory allocations and/or references, preventing them from spreading across the entire address space[170]. Finally, various authors improve **OS** scheduling operations per thread considering the number of hardware events (floating-point operations, number of memory accesses, etc.) each performs[16].

The input parameters of the latter are also useful for studying power consumption from other points of view. For instance, the compilers and programming languages development, the mathematical models based on a source code's operations[1], etc. These parameters can be obtained from applications such as ***Heaptrack***[125] and ***Hotspot***[126]. Furthermore, they can also show, among others, the link between the number of memory accesses of a running application and its corresponding lines of code

Finally, in the context of **already built applications**, some tools such as ***perf*** (Performance analysis tools for Linux)[180], ***Intel's Performance Counter Monitor***[230], or different types of physical sensors, allow studying the number of **RAM** accesses they perform when executed or when they already run. These tools are used by works that propose methods and models to interpret the **RAM** energy consumption. To do that, their authors usually consider certain types of workloads and the presence of certain amounts of data in this component[124][53][10].

The table 2.2 summarizes the techniques described above for measuring and optimizing **RAM** power consumption. It separates the approaches into four levels of abstraction: (1) Low level, which involves hardware development or configuration, including firmware and operating systems. (2) Medium level, which studies software design and development (from the programing language point of view). (3,4) High level, which comprises executing and running applications.

In this thesis, we study and optimize the scheduling of software components, which are running applications. For that reason, the next section focuses on deeply explaining the levels 3 and 4 mentioned in the last paragraph.

### 2.4.3 Proposal to measure the RAM's power consumption from the perspective of distributed algorithms

As we indicated in section 2.1, this research analyzes and optimizes the energy consumption of software components (in fact, running applications) through distributed scheduling operations. This analysis can occur from the moment an application is launched or when it is already running. In terms of **RAM**, these operations should consider two factors. On the one hand, they must be based on information accessible to the scheduling platform (usually middleware). On the other hand, they must attempt to get the **RAM** into low-power states.

Not many works have dealt with this topic since most focus on low-level aspects, as seen in the second part of the table 2.2. However, there are some exceptions. For instance, the creators of ***RAPL*** (Running Average Power Limit) proposed a weights-based model to relate the **RAM** power consumption, the read/write operations it performs, and its **CKE** state[53]. However, we didn't find the implementation of this model's calibration process (the model's weights). In addition, its accuracy has been re-evaluated, showing mixed results, especially when the system is in idle states[57].

As another example, one of the most relevant approaches for us is the one proposed by **Alexey Karyakin** et al.[124]. Their work provided and tested (using measurement tools and mathematical regression) a model to calculate the energy used by the **RAM** when performing database operations. This model considers in a differentiated way: (1) the **background consumption**, which depends only on the **RAM's** power state, and (2) the **active consumption**, which is generated by the reading/writing operations it performs.

To calculate the first type of consumption, the authors considered the percentage of a given time in which the **RAM** is in one of the following power states: (1) **Active**, which represents the "normal" working state (2) **CKEOFF**, which represents all intermediate states of power saving and (3) **self-refresh**, as the state of least power consumption. Thereby, the model they propose multiplied the said time by the corresponding power consumption. Moreover, since they used physical measuring instruments, the model considers the additional consumption generated by each **rank** in an **active** state as a separate attribute.

To calculate the **RAM's active** consumption, the authors considered the number of reading and writing operations carried out in a given time. That is, they multiplied the number of operations of each type by the energy consumption that each one generates. Here, the authors' model also considers the consumption of each **pre-charge** and **activation** cycle generated per operation.

For our part, to find an application's energy consumption in terms of **RAM**,

we use this same approach in a particular way. We also discover the **RAM's background** and **active consumption** but consider only the information we can obtain from the **GNU/Linux** interfaces. Furthermore, we differentiate between an application launching process and an application already running in the **active** energy consumption.

### The background energy consumption ($E_{BK}$)

In order to determine the **RAM's** power state residency from **OS** interfaces, we consider using two different approaches:

1. **Option 1: To use a processor counter monitor:**

   These powerful tools offer a set of easy-to-use programs that provide (among other functionalities) very verbose **RAM** information. For example, its internals' description, the global number of operations it performs in a given time, and (in a fairly exact way) the residence of the **RAM's ranks** in each of its power states.

   Among these tools, the **Intel Processor Counter Monitor - PCM** includes the **pcm-memory.x** and **pcm-power.x** applications to do this[10][150]. Other vendors like **AMD**[6] provide similar tools for their processors.

   Although **PCM-tools** are excellent for our purposes, they are compatible with a limited range of processor models (even from the same brand). That is why we consider using the relation between the memory and **OS** power states, as explained below:

2. **Option 2: To interpret the kernel power-states:**

   Following the **GNU/Linux kernel** specification[235], we relate its power states with those of the **RAM** as follows:

   - The **Suspend-to-idle state** and **Standby state** both have the equivalent of the **RAM's CKEOFF** state. The former suspends the timekeeping and puts all **I/O devices** into low-power states. The latter does the same but also disables low-level system functions.
   - The **Suspend-to-RAM state** puts the whole system into a low-power fashion. In the specific case of the **RAM**, it is set in the **self-refresh** state to allow the session's data retention.

   To learn the **OS** power-state, we can read the ***/sys/power/state*** file[234]. Its content usually is: 1) A string list of all available states if ***active state***, 2) **"freeze"** if ***StI state***, 3) **"deep"** if ***StR state*** and 4) **"standby"** if ***SbS state***.

Using any of these options, in our approach, we consider the amount of time the **RAM** was in each state ($T_x$), multiplied by its corresponding power consumption ($P_x$). That can be seen in the formula 2.2.

$$E_{BK} = T_{sf} \times P_{sf} + T_{ckeoff} \times P_{ckeoff} + T_{act} \times P_{act} + \sum_{i \in ranks} T_{act} \times P_{act_i} \quad (2.2)$$

Moreover, as aforementioned, the model of **Alexey Karyakin et al.**[124] considers additional power consumption **per rank** (row activations, reads, and writes) in the **CKEON** state. For the case of option two, we also take this consumption into account but consider the **OS's active state**.

| **Memory power state** | **Memory system state (proposed equivalence)** | | **Value (Watts)** |
|---|---|---|---|
| Self-refresh state(sf) | Suspend to RAM state (StR) | | 0.35 |
| CKEOFF state (ckeoff) | Suspend to Idle state (StI) | Standby State (SbS) | 0.89 |
| CKEON state (act) | Active state (act) | | 1.56 |
| Additional consumption CKEON state per rank ($act_i$) | Active state ($act_i$) | | 0.098 |

**Table 2.3:** Equivalence values to find RAM's background power consumption

The value of each factor in the formula 2.2 can be obtained by: (1) The tools described in the two previous options, (2) the ***dmidecode -t memory***[148] interface to know the number of memory **ranks**, and (3) the table 2.3, whose values were provided by the measurement and regression work of **Alexey Karyakin et al.**[124].

### The active energy consumption ($E_{act}$)

As **Alexey Karyakin et al.**[124] do, we calculate **RAM's** active energy consumption from the number of read-write operations carried out in a given time or by a process. That is, we multiply the number of operations ($N_x$) by the energy consumption ($E_x$) that each one generates.

The only difference with the mentioned authors is that we do not take into account the energy consumption of the **pre-charge/activation** cycles. That is because we couldn't access this information from any **OS** interface.

We describe the **RAM's** active energy consumption in the formula 2.3.

$$E_{act} = N_R \times E_R + N_W \times E_W \tag{2.3}$$

Following the chapter's methodology, we analyze an application's energy consumption from the moment it is executed and when it is already running. That is why we propose to obtain the values of each factor in the formula 2.3 in two slightly different ways.

1. **Option 1: Analyzing an application from the moment that it is executed:**

   In this option, the values of each factor in the formula 2.3 can be obtained via the following: 1) The ***perf (record | stat) -mem -p app*** command. It provides the number of page read-write operations performed in the **RAM** (this command also provides information about **cache hits** per level) from the moment the ***app*** is executed. 2) The table 2.4 whose values were provided by the measurement and regression work of **Alexey Karyakin et al.**[124].

| Operation | Value (nJ) |
|---|---|
| Page read operation | 6.6 |
| Page write operation | 8.7 |

**Table 2.4:** RAM's operations' energy costs

2. **Option 2: Analyzing an application already running:**

   In this option, we use the same sources as the previous one but make the **perf** command receive as parameters: 1) The **PID** of the process to be analyzed, and 2) the virtual folders of the **CPU/RAM** read-write events **(cpu/mem-stores and cpu/mem-loads): *perf (record | stat) -e cpu/mem* -p pid***

It is important to mention that tools based on **perf**, such as **Heaptrack** and **Hotspot**[54], allow analyzing the memory accesses an application performs more clearly (even visually). These help to understand its behavior at certain times and consider external variables, such as the number of requests, availability of other resources, etc.

### The overall application's energy consumption in the RAM ($E_{RAM_{app}}$)

As the Formula 2.4 shows, we consider both the **active** ($E_{act}$) and the **background** ($E_{BK}$) energy consumption to determine the energy consumed by an application in the **RAM**.

$$E_{RAM_{app}} = E_{BK} + E_{act_{app}} \tag{2.4}$$

Since the factors of the Formula 2.4 can be found from **OS** interfaces, we can use it either to analyze a single application (process) or entire **OS** instances. This consumption information is useful for distributed algorithms to make energy-conscious scheduling decisions.

### 2.4.4 Important considerations and future work

As the following chapters will show, the analysis of this section allows **RAM** to be an energy criterion for distributed scheduling algorithms. However, we are aware that our model may not be exact. That's because of the lack of analysis of the preload cycles, variations in the hardware model, or the impressions caused by normal **OS** interruptions.

For this reason, although our scheduling algorithms are based on comparative aspects and not on exact quantities, we will improve the proposed model using measurement devices and regression techniques in future work.

On the other hand, we obtained the model's values using **GNU/Linux** interfaces. That allows us to prove its viability and obtain the input values for our simulated environment. However, we know that we can learn other energy models with different tools and operating systems. We will also do this study in future work.

## 2.5 The NIC's energy consumption

The **NIC** (Network Interface Controller) is the hardware component used by a network node to communicate with other peers. To achieve this, a **NIC** supports several transmission protocols[24] and data sending/receiving abstraction levels, such as the proposed in the **OSI** model for **Ethernet**[119].

Different **NICs** types enable different network architectures, such as **Ethernet**, **Token Ring**, **Bluetooth**, or **Wifi**[95]. Each of these cards presents different ways of operation and, therefore, energy consumption.

In this work, we study **Ethernet** and **Wifi** devices since, to our understanding, these are the most used and studied in the distributed environments context.

### 2.5.1 Background

A **network interface**, whether it is embedded, internal, or external, typically consists of data buses, a memory, a processor, connectors, etc. [104][226][218].

Unlike our analysis of the **CPU** and **RAM**, we will not deeply detail the **NIC's** internals. That's because of two reasons. On the one hand, we have not found a way to relate the **NIC** circuitry's operations to its energy consumption. On the other hand, although network cards enable various energy-saving states (e.g., IEEE 802.11 power management specification[33]), we have not found a way to obtain their parameters from any **GNU/Linux** interface. That's why the power consumption models that we study/propose for this component are mainly based on: 1) the number and size of packets it receives-sends, and 2) the transfer rate it performs.

Finally, it is important to say that we study wired and wireless cards in the same way. We do that because several differential aspects are unmanageable for us. For example, the fixed wireless cards' transfer-power value (which impacts the distance of the signal) can be obtained from tools like ***wavemon - tx-power***[227]. However, it can only be modified by recompiling the corresponding drivers.

The following section explains some existing approaches and our proposal to find and optimize the **NICs'** power consumption.

### 2.5.2 Finding and optimizing the NIC's energy consumption

In the literature, several works propose mathematical models **to find** the **NIC's** power consumption from different abstraction points. To do that, some study an entire network deployment considering several aspects. Among these aspects, we can mention protocols and topologies[39], each node involved in the transmissions (**NICs**, routers, switches, etc.)[177][167], and the transmission power-money costs[177]. Furthermore, they also consider the simulation of scenarios where they relate variables such as transfer rate, the number of read/write operations, and energy consumption[90][246].

On the other hand, from the analysis point of a single device, some scientists study the last-mentioned relation to develop specialized platforms, applications, or APIs[172][183].

Other different approaches study the energy consumption of point-to-point communications. They take into account the entire network architecture[39], the execution of packet sending/receiving/discarding operations that specific services perform[238][73], etc.

Finally, as an example of a low-level perspective, some scientists consider the Mbit/Joule energy consumption in antennas and other network devices' circuitry[19].

Regarding **optimizing** the **NICs'** power consumption, diverse solutions also focus on different levels of abstraction. For instance, from a low-level perspective, the **GNU/Linux** Kernel may use **IOCTL** calls to request the network driver to put the **NIC** (or network subsystems) in a low-power state[129].

From a higher point of view, other works seek to improve the behavior of some transmission protocols. For example, they adjust the packet payload size and transfer patterns[153], modify the number of packets transmissions[121], improve the paths or the number of hops performed through the nodes[237][149], and others. These strategies enable network elements to intelligently go into sleep mode or adapt bandwidth to save energy[167].

Finally, some scientists optimize power consumption by improving data distribution among the nodes. They do that to shorten data-node distances[154], apply data consumption prediction algorithms[70], etc.

The Table 2.5 summarizes the techniques we described to find and optimize the **NIC's** power/energy consumption. We consider four abstraction levels within it: (1) The low level, which involves hardware configuration, operating system/drivers customization, or protocols improvement. (2) The medium level comprises software design and development (from the conception/programming point of view) considering variables such as the amount of sending operations performed. Finally, (3,4) the high-level analyzes executing and running applications considering variables such as transfer rate or the number of packets received/sent.

The next section explains our approach to find the **NIC** power consumption from the high-level perspective, which is compatible with distributed scheduling.

### 2.5.3 Proposal to measure the NIC's power consumption from the perspective of distributed algorithms

In this research, we analyze and optimize the software components' energy consumption by means of distributed scheduling operations. For this, we need to study these components' consumption from those **NIC-related** variables (among the others described in this chapter) that are possible to obtain from **OS** interfaces.

As shown in both parts of table 2.5, several works provide useful methods to analyze running processes from the **NIC** perspective. They consider variables such as 1) the current transfer rate, 2) the number of sent-received packets in a period, and 3) the current power status of the network card.

Although these models are very interesting and potentially useful for our research, they present some problems for us. For instance, the creators of **powerAPI**[172] offer a model based on the time that the network card spends in each of its power states. Although this model has already been tested by them, we have not found a way to obtain these states (**Power-up**, **idle 10/100/1000**, **LPI**, **S0**, **SX**, among others[116] using any **GNU/Linux** interface. Similarly, in order to find the **NIC's** power consumption, Feeney, L.M et al.[73] consider its power status as well as

| Techniques to measure NIC power/energy consumption | | | | |
|---|---|---|---|---|
| **Approach** | **Stage of:** | | | |
| | **Hardware manufacturing/ low level configuration** | **Software Design and development** | **Launching applications** | **Applications already running** |
| Study of topologies' and protocols' behavior | X | | | |
| Network architecture analysis | | X | | |
| Relate energy consumption with packets operations (read/write/ discard) performed | | X | X | X |
| Relate energy consumption with bandwidth or transfer rate | | | X | X |
| Study of devices' circuitry | X | | | |
| **Techniques to optimize NIC power/energy consumption** | | | | |
| IOCTL CALLS | X | | | |
| Adjust payload size and transfer patterns | X | | | |
| Enable devices either to go into sleep mode or to adapt their transfer rate. | X | | | X |
| Study network paths or the number of hops performed | X | X | | X |
| Intelligent distribution of data on the network | | X | | X |

**Table 2.5:** Approaches to measure and optimize NIC energy consumption

the operations of sending, receiving and discarding packages. In addition to our problem obtaining the power states, the packets discarding correspond to the protocol's operation and not necessarily to the behavior of processes. As a last example, Nedevschi et al.[167] propose models to find the power consumption of all the elements of a network considering the sending operations costs, processing of network packages, device's power states and the adaptation of transfer rates. Although this work is very interesting, the adequacy of transfer rates and packet processing is outside the scope of our work.

It is also important to say that, unlike previous devices, we couldn't analyze the **NIC's** consumption differently when a process is launched and when it is already running.

For all these reasons, we decided to propose our own energy models that relate processes' **NIC-load** with the corresponding power consumption. That is what the next section explains.

### Analyzing a process from the moment it is executed and when it is already running

As we said above, the existing methods focus mostly on analyzing transmitted packets, the bandwidth, and the **NIC's** power states to find its power consumption. To achieve this objective, we don't consider the device's power states. Instead, on the one hand, we propose a model based on the process's transfer rate. On the other hand, we propose another approach based on the number of packets a process transmits/receives.

1. **Analyzing a process from the transfer rate it generates**:

   To obtain the **NIC's** power consumption of a process, we start by finding this component's overall current consumption. For this, for a certain device **D**, we establish a $W_{u_d}$ and a $W_{i_d}$ values which represent the power consumption in **Watts** when its **NIC** is in an active and idle state, respectively.

   For a current transfer rate $L_D$ ,$Wu_D$ multiplies $L_D$ relative to its maximum transfer capacity $L_{MAX_D}$. Then, this result is added to the idle state consumption. For this, $W_{i_D}$ multiplies the complement of the **NIC's** load (i.e. $L_{MAX_D} - L_D$) relative to $L_{MAX_D}$.

   Finally, to obtain energy consumption in joules $Watts \times s$, we take into account the time $T$ in seconds. That is shown in the formula 2.5, which considers a symmetrical full-duplex mode:

$$E_{NIC_D} = (W_{u_D} \times \frac{L_D}{L_{MAX_D}} + W_{i_D} \times \frac{L_{MAX_D} - L_D}{L_{MAX_D}}) \times T_D \quad (2.5)$$

Before analyzing running processes, it's important to mention the last-formula factors' sources:

- The **NIC's** transfer capacity can be obtained from standardized specifications or datasheets[153], interfaces such as ***ethtool input***[127] for ethernet connections, or the ***/sys/class/net/Interface/device /max_link_speed*** (this could be different depending on the **GNU/Linux** distribution) interface for **wifi**, **ethernet** or other types of network cards.
- The current global transfer rate can be obtained from interfaces such as the ***iftop -i***[94] program.
- The power consumption of the **NIC's** working states can be obtained from datasheets (e.g., the intel **I219-LM** datasheet[116]), or some descriptive works like the one done by Chiaravalloti et al.[33].

Then, to model the energy consumption of a process **M** (identifiable by its **PID**) in terms of its workload in a device's (**D**) **NIC**, we use the equation 2.5; but only consider the mentioned workload ($L_M$). However, one of our challenges was figuring out how to handle idle time. In one first approach, given that at a given time $T$, the transfer rate is the average of the sending/receiving operations and idle states in $T$, we also considered that $M$ generates an idle state proportional in the same way that $L_M$ is for the current **NIC's** load ($L_D$). We show that in the equation 2.6.

$$E_{NIC_{D_M}} = (W_{u_D} \times \frac{L_M}{L_{MAX_D}} + W_{i_D} \times \frac{L_{MAX_D} - L_D}{L_{MAX_D}} \times \frac{L_M}{L_D}) \times T_D \quad (2.6)$$

This last formula implicitly considers the idle state at a certain moment of transmission. However, it may have a problem. Suppose several processes use the network card, keeping it with a relatively high load. In that case, the energy consumption of the idle state is attributed to each process, increasing its consumption in proportion to the load they generate. Although this may make sense, that increase is no longer logical and becomes artificial in different situations. For example, suppose that there is only one process that uses the **NIC** at a low transfer rate, and all the **NIC's** idle consumption is attributed to it. Here, high idle power consumption affects a process that uses the network card at a low frequency.

That is why in a second approach, we decided to consider the idle time independent of the processes and exclusive to the **NIC's** background consumption. In this way, the power consumption of a process depends exclusively on the operations it performs on the **NIC**. We show that in the formula 2.7:

$$E_{NIC_{D_M}} = (W_{u_D} \times \frac{L_M}{L_{MAX_D}}) \times T_D \tag{2.7}$$

This model is useful when it is only possible to obtain the consumption data of the **NIC's** idle and active state. However, it does not consider the power consumption differences that may exist between the sending and receiving operations. That is why, if this information is available, a pertinent approach is necessary. We will discuss it in the next paragraph.

2. **Analyzing a process from the packets it sends or receives**

   Another way we study a process's power consumption from its **NIC** usage is by analyzing the sending/receiving operations it performs. Indeed, this simple principle does not consider some protocol aspects, such as link-layer fragmentation or unsuccessful attempts to acquire media contention[73]. However, it enables quite accurate mathematical models based on data obtainable from **OS** interfaces.

   In our approach, we start defining the $W_{D_S}$ and $W_{D_R}$ values to represent a device's ($D$) **NIC** power consumption (in **Watts**) when sending and receiving packets. Then, to find the energy consumption ($E_{NIC_{D_M}}$) that a process ($M$) generates on the **NIC**, we calculate the time that said process keeps the **NIC** sending and receiving data. To define this time, we first multiply the number of packets sent ($NS_M$) or received ($NR_M$) by the respective average packets' size ($S_x$). Then, we divide the results by the corresponding total transfer speed ($L_{MAX_x}$).

   We describe that approach in the formula 2.8.

$$E_{NIC_{D_M}} = W_{D_S} \times \frac{NS_M \times S_s}{L_{MAX_s}} + W_{D_R} \times \frac{NR_M \times S_r}{L_{MAX_R}} \tag{2.8}$$

   Regarding the sources of the formula factors', let us explain two important points:

   - The data concerning the packets, such as the quantity, size, and type of operation performed per process, can be obtained from applications such

as ***atop -n***. For that, if the ***netatop-dkms*** kernel module is compiled, the tool provides information by protocols such as **UDP** and **TCP**. If it is the case, the model 2.8 must be extended.

- The maximum transfer rate per operation and the corresponding power consumption values can be found from the same sources described in the previous section.

## 2.6 Storage device energy consumption

The **Storage Device** is the hardware component that permanently stores users' and system's information. As we explained in section 2.4, each time an application is launched, the **OS** copies all or part of the application's data from the **storage device** to the **RAM** for efficiency reasons. Then, depending on the application's behavior, there could be data exchange between both components at a certain frequency. Moreover, if the amount of information exceeds the **RAM** capacity, the **OS** may use the **storage device** as a **swap** area, increasing its operational load[210].

All this flow causes the storage device to perform read and write **I/O operations**. These activate internal electrical components that consume energy when working.

In the following sections, we analyze the necessary information to understand this consumption from the perspective of running processes.

### 2.6.1 Background

Generally speaking, there are currently two types of storage devices for personal computers and servers: **HDD** (Hard Disk Drive) and **SSD** (Solid State Drive). The former comprises a stack of spinning metal disks known as platters. Each spinning disk has trillions of fragments that can be polarized to represent bits (1s and 0s in binary code). An actuator arm carries out this process. It magnetizes the fragments to write information and detects their magnetic charges to read it.

On the other hand, **SDDs** comprise trillions of semiconductors that store information by changing the electrical current of special internal circuits. Unlike **HDDs**, **SDDs** offer a higher transfer speed since their way of operating does not depend on moving parts[62].

Finally, both types of storage devices define four power states: **Active**, **Idle**, **Standby**, and **Sleep**. In this sequence, the more power a state saves, the longer it takes to return to the active state[194].

As we did with the network card, we will not detail more about the storage device's internals since the models we find-propose do not depend on them. Moreover,

for the same reason, we consider for both the **HDD** and the **SSD** only two working states and their corresponding power consumption: The **active** and the **idle** state.

The following sections explain our approach to model applications' power consumption from the perspective of storage devices.

### 2.6.2 Finding and optimizing the hard drive's energy consumption

In the state of the art we analyzed, not many works have studied the **storage device's** power consumption from a software management perspective. However, some do that based on simulation techniques variables. For example, they analyze previous physical measurements and specific hard drive's working states (rotation, writing, reading, etc.)[242], the application's execution stages considering **I/O** operations[246], etc. On the other hand, others offer energy models using different approaches. For example, they analyze the cost of virtual machines deployment[158], the size of in-disk files accessed by concurrent processes[103], and others.

From an **optimization** point of view, most works study how to handle disk accesses efficiently. For instance, some increase applications' execution efficiency through intelligent scheduling of **I/O** requests[41]. Other works reduce power consumption by caching/buffering processes'[29] or virtual machines'[239] disk accesses in a coordinated way. This last strategy allows the storage device to remain **idle** or **sleep** for the necessary time to meet energy savings.

Finally, the Table 2.6 summarizes the techniques described up to this point. As with the **NIC**, we couldn't implement any of these approaches with the values obtained from **OS** interfaces. The following section then explains this fact and our own consumption model.

### 2.6.3 Proposal to measure the Storage Device power consumption from the perspective of distributed algorithms: Analyzing a process from the moment it is executed and when it is already running

As we said in the previous section, we haven't found any approaches that we can implement via **OS** interfaces. Furthermore, the methods described in the table 2.6 analyze variables such as the size of files managed by applications or the **I/O** operations they perform. Both are outside the scope of our study.

| Techniques to measure Hard Drive power/energy consumption | | | | |
|---|---|---|---|---|
| **Approach** | **Stage of:** | | | |
| | **Hardware manufacturing/ low level configuration** | **Software Design and development** | **Launching applications** | **Applications already running** |
| Perform physical measures | X | | | X |
| Model consumption taking into account previous physical measures | X | | | X |
| Model consumption taking into account application's I/O stages | | X | | |
| Model consumption taking into account files/VMs size | | X | X | X |
| **Techniques to optimize NIC power/energy consumption** | | | | |
| Manage disk accesses to increase applications execution efficiency or to reduce the storage device energy consumption | X | | | |

**Table 2.6:** Approaches to measure and optimize storage device energy consumption

For the mentioned reasons, we propose a model based on the power consumption that the **hard drive** generates when it is in the **active** (performing read or write operations) or **idle** state.

Our model is similar to the equation 2.5 for the **NIC**. To find the **hard-drive energy** consumption, we establish the $W_{u_D}$ and $W_{i_D}$ values that represent the power expenditure when a device's ($D$) **hard drive** is in an **active** and **idle** state. Then, for a current transfer rate $L_D$, $W_{u_D}$ multiplies $L_D$ relatively to the maximum transfer capacity $L_{MAX_D}$. Next, we add this result to the **idle** state consumption. For that, $W_{i_D}$ multiplies the complement of the **hard drive** load (i.e. $L_{MAX_D}$ - $L_D$) relatively to $L_{MAX_D}$. Finally, we consider a time $T$ to obtain the **disk's** energy consumption in joules (Watts*s), as the equation 2.9 shows.

$$E_{HD_D} = (W_{u_D} \times \frac{L_D}{L_{MAX_D}} + W_{i_D} \times \frac{L_{MAX_D} - L_D}{L_{MAX_D}}) \times T_D \tag{2.9}$$

Then, to define the energy consumption ($E_{HD_{D_M}}$) that a process $M$ generates on the **hard drive**, we use the equation 2.9; but only consider the load generated by $M$ ($L_M$) relatively to $L_{MAX_D}$.

As in the case of the network card, we consider the **idle** time independent of the processes and exclusive to the **hard drive** background consumption. Thereby, a process power consumption depends exclusively on its operations on the **hard drive**, as seen in the formula 2.10.

$$E_{HD_{D_M}} = (W_{u_D} \times \frac{L_M}{L_{MAX_D}}) \times T_D \tag{2.10}$$

Concerning the formula's factors sources, we have not found a **GNU/Linux** interface for the $W_{u_D}$, $W_{i_D}$ and $L_{MAX_D}$ values. However, many **HDD** and **SSD** brands specify them in datasheets[195][193]. Furthermore, the transfer rate generated by a process on the **hard drive** can be obtained from programs such as ***iotop***[179].

## 2.7 Chapter Summary and Contributions

In this chapter, we have explained how various authors find and optimize the power consumption of the **CPU**, the **RAM**, the **NIC**, and the **storage device**. We have organized these approaches from a perspective of 1) hardware and **operating system** configuration, 2) software design and development, 3) application launching, and 4) applications already running.

The last two perspectives are compatible with our distributed algorithms' logic. For that reason, we have studied, adapted existing approaches, and proposed (if

concerned) our own mathematical models to describe the energy consumption of each of these components. Thereby, for the **CPU** and the **RAM**, we have selected and adapted already tested formulas to relate their workload, power state, and their power consumption. Finally, considering the same relation, we have proposed and tested our own energy consumption models for the **storage device** and the **NIC**.

It is important to say that each model's parameters' values can be obtained from different **GNU/Linux** interfaces (such as APIs, specialized programs, and others). In this chapter, we have described the still maintained and documented ones.

The next chapter studies and classifies important distributed scheduling approaches, detailing their operations, input variables, objectives, abstraction level, etc. As well as the models above, that study will be an input criterion for our scheduling approaches presented later in this thesis.

# CHAPTER 3

# THE DISTRIBUTED APPROACH

# 3 The Distributed Approach

## 3.1 Introduction

A few decades ago, traditional applications used to be only conceived in a traditional monolithic way. In this manner, all the development processes and deployment strategies considered only a single host. At runtime, monolithic applications' internals communicate with each other using method invocations or function calls to produce specific results. In this process, the operating system is the entity that manages the required access time to each hardware resource, such as **CPU**, **RAM**, etc[210].

Although this approach is still suitable for applications such as a kernel[210], more modular approaches are needed today. For example, applications such as **Netflix**, **eBay**, or **Zalando** engines[137] require performing massive data queries and having the independence of location and platform to work. Thereby, for them, a monolithic approach would present serious disadvantages such as unmanageable scalability, low cohesion, and high coupling[56]. In addition, there are scenarios where several nodes must interact to produce a result or satisfy a specific user's needs. That's the case of horizontal ambients such as sensor networks[184] or user device networks[51]. For these two reasons, the conception and-or the deployment processes are carried out through **distributed systems**.

A distributed system is defined by Tanenbaum et al.[205] as a collection of autonomous computing elements (in fact, software or hardware components) that appears to its users as a single coherent system. In their article, the authors explain that for a system to be distributed, it needs to meet some characteristics:

- Node autonomy: Each component fulfills its independent task and communicates with other peers using methods such as message passing. Therefore, they work concurrently without sharing a global clock or shared memory[166].
- Single coherent system: The set of components behaves transparently according to the users' interaction level and expectations. That is, preventing them from knowing and managing the system's architectural characteristics that they don't need to know.

Thereby, from an architectural point of view and depending on the user's role, it is required: 1) to know and or be able to manage the network architecture (often an overlay) and 2) to have a technology that allows, among others (execute security services, establish the communication between processes, etc.), executing scheduling operations to manage the nodes' hardware resources. This technology is known as **middleware**.

Several industrial technologies and authors use said scheduling operations to achieve different objectives. For example, as it is possible to see in some surveys[250][228][236][4][43][190][175][190][152], these objectives take into account variables such as centralized/non-centralized heuristics, mathematical/artificial intelligence approaches, types of application and virtualization, etc. All this is to optimize the deployment of applications time, energy consumption, load balancing, SLA assurance, etc. However, few of them prioritize power consumption as one of their main goals. As we will show in the following sections, they do not consider energy-based scheduling analyzing variables such as hardware heterogeneity, smart resources indexing, or special hardware power capabilities.

In this thesis work, we analyze and improve nodes' hardware resource management in a novel way to save energy without losing the notion of **QoS** (Quality of Service). We schedule (middleware operations: move, duplicate, start, stop) the components that make up a distributed application through different physical nodes using multidimensional/spatial special structures. With them, we index elements such as physical nodes and applications' components at execution time according to multiple criteria, such as the current availability of hardware resources, hardware's power-related features, or the application's resource requirements. This way, our objective is to have a system that performs the elements' search, insertion, and elimination at an efficient computational cost in order to perform energy-saving scheduling strategies.

The explanation of this entire strategy is the cornerstone of our next chapter. However, to justify and contrast it, we present an analysis of the state of art in the present chapter, proposing a taxonomy of different scheduling techniques that we consider important.

In the next section, we will explain the methodology that defines the structure of the rest of the document.

## 3.2 Methodology

This chapter analyzes the state of the art of distributed software components deployment and scheduling techniques. To easily contrast this analysis with our

proposal (briefly described in the previous paragraph), we present an algorithmic-based taxonomy that considers two scheduling moments:

- **Initial deployment**, which studies the correct nodes to deploy a software component, and
- **Execution-time scheduling**, which involves operations such as migration or duplication.

For each of these moments, we describe several approaches which may span one or multiple proposals in the form of an algorithmic flow. That means we analyze the concerning input variables, the operations that are executed, the method's heuristic (centralized and non-centralized), and the optimized variables.

Finally, as it is possible to see in figure 3.1, we consider it important to show the potential extensibility of our proposal. That's why we explain the mentioned moments and flows considering two different levels of abstraction: **Scheduling in the cloud** and **Scheduling close to data sources** which involve approaches such as grid computing, edge computing, and fog computing.

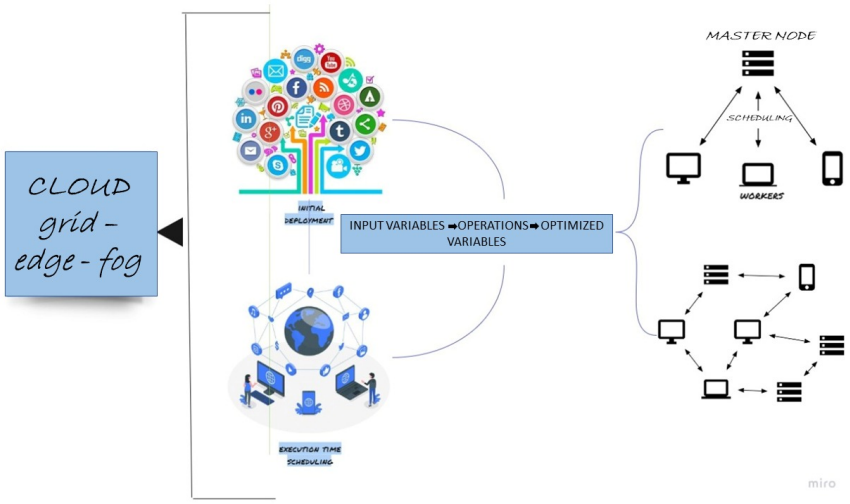


**Figure 3.1:** Taxonomy Structure

Section 3.3 explains the works that study **Scheduling in the cloud** techniques, and section 3.4 discusses the ones that study **Scheduling close to data sources**. In both sections, we analyze only the approaches that we consider important from the current state of the art. For that, we have filtered the research/industrial works considering particularity, popularity, or topicality criteria.

Our objective with this sample is to show the algorithms' trends in terms of the criteria they evaluate and the variables they optimize. To do that, we build two descriptive tables for each section: one for the analysis of the **initial deployment** approaches and another for the **execution-time scheduling** ones. Next, we have converted the number of approaches that coincide at each table point (input-optimized variable) in a ranking system that shows average trends. Finally, we deliberately score our work to compare it with the mentioned trends, evidencing the field of action of our contribution and its strengths.

It is important to emphasize that the scores **do not represent** the quality of the analyzed approaches. They only show optimization trends useful to define our proposal scope.

We finish the chapter by explaining the corresponding conclusions and contributions in section 3.5.

## 3.3 Scheduling in the cloud: Managing virtual entities in node clusters.

Cloud computing (whether public, private, or hybrid) is the on-demand availability of computing resources as services over the Internet (enabling the pay-per-use basis). These services are usually offered under three different models: 1) **IaaS** (Infrastructure as a service), which offers servers, storage, networking resources, etc., 2) **PaaS** (Platform as a service), which offers a develop-and-deploy environment to build cloud apps, comprising middlewares instances, development tools, data management systems, among others, and 3) **SaaS** (software as a service) which delivers applications functionalities as services to end users[35][162][163][164].

All these services comprise and-or manage applications deployed on clusters. By definition, a cluster consists of a set of nodes (from personal computers to super servers) connected through the network. They work together to perform common tasks providing fast processing speed, large storage capacity, efficient data integrity, reliability, and wide availability of resources[146]. Depending on the technology and its objectives (such as load balancing or high availability), a cluster can cover one or different geographical areas[60][11].

As in several distributed environments, in cloud computing systems, two problems must be solved when deploying applications and-or performing scheduling operations: 1) The independence of applications instances concerning a specific platform/operating system, and 2) the control of the resources quotas given to them. Both are solved by isolating these hardware resources (**CPU** time, **RAM** quantity,

etc.) and needed software (dependencies) in virtual execution environments. Depending on their operation and isolation level, they can be differentiated into **VM** (virtual machines) and **containers**[214].

- When using **VMs**, an entity called a **"hypervisor"** virtualizes each of the hardware components for complete instances of operating systems. Depending on its type, a hypervisor can run 1) directly on the hardware (type 1), serving as a lower abstraction layer than any operating system, or 2) as an application in a pre-existing operating system[67].
- When using **containers**, virtualization occurs using the kernel features of the host operating system. For this, the kernel isolates the processes of the virtualized applications in user-spaces-like systems by means of **namespaces** and **CGROUPS**. Moreover, they keep the involved dependencies (libraries, executables, etc.) described in a file and deployed in a folder hierarchy[92][9][213].

Typically, the cloud systems' engines schedule[61][12](i.e., deploy, migrate, start, stop) both of these technologies to fulfill the different providers' objectives related to their offered services (**IaaS**, **PaaS**, **Saas**). This process seeks to make the best possible use of the nodes' hardware resources to achieve the desired **QoS** (quality of service) while being frugal, both money and energy-wise. In this statement, it's important to mention that although it may have different definitions, the **QoS** is usually approached as the time taken to process a hardware resource request[143].

On the other hand, some scientific works optimize virtual entities' (re)deployment to improve and adapt the scheduling processes to emerging architectures. For example, to optimize the **VMs** deployment, some papers propose the proper, dynamic (resources capacities and duration) and anticipated creation of **VMs** based on the historical scheduling data[245], the applications' workflow[192], and the type of the involved tasks[245][192]. Other works make use of external/internal clouds resources (hybrid cloud) to deploy tasks based on internal **VMs'** overload situations, energy/cost considerations, tasks' delay bounds, and learning methods[252][241][185]. For their part, other approaches argue for applying linear programming models and graph theory to optimize the **VMs** deployment, taking into account variables such as **VMs'** resources requirements, resources availability, non-sliceable resources management, and in-host **VM** interference[143].

Furthermore, some approaches rely on serverless networks, such as the one proposed by Kirschnick et al.[131]. They propose a deployment architecture in decentralized networks of **nodes/VMS**. Herein, in any node that executes the system service, the user must provide the system specification considering the

desired application with its service dependencies. Then, the algorithm running in this initial node performs a breadth search to determine which **peers/VMs** can run the described software components. Finally, the system starts working when the initial node receives a positive notification of all the concerning **peers**.

In addition to studying the **initial deployment**, other approaches analyze different ways to perform **execution-time scheduling** operations. For example, Beenish et al.[89] migrate **VMs** to avoid overload situations and to shut down nodes in underload situations. For that, they consider thresholds in terms of **RAM** and **CPU**. Furthermore, they execute a matching algorithm based on the first elements of two lists to perform an energy-efficient deployment. On the one hand, **VMs** ordered in decreasing order considering their **CPU** requirements. On the other hand, a list of nodes in decreasing order considering their **CPU** MaxCapacity/Utilization Level and their peak/current energy expenditure. This algorithm has an algorithmic complexity of $O(nNodes \times \log(nNodes) + deployTime + nVMs \times \log(VMs))$.

Another interesting approach is the one proposed by Zhu et al.[251]. They use the rolling-horizon (in fact, planning ahead) optimization for task scheduling in the cloud. For this, the authors sort the tasks according to their deadlines. Then, they determine in advance whether or not a task can be properly terminated considering the current hardware resources of the deployed **VMs** in a set of nodes. If there are no candidates, the concerning nodes are load-balanced by performing **VMs** migration-to-create to retry the task deployment.

It is important to say that this entire process considers optimizing energy consumption. On the one hand, the deployment strategy selects the **VM** currently yielding minimal **CPU** energy consumption to execute tasks. On the other hand, it frees up hosts to shut down in a process called *scale down*. For the latter, the algorithm matches **VMs - nodes** by sorting the source hosts (hosts to free) by their **CPU** utilization in increasing order and the destination hosts oppositely. The entire task allocation process and the *scale down* operations have a complexity of $O(N + N \times \log(N))$ and $O(N^2)$ respectively.

In the same line of thought, many works seek to optimize the **initial deployment** and the **execution-time scheduling** through learning techniques. For the **initial deployment**, they focus on characterizing task types for scheduling monolithic or modular applications in hybrid or single clouds. In general, they analyze the applications' consumption history and their resource behavior variation to predict when, how often, and what resources they will need. Then, for the **execution-time scheduling**, they use the same variables in order to predict scaling (duplicate, close) and migration operations. All these procedures aim to optimize load balancing, monetary expenditures, system scalability, hardware resource usage, and service level agreement assurance[250]. For instance, Zhiheng et al.[249]

| INITIAL DEPLOYMENT IN THE CLOUD- OPERATIONS AND FEATURES - PART 1 | | | | | | |
|---|---|---|---|---|---|---|
| **Ref.** | **Input Variables** | **Operations** | **Approach** | **Require pre-processing/ learning** | **Consider special energy-related hardware features of CPU, RAM, HDD, and NIC.** | **Study computational complexity** |
| [245] [192] | - Scheduling historical Data<br>- Taks' HW load/needs<br>- VMs' HW resources<br>- Applications' workflow | - Early/dynamic creation of VMs<br>- Proper VMs-tasks matching. | Centralized | YES | NO | NO |
| [252] [241] [185] | - VMs' HW resources<br>- Taks' hw load/needs<br>- Tasks' deadline<br>- Price and energy expenses. | - Perform learning/prediction methods<br>- Deploy tasks to internal/external clouds | Centralized | YES | NO | NO |
| [143] | - VMs' HW resources<br>- Nodes' resources ((non)sliceable) | - Perform linear programing and graph-based methods<br>- Proper VMs-nodes matching. | Centralized | YES | NO | NO |
| [89] | - VMs' HW resources<br>- Nodes' resources | - Sort nodes and VMs based on hardware load to match efficiently. | Centralized | YES | YES:<br>CPU' MIPS and RAM's states | YES |
| [251] | - VMs' AND nodes HW resources<br>- tasks' deadline | - Sort nodes based on hardware load and tasks based on deadlines to match efficiently<br>- Apply rolling horizon optimization for deployment. | Centralized | YES | YES:<br>CPU' MIPS | YES |
| [78] [76] [82] | - Tasks HW resources<br>- Nodes' resources | - Create task queues<br>- Deploy tasks in order of arrival and proportionality of resources. | Centralized | NO | NO | NO |
| **Our Work** | **- Nodes' HW resources.<br>- Applications' HW consumption.** | **- Scheduling based on indexation in multidimensional data structures.** | **Centralized and Decentralized** | **NO** | **YES** | **YES** |

**Table 3.1:** Cloud initial-deployment approaches: Input Variables | Operations performed - PART 1

| INITIAL DEPLOYMENT IN THE CLOUD-Optimized/Analyzed Variables-PART 1 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Ref.** | **HW res. usage** | **Deploy. time** | **QoS** | **Energy cons.** | **Monet. costs** | **Scalab.** | **High avail.** | **Efficient search/ insert/ delete for sched.** |
| [245] [192] | X | X | X | | X | | | |
| [252] [241] [185] | X | X | X | X | X | X | | |
| [143] | X | X | X | | X | | | |
| [89] | X | X | X | X | | | | |
| [251] | X | X | X | X | | X | | |
| [78] [76] [82] | X | | X | | | X | X | |
| **Our work** | **X** | **X** | **X** | **X** | | **X** | | **X** |

**Table 3.2:** Cloud initial-deployment approaches: Target Variables - PART 1

apply the **k-means** algorithm to classify tasks based on **CPU**, **RAM**, **network**, and **storage consumption**. They then deploy long-running and short-running applications, taking into account idle resources and ranking functions such as the Least Requested priority (lowest resource utilization first). Then, for **execution-time scheduling**, they use the *Shortest Runtime Rescheduling algorithm* to move the lightest application that has been running for the shortest time, thus ensuring a reduced impact on the **QoS**. For its part, the approach of Haitao et al.[244] is also based on the prediction of resources for the deployment and migration (for them, destroy and redeploy) of containers for video processing. For this, they use service similarity matching and a time-series nearest neighbor regression on the consumption data of the corresponding containers.

Other works seek to migrate virtual machines, especially for energy-saving purposes. For instance, Azmy et al.[14] create a power panel that defines the cluster's power states. On the one hand, these states consider **VMs'** allocation policies based on thresholds or medians. On the other hand, they consider migration policies such as the Maximum Correlation policy (**VMs** that consume the most in the host), Minimum migration time policy, and Random selection policy. The authors' objectives are to avoid physical nodes' overload conditions and/or achieve underload hosts shutdown. Similarly, Siddavatam et al.[198] pursue the same goal

| INITIAL DEPLOYMENT IN THE CLOUD- OPERATIONS AND FEATURES - PART 2 | | | | | | |
|---|---|---|---|---|---|---|
| **Ref.** | **Input Variables** | **Operations** | **Approach** | **Require pre-processing/ learning** | **Consider special energy-related hardware features of CPU, RAM, HDD, and NIC.** | **Study computational complexity** |
| [136] [247] [140] [240] | - Containers HW resources.<br>- Containers constraints. | - Filter nodes based on specified criteria<br>- Sort nodes based on HW resources and specified criteria. | Centralized | YES | NO | NO |
| [86] | - Nodes' HW resources.<br>- Tasks' HW load. | - Use linear equations to deploy and equitably deliver resources based on dominant resources. | Centralized | YES | NO | NO |
| [131] | - System/ Application specification. | - Create a component-based tree.<br>- Perform breadth search in the tree. | Decentralized | NO | NO | NO |
| [248] | - Task execution time.<br>- Nodes' HW resources. | - Sort tasks and nodes.<br>- Perform matching: Most utilized instance with enough resources. | Centralized | YES | NO | YES |
| [250] [249] [244] | -Applications' consumption history.<br>-Applications' Behavior. | - Execute prediction algorithms. | Survey: Centralized and Decentralized | YES | NO | YES |
| [191] | - Connected containers graph.<br>- Nodes Graph.<br>- Nodes'/ Containers HW resources. | - Linear Programming Minimization. | Centralized | YES | NO | NO |
| [203] | - Nodes' HW resources.<br>- Containers' HW load. | - Random selection<br>- Selectionbased on HW usage and container number. | Centralized | NO | NO | NO |
| **Our Work** | **- Nodes' HW resources.<br>- Applications' HW consumption.** | **- Scheduling based on indexation in multidimensional data structures.** | **Centralized and Decentralized** | **NO** | **YES** | **YES** |

**Table 3.3:** Cloud initial-deployment approaches: Input Variables | Operations performed - PART 2

| INITIAL DEPLOYMENT IN THE CLOUD-Optimized/Analyzed Variables-PART 2 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Ref.** | **HW res. usage** | **Deploy. time** | **QoS** | **Energy cons.** | **Monet. costs** | **Scalab.** | **High avail.** | **Efficient search/ insert/ delete for scheduling** |
| [136] [247] [140] [240] | X | | X | | | X | X | Linear search |
| [86] | X | | X | | | X | | |
| [131] | | X | | | | X | X | X |
| [248] | X | X | X | X | X | X | X | |
| [250] [249] [244] | X | X | X | X | X | | | |
| [191] | X | | X | X | | | | |
| [203] | X | | X | | | X | X | |
| **Our work** | **X** | **X** | **X** | **X** | | **X** | | **X** |

**Table 3.4:** Cloud initial-deployment approaches: Target Variables - PART 2

using a similar migration strategy, but based only on **CPU** usage. For their part, Pawlish et al.[182] aim to avoid overhead costs. They migrate tasks from local data centers to clouds if the average **CPU** load of its nodes is greater than a certain percentage threshold.

On the other hand, important companies have developed technologies that optimally manage the clusters' hardware resources to achieve specific objectives. For instance, several of them seek to improve massive data processing in parallel using nodes in a structured way. That is the case for those who use the **MapReduce** paradigm. It uses hash operations to parallel distribute and process data of common characteristics in common nodes known as mappers and reducers. Here, the containerized tasks' execution order and the corresponding resources allocation can be based on structured methods. For example, in the **FIFO** approach, each task is executed by priority and in order of arrival. On the other hand, the **fair-capacity** scheduling creates different queues of tasks, assigning them a proportional, convenient, and fixed amount of resources[78][76][82].

As another example, one of the best-known technologies for container management is **Docker**. This platform allows developers to easily deploy, edit or delete their applications in **GNU/Linux-based containers**[203]. **Docker** schedules con-

| EXECUTION-TIME SCHEDULING IN THE CLOUD - OPERATIONS AND FEATURES | | | | | | |
|---|---|---|---|---|---|---|
| **Ref.** | **Input Variables** | **Operations** | **Approach** | **Require pre-processing/ learning** | **Consider special energy-related hardware features of CPU, RAM, HDD, and NIC.** | **Study computational complexity** |
| [89] | - Nodes' load thresholds set. | - Migrate, close and deploy VMs. | Centralized | NO | YES: CPU' MIPS and RAM's states | YES |
| [251] | - VMs' HW resources.<br>- tasks' deadline. | - Migrate, close and deploy VMs. | Centralized | NO | YES: CPU' MIPS | YES |
| [136] | - Containers affinity, tolerations, etc.<br>- Scalability, high availability variables.<br>- Custom schedule criteria. | - Migrate, destroy, start, duplicate containers. | Centralized. | NO | NO | Linear search |
| [248] | - Node's hardware load.<br>- Node's load time.<br>- Images size.<br>- Bandwidth. | - Migrate, destroy, start, duplicate containers. | Centralized | NO | NO | YES |
| [14] [198] | - Node's hardware load.<br>- Applications' Images size.<br>- Bandwidth. | - Migrate VMs/tasks. | Centralized | NO | NO | NO |
| [250] [244] | -Applications' consumption history.<br>-Applications' Behavior. | - Predict optimal execution-time scheduling operations.<br>- Perform execution-time scheduling. | Centralized and Decentralized | YES | NO | YES |
| [249] | - Node's hardware load.<br>- Applications' Images size.<br>- Applications' running time. | - Migrate the lightest application with the shortest running time. | Centralized | NO | NO | NO |
| [203] | - Node's hardware load<br>- Nodes health | - Migrate container to the node with enough resources | Centralized | NO | NO | NO |
| **Our Work** | **- Nodes' HW resources.<br>- Applications' HW consumption.** | **- Scheduling based on indexation in multidimensional data structures.** | **Centralized and Decentralized** | **NO** | **YES** | **YES** |

**Table 3.5:** Cloud execution-time approaches: Input Variables | Operations performed

| EXECUTION-TIME SCHEDULING - Optimized/Analyzed Variables | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Ref.** | **HW res. usage** | **Deploy. time** | **QoS** | **Energy cons.** | **Monet. costs** | **Scalab.** | **High avail.** | **Efficient search/ insert/ delete for scheduling** |
| [89] | X | | | X | | X | | |
| [251] | X | | X | X | | X | | |
| [136] | X | | X | | | X | X | Linear search |
| [248] | X | | | X | X | X | | |
| [14] [198] | X | | | X | | | | |
| [250] [244] | X | | X | X | X | X | X | |
| [249] | X | | X | | | X | | |
| [203] | X | | X | | | X | X | |
| **Our work** | **X** | **X** | **X** | **X** | | **X** | | **X** |

**Table 3.6:** Cloud execution-time approaches: Target Variables

tainers on a cluster's physical nodes using tools like **Docker Swarm**. This tool offers centralized algorithms for cluster management such as automatic load balancing, high availability, scalability, roll-back execution, etc. In general, for the **initial deployment** of a container, the manager device supports by default three approaches based on resources and the number of containers: 1) The random selection of a node that contains enough resources, 2) the selection of nodes with more containers to avoid application fragmentation and 3) The choice of nodes with fewer containers to improve load balancing. Then, in order to maintain proper load balancing and high availability, **Docker Swarm** destroys containers and redeploys them on nodes with sufficient hardware resources. **Swarm** allows developers to define their own reschedule policies[8][20].

**Kubernetes**, for his part, groups containerized tasks in entities called **PODs**. Then, it seeks to schedule **PODs** on physical nodes or **VMs** according to the tasks' constraints and/or specifications. For that, **Kubernetes** uses 1) **Strong** task constraints, with which the scheduler filters available nodes by creating a list of candidates, and 2) **Soft** task constraints, with which the scheduler scores and sorts the mentioned list especially considering nodes **CPU** and **RAM**. Here, the first element in the list is the node on which the task will run[136][15]. Also, it is

important to say that to perform execution-time scheduling on **PODs** (migrate or duplicate), **Kubernetes** uses a controller called **ReplicaSet**. This special process monitors the **PODs'** status based on a linear search of all their instances identified with certain labels in the entire cluster[135]. Finally, **Kubernetes** allows running different **PODs-node** matching criteria such as affinity, selectors, tolerances, or taints.

It is important to mention that some scientific works seek to improve certain **Kubernetes** methods. For example, some approaches avoid the execution of the scheduling algorithm when there is only one node. They consider additional score criteria such as network bandwidth, disk storage[247], and GPU[240]. Other works improve the **Kubernetes** scheduling by scaling the multi-resource approach to the multi-clusters environments[140]. On the other hand, some authors use the **Kubernetes** base system to create different services and reimplement the default scheduling approach. These services aim to execute different complex deployment and in-execution heuristics.

An important example of the last point is the one proposed by Zhiheng et al.[248]. They implement four entities to perform scheduling approaches: 1) The **Resource Profiler**, which maintains an updated snapshot of **VMs** and **tasks**, 2) the **Cloud Adaptor**, which performs the deployment and migration operations, 3) the **Task Packer** entity, which makes scheduling decisions, and 4) the **Instance cleaner** which maintains only useful instances alive.

In their system, the authors ensure that migrations are executed while maintaining the running state of the containers. On the other hand, they achieve intelligent scheduling and elastic clusters by creating special entities and services. For that, they predict the execution time of the tasks to be deployed and group them into two sets: 1) **Permanent execution tasks** (long-running) and 2) **temporary execution tasks** (batch jobs). Then, they deploy the first group to nodes that imply low energy and monetary cost (time cost per billing) and the second group to nodes with convenient hardware features. Thereby, to treat the **temporary execution tasks**, they arrange each **POD** to the most utilized node instance with enough requested resources in terms of **CPU**, **RAM**, and execution time. Then, to scale and shut down some nodes, they select those being used at less than 50% capacity. The algorithm releases them using migration operations considering virtualized images, execution time, and available bandwidth.

It is important to say that the authors show a linear algorithmic complexity for their deployment and scaling algorithm.

On the other hand, some technologies dynamically manage the different physical nodes used/shared by different cluster schedulers from a higher level of abstraction. That is the case with the **Apache Mesos**[80] distributed kernel. Using a master

entity, **Mesos** receives the available resources from each physical node and manages them among the different running cluster schedulers (two-level scheduling)[87]. For that, **Mesos** utilizes the **DRF**[86] (Dominant Resource Fairness) algorithm by default. It uses linear equations to ensure that: For each (non)containerized (default: **Mesos** container) task, the percentage of its dominant resource (the most demanded one) type that it gets cluster-wide is the same for all other tasks[159].

It's important to mention that some technologies enrich the **Mesos** scheduling approach. For instance, **Aurora**[75] runs on the top of **Mesos** kernel in order to perform containerized task grouping, scalability operations, set replica criteria, etc. For its part, **Marathon**[79] is able to launch tasks at the desired times using the **Chronos** scheduler[77].

Following the same goals as **Kubernetes** or **Swarm**, some scientific works seek to efficiently schedule containers in hardware resources[4]. For example, Rodrigues et al.[191] optimize container deployment through Mixed Integer Linear Programming. For this, they study the graph of physical nodes of a cluster, and the graph made up of the connected containers deployed in it. Regarding resources, they analyze the **RAM**, the **CPU**, and the transfer rate involved in both graphs to create weights for each edge and vertex. Then, they determine the most optimal deployment by minimizing an objective function that considers the quality of service and energy consumption. Another interesting approach is the one proposed by Pongsakorn et al.[222]. They focus on container migration based on four stages. First, classify containers into groups of high and low duration. Secondly, group the long-lived containers based on their allocated resources and hosts. Next, on the one hand, find a pair of hosts whose difference in resource usage is significant. On the other hand, select the containers whose difference between allocated and used resources is also significant. Last, perform a swap of both containers to balance the load.

Finally, another group of works seeks to optimize the communication between deployed tasks. For example, Wu et al.[232], in addition to a redeployment algorithm, they propose a routing improvement of multicast tasks in data centers of different zones based on optimization techniques. This area of study is beyond the scope of our research work.

### 3.3.1 Analysis and considerations

So far, we have explained some works whose objective is to figure out the convenient **initial deployment** and **in-execution scheduling** of virtual entities in **cloud clusters**. We have selected and analyzed several approaches proposed by

scientists and applied by well-known platforms and tools. We believe that overall, the approaches analyzed here represent a strong trend of approaches today.

Following the methodology described in section 3.2, we study this trend from an algorithmic approach. To do that, in Table 3.1, 3.2, 3.3 and 3.4 (Tables 3.3 and 3.4 are the continuation of Tables 3.1 and 3.2), we characterize the works concerning the **initial deployment** of applications. Tables 3.1 and 3.3 describe the approaches' input variables, operations performed, or their type of policy (centralized/non-centralized). Tables 3.2 and 3.4 illustrate the variables they analyze and/or optimize, such as **QoS** or energy consumption.

Then, in Tables 3.5 and 3.6, we characterize the works concerning **execution-time scheduling** considering the same analysis fields.

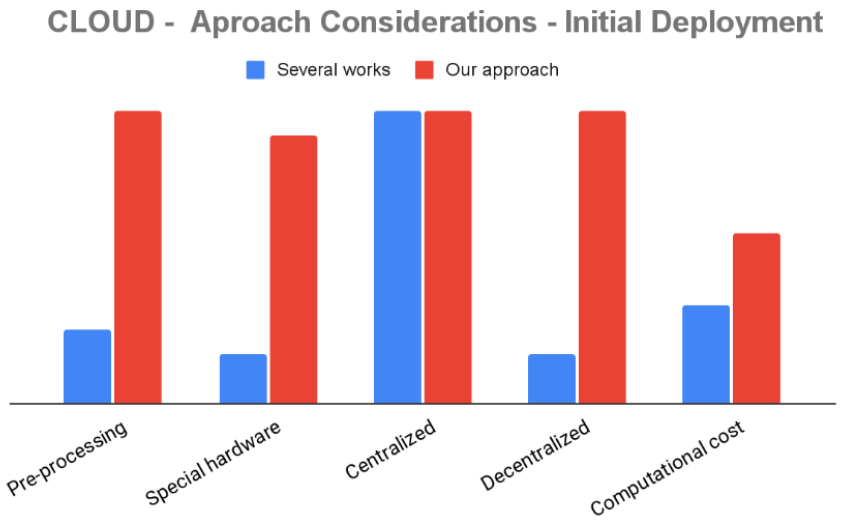


**Figure 3.2:** Cloud computing: Initial Deployment Considerations Trend

As shown in Figure 3.2, most works are based on pre-processing techniques to **deploy** a software component ideally. In our case, our goal is not to ensure the best possible initial deployment. Instead, we use a distributed data structure to have a stable system, which on the one hand, is reactive to hardware load changes, and on the other hand, does not need to perform expensive calculations to work.

Then, most approaches do not consider the power characteristics of several hardware components to perform scheduling decisions. For our part, we believe that we can profile hardware and software components from mathematical models and operating system interfaces (see chapter 2). That allows us to have a more extensible and clear view of a system's energy consumption. For example, we

study the **CPU**, **RAM**, network, and storage components' features to perform our scheduling algorithm.

Next, we can see that most works use centralized models to schedule. That is due to ease of components control, stability, and scalability, among other reasons[168]. About this point, we believe in the elasticity of structured environments. Thereby, although centralized distributed multidimensional data structures are applicable for scheduling, we have preferred to implement our algorithm in a **P2P** system. In addition to validating it for different deployment levels (cloud, grid, etc.), our goal is to demonstrate that said architecture is an excellent candidate for energy savings purposes. Moreover, we believe that **P2P** networks in the cloud would offer many advantages such as security, privacy, autonomy, etc.

Finally, most works do not consider the computational cost of their algorithms. In our case, we consider it important since our approach's algorithmic behavior is reflected in the number of network operations.

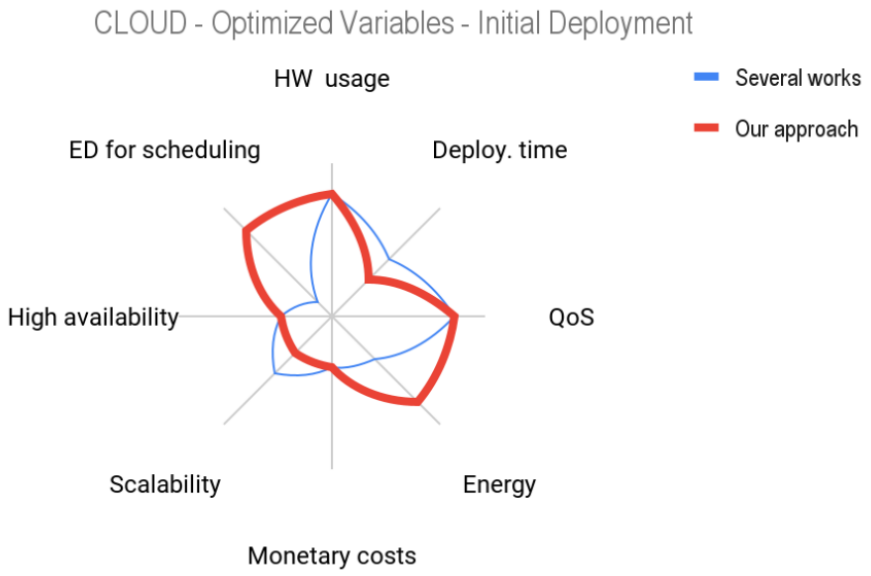


**Figure 3.3:** Cloud computing: Initial Deployment Variables Trend

On the other hand, Figure 3.3 shows the trend of the approaches' **analyzed / optimized** variables. Here, most of the works seek to correctly manage hardware resources to mainly improve scalability, quality of service, monetary costs, and the efficient deployment of applications.

For us, our main objective is to save energy without losing the notion of quality

of service. To do this, we efficiently use multidimensional data structures based on a correct analysis of hardware resources load. Furthermore, our technique allows us to consider all the other variables indirectly. For example, the scalability of our approach is addressed in the overlay used; the deployment time depends on the nature of the data structure; the monetary costs are related to power consumption, and the high availability is related to the mirroring operations we consider.


**Figure 3.4:** Cloud computing: Execution-time scheduling Considerations Trend

Regarding the **execution time scheduling**, in Figure 3.4, we show that, unlike the **initial deployment** stage, most works do not base their approaches on pre-processing operations. We believe this is because migration scenarios tend to be much more dynamic and less predictable. Another significant difference is a major concern here for the computational cost. We believe that this is because the number of operations through the network influences energy consumption, quality of service, and others.

We address both aspects with the efficient use of spatial data structures since it allows us to analyze computational costs and maintain a dynamic system. The analysis of the rest of the variables is the same as we did for Figure 3.2.

Regarding the trend of the **analyzed / optimized** variables at the **execution-time** scheduling, Figure 3.5 shows that, unlike the **initial deployment**, the works have a higher tendency to evaluate the energy consumption and scalability variables.

In our case, energy consumption is our primary objective. However, we indirectly manage the system's scalability performance through our chosen data structure. We analyze the rest of the variables in the same way as we did for figure 3.3.

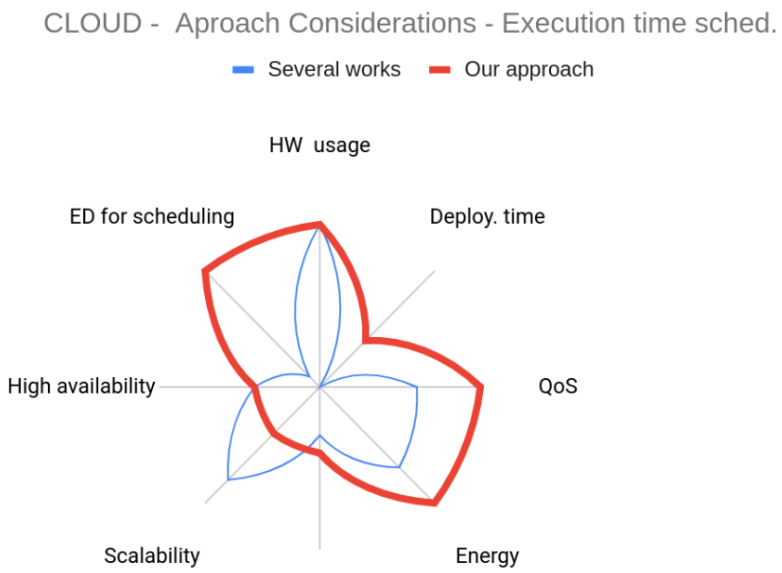


**Figure 3.5:** Cloud computing: Execution-Time scheduling Variables Trend

The next section analyzes several works that seek to optimize the **initial deployment** and the **in-execution scheduling** in devices beyond the cloud.

## 3.4 Scheduling close to data sources.

As seen in the previous section, cloud computing studies data processing and the execution of different workloads in clusters of (generally) homogeneous nodes. However, these nodes, usually located in data centers, are often physically far from the concerning data sources. This fact represents a load management problem since, as mentioned in **IEEE innovation**[102], more and more connected devices generate data to process. For instance, billions of them generated an unprecedented amount of data in the last year, increasing the difficulty of managing bandwidth (despite 5g, network capability is still heterogeneous globally), energy consumption, and computing resources are growing[102].

For these reasons, scientists and enterprises developed different systems hierarchies. They seek to take advantage of the computational capabilities of nodes closer to the data and workloads sources. That is the case of **edge**, **fog**, and **grid computing**.

Regarding the definitions of each, some scientists consider the first two as syn-

onyms[175][96]. For them, both approaches consist of processing workloads considering two types of closeness. On the one hand, closeness to the concerning data source. On the other hand, closeness to where the actions related to that data are taken[36]. Others, however, define them differently[101]. They mention that one (**fog** or **edge**) is a kind of interface between **edge** devices and the cloud.

Concerning **grid** computing, we did not find a universal definition either. Some authors define it as a paradigm in which heterogeneous devices connected from different places perform tasks without centralized control[17][117].

This section indistinctly analyzes the works that study scheduling techniques in these three areas. The reason is that, unlike the cloud level, the three involve the management of software components in probably very heterogeneous hardware, in much more diverse and probably dynamic[25] locations, and with a greater tendency towards decentralized philosophies. To do said analysis, this section applies the chapter's methodology, studying the techniques concerning the **initial deployment** and the **execution-time** scheduling phases.

Among the works mentioned in the previous paragraph, some focus on improving the overall performance of applications running on heterogeneous devices. For example, Manjot[17] seeks to achieve this goal by improving the tasks' **initial deployment** process. For this, he maintains a centralized entity that stores all the **memory** and **CPU** characteristics of the involved nodes. He then displays each task considering two perspectives: 1) a first-come, first-served basis or 2) applying a heuristic based on the time a task waits for a resource, the time that task typically uses that resource, and the completion time of the job.

Studying the same objective, Chauhan et al.[27] conceive a dependencies-based application as a directed acyclic graph and a set of nodes as a structured overlay. Using these, they propose a fully decentralized scheduling algorithm that aims to execute in parallel the largest number of application modules of the same dependencies level. This algorithm maintains two types of information in the node which starts an application (it can be any node on the network): 1) A table with information on its neighbors such as name, **IP**, **CPU MPIS**, running tasks and subtasks, etc., and 2) the execution characteristics of the application. Then, the node deploys either any task of the same level of dependencies or the longest task of a higher level of dependencies on the most suitable node. This selection considers the peer's load, processing capacity, task dispatch time, previous processing information, etc. For this, the authors propose heuristics based on weights.

In another of their works, the authors use similar heuristics to implement high availability strategies. In them, the nodes monitor their neighbors to check that they are still working. If this is not the case, the involved task list is forwarded to available nodes, and their corresponding tables are updated accordingly[26].

In addition to improving application performance, other researchers consider maintaining the network balance during the **initial deployment** process. For example, Dias et al.[132] choose the best node to deploy an application from a central entity. To do that, they consider variables such as deployment time, download queue, and bandwidth involved. For their part, Drost et al.[63] execute in parallel the tasks started at a node of a decentralized network. For that, the initial node broadcasts with a specified depth looking for a peer with the necessary resources to execute one of the tasks. The nodes that have such resources accept the request. Those who do not, as well as those who receive the request twice, deny it. The algorithm achieves the network balance considering the physically closest candidates to the origin node. Finally, pursuing the same goal, Maheswaran et al.[72] make a device capable of connecting to several fogs that can fulfill their workloads. The algorithm approach considers the criteria of resource availability and network latency.

Otherwise, some works base their deployment approaches on learning techniques. For example, Lordache et al.[118] consider a set of nodes capable of executing the tasks to be deployed and/or executing scheduling operations based on genetic algorithms. Thereby, each node works as an agent that can process/designate tasks on the same processor unless they have exclusivity restrictions. Then, the first population is initialized stochastically, and the tasks are assigned to specific processors. Thereupon, the deployment space is explored by starting chromosomes using random generators. Here, 1) each agent displays different probability distributions such as normal or Poisson, and 2) the fitness function is defined by the number of processors and the total execution time of all tasks assigned (previous and current) to each processor. For their part, Toka et al.[219] focus on improving the horizontal scaling of Kubernetes edge clusters (explained in the previous section). To achieve this, instead of relying on the current scaling period's observations, the authors perform scaling decisions based on machine learning algorithms. These analyze request intensity over time, considering the **CPU** as the main component.

Besides studying the deployment of processes, other types of work focus on improving the localization of data to process. For example, Neumann[169] et al. propose an architecture for **P2P** networks called *STACEE*. In it, smart services enable data characterization based on its lifetime, intermittency, and access quantity. It allows the definition of treatment metrics based on monetary costs, energy consumption, or data proximity. Furthermore, *STACEE* offers an objective-functions minimization model, which comprises variables of existing hardware (energy costs, **CPU** available, prices, et.) and customer requirements (content size, release date, distance, et.). All this aims to deploy and migrate data and processes optimally.

For their part, other works improve data access and management through clus-

tering techniques[13]. For example, Li et al.[142] proposes that all the nodes of a **P2P** network are indexed in a multi-dimensional Cartesian plane divided into zones. For that, one or more peers have the bits-string-based metadata of each zone. Then, the latter are grouped hierarchically (clustering process) so that the whole network is seen as a binary tree for efficient queries. This tree is known as **VPTree**.

Finally, other scientists focus on optimizing data caching on **edge** devices. About this point, Shuja et al.[197] propose a wide survey that describes the approaches concerning the when, what, where, and how to cache. They categorize the works according to whether they focus on supervised, unsupervised, reinforcement, transfer learning, and neural networks. This document explains, for example, that to do caching at the **edge** and to predict data mobility and popularity, most authors use reinforcement learning and neural networks. That is because the popularity of content can be learned from no data or little data sets. On the other hand, the authors describe the methods to increase the cache hit rate. This objective is one of the most sought-after and can be achieved with supervised learning, clustering techniques, etc. Moreover, the authors explain some device-to-device communications techniques for energy savings or transmissions balance. We do not develop this point further since, although we assume data management, the optimization of their treatment is outside the scope of our research work.

Other researchers consider **execution-time scheduling** in order to achieve load balancing objectives. For example, Charantola et al.[25] perform migration operations when the **cloudlets** (small-scale datacenter) close to the respective data sources are saturated. The authors intelligently select which applications should be migrated to the cloud whenever a **cloudlet** is overloaded. They base this selection on the applications' connections, the users consuming them, and the **CPU** load/requirements. For their part, Puthal et al.[189] propose a system in which each edge-cloudlet is recorded in a central entity in the cloud. Then, each cloudlet broadcasts to register all other peers performing and offering all necessary security methods. Thereby, if a cloudlet enters an overload condition, it performs a breadth search for resources. It broadcasts its identity and loads data, looking for a peer able to execute part of this load. If the security filters are reached, and another peer has sufficient resources, the cloudlet performs migration operations.

On the other hand, other approaches consider load balancing of **IoT** devices. That is the case of Chhikara et al.[32] who deploy an energy-savings container migration strategy for **IoT** networks. Their proposal applies the **K-means** and **hierarchical clustering** algorithms to classify hosts. They consider three groups that define three host states: overload, underload, and load-balanced state. They then select the first two groups to balance their load by freeing them from their

| INITIAL DEPLOYMENT IN THE EDGE/FOG/GRID - OPERATIONS AND FEATURES | | | | | | |
|---|---|---|---|---|---|---|
| **Ref.** | **Input Variables** | **Operations** | **Approach** | **Require pre-processing/ learning** | **Consider special energy-related hardware features of CPU, RAM, HDD, and NIC.** | **Study computational complexity** |
| [17] | - Nodes' CPU and memory capacity.<br>- task: Waiting time for a resource, estimated time of resource use | Apply deployment heuristics | Centralized | YES | NO | NO |
| [27] | - Nodes' CPU capacity.<br>- task: Dependencies (Graph) and requested resources.<br>- Nodes overlay | Deploy tasks of the same dependency level on different nodes | Decentralized | NO | NO | NO |
| [118] | - Nodes' CPU capacity.<br>- Nodes' load.<br>- Taks' CPU consumption. | Apply genetic algorithms to find the best tasks deployment | Decentralized | YES | NO | NO |
| [169] | - Nodes' CPU, Storage, and network capacity.<br>- User requirements<br>- Data Characteristics | - Minimization of objective functions.<br>- Efficient deployment of data | Decentralized | YES | NO | NO |
| [13] [142] | - Data index<br>- Node's index | - Execution of clustering techniques | Decentralized | YES | NO | YES/ Multid. data structures |
| [132] | - Size of the component to display.<br>- Nodes' Bandwidth.<br>- Node queue. | - Deployment based on time and resources | Centralized | YES | NO | NO |
| [197] | - Data history.<br>- Node Resource Usage.<br>- Bandwidth Usage.<br>- Cache hit level. | - Application of supervised, unsupervised, reinforcement or transfer learning methods. | Centralized/ Decentralized | YES | NO | NO |
| [63] | - Nodes' CPU, disk, and RAM capacity.<br>- Workloads' CPU, disk, and RAM requirements. | - Search for peer resources by broadcasting.<br>- Deployment and execution of tasks in parallel | Decentralized | NO | NO | NO |
| **Our Work** | **- Nodes' HW resources.<br>- Applications' HW consumption.** | **- Scheduling based on indexation in multidimensional data structures.** | **Centralized and Decentralized** | **NO** | **YES** | **YES** |

**Table 3.7:** GRID/FOG/EDGE initial-deployment approaches: Input Variables | Operations performed

| INITIAL DEPLOYMENT IN THE EDGE/FOG/GRID - Optimized/Analyzed Variables | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Ref.** | **HW res. usage** | **Net. Bal.** | **QoS/ resp. time** | **Energy cons.** | **Monet. costs** | **Scalab.** | **High avail. / Data Mgmt.** | **Efficient search/ insert/ delete for scheduling** |
| [17] | X | | X | | | | | |
| [27] | X | | X | | | | | |
| [118] | X | | X | | | | | |
| [169] | X | X | X | X | X | | X | |
| [13] [142] | X | | | | | | X | |
| [132] | X | X | | | | | | |
| [197] | X | X | X | X | | | | |
| [63] | X | X | X | | | | | |
| **Our work** | **X** | | **X** | **X** | | **X** | **X** | **X** |

**Table 3.8:** GRID/FOG/EDGE initial-deployment approaches: Target Variables

most **CPU/RAM**-consuming containers. A list of new nodes in the system is sorted according to **CPU** usage.

Using clustering methods as well, Elbamby et al.[68] migrate load from the end devices to the cloudlets to improve the energy and computing resources management. They propose a decentralized method that groups nodes into disjoint sets based on spatial proximity and mutual interest in certain tasks. Then, the authors get a task popularity matrix to deploy a joint task distribution and caching scheme. In it, each cloudlet hosts as many results as possible, replacing old and less frequently used data with new insertions. That seeks to optimize the latency time of the final device.

Another way to migrate load from end devices to the edge is through a collaborative system such as the one proposed by Zhang et al.[243]. In their proposal, fog nodes register as volunteers in the non-centralized system. The system then accepts the load transaction: *End device- volunteer node* based on the energy consumption, delay history, and computing capabilities.

On the other hand, some authors also perform **Execution-time scheduling** to balance network load and manage physical distances. For example, Puliafito et al.[188] keep **IoT** devices as close as possible to the edge containers they use. When an **IoT** device moves away from the container that consumes more than a certain

| EXECUTION-TIME SCHEDULING IN THE EDGE/FOG/GRID - OPERATIONS AND FEATURES | | | | | | |
|---|---|---|---|---|---|---|
| **Ref.** | **Input Variables** | **Operations** | **Approach** | **Require pre-processing/ learning** | **Consider special energy-related hardware features of CPU, RAM, HDD, and NIC.** | **Study computational complexity** |
| [25] | - Cloudlets' hardware load.<br>- Components CPU cons. | - Smart migration to cloud | Centralized | NO | CPU MIPS | NO |
| [26] | - Nodes' CPU capacity.<br>- task: requested resources.<br>- Neighbor nodes health signal. | - Monitor the health of neighboring nodes.<br>- In case of nodes dead, redeploy the tasks involved in other nodes. | Decentralized | NO | CPU MIPS | NO |
| [169] | - Nodes' CPU, Storage, and network capacity.<br>- User Reqs.<br>- Data Char. | - Migration and duplication of data and processes. | Decentralized | YES | NO | NO |
| [219] | - PODs' CPU usage over time.<br>- Workload data over time.<br>- PODs quantity. | - Application of machine learning techniques.<br>- PODs scaling (including scale down) | Centralized | YES | NO | NO |
| [32] | - Nodes CPU and RAM usage.<br>- Containers load. | - Cluster nodes by their load.<br>- Order nodes by their load.<br>- Migrate containers | Centralized and Decentralized | YES | NO | YES |
| [189] | - Edge Cloudlet load. | - Perform a broadcast to find another peer and negotiate load migration | Decentralized | NO | NO | NO |
| [68] [72] | - Power and resource status of end devices.<br>- Spatial proximity of nodes.<br>- Node interest in common tasks.<br>- Data Usage Frequency. | - Migrate load from end devices to cloudlets.<br>- Execute clustering technique for cache and data popul. OR<br>- Dynamically connect to specific fogs according to network load | Decentralized | YES | NO | NO |
| [188] | -Distance between container provider and consumer devices.<br>- Nodes CPU and RAM capacity. | - Migrate containers from one edge node to the other peer that is closest to the IoT device | Centralized | NO | NO | NO |
| [243] | - Nodes' HW capacity.<br>- task: requested resources.<br>- Delay of tasks | - Fog node: Volunteer.<br>- Migrate load: end device - fog node | Decentralized | NO | CPU Joule/cycles Transmission power: Shannon capacity for wireless | NO |
| **Our Work** | **- Nodes' HW resources.<br>- Applications' HW consumption.** | **- Scheduling based on indexation in multidimensional data structures.** | **Centralized and Decentralized** | **NO** | **YES** | **YES** |

**Table 3.9:** GRID/FOG/EDGE execution-time scheduling: Input Variables | Operations performed

| EXECUTION-TIME SCHEDULING IN THE EDGE/FOG/GRID - Optimized/Analyzed Variables | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Ref.** | **HW res. usage** | **Net. Bal.** | **QoS/ resp. time** | **Energy cons.** | **Monet. costs** | **Scalab.** | **High avail. / Data Mgmt.** | **Efficient search/ insert/ delete for scheduling** |
| [25] | X | X | X | | | | | |
| [26] | X | | | | | | X | |
| [169] | X | X | X | X | X | | X | |
| [219] | X | | X | | | X | X | |
| [32] | X | | | X | | | | |
| [189] | X | | | | | | | |
| [68] [72] | X | X | X | X | | | | |
| [188] | | X | X | | | | | |
| [243] | X | X | X | X | | | | |
| **Our work** | **X** | | **X** | **X** | | **X** | X | **X** |

**Table 3.10:** GRID/FOG/EDGE execution-time approaches: Target Variables

threshold, the system executes a migration strategy. That consists of redeploying said container in the node-edge that 1) Is the closest possible to the new location of the device and 2) has sufficient resources in terms of **CPU** and **RAM**.

Finally, some works study the communication between tasks and distributed data management. For example, Corneo et al.[40] propose an optimal way to retrieve data from sensors. They aim to avoid bandwidth overload and lose the novelty of the data. As in the previous section, this area of study is outside the scope of our research work.

### 3.4.1 Analysis and considerations

This section has described some approaches that we consider important for **initial deployment** and **in-execution scheduling** at the **edge**, **fog**, and **grid** levels.

Again, following the methodology described in the first section, we study this trend from an algorithmic approach. However, unlike in the previous section, we have changed some analysis criteria due to the natural differences between **Cloud** and **edge/grid/fog** heuristics. Thereby, in Tables 3.7 and 3.8 , we characterize the works concerning the **initial deployment** of a certain workload, taking into

account: 1) The input variables, the operations performed, the type of approach or special hardware considerations, and 2) the variables that the approaches analyze and/or optimize such as **QoS**, Energy consumption, network balance or data management. Then, in Tables 3.9 and 3.10, we characterize the works concerning **execution-time** scheduling considering the same analysis fields.

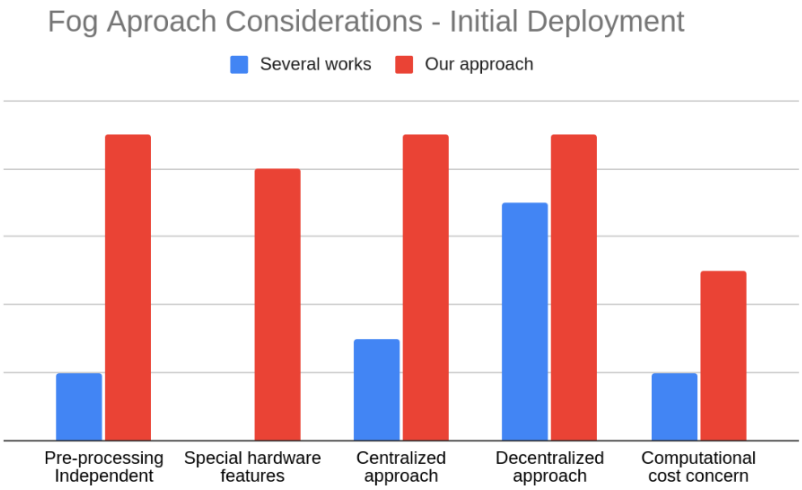


**Figure 3.6:** GRID/FOG/EDGE: Initial Deployment Considerations Trend

Figure 3.6 shows that most works tend to perform preprocessing operations before deploying software components. Moreover, it is interesting to see that very few authors analyze the characteristics of hardware components despite their heterogeneity. In our case, we manage an efficient initial-deployment process only with current hardware requirements information. Nevertheless, we also consider the energy characteristics of components such as the **CPU**, **RAM**, **hard disk**, and **network card**. This analysis is very important for us, especially when battery-dependent devices are present, as in the case of **grid** or **fog** environments.

Another interesting fact is the more significant trend towards decentralized scheduling approaches. That is due to the physical distances between central entities and the worker nodes deployed close to the data sources. It's important to say that said central entities may even belong to higher abstraction layers such as the cloud.

Finally, the figure does not show a great tendency to analyze algorithmic complexity. In our case, we need to study that since it is directly related to the number of operations that we perform through the network.

On the other hand, Figure 3.7 shows that most works that study the **initial**

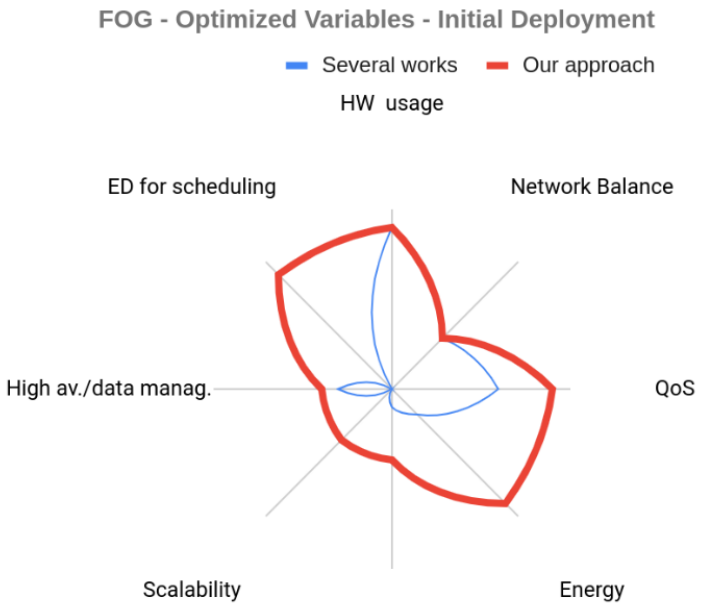


**Figure 3.7:** GRID/FOG/EDGE: Initial Deployment - Scheduling Variables Trend

**deployment** in fog-type environments focus on the proper use of hardware components, the quality of service, and the efficient network balance. The latter, unlike cloud environments, is often due to having more layers of abstraction in the architecture. Thus, having intermediate nodes between the end devices and the cloud nodes makes it imperative to manage the load at the hardware and software resources level.

In our case, although we analyze the efficient use of hardware components (including network load balancing), we have not studied the application of our method in vertical hierarchies. This point represents for us one of the future works.

The analysis of the rest of the variables is the same as the one we did for figure 3.3.

Regarding the **execution-time scheduling**, figure 3.8 shows that several works depend on pre-processing operations. That is due to the trend of using clustering and prediction algorithms on variables such as resource usage, data popularity, etc.

Although we do not study these points directly, we consider them compatible with the use of multidimensional structures and correct distance functions. That represents a more dynamic and less computationally expensive approach.

On the other hand, Figure 3.8 shows a greater tendency to discuss special hard-

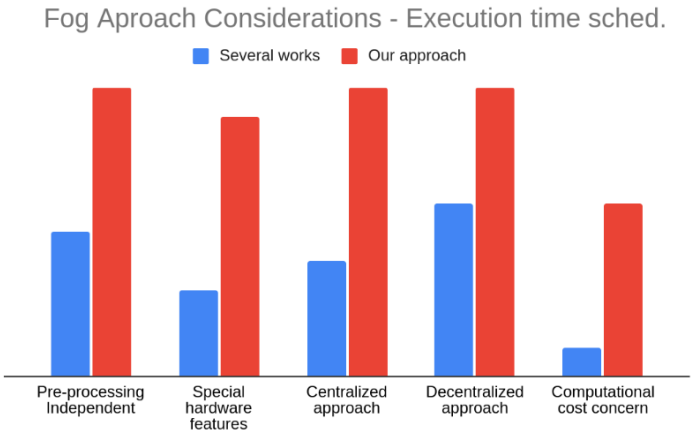


**Figure 3.8:** GRID/FOG/EDGE: Execution-time scheduling Considerations Trend

ware features than the **initial deployment moment**. It is important to say that our approach addresses this point at both scheduling moments.

The analysis of the rest of the variables is the same as the one we did for figure 3.6.

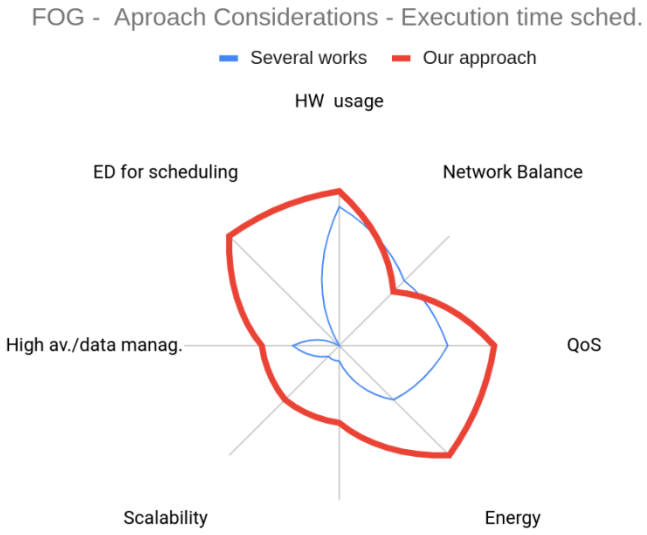


**Figure 3.9:** GRID/FOG/EDGE: Execution-time scheduling Variables Trend

Finally, figure 3.9 shows a greater tendency to maintain an optimal network balance and energy use. That is because many works perform scheduling operations based on the position changes of some devices.

Although we do not deal directly with this, we consider that position data can belong to an extra dimension of a multidimensional data structure. In this way, searches for nearby nodes can be performed in an efficient logarithmic complexity.

It is important to say that, as the Figure describes, the efficient use of energy is one of our primary concerns. We study that through efficient scheduling operations.

The analysis of the rest of the variables is the same as the one we did for figure 3.7.

## 3.5 Chapter summary and contributions

In this chapter, we have selected and analyzed works that study distributed deployment and scheduling strategies. We made this selection considering the criteria of diversity, strength, and the variables shown in all the presented tables.

Unlike many surveys, we propose an algorithmic way of analysis to study these works. In it, we evaluate the problems and interest variables concerning two scheduling moments: 1) when an application is being deployed (**Initial deployment**), and 2) when it is being scheduled at run-time (**execution-time scheduling**).

So many interesting aspects to evaluate become visible, such as the differences between **cloud** and **fog/grid/edge** architectures, the management of physical distances between devices, the energy consumption, etc. For example, while the management of **cloud** clusters has a greater tendency to evaluate scalability factors, in **fog/grid/edge**, there is a greater tendency to consider the physical location of nodes.

All these aspects have allowed us to define the scope of our work and explain some of its strengths. For example, some approaches are based on learning techniques. Usually, in addition to requiring expensive computational operations, these methods can be affected by unpredictable data variations. In our case, we believe that the efficient configuration of our approach allows a safe deployment and scheduling based on current and dynamic data. Furthermore, many authors propose the optimization of some variables, such as only the quality of service or energy consumption. For our part, our approach can manage naturally as many aspects as needed at the same time.

As we said in the introduction of this chapter, we use multidimensional data structures to index data from both the nodes running the applications and the applications themselves. Depending on their type, these structures are compatible with centralized and decentralized network heuristics. They allow us to have a representation of a custom universe of as many dimensions as needed. Based on

them, we can perform efficient queries enabling different scheduling objectives such as energy savings, security, location dynamism, etc.

The next chapter will present our approach's structure, algorithm, and deployment.

CHAPTER 4

# MULTIDIMENSIONAL entities for energy savings

# 4 Multidimensional entities for energy savings

## 4.1 Introduction

In the previous chapter, we have explained some of the different techniques for applications' **initial deployment** and **execution-time scheduling** in distributed systems. They aim to optimize several variables such as deployment time, scalability, or energy consumption. To achieve this, they analyze the state of a system deployment at a given time (input variables: hardware usage, physical location, load, etc.) to trigger scheduling operations (i.e., tasks deployment, **containers** migration, **VMs** duplication, etc.).

In the specific case of energy consumption, they have shown that it can be decreased with efficient deployment and load balancing techniques or the proper hardware features usage (prudent shutdown, CPU frequency reduction, etc.). However, they do not consider energy-based scheduling analyzing variables such as hardware heterogeneity, intelligent and multiple hardware resources indexing, or special hardware power capabilities.

On the other hand, as mentioned in chapter 2, we profile hardware and software components through mathematical models. Once identified an application by its **PID**, these allow us to relate the workload of the concerning hardware devices to their energy consumption using **OS** (operating system) interfaces. In this way, we can make deployment and scheduling decisions based on current consumption/workload information without specialized measurement artifacts. Although these models may not give perfectly accurate results, they allow comparing consumption information based on time and the heterogeneity of devices.

In this chapter, we present our distributed scheduling approach for load balancing energy savings. However, before explaining it, it is necessary first to choose the architecture of the applications to study, taking into account scheduling-based features. For example, applications based on virtual machine instances enable migration or duplication of these entities. On the other hand, monolithic processes running directly on the host **OS** let only balance the load generated from requests. Finally, applications based on containers enable light migration-duplication, efficient deployment, modular scalability, etc.

In this thesis work, we have chosen to analyze containerized applications based on microservices (**MS**). According to *Paolo Di Francesco*, they are small services

that make up an application, each running in its own process and communicating with lightweight mechanisms[58]. We chose this architecture because it enables many scheduling advantages[74][215]:

- Microservices generally use containers as their deployment technology. Therefore, each instance is identifiable from its **PID**[92].
- Easy updates via modules. A system administrator only needs to update only one microservice to update an application functionality.
- Heterogeneity. This architecture facilitates application deployment between different types of devices and heterogeneous platforms.
- Non-centralized storage operations.
- Extra security. The failure of one functionality does not affect the entire system. Thus, the errors can be treated and corrected separately.
- Easy to implement new content. To add new features to an application, a system administrator only needs to add new microservices and re-deploy only the components that will use it.

In our work, we consider the microservices approach only as a use case. Thereby, in a microservices deployment, we analyze and improve hardware resource management in a novel way to save energy without losing the notion of **QoS** (Quality of Service). We schedule (middleware operations: move, duplicate, start, stop) the containerized microservices that make up a distributed application through different physical nodes using multidimensional/spatial special structures. With them, we index elements such as physical nodes and microservices at execution time according to multiple criteria, such as the current availability of hardware resources, hardware's power-related features, or the container's resource requirements. This way, our objective is to have a system that performs the elements' search, insertion, and elimination at an efficient computational cost in order to execute energy-saving scheduling strategies.

Although we consider our technique extensible for any other type of architecture, we leave this analysis for future work.

Another important issue is scheduling heuristics. These can be oriented to either centralized or decentralized philosophies. Each has advantages and disadvantages depending on the type of application, deployment politic, or deployment level (i.e., Cloud, grid, or host). For example, important technologies like **Netflix** prefer centralized microservices orchestration (**Netflix** uses its well-known Conductor)

mainly for scalability matters, tight coupling, **SLA** agreements, etc.[168]. For their part, *de langue et al.*[138] took advantage of a decentralized environment to develop a non-formal learning system in CoPs (Communities of Practice) based on a microservices architecture. In terms of platforms, **Kubernetes** offers centralized algorithms for the efficient deployment and scale of **PODs**[136]. For his part, **Kalimucho** [52] provides a middleware platform to manage microservices; but in a **p2p** network of heterogeneous devices. **Kalimucho** also enables intelligent microservices movement, duplication, and routing operations. Chapter 3 summarizes both heuristics more fully.

Although our approach is extensible for both philosophies, we will study a **decentralized** system in this instance. We believe that in addition to advantages such as scalability and the absence of a single point of failure[26], a **non-centralized** network is a perfect fit for energy-based scheduling. We will study other approaches in future works.

## 4.2 Methodology

This chapter describes the non-centralized scheduling algorithms that we propose for distributed environments. In general terms, these strategies perform device self-analysis and negotiation procedures for load balancing and energy savings.

In order to understand how these procedures work, the chapter begins by explaining the scheduling instruments that each device runs. Thus, section 4.3 describes our **Kaligreen** middleware that consists of per-device instances of a special portable microservice. That aims to analyze its device's resource status, negotiate with other peers, and execute scheduling algorithms.

Among the algorithms we implement, **Kaligreen** executes neighborhood microservice exchanges based on overloading and underloading situations. That considers hardware heterogeneity and different microservices restrictions.

To enrich the operation of our middleware, section 4.3.5 explains the architecture of **Kaligreen V2**. That implements special tables to describe the relationship between running software features and hardware special characteristics.

Finally, section 4.4 explains our algorithms based on multidimensional spaces. Such procedures use **Kaligreen** in its two versions and special data structures compatible with these spaces. Subsections 4.4.1 to 4.4.5 describe the theoretical framework and the abstract structures we define. Then, from subsection 4.4.6 to the end of the chapter, we describe our algorithms' procedures per se.

## 4.3 Kaligreen Middleware: Establishing entities and tools for inter-device negotiation

As mentioned in the previous section, the **Kaligreen middleware**[5] represents the first step of our proposal. **Kaligreen** works across multiple connected devices without belonging to specific network topology. In this manner, the devices are not guaranteed to connect to other peers all the time (i.e., they turn off, the connection signal is weak, etc.). For example, let us consider a network of cloud-independent user devices. Such devices (smartphones, televisions, watches, etc.) can connect to produce deployed applications' results using different interfaces (Wifi, Bluetooth, Ethernet, etc.). Here, intelligent device negotiations are the natural way to schedule one or more applications' microservices. In **Kaligreen's** case, the main objective of these negotiations may be to save the most significant amount of energy, particularly for devices that depend on a battery.

The non-centralized nature of scheduling algorithms in **Kaligreen** enables each device to offer or request resources from peers. If two devices agree, they can execute a container migration or duplication process. However, this methodology needs mechanisms and protocols to perform correctly.

In the next section, we explain some of **Kaligreen's** features, which allow the execution of scheduling strategies.

### 4.3.1 The Kalimucho middleware and devices for Kaligreen

In order to execute our algorithms, we need a tool capable of managing highly heterogeneous decentralized environments. For that reason, we base our approach on the **Kalimucho** middleware[51]. It allows to move, stop, and (re)start application's microservices efficiently and transparently. For that, **Kalimucho** reads and understands distributed Java applications as a set of containerized components installed on Android and desktop devices. Each component is connected to others through special connectors that can be in an active or inactive state. That means a component can have many connectors with several other peers without saturating the network since only the necessary connectors are active. Additionally, **Kalimucho** is able to move components from one device to another without losing their execution state. To do that, if a component requests to be migrated to another device, **Kalimucho** will do it by shifting its java source code along with its current execution state using special storage files.

For **Kalimucho**, all these operations are platform-independent (operating system, firmware, etc.) thanks to the portability offered by the Java virtual machine.

In our case, we take advantage of this feature to efficiently save the energy of user devices. We consider this a significant problem since, according to several studies[200][223][225][176], by 2016, at least 70% of people in countries like South Korea had mobile devices (phones, tablets, etc.). Moreover, more than 81% of people in the world use mobile devices with the Android operating system. On the other hand, one billion PCs run Windows, 100 million devices (desktops and smartphones) use MacOS, and 1.6% of personal computers run **GNU/Linux**.

In **Kaligreen**, we differentiate between battery-dependent devices (laptops, smartphones, tablets) and battery-independent devices (desktop computers). This differentiation allows us to establish scheduling politics considering hardware heterogeneity. For example, algorithm 1 considers battery-independent devices prioritize being freed from **Kalimucho** containers but not receiving them.

For all these devices to be able to communicate and execute their scheduling algorithm, it is necessary to implement a standard protocol. We define this protocol through a metadata vector, which we will detail in the next section.

### 4.3.2 The Descriptor Vector

In order to execute a non-centralized scheduler, a device in a network must be able to describe essential hardware resources information to other peers. In **Kaligreen**, each device $D_i$ uses a descriptor vector $V_{i_k}$ to depict five possible types of metadata: 1) Its resources capacity, 2) its current average hardware resource load, 3) its hardware average resources availability, 4) the execution requirements of a microservice $M_{i_j}$, and 5) the average current load generated by $M_{i_j}$ in each resource. This way, for example, a device $D_1$ can send a vector $V_{1_3}$ to other peers seeking to establish a negotiation process to migrate $M_{1_2}$. One goal may be to find a candidate capable of executing $M_{1_2}$ more frugally.

**Kaligreen** considers the hardware resources described in chapter 2. As a first approach, all vector $V_i$ stores **CPU**, **RAM**, **network**, and **storage resources** metadata in terms of **GHz**, **MB**, **MB/s**, and **MB/s**, respectively. In figures 4.1 and 4.2, we show the five possible uses of $V_i$. The first shows the metadata sending process concerning the hardware resources of the first device. The second shows this process for the hardware resources that an application needs and obtains from the device.

In order to complete our scheduling approach, we also categorize the containerized microservices. That is to restrict some operations such as migration to avoid damaging the user experience and the applications' functionality.

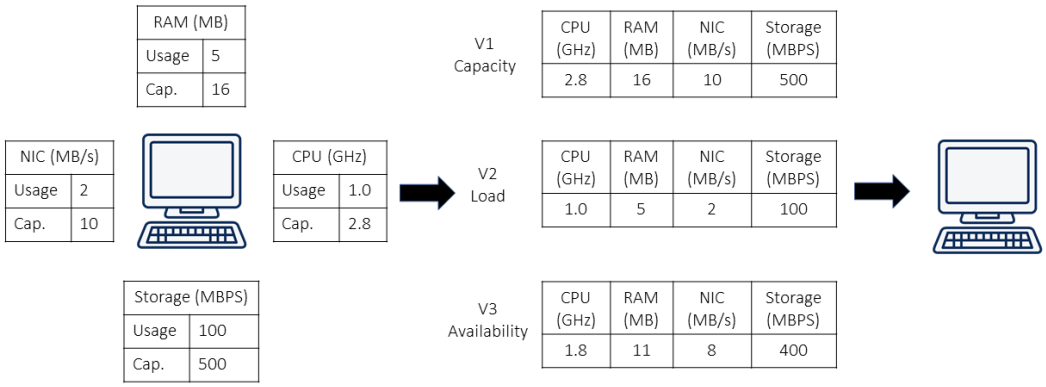


**Figure 4.1:** Hardware resources' metadata sending process

### 4.3.3 Microservices for applications and devices' supervision

As we explained in section 4.1, we consider that a distributed application comprises microservices working together. One of the problems we analyze in **KaliGreen** is categorizing and scheduling these microservices without damaging the user experience or an application workflow. We treat the first issue as follows:

- **UI Microservices** (interactive part of the application): **Kaligreen** will not migrate this type of microservices since the user is the only one who can decide with which device to interact.
- **Sensors-management Microservices: Kaligreen** will not migrate this type of microservices since this would imply stopping obtaining data from the environment involved.
- **Computation Microservices:** This group includes artificial intelligence training, video treatment, or image compression. **Kaligreen** can migrate or duplicate the corresponding containers to save a device's energy.

In order to address the second issue, we have designed a special type of microservice that we call **"Supervisor"**. In the **Kaligreen** environment, it is a demon-like process that runs by default in each device $D_i$. Once it starts running, it enables in $D_i$ the following operations:

- Perform periodic analysis of each hardware component considering load and energy consumption.
- Manage a list of metadata of all the running microservices.

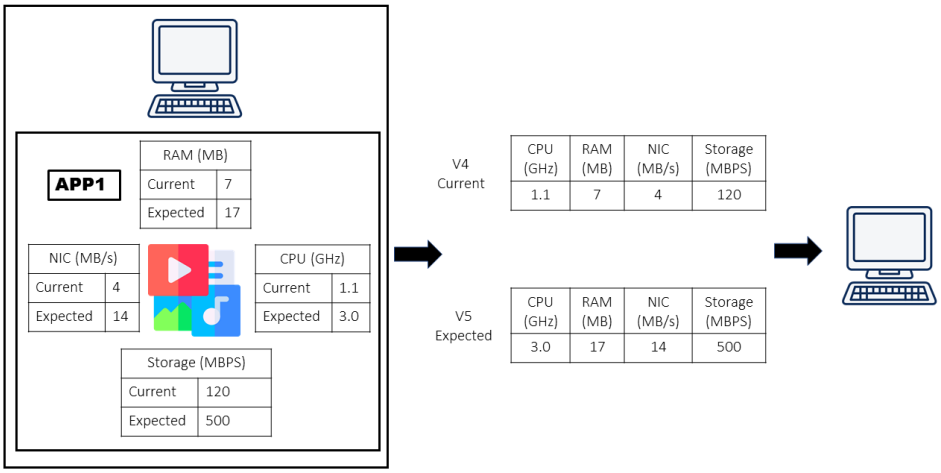


**Figure 4.2:** Application needs metadata sending process

- Send **descriptor vectors** to other devices.
- Receive and analyze **descriptor vectors** from other devices.
- Execute scheduling algorithms, which may involve container operations such as stop, start, migration, and duplication.

All these operations allow us to define different scheduling heuristics. If all devices connect to a single peer, the latter could run a distributed centralized algorithm. Otherwise, the heuristic will be decentralized.

As we said in section 4.1, our goal with **Kaligreen** is to implement **non-centralized** approaches. The following section explains our first algorithm based on devices' inter-neighbor negotiations.

### 4.3.4 A first distributed algorithm in Kaligreen

Let us define a heterogeneous user network $N$ similar to the one explained in section 4.3. $N$ comprises battery-dependent and non-battery-dependent devices identified by $D_i$. Depending on their hardware load and battery status, $D_i$ can have two roles: 1) **help seeker** or 2) **resource provider**. Furthermore, $D_i$ can migrate a microservice $M_{i_j}$ to a device $D_j$ after a negotiation process. We consider the devices' inefficient energy situation (for example, a smartphone that is about to finish its battery) to trigger these migrations.

We validate these situations in a function called ***isInEmergencySituation()***. Then, if $D_i$ enters a state of emergency, it will select the microservices to migrate. For this, $D_i$ executes the function ***selectHeaviestMicroservice(n)***.

This last function selects the $nth$ microservice $M_{i_j}$ that generates the most significant impact on the device based on **CPU**, **RAM**, **network load**, and the current remaining **battery** if concerns. For this selection to be possible, we set priority values for each component (based on a previous trivial analysis of random applications). These values are 1 for **Network**, 0.8 for **CPU**, and 0.7 for **RAM**. With them, the function determines the most affected component by 1) multiplying the load that $M_{i_j}$ generates (in percentage terms) by the corresponding factor and 2) selecting the component with the highest value.

Once $D_i$ selects a microservice $M_{i_j}$, it extracts $M_{i_j}$ metadata and writes it in a descriptor vector $V_{i_j}$. Then $D_i$ sends $V_{i_j}$ to the connected peers using the $sendVector(V_{i_j}, D_k)$ function. Then, $D_i$ will wait a reasonable time to get candidate devices that can continue executing $M_{i_j}$. We fixed that time in $100ms$ as an average ping result for our first implementation[59]. For this first implementation, $D_i$ will consider that the best candidate peer $D_j$ is the first one who answers. As in **Kalimucho**, we base this decision on the inverse relationship between physical location and response time.

Finally, $D_i$ notifies $D_j$ using the $sendConfirmation(D_j)$ function to avoid other peers considering $D_j$ as a candidate. If other candidates arrived after, $D_i$ invokes the $sendNegativeConfirmation(D_w)$ function.

As a **help seeker**, $D_i$ succeeds in the algorithm migrating the microservice $M_{i_j}$ to $D_j$ and quitting the emergency situation.

It is important to say that if $M_{i_j}$ cannot be moved, the algorithm tries to migrate the next heaviest one. This process finishes when $D_i$ is no more in an emergency situation or there are no possible microservices to move.

On the other hand, Algorithm 2 describes the procedure to make a device $D_j$ a **resource provider**. Herein, $D_j$ must have sufficient computational capacity to run a foreign microservice $M_{i_j}$ from $D_i$ without entering an emergency state. For this, it analyzes the received concerning vector $V_{i_j}$ and adds its values to its components' current loads' data. If any results exceed the 80% set, $D_j$ will not offer its resources.

The supervisor microservice runs this algorithm periodically and removes all candidate vectors after a migration process or after a maximum response time.

As we can see, **kaligreen's supervisor microservice** can execute scheduling algorithms iteratively. This fact has some advantages. For example, in our first algorithm, the devices' load distribution is sensitive to new changes caused by the user (opening or closing applications) while saving energy. However, to improve energy savings strategies, **Kaligreen** should also learn to use the appropriate

**Algorithm 1:** Help seeker for a device $D_i$

```
1  c = 1;
2  M_ij ← selectHeaviestMicroservice(c);
3  while isEmergencySituation() do
4      if M_ij = NULL then
5          break;
6      V_ij ← buildVector(M_ij);
7      foreach connectedDevices D_l do
8          sendVector(V_ij, D_l);
9      WaitForCandidatsReponses(20)
10     foreach deviceCandidates do
11         D_j ← selectBestCandidate();
12     if D_j = NULL then
13         M_ij = selectHeaviestMicroservice(c + 1);
14         continue;
15     sendConfirmation(D_j);
16     foreach NonSelectedCandidatesD_w do
17         sendNegativeConfirmation(D_w);
18     moveMicroservice(M_ij, D_j);
19 wait()
```

**Algorithm 2:** Resource provider procedure for a device $D_j$

```
1  if isEmergencySituation() ≠ true then
2      foreach Vectors V_k of each device D_k do
3          EvaluateEnergyImpact(V_k);
4          if triggersEmergencySituation(V_k) then
5              delete(V_k);
6      V_c ← selectHeaviestMicroserviceVector();
7      sendConfirmation(D_k);
8      receiveMicroservice(M_ij);
9      deleteAllRemainingVectors;
```

hardware components optimally. For that, our middleware needs to consider the components' usage time, capabilities, behavior, and energy savings features.

The following section shows **Kaligreen's** improvement design, enabling it to be aware of the relation of two factors: 1) The microservices processing requirements and 2) the awareness of as many hardware features as possible. That will allow intelligent microservices filtering as a part of scheduling procedures.

### 4.3.5 Kaligreen V2: A middleware aware of hardware opportunities to save energy

To optimally save energy, a middleware that performs distributed scheduling algorithms needs to consider the consumption of the devices' components. On the one hand, Chapter 2 explained that the four components that we analyze in this thesis are the **CPU**, the **RAM**, the **network interface**, and the **storage device**. On the other hand, Chapter 3 explained how some distributed approaches save energy based on these components.

As a first step, **Kaligreen** only considered the load of the four components regardless of their heterogeneity. This second version introduces microservices filtering based on their execution characteristics and the mentioned components' capabilities.

To explain the methodology of this section, let us define the mentioned execution features through two variables:

- Runtime: We define a microservice as persistent when it has no known termination point.
- Access time to a hardware component: We define that a microservice $M$ generates a ***high*** average load $L$ in a component $C_j$ of a device $D_j$ when $L$ exceeds a threshold based on the characteristics of $C_J$. This threshold must be set manually before applying the scheduling algorithm.

The following section explains the analysis of the first component: The **CPU**.

#### CPU analysis

In Chapter 2, we mentioned that a **CPU**, at the hardware level, can implement **C-States** and **DVFS** to save energy. Furthermore, it can implement **turbo boost** technology to increment the execution frequency for a prudent moment. In a scheduling algorithm, migration operations can be made cleverer by only considering "compatible" microservices with these features. Table 4.1 shows this correlation.

| Microservices' Features in $D_i$ | | CPU features in $D_j$ | | |
|---|---|---|---|---|
| **Persistent Microservice** | **High CPU Load** | **Boosting** | **PCPG** | **DVFS** |
| NO | YES | Candidate | Candidate | Candidate |
| NO | NO | – | – | – |
| YES | YES | – | Candidate | Candidate |
| YES | NO | Candidate | Candidate | Candidate |

**Table 4.1:** Correlation between Microservices features and **CPU** capabilities

| Microservices' Features in $D_i$ | | Scheduling Operations |
|---|---|---|
| **Persistent Microservice** | **High RAM Load** | **Migration** |
| NO | YES | Candidate |
| NO | NO | – |
| YES | YES | Candidate |
| YES | NO | – |

**Table 4.2:** Correlation between **RAM** microservices features and middleware operations

In **Kaligreen**, the **supervisor microservice** implements the table 4.1 from the microservices metadata list. For example, suppose a device $D_i$ executes a persistent microservice $M_i$ that generates a high **CPU** load. In that case, an algorithm may evaluate migrating it to a device $D_j$ that does not contain the boosting characteristic. The reason is that, as we explained in chapter 2, frequencies beyond those supported by the **TDP** considerably increase power consumption (the non-linearity of the model 2.1 and the work of fans). On the other hand, the algorithm could try to keep a persistent low-frequency demandant microservice if the **CPU** of $D_i$ can lower its frequency or turn off cores and the one of $D_j$ does not.

### RAM analysis

As explained in chapter 2, the consumption that a microservice $M_i$ generates in the **RAM** is determined by the number of accesses $M_i$ executes. Since we do not find potential **RAM** energy features for filtering operations, the supervisor microservice may only consider the **RAM** usage.

In this way, a microservice that generates a high load in **RAM** (triggering, for example, **swap** operations) is a candidate to be migrated.

### The storage device analysis

In a similar way to the **NIC**, Chapter 2 explains that the storage device's power consumption is determined by its transfer rate. However, the filtering opportunity we find here relates to the component type: 1) **SSD** or 2) **HDD**.

The **supervisor microservice** can implement table 4.3 for filtering. For example, a scheduling algorithm may migrate a non-persistent microservice $M$ with an **HDD** to a device $D_j$ if this last has an **SDD**. This operation might involve freeing $D's$ hard drive and quickly terminating $M$.

| Microservice Features | | Storage Type | |
|---|---|---|---|
| **Persistent Microservice** | **High Storage Load** | **SSD** | **HDD** |
| NO | YES | Candidate | – |
| NO | NO | – | Candidate |
| YES | YES | Candidate | – |
| YES | NO | Candidate | Candidate |

**Table 4.3:** Correlation between the **storage-related** microservices and storage features

### The Network device analysis

Chapter 2 described that it is possible to know the energy consumption of a network interface from its different power states. On the other hand, although some authors perform some **NIC** power management operations (e.g., automatic D'link green ethernet[50], rate adaptation[153], etc.), they are not accessible from **OS** interfaces. For these reasons, the filtering implemented by the **supervisor microservice** considers, on the one hand, the following criteria for a microservice $M$:

- Bandwidth consumption.
- The size of $M$, which involves migration cost.
- The size of the data coupled to $M$.
- The load generated by links from $M$ to other microservices.

On the other hand, to establish the relation with the variables above, **the supervisor microservice** considers the following operations:

- Data migration: This operation depends on the degree of coupling of $M$ with its data.
- Microservice duplication: Useful for load balancing when it is due to foreign connections.
- Microservice migration.

Thereby, **the supervisor microservice** uses table 4.4 to implement microservice filtering considering the devices' **NIC**. For example, suppose a device $D_i$ executes a persistent microservice $M_i$ that occupies $500kb$ and saturates more than 90% of the $D_i$ **NIC's** capacity. The scheduling algorithm may try to move it, as the first operation, to another device that does not depend on the battery or has a better type of network card. Suppose now that $M_i$ is non-persistent, occupies 1Gb, and does not saturate the $D_i$ bandwidth. The algorithm, in this case, may avoid scheduling $M_i$.

In future works, we will analyze different network interfaces such as Ethernet, Wifi, or 5G.

It is important to remember that, as we describe in section 4.3.5, we define the term ***"high"*** as exceeding a threshold based on the characteristics of each device. This threshold must be set manually before applying the scheduling algorithm.

So far, we have explained the operation and the tools that **Kaligreen** offers to execute distributed scheduling algorithms. Among these tools, the **supervisor microservices** can analyze the state of its device, build **descriptor vectors**, negotiate with peers, and perform scheduling operations. It also manages a list with the metadata of the running microservices, which can be filtered, if necessary, in descriptive tables. The purpose of these tables is to improve the scheduling process. These relate the microservices execution characteristics with hardware components capabilities or possible migration operations.

In the next section, we explain one of the cornerstones of our proposal: A distributed algorithm based on distributed multidimensional data structures and **Kaligreen** tools.

| Microservices' Features in $D_i$ | | | | | Scheduling operations | | |
|---|---|---|---|---|---|---|---|
| **Persistent MS** | **High MS size** | **High Bandwidth Load** | **High load generated by the MS connections** | **High data size** | **MS Migration** | **MS Duplication** | **MS Data Migration** |
| YES | YES | YES | YES | YES | Candidate | Candidate | – |
| YES | YES | YES | YES | NO | Candidate | Candidate | Candidate |
| YES | YES | YES | NO | YES | Candidate | Candidate | – |
| YES | YES | YES | NO | NO | Candidate | Candidate | Candidate |
| YES | YES | NO | YES | YES | – | Candidate | – |
| YES | YES | NO | YES | NO | – | Candidate | Candidate |
| YES | YES | NO | NO | YES | Candidate | Candidate | – |
| YES | YES | NO | NO | NO | Candidate | Candidate | Candidate |
| YES | NO | YES | YES | YES | Candidate | Candidate | – |
| YES | NO | YES | YES | NO | Candidate | Candidate | Candidate |
| YES | NO | YES | NO | YES | Candidate | Candidate | – |
| YES | NO | YES | NO | NO | Candidate | Candidate | Candidate |
| YES | NO | NO | YES | YES | Candidate | Candidate | – |
| YES | NO | NO | YES | NO | Candidate | Candidate | Candidate |
| YES | NO | NO | NO | YES | Candidate | Candidate | – |
| YES | NO | NO | NO | NO | Candidate | Candidate | Candidate |
| NO | YES | YES | YES | YES | – | – | – |
| NO | YES | YES | YES | NO | – | – | – |
| NO | YES | YES | NO | YES | Candidate | Candidate | – |
| NO | YES | YES | NO | NO | Candidate | Candidate | Candidate |
| NO | YES | NO | YES | YES | – | Candidate | – |
| NO | YES | NO | YES | NO | – | Candidate | Candidate |
| NO | YES | NO | NO | YES | – | – | – |
| NO | YES | NO | NO | NO | – | – | – |
| NO | NO | YES | YES | YES | Candidate | Candidate | – |
| NO | NO | YES | YES | NO | Candidate | Candidate | Candidate |
| NO | NO | YES | NO | YES | Candidate | Candidate | – |
| NO | NO | YES | NO | NO | Candidate | Candidate | Candidate |
| NO | NO | NO | YES | YES | – | – | – |
| NO | NO | NO | YES | NO | – | – | Candidate |
| NO | NO | NO | NO | YES | Candidate | Candidate | – |
| NO | NO | NO | NO | NO | Candidate | Candidate | Candidate |

**Table 4.4:** Correlation between the **network-related** microservices features and middleware operations

## 4.4 An energy-saving approach: Understanding microservices and devices as multidimensional entities

In Chapter 3, we explained various scheduling approaches used by scientists and well-known tools. On the one hand, these techniques can be executed in different deployment levels such as cloud or grid. On the other hand, they can have different objectives, such as quality of service assurance, high availability, scalability, or energy savings. In terms of the latter criterion, most of the authors we analyzed do not consider the characteristics of each component in depth. In addition, they base their proposals on learning techniques, centralized migration algorithms, or the treatment of mathematical functions.

In our case, we believe that **multidimensional data structures** allow having a clear and updated view of an entire system. With them, we can index the current data of the average consumption of the software components, loads of the devices' hardware elements, their energy-saving capacities, the devices' geographical positions, the battery status, etc. Then, for scheduling purposes, we can perform (re)insertions, deletions, and range searches at optimal computational cost depending on the multidimensional structure we choose.

It is important to say that **our approach is abstract**. Researchers can select different data structures compatible with centralized, non-centralized, or hybrid approaches at any level of deployment (cloud, grid, etc.). In this way, they will be able to execute various scheduling heuristics analyzing and optimizing as many variables as necessary.

In this thesis work, we implement this idea for the case of decentralized networks using **Kaligreen**. Our algorithm identifies "ideal" host candidates for microservices' execution and applies runtime scheduling operations (migration or duplication) to reduce energy consumption. To do this, an overlay called **MAAN** (Multi-Attribute Addressable Network)[23] allows us to interpret a decentralized network as a multidimensional resource (capacity-demand) space, which supports range queries in a logarithmic quantity of hops.

We use **MAAN** to contrast the microservices' requirements and the hardware availability that each device has. Thereby, in a certain period or under certain circumstances (i.e., battery problems, overload, etc.), one device running microservices can:

- Map them in terms of their execution requirements (i.e., **CPU** frequency, **RAM** operations, **Network** rate, and **disk** speed),

- Select an ideal microservice $M$ to be moved or duplicated,
- Find ideal peer(s) that meet the $M$ requirements in an optimal computational complexity, and
- Negotiate the migration or duplication of $M$ considering energy consumption and **QoS** criteria.

Before explaining this approach in detail, it is first necessary to understand the concept and operations of multidimensional spaces. We detail this in the following section.

### 4.4.1 Multidimensional spaces

Data management is a big concern in computer science. For this reason, some structures allow modeling data sets as spaces with as many dimensions as the data sets have characteristics. They allow data insertions, search, and recovery at efficient operation costs. For example, all binary-based trees like AVL, RedBlack, and B* perform one-dimensional data management operations in $O(log(N))$ for the average case. At the same time, hash-based structures do the same in $O(1)$ for the best case.

However, these operations become more complicated with the increment of dimensions (data characteristics). Several solutions based on metrics (defined by distance functions) and-or n-dimensional spaces deal with this problem. They can manage multi-dimensional data sets from the viewpoint of 1) The distances among its elements[28] (e.g., BKT, BT, or LAESA) or 2) the abstraction of the universe in which they exist[81] (e.g., K-D-Tree, R*tree or Z-ordering).

All these data management approaches allow, on the one hand, to process queries based on a single characteristic/dimension to obtain a single element. On the other hand, they also retrieve all data that meet various criteria in a specific range or region. Furthermore, depending on the structure heuristic, they operate from the viewpoint of one particular element or the viewpoint of the entire universe. For example, while **BKT** organizes the elements from a centroid-element, the **RTree** models all the data universe in a rectangle-based heuristic.

Edgar Chavez et al.[28] describe the possible queries in multidimensional metric spaces, which are also applicable in spatial access methods.

- Range query $(q, r)_d$. Retrieve all elements which are within distance $r$ to $q$. This is, $\{u \in U \mid d(q, u) \leq r\}$.

- Nearest neighbor query $NN(q)$. Retrieve the closest element to $q$ in U. This is, $\{u \in U \mid \forall v \in U, d(q,u) \leq d(q,v)\}$.

- K-Nearest neighbor query $NN_k(q)$. Retrieve the $k$ closest elements to $q$ in U. This is, retrieve a set $A \subset U$ such that $|A| = k$ and $\forall u \in A, v \in U - A, d(q,u) \leq d(q,v)$.

Our first structure implements a multidimensional space of hardware resources for a microservices environment. In this space, we can perform energy-savings strategies considering as many hardware features as possible (i.e., capacity, use of special features, etc.). This way, for a microservice $M_{i_j}$ executed in a device $D_i$ generating a load $L$ and an energy consumption $E$, it is possible to find a device $D'$ that: 1) has at least availability of resources $C$, in such a way that $C \geq L$, and 2) the energy consumption $E'$ generated by $M$ in $D'$ is less than $E$ because of final load or particular hardware's characteristics of $D'$.

The following section will explain our hardware resource space and its features.

### 4.4.2 The multidimensional resources space *U*

Managing microservices in a decentralized network environment is challenging. Operations such as load balancing, scheduling, microservices discovery, or energy-saving are difficult to analyze without a central entity. That is because this central entity makes all information about the state of a system readily available.

However, as we explained with **Kaligreen**, we believe that it is also possible to perform scheduling algorithms by intelligently organizing the devices of a P2P network for opportune negotiations. Herein, a **helper seeker** (see section 4.3.4), on the one hand, can migrate microservices to another peer to decrease its load and-or save energy. For his part, a **resource provider** can even proactively offer resources to other peers to process microservices more frugally.

Nevertheless, for a device to find an "ideal" peer to trade with is still a problem.

That is why multidimensional spaces are the cornerstone of our work. Depending on the scheduling strategy, these can be built to manage different data types. Let us define some instances of this relation, considering heterogeneous devices as indexable elements.

- Index current power consumption to negotiate load: In the space, devices can search for peers with a certain power consumption value in their components. Then, to migrate or duplicate microservices, a device may evaluate candidates based on their hardware resources‘ load (availability).

- Index physical positions to trade power consumption: Here, devices may search for physically close peers as candidates. Then, their evaluation for microservices scheduling would be done considering energy criteria (hardware components' capabilities and load, battery status, and others.).

- Index current load of hardware components to negotiate power consumption: Here, devices can search for peers able to run a certain microservice. Then the candidates will be evaluated by energy criteria.

Going further, we can even combine criteria to execute mixed queries. For example, a device could query a heterogeneous space to find all physically close candidates that are not battery dependent, that have a given current power draw, and whose **RAM** is not saturated.

In our case, we randomly decided to implement first the space described in the third point above, leaving others for future works. Thereby, our approach abstracts and queries hardware resources in a 4-dimensional space. It contains the devices' available capabilities in **CPU** frequency, **RAM** capacity, **NIC** rate, and **storage device** speed. We show this in the definition 4.1.

▶ **Definition 4.1 (The space *U*).** Given a set of devices connected to a shared network, we define a space ***U*** made up of 4 dimensions/axis: 1) **CPU** frequency in **GHz**, 2) **RAM** capacity in **MB**, 3) **Network transfer rate** in **MB/s**, and 4) **Storage** speed in **MB/s**. Devices are elements of ***U*** in terms of their current-average hardware resources' availability. ◀

Queries in ***U*** allow a device to know which peers have sufficient resources to run specific microservices in an optimal time. Then, using the **Kaligreen's** tools, the device may select the peers with outstanding features to perform migration-duplication operations (e.g., a low **CPU** requirement microservice and a device whose **CPU** can turn off some cores). It is important to say that ***U*** can consider many more variables such as special hardware features or other functional aspects. However, we consider them at the candidate selection time in this first implementation.

The following section discusses some distributed data structures which allow devices and processes to be indexable nodes. The section describes centralized and decentralized approaches as well as our choice to implement ***U***.

### 4.4.3 Multidimensional P2P structures

As previously mentioned, this implementation seeks to take advantage of some features offered by decentralized network environments for scheduling. For instance, the fact that no single central unit realizes decision-making allows devices to be autonomous. Each peer can analyze its load or energy circumstances to perform ad-hoc scheduling operations (i.e., negotiation, load balancing, query, etc.) Thus, in addition to enabling an entirely dynamic system, this approach allows us to eliminate a single point of global failure.

However, for a device to find other peers to negotiate or request information, it must belong to a standard comprehensible system. For this reason, several network overlays enable these operations in an efficient number of hops. These abstract the network deployment as data structures whose nodes and information are indexed through the same universe of metadata. Some notable examples are 1) an ordered circular list iterated through hash techniques[206][134], 2) a binary tree that can be explored through bit space and logical operations like XOR[157], and (3) a linked list of nodes that can search for elements by making smart hops in the structure[156].

Nevertheless, there are circumstances where a structure that supports more than one criterion (dimension) is needed. For example, in our space, we query a p2p device network to obtain the best candidates in terms of **CPU**, **RAM**, **network**, and **storage** availability to process a microservice. For that, some existing approaches implement multidimensional data structures such as Rtree[66] or KD Tree[83] in a distributed network. As in the original structure, they also keep the parent-child relation of the tree nodes in the network devices.

All these structures allow multidimensional range querying efficiently. However, balancing or restructuring operations when inserting or deleting nodes may be expensive. They could involve the participation of many devices in the network that will perform complex data movements.

On the other hand, other approaches seek to partition multidimensional spaces into more independent structures. For example, Znet[196] divides the space using Z-order curves. Its creators use the devices of the p2p network to form a Z-zone-based skip graph, where each device stores part of the description of a rectangular zone. In this way, when a query arrives at a node, it first verifies if the element (or the range) belongs to the zone it saves by exploring the highest element of its list. If it cannot find it, it derives the search to the node closest to the destination area, descending one level each time the search gets closer to its destination. Figure 4.3 shows the mapping procedure.

Another interesting strategy is the one proposed by Cai et al.[23]. **MAAN**

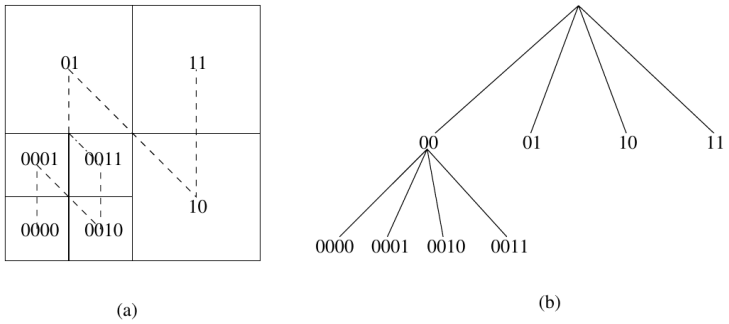


**Figure 4.3:** Space partitioning and mapping: (a)Z-curves at different orders, (b) corresponding partition tree[196]

generalizes the **Chord**[207] approach to deploy a multidimensional space in as many circular devices lists as the space has dimensions.

In our case, we have chosen **MAAN** to implement our space ***U***. This deliberate election is because this thesis does not study the (dis)advantages of different overlays. However, we will explore other structures in future works.

Before explaining the structure of **MAAN**, it is necessary to explain the **Chord** system. We do this in the next section.

### 4.4.4 The Chord system

**Chord** deploys an ordered circular list of devices organized in a clockwise fashion according to an **ID**. Furthermore, using the same **IDs'** family, the system also identifies the elements to look for (i.e., data, processes, and others.).

For their part, each device (node) keeps a table of addresses called "Finger table". This structure has as many elements as the bit string length representing the system's highest possible **ID**. For instance, in Figure 4.4, the system supports up to 63 (111111 in the binary system) devices. Therefore, the devices' finger tables can contain up to 6 elements.

Next, each index of a device's table is defined by its **ID** added to $2^i$, where $i$ goes from 0 to the length of the bit string above. This way, every time an element **E** is searched through a node **N**, **N** looks for the **E's** index successor in its finger table. That is, the highest possible index that does not exceed **E's ID**. For example, suppose that in the topology of figure 4.4, the element **"22"** is searched through node **"8"**. The latter will find that element **"22"** should be in node **"21"** since it

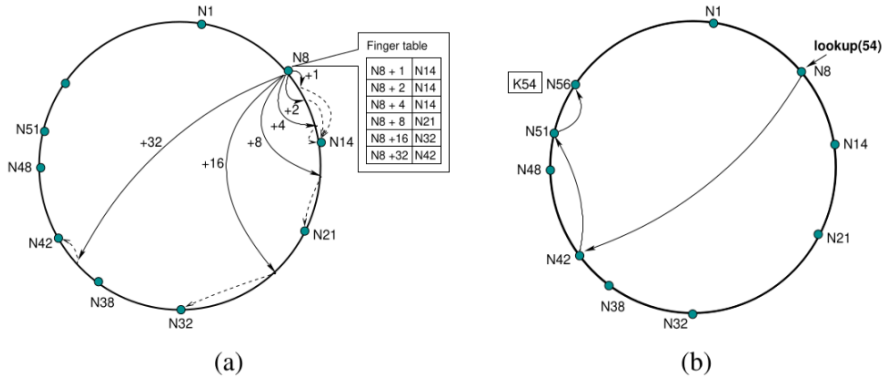


**Figure 4.4:** (a) Finger table entries for node 8. (b) Path of a query for key 54 starting at node 8[207]

is stored in the index **"16"** (n8+8), which is the highest possible **ID** that does not exceed **"22"** (otherwise, N8+16 = 24). In this way, the data structure used to search for elements is a **consistent hashing table**.

On the other hand, because **Chord** manages an ever-changing network of devices, nodes can join and leave at any time. The following section explains how each join-leave operation achieves said dynamism.

### The join operation

There are several procedures a node $nN$ and other devices in the network must perform in order for $nN$ to join the **Chord** system.

First, the entering node nN gets assigned a unique **ID** using a **SHA-1** function that hashes the $nN's$ IP address. Then, an external mechanism guarantees a connection between $nN$ and a **Chord** network pre-existent node. Here, the latter is found by executing a query inside the network to find a node representative of an immediate successor of $nN$ : $nN1$.

The knowledge of $nN1$ consists of an immediate circle predecessor node reference and a **finger table**. $nN$ then uses this knowledge to assign its circle predecessor reference to the one of $nN1$ and fill its own finger table by copying the one of that same node. Finally, the predecessor reference of $nN1$ is re-set to $nN$.

Once $nN$ has been initialized, an update must be issued by walking counter-clockwise along the circle to update previous nodes' finger tables. The goal is to acknowledge $nN's$ existence preserving the finger tables' consistency. This update is done by assigning $nN's$ **ID** to the appropriate finger table references of each predecessor node $P$ if both of the following conditions are met.

1. $P$ precedes $nN$ by at least $2^{i-1}$, where $i$ is delimited from 1 to the length of finger tables.

2. The last finger entry of node $P$ succeeds $nN$.

This update operation is sustained until a previous node $P$ whose $i^{th}$ finger precedes $nN$ is reached. Finally, the responsibility of corresponding keys gets delegated from $nN1$ to $nN$.

#### The leave operation

For a node $LN$ to leave the network, it first informs its predecessor node $pLN$ that the $LN's$ immediate successor in the circle, $LN1$, is now its immediate successor. Then, the circle predecessor reference of $LN1$ is set to $pLN$. Next, an update operation similar to the one described for the join operation is performed, but using the $LN1's$ **ID**. This last operation aims to keep the finger tables of nodes located counter-clock along the circle updated. Finally, the responsibility for $LN's$ keys is delegated to $LN1$.

### 4.4.5 MAAN

As mentioned above, **MAAN** generalizes the **Chord** approach to support n- dimensional and range queries. To begin, consider a space whose nodes have several attributes to index. On the one hand, **MAAN** uses an SHA-1 hashing function to assign an $m - length$ bits identifier to each node. On the other hand, it implements a locality preserving hashing function that, unlike **Chord**, assigns to each node's attribute an identifier in the $m - bit$ space according to its value (the same space shared by the $m - bits$ node identifiers specified before). The latter is crucial because the SHA-1 function would destroy the locality of keys since **MAAN** seeks to pair numerical attribute values to them instead of objects' names. That allows the execution of two different multi-attribute query resolution approaches that would not be possible within **Chord's** original system:

1. The iterative approach: It consists of a query originated in a node $N$. That is composed of an $M$ number of following sub-queries according to the $n$ number of attribute dimensions. The query returns a series of candidate lists later intersected in $N$ to find the fittest candidates.

2. The single attribute query resolution: It aims to find a set of candidates $X$ by performing queries fulfilling the conditions of a single dominating attribute.

Then, sub-queries are applied to $X$ to obtain a sub-set of candidates whose members meet the conditions of the other attributes. Finally, the origin node $N$ selects the fittest candidate by correlation.

In **MAAN**, join and leave operations are the same as the original **Chord** system.

The following section explains how we tune **MAAN** to implement our multidimensional resource space.

### 4.4.6 Implementing the space of resources *U* in MAAN

As previously described in section 4.4.2, we index device resources in the $U$ space considering the availability of their processing-related capabilities. With that, we aim to find ideal resources to migrate or duplicate microservices and save energy efficiently.

#### Resources to index

According to the definition of $\boldsymbol{U}$, four different dimensions/axes relate to four different hardware resources: (1) **CPU** frequency in **GHz**, (2) **RAM** capacity in **MB**, (3) **network** transfer rate in **Mbps**, and (4) **storage** speed in **MBps**. For that, we use **MAAN** to index each device's resource availability values on the corresponding $\boldsymbol{U}$ axis.

#### The join operation

When a device intends to join our **MAAN** network, four different logical nodes are created in four different dimensions (axis) of the $\boldsymbol{U}$ space. These nodes follow the same operations described in section 4.4.4 for the **Chord** system but with some technical differences.

In our approach, we lack an IP address identifier for each device. Instead, we created a locality preserving formula that, for any dimension, transforms a numerical value representing the availability of a resource into an **ID** coherent to the **ID** space available. Formula 4.1 states a superior limit for numerical entry values and a maximum number of nodes supported. Thus, the resource availability value $V$ is multiplied by the maximum number of supported (at any time) nodes $N$. Then, the product is divided by the maximum device's resource limit value $SL$, where $SL$ and $N$ values are relative to each dimension's constraints.

$$ID = \left\lceil \frac{V * N}{SL} \right\rceil \tag{4.1}$$

This model allows us to preserve a unique universe of integer **IDs** that can be assigned to nodes according to their dimensions. Thereby, in our approach, the finger tables contain records in pairs of $m-bit$ space identifiers and node references. These elements grant us the execution of n-dimensional queries by applying an iterative approach like **MAAN's** original procedure.

On the other hand, it is necessary to say that nodes with a repetitive value for any attribute can join the network resulting in an identical **ID** assignment as a pre-existing node. That would result in a collision causing the entering node to be placed in the circle between said pre-existing node, further referenced as **twin-node**, and its successor. We solve that in the following way. When a twin node $T$ joins the network and finds a node $rN$ identified with its same $ID$, it enters a twin list of $rN$, being $rN$ the root of the list. Herein, each member has both predecessor and successor **twin-node** pointers.

### The leave operation

When a device intends to leave the network, our structure performs mostly the same operations as the original **Chord's** system but handles the **twin nodes** as an added feature. For any twin list with a root node $rN$, if $rN$ intends to leave the network, its first twin successor $TN$ gets assigned as the new $rN$. Then, $rN$ copies in $TN$ its circle predecessor reference and its finger table. On the other hand, if a node inside the $rN's$ twin list leaves, the system removes it from the said list and resets the concerning predecessor and successor **twin-node** pointers.

### Data frames and the reindexing process

Section 4.4.6 explained that our structure indexes nodes based on the availability of their resources. However, the latter is very dynamic in real environments since it can be altered by the **OS** algorithms, the running applications' behavior, etc. For that reason, we implement the concept of "**data frames**" in our approach.

A **data frame** is a FIFO-like collection of availability measurement values related to a resource. Each **supervisor microservice** implements four data frames for the four axes of the $U$ space. The implementation considers the structure size, frequency of data recollections and insertions, and run-time modifications. Then, the **supervisor microservice** uses the average of the data frames values and the formula 4.1 to find its resource's corresponding **IDs**. That allows us to control the frequency with which nodes get dynamically reindexed into our MAAN-based approach.

### Querying for resources

In order to perform queries, we follow the **MAAN's** iterative approach but consider two new operations to treat any candidate list $X$.

To begin, we implement a **"candidate lock"** function to prevent resource selection collisions. That phenomenon occurs when several query resolutions select the same node candidate as the fittest one. Thereby, simultaneous microservices migration/duplication operations may occur towards this node, causing overload and over energy-consumption conditions.

A device $D$ applies candidate lock to each member $x \in X$, meaning that they can be candidates to exclusively one query at a time. If $x$ was already taken as a candidate by another peer, $D$ removes it from $X$ (see section 4.3.4).

On the other hand, the system must avoid the "circular saturation" phenomenon. That occurs when $D$ migrates its microservices to a candidate $x$, causing in it an overload condition. The latter may trigger in $x$ mechanisms to mitigate thermal energy elevation (such as a fan) and-or initiate chain migrations in the system resulting in further network strain.

For that, the **supervisor microservices** implement methods that enable projecting the future percentage of each resource consumption from numeric values. In this instance, these methods are approached in the scheduling algorithm, which must be aware of the system's stability.

## 4.4.7 The Scheduling algorithm

Once a device is inserted into our **MAAN's** network configuration where the space $U$ is deployed, its **supervisor microservice** starts executing a scheduling algorithm (see section 4.3.3).

This section explains our scheduling algorithm that we have implemented in this thesis. That considers running microservices deployed in our $U$ space.

As we said in the previous section, it implements projection functions for each hardware component. That allows devices to know the load and power consumption they would generate if they perform microservices migrations or duplications.

### The CPU Projection

Chapter 2 explained that the **CPU** power consumption is found from its capacitance, frequency, and voltage. However, heterogeneity must be considered when cross estimating (estimating from another device). That is, the frequency of a **CPU** executing a process is not necessarily the same as for another.

For that reason, for a microservice $M$, we deduce the number of $M's$ program instructions considering the ones that the **CPU** can execute per unit of time. We can get this information from the ***/proc/cpuinfo*** file).

Thereby, if (1) $D's$ **CPU** is capable of executing $I_D$ program's instructions in a time $T1$, and (2) M consumes $X\%$ of this component frequency $X_{M_D}$ in $T1$, then $M$ executes $I_{M_D}$ program's instructions in a proportional way too:

$$I_{M_D} = \frac{X_{M_D} \times I_D}{100}(T1) \tag{4.2}$$

Then, for another device, $D2$, it is possible to deduce the percentage $X_{M_{D2}}$ of its **CPU** load if it would execute $M$. For that, we consider the number of instructions $I_{D2}$ in $T1$ that $D2$ can execute:

$$X_{M_{D2}} = \frac{X_{M_D} \times I_D}{I_{D2}}(T2) \tag{4.3}$$

Finally, the energy that $D2's$ **CPU** would consume to process $M$, $E_{M_{D2}}$, can be calculated following the relation shown in equation 2.1; but considering the proportions that we explained in the last two paragraphs.

$$\begin{aligned} E_{M_{D2}} =& \frac{X_{M_D} \times I_D \times C_{D2} \times V_{D2}^2 \times F_{D2}}{100 \times I_{D2}}(T) + fans_s(t), where \\ & \\ C_{D2} =& \frac{0.7 \times TDP_{D2}}{f_{TDP_{D2}} \times V_{TDP_{D2}}^2}, and \\ fans_s =& J \forall K(f_i > f_k > f_j) \end{aligned} \tag{4.4}$$

Equation 4.4 allows obtaining from $D$ an estimation of the energy that $D2's$ **CPU** would consume if it executes $M$. Thus, in a negotiation process, this consumption can be analyzed before a migration or duplication operation.

It is important to remember that the formula considers the fan's consumption at a speed $s$. Said speed is related to a certain processor frequency $F_k$.

### The RAM Projection

In chapter 2, we explained that the energy consumption of the **RAM** is divided into both active and background consumption. In addition, we also explained that the number of operations executed by **RAM** is strongly related to the **CPU's** work. Finally, the last section described a method to represent the work differences among

several **CPUs**. Therefore, we consider implicit the heterogeneity of the **RAM** of various devices in terms of said number of operations.

Thereby, to find the energy consumption that a microservice $M$ running in $D$ would produce in a device $D_2's$ **RAM**, we use the formula 4.5.

$$E_{act} = E_{BK_{D2}} \times (fac) + N_{R_D} \times E_{R_{D2}} \times (fac2) + N_{W_D} \times E_{W_{D2}} \times (fac3) \tag{4.5}$$

It is important to say that the factors of the formula 4.5 are the same as the models 2.2 and 2.3. Nevertheless, we have considered slight random differences in each device's $E_{BK}$, $E_R$, and $W_R$ for our experiments ($fac1$, $fac2$, and $fac3$ multipliers). We did this since the model's original values were obtained from linear regression and measurement operations (see section 2.4.3) on a single device, not representing heterogeneity criteria.

In future works, we will look for more precise ways of projecting the RAM's energy consumption

### The NIC Projection

Chapter 2 describes two ways to find the energy consumption of the **NIC**. One that studies its **load** and another based on the number of **packages sent** or **received**. Thus, to calculate the energy consumption that $M$ running in $D$ would produce in the $D_2's$ **NIC**, we use models 2.7 and 2.8 but in terms of $D_2$:

- Analyzing $M$ from the average transfer rate it generates: This projection approach considers (1) the $D2's$ **NIC** power consumption in its active state, $W_{u_{D2}}$, (2) the average load that $M$ produces in $D's$ **NIC**, $L_{M_D}$, and (3) the maximum transfer capacity of $D2$, $L_{MAX_{D2}}$.

$$E_{NIC_{D_M}} = (W_{u_{D2}} \times \frac{L_{M_D}}{L_{MAX_{D2}}})(T_D), \tag{4.6}$$

- Analyzing $M$ from the packets it sends or receives: This projection approach considers (1) the $W_{S_{D2}}$ and $W_{R_{D2}}$ values to represent $D2's$ **NIC** power consumption when sending and receiving packets, (2) the number of packets sent ($NS_{M_D}$) or received ($NR_{M_D}$) in $D$, (3) the respective average packets' size ($S_{x_D}$) in $D$, and (4) the corresponding total transfer speed ($L_{MAX_{x_{D2}}}$) of $D2$.

$$E_{NIC_{D2_M}} = W_{S_{D2}} \times \frac{NS_{M_D} \times S_{s_D}}{L_{MAX_{s_{D2}}}} + W_{R_{D2}} \times \frac{NR_{M_D} \times S_{r_D}}{L_{MAX_{R_{D2}}}}, \tag{4.7}$$

### The Storage Device Projection

As Chapter 2 explained, we propose energy models for the **storage device** similar to those we use for the **NIC**. Thereby, we use the same arguments as the previous section for the **storage-device** energy projection approach.

Formula 4.8 considers (1) the $D2's$ **storage-device** power consumption in its active state, $W_{u_{D2}}$, (2) the average load that $M$ produces in $D's$ **storage device**, $L_{M_D}$, and (3) the maximum **storage rate** capacity of $D2$, $L_{MAX_{D2}}$.

$$E_{HD_{D_M}} = (W_{u_{D2}} \times \frac{L_{M_D} \times fac}{L_{MAX_{D2}}})(T_D), \tag{4.8}$$

So far, we have explained the projection operations for each ***U***-space resource. The following sections explain each device's scheduling algorithms, considering these operations.

## 4.4.8 The scheduling algorithm: Analyzing microservices in the U space

Our algorithm considers a device-microservices deployment with the following characteristics:

- All devices are indexed in our **MAAN** configuration, which implements the ***U*** space.
- Each device is indexed on the four edges of ***U*** considering its **CPU**, **RAM**, **NIC**, and **storage device** availability.
- A device $D$ can run one or several microservices $[M_{D_i}...M_{D_j}]$, which have initial resource requirements for **CPU**, **RAM**, **NIC**, and **storage** rate.
- Each microservice is identifiable by a **PID**.
- The **OS** of each device executes a single-priority round-robin scheduling algorithm. Thus, competing microservices obtain resources proportionally to what they request.

Then, each device $D_i$ executes the scheduling algorithm 3, extending the **Kali-green helper seeker** procedure (see algorithm 1). Thus, each **supervisor microservice** considers that a device is in an emergency state when any resource load exceeds 85% of its capacity. Furthermore, this entity also considers a device in an underload state when its resources (all of them) have a load under 5%.

**Algorithm 3:** Help seeker for a device $D_i$ indexed in **U**

```
microserviceList ← KaligreenV2Filtering();
sortByKaligreenV1factors(microserviceList);
size ← size(microserviceList);
while isEmergencySituation() or isUnderLoadSituation() do
    M_ia = microserviceList[size];
    M_ib = microserviceList[size/2];
    M_ic = microserviceList[0];
    if MoveMicroservice(M_ia) = true then
        microserviceList.delete(M_ia);
        size = size − 1;
        continue;
    else if MoveMicroservice(M_ib) = true then
        microserviceList.delete(M_ib);
        size = size − 1;
        continue;
    else if MoveMicroservice(M_ic) = true then
        microserviceList.delete(M_ic);
        size = size − 1;
        continue;
    else
        break;
if loadEmpty(allResources) then
    sleep();
wait();
```

Whenever a device **(D)** enters an emergency or underload state, the algorithm tries to migrate microservices. To do that, it first uses the **KaligreenV2** tables (section 4.3.5) to select a list of migratable microservices. Then, it sorts that list using **Kaligreen's** consumption factors approach. Finally, the algorithm attempts to move either the heaviest microservice, the middle-positioned, or the lightest one, recomposing the list in each iteration. We have considered only these three "pertinent" possibilities to avoid overwhelming the network with repetitive queries.

If $D$ does not have microservices running (after the underload state), it enters sleep mode to save energy.

For its part, algorithm 4 describes the migration function. It looks for device candidates in the **MAAN-based** structure for each resource requested by the microservice $M$. Then, it finds a single metadata list of devices capable of running $M$. If the candidates are not locked and simultaneously are compatible with some

**KaligreenV2** criteria, the method projects the power consumption considering the approaches explained in section 4.4.6. The function ends by migrating $M$ to the device where it will consume the least amount of power.

**Algorithm 4:** ***MoveMicroservice(M)***

```
1  V ← buildVector(M);
2  ResourceCandidateLists ← iterativeFindInU(V);
3  CandidateList ← intersect(ResourceCandidateLists, V);
4  foreach D_l in CandidateList do
5      sendVector(V, D_l);
6      CONF_{D_l} ← WaitForCandidateReponse(20);
7      D_{l_KaligreenV2} = isCandidatePossible(D_l, M);
8      if CONF_{D_l} and D_{l_KaligreenV2} then
9          lock(D_l);
10     else
11         DeleteFromCandidateList(D_l);
12 if isEmpty(CandidateList) then
13     return false;
14 foreach D_l in CandidateList do
15     cpuP ← projectCPU(V);
16     ramP ← projectRAM(V);
17     nicP ← projectNIC(V);
18     storageP ← projectStorage(V);
19     c ← calculate(cpuP, ramP, nicP, storageP);
20     if c < calculateConsumption(cheapestDevice) then
21         cheapestDevice ← D_l;
22 moveMicroservice(M, cheapestDevice);
23 return true;
```

## 4.5 Chapter summary and contributions

This chapter presents our distributed scheduling algorithms and the tools they need to be deployed. Among the latter, our middleware **Kaligreen** implements resource analysis and device negotiation operations in each network node. These tools have allowed us to develop procedures considering various heuristics. For example, from a design point of view, we have defined various types of microservices, data, and application graphs. Then, we implemented distributed scheduling using smart microservice-data selection, simple negotiations, multidimensional spaces, and

special data structures to schedule them. To the best of our understanding, this is the first work that approaches distributed scheduling for energy savings in this way.

The next chapter will describe **PISCO**, the simulator we have created to execute and test the architecture and algorithms explained here. This tool allowed us to define heterogeneous applications, devices, and networks easily. Moreover, it allowed us to run and evaluate our scheduling algorithms efficiently.

CHAPTER 5

# THE PISCO SIMULATOR

# 5 The PISCO Simulator

## 5.1 Introduction

Nowadays, many architectures for software components (e.g., Docker-Kubernetes, Kalimucho, etc.) aim to create modular applications efficiently. As described in chapter 3, these entities are scheduled to optimize variables such as quality of service, the physical location of devices, load balancing, or energy consumption.

As explained in chapter 4, we propose distributed scheduling strategies for load balancing and energy savings using intelligent device negotiations and multidimensional data structures. To do that, we use microservice-based applications as the input to our algorithms and the formulas in Chapter 2 as energy measurement tools.

This chapter explains **PISCO**, an innovative simulator to deploy energy-saving methods in microservices-based networks. We have designed and implemented it as a deployment and evaluation tool for the techniques described in the previous paragraph. **PISCO** enables its users to focus uniquely on scheduling approaches and their hardware-software repercussions. This way, they don't need to worry about low-level network configurations or **OS** issues to evaluate their methods.

**PISCO** can deploy and schedule (move, duplicate, start/stop) **MS** (microservices) and their dependencies on various devices supporting heterogeneity criteria (**MS's** types, data and restrictions, and devices' CPU, bandwidth, RAM, Battery, etc.). Moreover, it manages scheduling algorithms and their repercussions, considering the following aspects:

- Both the capabilities and the particular characteristics of each device's hardware component.
- The simple configuration (through a GUI or an API) of centralized (i.e., client-server), decentralized (i.e., p2p architectures like kademlia[157], chord[206], or multidimensional systems such as **MAAN**[23]), or hybrid networks.
- The definition and evaluation (tracking and application of linear regression techniques) of QoS, energy metrics, and quantity of operations (movements, duplications, deployments, and deletions) related to an algorithm.

- The unexpected network circumstances or user strategies consisting of devices appearing, disappearing, turning on, or off.

Finally, **PISCO** is agnostic and not dedicated to a particular middleware. It can virtualize an environment controlled by any software that deploys and schedules microservices. That allows researchers to create any scheduling algorithm and measure its consequences in the terms they define.

## 5.2 Methodology

This chapter explains our **PISCO** simulator's context, entities, and functioning. To understand its creation and scope, section 5.3 explains and compares the framework of several important simulators considering different abstraction levels. Then, to explain the **PISCO's** engine, section 5.4.1 details the simulator entities, which are typically instances of java objects. Finally, pursuing the same objective, section 5.4.2 explains the operations that our tool supports.

We finish the chapter by explaining the corresponding conclusions and contributions in section 5.5.

## 5.3 Existing tools

To understand and evaluate any distributed environment, the scientists may use modeling tools supporting several appropriate criteria. For example, **packet-oriented** simulators allow evaluating network traffic and specific transmission protocols. Among them, Packet Storm[178], IP WAN Emulator[100], SENS[209], and Cisco Packet Tracer[34] allow modeling scenarios to evaluate data streaming techniques and features, latency, data loss and duplication, and others. For its part, the O-ICN Simulator[3] allows evaluating the network status while modeling a particular architecture called Information-Centric Networking (ICN). Finally, GreenCloud[88] allows understanding of energetic issues at the packet network-protocol level.

The simulators above are helpful for modeling protocol-oriented scenarios and studying connection phenomena. However, we consider that these tools cannot replicate the behavior (i.e., use of resources, application **QoS** and energy consumption) of service-based architectures, where a higher level of abstraction is needed.

Considering a higher abstraction level, Petr Novotny et al.[173] propose a framework for simulating architectures based on MANET (Mobile ad hoc network) service networks. It includes service, messaging, and network layers to model services'

complex message exchange behavior. For this purpose, they consider approaches like cascading flows of messages in challenging conversations, comprehensive client-driven workload profiles, and the propagation of faults through services.

This simulator is very efficient in studying services information exchange. However, it does not consider the energy repercussions of services deployment, nor does it allow to know the best way to schedule them in a given network.

At the services deployment level, Cloudsim[37] is a complete simulator that allows modeling service-based architectures with different virtualization policies (i.e., VMs, containers, etc.) at various CLOUD/GRID layers. It enables the study of diverse repercussions concerning energy and **QoS** when managing services. For its part, DockerSim[171] considers not only strategies at service-based architecture but also entire packet-level network and protocol behaviors. Furthermore, μqSim[246] evaluates the execution of microservices deployed with a centralized scheduler, analyzing their latency, requests, and **QoS**. This simulator has a power management engine and handles dependencies between microservices. However, it does not consider decentralized scheduling or heterogeneity in the execution environment. Next, the iFogSim[90] simulator offers an approach to ensure **QoS** and efficient power management by moving services from the cloud to the network's edge. For its part, InterSCity[71] is a robust microservices-based platform that counts on the *InterSCSimulator*. It performs load balancing and microservices scheduling while keeping awareness of power consumption and resource availability.

These simulators are very competent in analyzing scenarios in which services and containers are deployed. However, we consider that they do not offer a straightforward way to execute and evaluate distributed algorithms of the level at abstraction that we propose. That is, considering heterogeneous devices belonging to different network topologies, building network overlays compatible with multidimensional structures, analyzing the relation between energy consumption and a defined QoS, and others.

Table 5.1 shows the characteristics above of some simulators offered by important scientists. Herein, each letter has the following meaning:

- Feature A. Considers behavior aspects at the connection level (protocols, data loss, etc.).
- Feature B. Considers energy consumption at connection level (protocols, data loss, etc.).
- Feature C. Supports microservice deployment in the cloud and local network environments.

| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| PacketStorm | X | | | | | | | | |
| IPWAN | X | | | | | | | | |
| PacketTracer | X | | | | | | | | |
| GreenCloud | X | X | | | | | | | |
| SENS | X | | | | | | | | |
| O-ICN | X | | | | | | | | |
| [Petr Novotny et al. 2016] | | | X | | | X | X | | |
| CloudSim | | | X | | X | X | | | |
| DockerSim | X | | X | | | X | X | | |
| µqSim | X | | | X | X | | X | | |
| iFogSim | | | X | | X | X | X | X | |
| InterSCity | X | | X | X | X | X | X | | |
| **PISCO** | | | **X** | **X** | **X** | **X** | **X** | **X** | **X** |

**Table 5.1:** Simulators' features

- Feature D. Considers special types of microservice management.
- Feature E. Considers energetic issues when operating microservices.
- Feature F. Considers communication phenomena among services.
- Feature G. Considers Microservices dependencies when operating them.
- Feature H. Supports centralized and non-centralized scheduling algorithms, allowing to model dynamic architectures and energy-QoS criteria easily.
- Feature I. Allows a transparent way of creating custom network overlays such as multidimensional P2P networks.

For us, it's important to enable deploying and scheduling modular applications, focusing mainly on their performance and impact in any distributed environment. For this reason, table 5.1 shows the philosophy of our simulator. **PISCO** is strongly focused on applying both centralized and distributed scheduling algorithms, allowing to study at deployment/execution time the following criteria: 1) The definition, measurement, and modelization of applications QoS, 2) the workload of each node's hardware components, and 3) the energy consumption of a particular device (or even one/some of its hardware components) as well as of the entire custom network.

Furthermore, regarding this last point, **PISCO** also allows studying, simply and visually, the distributed scheduling considering multidimensional aspects.

Moreover, in our algorithms context, we need a tool capable of deploying network overlays with the same dynamism and multidimensionality as several data structures. Within these, we also needed the tool to execute and evaluate different scheduling heuristics dynamically. This dynamism consists in the definition of heterogeneous devices, software, and topologies. As no tool we explored fulfilled these functions, we decided to design and develop **PISCO**.

The next section presents **PISCO's** structure, capabilities, and operations.

## 5.4 The PISCO simulator

To study the "best" way to deploy and schedule distributed applications in custom networks of heterogeneous devices, we developed **PISCO**. **PISCO** is a desktop and portable java-simulator that can manage (deploy/schedule) any distributed application in any network architecture (i.e., non-centralized, centralized, multidimensional, horizontal, and others.).

In our simulator context, an application is defined as a directed graph whose nodes are microservices and edges the connections among them. Moreover, every application is deployed on a graph of connected devices, where the nodes could be: (1) devices with heterogeneous characteristics (dependant on batteries, static or mobile, devices with different network interfaces, etc.) or (2) abstract entities such as clusters or cloudlets. Furthermore, the device graph's edges are the network connections among nodes (ethernet, wireless, 4g, or Bluetooth), which can be: (1) physically direct (cable or wireless), or (2) logical (through an overlay structure). Both types of connections allow the transfer rate at which the microservices can send or receive data from each other.

The following section explains the operation and interaction of each of the elements of both graphs.

### 5.4.1 The simulator entities

Entities in **PISCO** are instances of Java classes. To understand their interaction, Figure 5.1 shows a condensed version of our simulator's class diagram. That schema is deeply explained in the following sections.

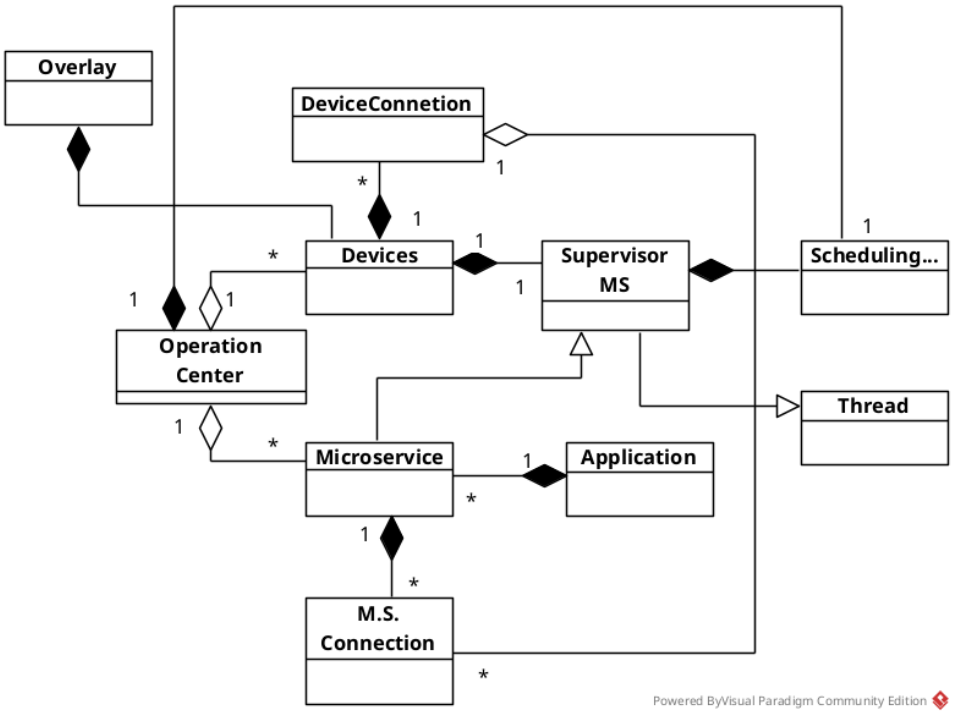


**Figure 5.1:** PISCO's condensed class diagram

### Device

In **PISCO**, a device is any processing entity identifiable by a unique ID. By default, it supports desktop computers, laptops, and smartphones. Moreover, users can also define and configure their own type of device.

Each device has different capabilities in terms of CPU, RAM, hard drive, network, and battery (optional), depending on its type and custom configuration. However, users can also create different hardware components for each device.

On the other hand, each device can execute microservices, providing hardware resources according to their needs. For that, **PISCO** supports scheduling philosophies such as proportional share scheduling, round-robin, or a simplified version of CFS[128], which uses a red and black tree to organize processes.

Finally, each device implements a distributed middleware (like the supervisor service in section 4.3). On the one hand, the latter manages the default device's hardware components according to the following criteria:

- **CPU**:

  - Current load and availability in percentage relative to the clock frequency.
  - Capacitance, TDP, and voltage.
  - Presence or absence of PCPG, DVFS, or Overclocking.
  - Custom energy formula.
  - Power expenditure in Watts.
  - Energy expenditure in Joules.

- **RAM**:
  - Current and available load in percentage and MB.
  - Current read and write operations.
  - Custom energy formula.
  - Power expenditure in Watts.
  - Energy expenditure in Joules.

- **NIC and storage device**:
  - Current and available rate in GB/s/MB/s/percentage.
  - Custom energy formula.
  - Power expenditure in Watts.
  - Energy expenditure in Joules.

- **Battery**:
  - Used and available in terms of Watt-Watts hours.

- **GPS position**:
  - Latitude and longitude values.

And on the other hand, it executes distributed scheduling algorithms (i.e., moving or duplicating a microservice to another device, performing negotiations, managing load balancing, tracking the number of operation hops, etc.).

It is important to mention that the middleware running on each device also enables centralized scheduling. For this, **PISCO** configures a node as a master, being able to invoke the execution operations of the peers it controls.

### Network overlay

**PISCO** implements networks in the form of data structures in which the indexed elements are devices. By default, our simulator supports graphs representing networks of direct connections, n-trees representing hierarchical networks, and multidimensional lists representing multi-attribute spaces.

The screenshots in figures 5.2 and 5.3 show two network configurations. The first one shows a devices graph, where each device is physically connected with some peers. For its part, the second one shows a **MAAN** network overlay (see chapter 4) in its **CPU** dimension.

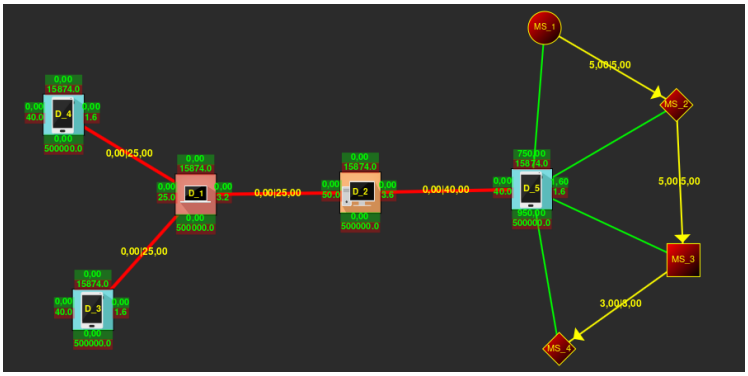


**Figure 5.2:** Network data structure: Devices graph

Finally, as with the other entities, the user of our simulator can create other overlays.

### Microservice

For **PISCO**, a microservice is a functional entity identifiable by a unique ID and with a defined function. The latter is a string code representing the microservice functionality and scheduling restrictions (see section 4.3.3). For instance, by default, PISCO defines three types of microservices: graphical interface, calculation, and data management. The differences among them are their default resource consumption, the amount of data they send/receive, their size, and the restriction of being moved or not from one device to another. On this point, it is also important to point out that the simulator users can create their own types of microservices.

When deployed on a device, each microservice claims a certain amount of CPU, network, RAM, and disk resources. To manage that, **PISCO** defines the following two attributes for each resource:

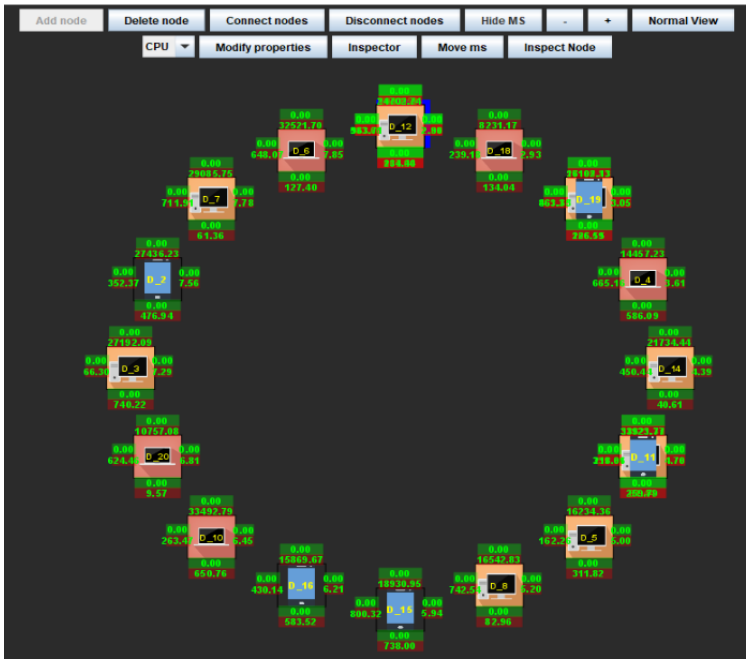


**Figure 5.3:** Network data structure: Multidimensional circular lists

- The average expected resources: It defines the microservice's needs to run at 100% of QoS.

- The average current resources: It defines the resource status of a microservice, which depends on the other peers competing for resources and the OS heuristic.

### Connection

**PISCO** defines and manages two kinds of connections: Physical/logical among devices and functional connections among microservices. A physical connection logically supports many microservices' connections. That defines the current and maximum possible transfer rate (i.e., between 2 devices: the maximum transmission capacity of the device with the **NIC** with less transmission capacity).

Our simulator also allows graphically analyzing, on the one hand, the status (i.e., at energy and load level) of devices' **NICs** (even if they belong to abstract entities such as clusters or cloudlets). On the other hand, it tracks the microservices' connections (i.e., expected transmission rate vs. real transmission rate) using physical ones. For that, connections among microservices logically store the device

connections they use to communicate. Thereby, the supervisor microservices are aware of the device's path they use and their current/expected transfer rate.

#### Application

**PISCO** defines applications as directed graphs of microservices. Herein, each node has connections as dependency relationships with other microservices that may run on different devices. In **PISCO**, applications are also identified with a unique ID.

#### Operations center

If needed, **PISCO** holds a centralized entity that stores all references to connections and existing devices of a deployed scenario. In this way, the simulator users can track their algorithm's behavior and execute centralized scheduling algorithms.

#### Abstract entities

Additionally, **PISCO** also implements entities such as cloudlets and clusters as resource providers. Both can be deployed and connected similarly to devices but considering different intra-scheduling politics. With them, our simulator is compatible with small networks of user devices and networks based on cloud/edge technologies.

### 5.4.2 The simulator operations

**PISCO** can apply different heuristics for microservices (re)deployment and scheduling through a network. For that, the simulator implements the following operations:

#### Devices and abstract entities deployment

Any simulator user can declare default or custom devices and abstract entities (i.e., desktop computer, laptop, smartphone, etc.) by specifying the desired resources (CPU, RAM, NIC, storage device, or battery). Then, the devices will be indexed by **PISCO** as indicated in the *add* method of the network overlay. This method may consider load, hierarchy, or energy criteria.

On the other hand, **PISCO** also supports physical positions in the form of *longitude* and *latitude* points on a Cartesian plane, as shown in figure 5.4. That allows studying deployment techniques based on distance heuristics or methods to improve communication paths.

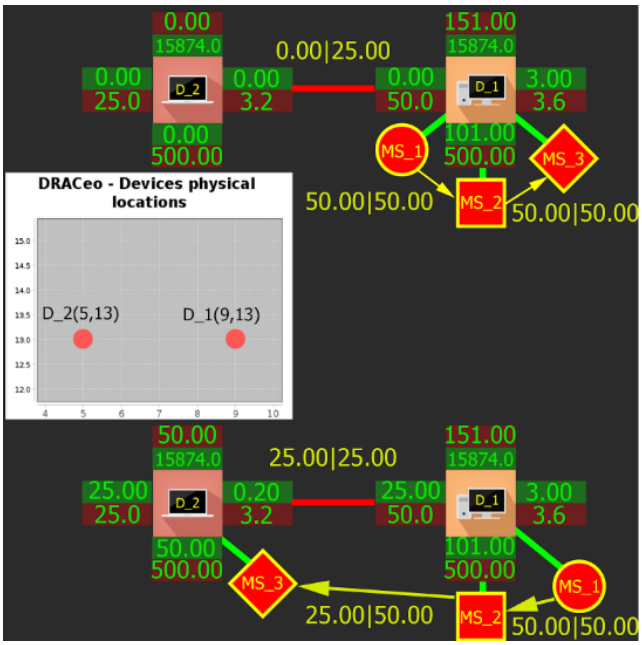


**Figure 5.4:** Devices in PISCO and their positions

### Devices and abstract entities suppression

**PISCO** implements two device suppression methods: 1) Manually, by the user's decision, and 2) automatically, according to the desired strategy. A user can activate a "disappear" option for each device, causing the simulator to delete it after a random, established, or battery-related time. This option simulates unexpected loss of connection (i.e., a mobile device entering a building, battery loss, system crash, etc.).

The device suppression invokes the *delete* function of the network overlay.

### Microservices deployment

**PISCO** allows simulating the execution of microservices on devices. Once deployed, a microservice uses a quantity of resources (see figure 5.4) for a determined time or indefinitely. This time can be set by: (1) the **PISCO** UI in an interactive way or (2) automatically by an operation center's function. Furthermore, the user must also specify the microservices' size, defining the amount of data sent over the network when it is moved or duplicated. This attribute allows analyzing the cost of reconfigurations in terms of efficiency and energy.

### Microservices suppression

Like deployment, a microservice can be killed using a simulator UI button or according to a predefined amount of time in the operation center (also set using the simulator UI). When suppressed, the microservice will stop using the device's resources and will no longer be available for any item.

### Microservices migration and duplication

**PISCO** can move or duplicate microservices from one device to another. On the one hand, these operations can be done from the operations center, which maintains the full list of microservices and connections. On the other hand, the operations can be performed from a particular device (it knows only its connected devices) according to the algorithm its runs.

When a microservice is migrated, it first releases the resources of its current device. Then, **PISCO** simulates the moving process by using the active network connection for a time based on the microservice's size. Finally, it starts to compete (as there could be other microservices) for the resources of the new device.

The duplication operation is similar to the previous one but without killing the microservice on the source device.

About these operations, it's important to mention that **PISCO** always keeps (non)centralized microservices-devices graphs updated. Thus, for example, if a microservice is moved and becomes unreachable for one of its dependencies, the user can specify search mechanisms for the same instance in the devices graph.

Figure 5.4 is a screenshot of one of the **PISCO's** UI. It shows a simple scenario where we deployed an application consisting of three microservices ($MS\_1$, $MS\_2$, and $MS\_3$) communicating with each other at $50MBps$ and two connected devices (a laptop, $D\_2$, and a desktop computer, $D\_1$). Then, each device shows its status in the outline of its icon in terms of used/available network (left: $\frac{X}{50}MB/s$), RAM (top: $\frac{Y}{15874}MB$), CPU (right: $\frac{Z}{3.6}GHz$), and storage rate(botton: $\frac{W}{500}MB/s$).

In this example, if $MS\_3$ is moved to the device $D\_2$, the transmission quality rate between both microservices $MS\_2$ and $MS\_3$ declines. That is because the physical network connection is limited to $25MB/s$ (between the two **NICs**, $D\_2's$ one has the least transfer capacity). It is important to say that this does not necessarily affect an application's **QoS** level considering its definition and type. In our case, our algorithms always seek to have the highest possible **QoS** assuming the approaches described below.

### Microservices and devices Start/Stop

**PISCO** allows starting and stopping devices or microservices in a scheduled or manual way. Both operate as the mentioned deployment and suppression operations, but without deleting these entities permanently. Thereby, they can also be restarted manually or in a scheduled way.

### QoS definition

**PISCO** allows users to define their own **QoS** philosophies, either for microservices or applications. However, the simulator offers two default heuristics: One, "non-dependent," and another, "dependent."

These default definitions aim to model different application behaviors. The first, for example, describes applications whose modules can have: (1) different replicas, enabling load-balancing strategies in the event of **QoS** affection, and (2) non-critical functionalities, such as batch processing or backups creation. Here, users must specify the module's importance, as shown in the following paragraphs. On the other hand, the second describes applications whose modules are unique and important, causing that if one of them is affected, the behavior of the entire application changes.

For the non-dependent approach, **PISCO** calculates the microservices' **QoS** value in proportion to the hardware resources needed-obtained. On the other hand, the applications' **QoS** is obtained proportionally to its microservice's **QoS** value.

The non-dependent microservices' **QoS** is defined as follows (including equation 5.1):

- For a microservice $M$ being executed in a device $D$, **PISCO** defines a QoS=100% considering the resources that $M$ requires to run in terms of **CPU** frequency in $GHz$, **RAM** consumption in $MB$, **network** transfer rate in $MB/s$, and **storage** transfer rate in MB/s.

- Then, there might be some differences between the resources $M$ requires to run and those it gets from $D$. We express those differences as percentages (from 0 to 100) using the following symbols: (1) $D_C$, for **CPU** frequency, (2) $D_R$, for **RAM**, (3) $D_N$, for **network** transfer rate, and (4) $D_H$ for **storage** transfer rate.

- For each of these variables, a user must specify an impact value. This value must be considered from 0 to 1 so that all the values add up to 1.

$$\begin{aligned} QoS_M =& I_1 D_C + I_2 D_R + I_3 D_N + I_4 D_H \\ where, & \sum_{i=0}^{4} I_i = 1 \end{aligned} \tag{5.1}$$

For example, let us define that (1) $M$ needs a certain quantity of **GHz** of **CPU**, **MB** of **RAM**, and **MB/s** of **NIC** and **storage** transfer rate, (2) $M$ finds only half of these resources in $D$ ($D_C = D_R = D_N = D_H = 50\%$), and (3) the user decides to configure the impact factor of each resource equally to 0.25 for each component ($I_1 = I_2 = I_3 = I_4 = 0.25$). Then, the **QoS** of $M$ is defined as follows:

For example, let us define that (1) $M$ needs a certain quantity of **GHz** of CPU, **MB** of RAM, and **MB/s** of **NIC** and **storage** transfer rate, (2) $M$ finds the following resources available in D:$[D_C = 90\%, D_R = 6\%, D_N = 9\%, D_H = 5\%]$, and (3) the user decides to configure the impact factors of each resource as follows:$[I_1 = 0.7, I_2 = 0.1, I_3 = 0.1, I_4 = 0.1]$. Then, the **QoS** of **M** is defined as:

$$\begin{aligned} QoS_M &= 0.7 \times 90 + 0.1 \times 6 + 0.1 \times 9 + 0.1 \times 5 \\ QoS_M &= 73.2\% \end{aligned} \tag{5.2}$$

As seen in the equation 5.2, the $M's$ **QoS** remains at an acceptable value despite the unavailability of **RAM**, **network**, and **storage** rate. That is because the user has defined that the **CPU** is the most relevant component for quality of service. For example, this scenario may belong to an application for calculating machine learning predictions. Here, the **CPU** is highly required to do the calculations. However, the only use of the rest of the peripherals may be the storage of data executed by independent functions.

Next, the non-dependent application's QoS, $QoS_{app}$, is defined by the quality of service of each of the $N$ microservices, $QoS_i$, that compose it. Again, we consider the impact value $I_i$ ,that the user must define in the same way as the last method. The following formula shows this approach:

$$QoS_{app} = \sum_{i=0}^{n} QoS_i * I_i, where \sum_{i=0}^{n} I_i = 1 \tag{5.3}$$

On the other hand, **PISCO** also offers by default a "dependent" heuristic, in which the **QoS** of a microservice $M$ is limited by the least satisfied demanded resource. As before, we consider here the impact parameter, $I$, specified by the user. The following formula shows this approach:

$$QoS_M = Min(I_C \times D_C, I_R \times D_R, I_N \times D_N, I_H \times D_H) \tag{5.4}$$

Then, the **QoS** of an application, $APP$, composed by a set of microservices, $M = [M_0...M_n]$, is limited by the microservice $M_i \in M$ with the lowest **QoS** value, as shows the following formula:

$$QoS_{APP} = Min(QoS_{M_i}) \; \forall \; M_i \in M \tag{5.5}$$

### QoS analysis

Our simulator can show the **QoS** of applications and microservices at a specific rate and time interval defined by the user while executing a scheduling algorithm. For that and analyzing purposes, **PISCO** supports graphical PLOT, performance metrics storage, and linear/polynomial regression.

### Power consumption parameters definition

**PISCO** uses, by default, the formulas described in Chapter 2 to find the power consumption of the **CPU**, **RAM**, **NIC**, and **storage** devices.

### Power/Energy consumption analysis

Our simulator analyzes the power and energy consumption in the same way as with the **QoS**. It supports graphical PLOT, performance metrics storage, and linear/polynomial regression. For example, the screenshot in Figure 5.5 shows the **QoS** and power consumption metrics of **Kaligreen's** default algorithm over some of its iterations.

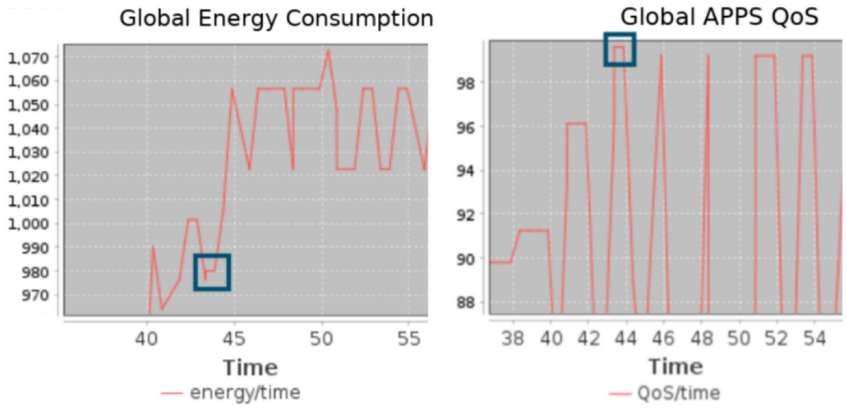


**Figure 5.5:** PISCO evaluation window

Herein, **PISCO** detects the iteration in the second 45 as the best since the **QoS** is the highest possible (100%) and the network power consumption the lowest ($978W$). The rest of the curve results from deploying several microservices after said second, provoking the simulator to restart the algorithm looking for another solution.

### (Non)centralized scheduling algorithm Start/Stop

When **PISCO** launches a scheduling algorithm, it starts the concerning middleware-instances threads (the master's one for centralized and all the devices' middleware for non-centralized). Then, the simulator automatically displays the following metrics for performance analysis: (1) Run time, (2) number of operations performed, (3) data transmitted by movements/duplications on a device or across the network, (4) energy used for movements/duplications on a device or across the network, and (5) per device and global load. For example, the screenshot in the figure 5.6 shows these variables.

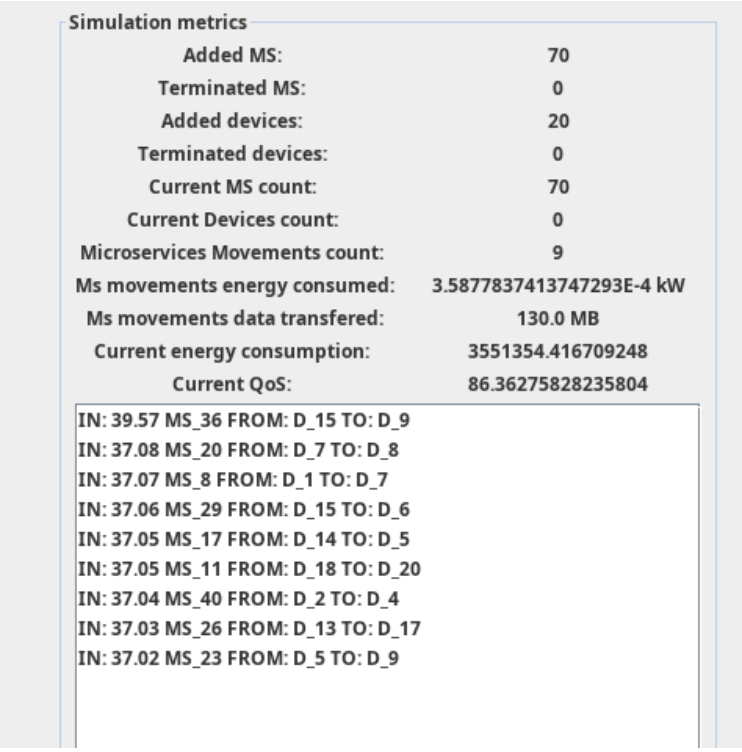


**Figure 5.6:** PISCO metrics window

#### Save/Load scenario from an initial or an already modified state

To facilitate the evaluation of distributed algorithms, **PISCO** can save all the objects of a scenario (including their state) in JSON files. Among these objects, we can mention devices, connections, overlays, and other instances.

## 5.5 Chapter summary and contributions

This chapter explains the justification and operation of the simulator we created: **PISCO**. Regarding the first point, we needed a tool capable of deploying network overlays with the same dynamism and multidimensionality as several data structures. Within these, we also needed the tool to execute and evaluate different scheduling heuristics dynamically. This dynamism consists in the definition of heterogeneous devices, software, and topologies. Moreover, concerning the second point, the chapter explains the entities and operations of the simulator.

To the best of our understanding, only **PISCO** fulfills all these needs. For this reason, we use our tool to study our distributed algorithms.

On the other hand, given the novelty of our simulator, it was selected by SATT Aquitaine (AST Innovation) for a maturation process. Next, a software license and a patent were registered at INPI. Finally, two licenses are currently deployed (Univ. La Rochelle, Univ. Paris 1 - Sorbonne), and a technological transfer is under discussion with the international group SPIES.

The next chapter describes the experiments that demonstrate the effectiveness of our methods from chapter 4, which we implemented in **PISCO**.

# CHAPTER 6

# EXPERIMENTS AND RESULTS

# 6 Experiments and results

## 6.1 Introduction and methodology

This chapter explains the experiments and results of the measurable methods proposed throughout this thesis. In each part, the latter offers a contribution according to what it studies. Chapter 2 proposes mathematical models to estimate the power consumption of the network card and storage device. Then, chapter 3 describes an algorithmic survey approach for distributed scheduling techniques. For its part, chapter 4 explains our distributed scheduling algorithms. Finally, chapter 5 details the design structure of the simulator that we have created.

Here, we will analyze the proposals that we can evaluate through experiments. In section 6.2, we analyze the approaches of Chapter 2, and in section 6.3, the ones of chapter 4. In each of these, we describe the type of experiment, its settings, its results, and the corresponding conclusions.

## 6.2 Analyzing the power formulas for the NIC and storage device

To analyze the power consumption of both components, we have run specialized benchmarks and applications for their saturation on a desktop computer. We do this experiment while waiting to make measurements on the SPIE company servers as part of the partnership with our laboratory and the SATT.

The specifications of our testing elements are the following:

- Desktop computer:
  - **Motherboard**: Gigabyte GA-890GPA-UD3H.
  - **CPU**: AMD Phenom II x4 955 - 3213.57 MHz.
  - **RAM**: 4096 MB Dual Channel.
  - **Graphics card**: Nvidia GeForce GT 710.
  - **Storage device**: Western Digital WDC WDS 480 GB.
  - **OS**: Windows 8 (free of installed programs).

  - **NIC**: TP-Link 72.2 Mbps

- Measuring instrument: Wiring Type Power Monitor.

### 6.2.1 The storage device

To measure the storage device's power consumption, we ran for half an hour the following benchmarks with our power monitor connected to the PC's power source: AS SSD Benchmark[217], Crystal Disk Mark[49], SiSoftware Sandra[199]. These saturate the hard disk in its read and write operations, using the rest of the hardware components as little as possible.

We did four types of experiments to study four levels of workloads in the component. We ran the benchmarks so that they saturate 100%, 75%, 50%, and 25%, of the storage's read and write capacity. We choose this method to study the relationship between workload and power consumption (see section 2.6). To perform these percentages, we have programmed a script that interrupts the benchmark proportionally to these values during each experiment's time. We did this by mimicking the operation of any operating system. That is, giving the full access of the resource to a process for a time calculated from its priority and other competing processes.

| Operation | Without Benchmark (Watts) | With Benchmark (Watts) |
|---|---|---|
| Workload at 100% | | |
| Writing op. | 64.27 | 72.88 |
| Reading op. | 64.4 | 71.06 |
| Workload at 75%. | | |
| Writing op. | 66.01 | 70.24 |
| Reading op. | 66.66 | 69.1 |
| Workload at 50%. | | |
| Writing op. | 64.33 | 68,36 |
| Reading op. | 66.72 | 68.23 |
| Workload at 25%. | | |
| Writing op. | 65.63 | 67,5 |
| Reading op. | 67.20 | 67.7 |

**Table 6.1:** Measurement results for the Storage Device

Table 6.1 shows the measurement results. It is important to note that the difference in consumption between using and not using a benchmark is as follows:

- With a workload of 100%: 8.61 for Writing op. and 6.66 for reading op.
- With a workload of 75%: 4.2 for Writing op. and 2.5 for reading op.

- With a workload of 50%: 4.03 for Writing op. and 1.5 for reading op.
- With a workload of 25%: 1.95 for Writing op. and 0.59 for reading op.

These results are proportional to the load generated with the benchmarks. Inaccuracies shown may be due to irregularities in power consumption and management of other PC's internals.

On the other hand, it is important to point out that the average consumption found for a load of 100% is approximately 6 and 4 watts higher than indicated in the disk datasheet[187], respectively, for each operation. These differences correspond to the other peripherals' functions.

Finally, we also find that the write operation is, on average, more expensive at 1.9W than the read operation.

### 6.2.2 The NIC

Using the same instruments and methodology as the previous device, we run the SpeedTest[204] and nPerf[174] online tools at the following four intensities: 100%, 75%, 50%, and 25% of the **NIC** Download-Upload capacity.

| Operation | Without Benchmark (Watts) | With Benchmark (Watts) |
|---|---|---|
| Workload at 100% | | |
| Down.op. | 64.5 | 101.57 |
| Upl. op. | 65.61 | 78.07 |
| Workload at 75% | | |
| Down. op. | 64.3 | 91.49 |
| Upl. op. | 65.6 | 74.59 |
| Workload at 50% | | |
| Down. op. | 64.5 | 79.88 |
| Upl. op. | 65.61 | 70.9 |
| Workload at 25% | | |
| Down. op. | 64.2 | 71.14 |
| Upl. op. | 65.39 | 67.8 |

**Table 6.2:** Measurement results for the NIC

Table 6.2 shows the measurement results. The difference in consumption between using and not using a benchmark is as follows.

- With a workload of 100%: 37.07 for Downloading op. and 12.46 for Uploading op.

- With a workload of 75%: 27.19 for Downloading op. and 8.99 for Uploading op.
- With a workload of 50%: 15.38 for Downloading op. and 5.29 for Uploading op.
- With a workload of 25%: 6.94 for Downloading op. and 2.41 for Uploading op.

These results are proportional to the load generated with the benchmarks. Inaccuracies shown may be due to irregular power consumption of the PC's internals. Moreover, we also see that the download operation is, on average, more expensive at 14.35W than the upload operation.

### 6.2.3 Conclusions and considerations

The results obtained from the measurements of the **storage** and **NIC** devices show the following findings:

- The power consumption is proportional to components load. That verifies our formulas 2.7 and 2.10. However, it is necessary to consider input and output operations differently.
- Reading, writing, uploading, and downloading operations may have different power consumption values.
- To improve the accuracy of consumption formulas for any component, it is necessary to consider the behavior of other PC internals.

To enrich the results of our following experiments, we will use the values shown here to describe the consumption of some devices in our scenarios.

## 6.3 Analyzing the distributed scheduling algorithms

This section describes the experiments and results for our two main scheduling algorithms. On the one hand, the one that **Kaligreen** implements by default. On the other hand, the distributed algorithm that based on multidimensional spatial structures.

To perform our experiments, we considered a collection of metrics to define devices' capabilities and hardware consumption variables. We gathered these values

from data sheets, public hardware benchmarks, and the last section's experiments. That is to approximate results to real scenarios, facilitating the interpretation of results.

Finally, we use our **PISCO** simulator (see section 5) as a tool for the algorithms' deployment, execution, and analysis of results.

### 6.3.1 Defining devices' capabilities

Chapter 2 considers four different devices' hardware capabilities to study the relation between **load**, **QoS**, and **energy** consumption: **CPU** frequency (GHz), **RAM** capacity (MB), **storage** transfer rate (MB/s), and finally, **NIC** data transfer rate (MB/s).

In our test scenarios, we considered devices with random delimited values for each hardware component capacity. For that, we choose tools that provide us with an insight into hardware resources available in the market.

Table 6.3 shows this heterogeneity approach.

| CPU (GHz) | RAM (MB) | NIC (Mb/s) | Disk (MB/s) |
|---|---|---|---|
| 1.2-4.8 | 2000-32000 | 101.0-1000 | SSD: 101.0-800<br>HDD: 80-160 |

**Table 6.3:** Devices' capabilities

To define the **CPU** and **RAM** intervals, we studied public hardware benchmarks performed independently by millions of users[221][201]. Next, to define the **NIC** and **storage** intervals, we used hardware and software specialized websites that perform independent tests[220][122].

### 6.3.2 Defining power consumption values

As for the devices' capabilities, we selected **CPU** consumption values following information sheets which consider overclocking, voltage, capacitance, and others[109][107][113]. Then, we obtained the other components' values from manufacturers' datasheets, our last experiment, and scientific hardware specifications[46][195][193][33].

Table 6.4 shows this heterogeneity approach.

| CPU | RAM | NIC | Disk drive |
|---|---|---|---|
| Capacitance: 10 pF Voltage: 1.2v | 3-5W | Idle: 0.4944W Working: (Uploading & Downloading) 1.1349W | Idle (SSD/HDD): 0.05/5.4W Working (Reading & Writing) (SSD/HDD): 2.2/8.0W |

**Table 6.4:** Devices' consumption values

### 6.3.3 Defining MS consumption values

Following the same methodology as in the previous sections, we have defined consumption intervals for three types of microservice (see chapter 4). Our goal is to automate the deployment of software components with heterogeneous hardware needs. Table 6.5 shows this heterogeneity practice.

| Resources | GUI MS. | Control MS. | DB. MS. |
|---|---|---|---|
| CPU (GHz) | 1.2-1.7 | 1.8-3.2 | 0.8-1.2 |
| RAM (MB) | 200-300 | 100-300 | 150-300 |
| Network Upload(MB/s) | 3-5 | 4-7 | 2-5 |
| Network Download(MB/s) | 2-3 | 4-9 | 3-5 |
| Disk (MB/s) (both op.) | 0.1 - 5 | 50 - 100 | 50 - 200 |
| Microservice size (MB) | 50 - 100 | 10 - 80 | 60 - 500 |

**Table 6.5:** Devices' consumption values

### 6.3.4 Scalability test

The objective of this test was to challenge our scheduling algorithms with increasing numbers of microservices and devices over time. The test begins with 20 heterogeneous devices and 40 heterogeneous microservices.

The duration of the experiment consists of 100 seconds, deploying ten random microservices and ten random devices every 10 seconds for the first 80 seconds, leaving 20 final seconds to the scheduling algorithm for stabilizing.

We performed this experiment 100 times to ensure repeatability of testing and consistency of the results.

### 6.3.5 Stress test

The objective of this experiment was to prove the resilience of the algorithms when faced with a constant change in microservices execution. For that, we introduced a

time constraint feature for the latter, which gave them a specific "time to finish." That time ranged from 5 to 30 seconds for the whole test duration of 100 seconds.

This test begins with 250 random microservices deployed among 50 heterogeneous devices. Then we randomly added microservices while the existing ones disappeared. Here, the test guaranteed the network to be stressed.

This experiment was also performed 100 times to ensure repeatability of testing and consistency of the results.

### 6.3.6 Tests metrics definition

To evaluate the performance of our approach, we analyzed five variables in our two experiments.

- The energy consumed by the entire network (Joules).
- The QoS achieved on average by all microservices executed in the network.
- The average number of movements performed by the algorithm.
- The data is transferred by microservices movements (MB).
- Energy cost of microservices movements (Joules).

### 6.3.7 Definition of "success"

To correctly interpret the data results as a "success," we compared the ones of our two algorithms against the original deployment consumption. For that, we considered the same conditions and number of experiment repetitions.

We cataloged our algorithms as successful if, in their results, the energy consumption is either lowered or kept the same while increasing on QoS. That is because the latter is always a crucial variable in any distributed system.

### 6.3.8 Scalability test results

As table 6.6 shows, in the experiments, the multidimensional-based algorithm proved to consume a total average of 27% less energy (692020.73 joules) than the scenario with no algorithm (95452.72 joules) and 11% less energy against the Kaligreen algorithm (98570.94 joules).

Furthermore, the average QoS sustained by the multidimensional-based algorithm was more significant than in the other experiments. The first algorithm

scored an average of 99.24% of QoS against 94.04% of the no-algorithm experiments and 99.21% of the kaligreen algorithm.

It's important to point out that the multidimensional-based and Kaligreen's algorithms performed an average of 55.70 and 50.35 movement operations, transferring 850.29 MB against 775.83 MB, respectively. However, although the number of operations it executed was higher, the multidimensional algorithm proved to have better overall power management.

| | Multid. Algorithm | No algorithm | Kaligreen Alg. |
|---|---|---|---|
| Average of total energy | 69202.7337 | 95452.72 | 78570.94 |
| Average of QoS | 99.2414925 | 94.04 | 99.21 |
| Average of movements | 55.7014925 | 0 | 50.35 |
| Average of data transfer | 850.298507 | 0 | 775.83 |
| Average of movements' Energy consumption | 0.00210982 | 0 | 0.00176535 |

**Table 6.6:** Devices' consumption values

After evaluating these results, we can consider our experiment a success, stating that our algorithm is scalable and stable.

### 6.3.9 Stress test results

This experiment revealed an interesting behavior for the multidimensional approach against the other alternatives. As it is possible to see in table 6.7, this algorithm consumed a total average of 69% more energy (578315.4 joules) than the no-algorithm experiments (340372.9 joules. Nevertheless, that keeps a low level of QoS because of the lack of scheduling operations) and 4% less energy than the naive algorithm (598737.0 joules).

However, the most revealing information shown by this test was the average QoS sustained by each approach.

| | Multid. Algorithm | No algorithm | Kaligreen Alg. |
|---|---|---|---|
| Average of total energy | 578315.444 | 340372.995 | 598737.071 |
| Average of QoS | 92.7182353 | 87.05 | 87.6183333 |
| Average of movements | 912.647059 | 0 | 798.583333 |
| Average of data transfer | 12610.2941 | 0 | 13243.3333 |
| Average of movements' Energy consumption | 0.03020178 | 0 | 0.03184407 |

**Table 6.7:** Devices' consumption values

The multidimensional algorithm scored an average of 92.71% of QoS against the average of 87.61% sustained by Kaligreen's algorithm and 87.05% sustained when no algorithm was applied. That demonstrates that the movements performed by

the first procedure did not only save energy compared to the second one; but were also smarter, keeping the QoS level as high as possible under heavy loads.

On the other hand, these movements were higher than the Kaligreen algorithm's, showing an average of 912.6 operations against 798.5, respectively. Even though this involves a higher average of data transferred (2610.29 MB and 13243.3 MB), the difference became non-significant considering they consumed 0.030 joules against 0.031 joules consumed by the Kaligreen's procedure. That is explained because mainly small microservices are moved under a heavy network resource load.

Overall, our multidimensional algorithm proved to be resilient under heavy workloads, keeping energy as low as possible and, in an inverse way, QoS as high as possible.

### 6.3.10 Conclusions and considerations

The results obtained in these experiments allow us to get the following findings:

- Running any of our distributed scheduling algorithms gives better results than the initial deployment.
- A distributed algorithm based on multidimensional data structures may perform fewer operations than a naive one, such as Kaligreen's default procedure. If the opposite happens, it is always because the QoS value increases. That is because the structure always finds available candidates in an optimal search time.
- Kaligreen's default algorithm does not avoid oscillations or look for other candidates beyond its direct neighbors. For that reason, it does not always find good solutions in terms of consumption or QoS.
- The **PISCO** simulator allows efficient modeling, executing, and comparing of different planning algorithms.

# CHAPTER 7

# CONCLUSIONS AND FUTURE WORK

# 7 Conclusions and Future work

This thesis work seeks to create and execute distributed tools and algorithms to save energy while maintaining the notion of **QoS**. For this, its chapters describe our proposal of a way forward toward intelligent distributed scheduling.

The first step on this path is to have energy information usable by any distributed middleware and simulation tools. Chapter 2 studies this problem. In it, we provide energy formulas for four important hardware components (**CPU**, **RAM**, **NIC**, and **storage**) to describe the consumption of running applications in heterogeneous devices.

For the **CPU**, we have adapted equations that consider its frequency, capacitance, and voltage. Although this model has been tested, we believe that in the future, we need to seek or implement new approaches that consider last-technology capabilities such as dynamic overclocking or P-states. Then, we have also adjusted an already proven model for the **RAM**. Nevertheless, we need to study more heterogenous-representative equations for this component in the future. Finally, For the **NIC** and **storage** device, we have proposed and tested our formulas based on workload. Although they proved to be quite accurate, we have to consider further studying other interfaces like 5G, Bluetooth, different types of SSD, and others.

All the formulas will soon be tested on heterogenous environments provided by companies such as SPIES ICS.

Given the effectiveness of the four models described, we can argue that an application can be modeled with data obtained from the operating system. We have verified this fact using GNU/Linux interfaces, leaving the analysis of other operating systems for the future.

Our next step is to understand how distributed approaches work. For this, chapter 3 shows a different way of categorizing research works based on an algorithmic procedure. The strategy considers 1) important input variables such as hardware components load and devices positions, 2) scheduling operations as migrations or duplications, and 3) optimized variables such as **QoS** and energy consumption. Moreover, we study this flow in (de)centralized environments at different deployment levels, such as cloud, grid, or edge. That makes it easy to understand the scope of new scheduling proposals.

To this day, we continue to study new techniques to enrich, publish, and validate our categorization policy.

Having explored state of the art in distributed environments, Chapter 4 describes our scheduling strategies. We have designed a distributed middleware called **Kaligreen**. It is aware of the software and hardware components of the devices on which it runs. Furthermore, it implements special communication methods that enable centralized and decentralized strategies in any network topology. **Kaligreen** includes a default algorithm that performs neighborhood microservices exchanges after negotiation and microservices-filtering processes.

Using **Kaligreen**, we have designed, implemented, and tested a fully decentralized algorithm inspired by multidimensional data structures, representing one of this thesis's major contributions. That considers the devices as nodes in a space with as many edges as the characteristics of said nodes. In this space, each device can find peers to negotiate microservice migrations or duplications at an optimal computational cost. We have chosen the MAAN network as our multidimensional space test instance. However, we have left evaluating and implementing other structures such as skip-graph for future work.

Observing the results of our algorithm, we can conclude that it is efficient to save energy while preserving the quality of service notion. In addition, our approach allows users to customize and prioritize their variables of interest. That is useful in multi-cloud environments, where the prices of the contracts are directly related to the dynamism of the quality of service offered, energy consumption, high availability, and others. On this matter, we are in the process of coordinating with important companies such as SPIES ICS.

On the other hand, to this day, we are implementing and evaluating an algorithm that considers data (based on barycenter heuristics) and microservices connections. This approach will soon be available and submitted for publication.

Finally, chapter 5 shows the design and implementation of **PISCO**, the simulator we created and use to deploy and evaluate our scheduling algorithms. The importance of our tool is to allow its users to focus solely on their scheduling heuristics. For this, **PISCO** implements an overlay as any data structure (graphs, trees, etc.), devices and software as dynamic objects, and middlewares as threads to execute any scheduling policies.

## 7.1 Long-term perspectives

One of the reasons for the energy waste linked to digital technology is the lack of knowledge about software and hardware behavior. That concerns any user, even

novice, advanced, or expert users. For that reason, the future of **PISCO** and our **algorithms** have an industrial and scientific outlook. One free license is currently deployed in CRI/University Paris 1-Sorbonne. Moreover, a second one is now under deployment at L3I/ University La Rochelle. Next, a technological transfer is under discussion with SPIES ICS with AST Innovation (SATT Aquitaine) – this latter is at the juridic discussion level. In the coming months, we hope to verify scalability and robustness with real use cases with SPIES ICS.

Moreover, our works will also be used as a software-conception validation tool. In the context of a Ph.D. project (in collaboration with Spain), researchers are analyzing the software "footprint energy cost." The objective is to label services (individually) and applications (set of services collaborating).

The project above, jointly with ours, may help design applications while estimating their footprint according to their composition and hardware use. For that, it will integrate a software engineering part.

Furthermore, our contribution set (**PISCO** and **algorithms**) is one of the two core aspects of a future industrial chair. It will be joined to a project on the domain of warehousing (Data Lakes) to help prove new architectures. On the one hand, the objective here is to apply energy-saving techniques more commonly found in dynamic approaches such as Edge/Fog Computing to the massive data analysis architectures, such as data lakes. On the other hand, it aims to perform information processing closer to their sources, enabling new analysis primitives (access/collection, cleaning, examination, consolidation, reporting) such as redundancies and sequencing awareness.

In this project, we need to perform two main operations: 1) Energetically quantify the different functional primitives of data analysis, and 2) quantify the analysis chains and weight them according to the architecture of the information system (massively distributed, sparsely distributed) and external operations (data refresh frequency, TTL - Time To Live, CRUD operation, format, and types, etc. ).

# Bibliography


[1] H. Acar, G. I. Alptekin, J. Gelas, and P. Ghodous. **Beyond CPU: Considering memory power consumption of software**. In: *2016 5th International Conference on Smart Cities and Green ICT Systems (SMARTGREENS)*. 2016, 1–8 (see pages 10, 22).

[2] Hayri Acar. **The Impact of Source Code in Software on Power Consumption**. *International Journal of Electronic Business Management* 14 (Jan. 2016), 42–52 (see page 13).

[3] Suvrat Agrawal, Samar Shailendra, Bighnaraj Panigrahi, Hemant Kumar Rath, and Anantha Simha. **O-ICN Simulator (OICNSIM): An NS-3 Based Simulator for Overlay Information Centric Networking (O-ICN)**. In: *Proceedings of the 1st Workshop on Complex Networked Systems for Smart Infrastructure*. CNetSys '18. New Delhi, India: Association for Computing Machinery, 2018, 13–15. ISBN: 9781450359276. DOI: 10.1145/3265997.3266000. URL: https://doi.org/10.1145/3265997.3266000 (see page 106).

[4] Imtiaz Ahmad, Mohammad Gh. AlFailakawi, Asayel AlMutawa, and Latifa Alsalman. **Container scheduling techniques: A Survey and assessment**. *Journal of King Saud University - Computer and Information Sciences* (2021). ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2021.03.002. URL: https://www.sciencedirect.com/science/article/pii/S1319157821000562 (see pages 42, 54).

[5] Hernán Humberto Álvarez Valera, Philippe Roose, Marc Dalmau, Christina Herzog, and Kyle Respicio. **KaliGreen: A distributed Scheduler for Energy Saving**. *Procedia Computer Science* 141 (Jan. 2018), 223–230. DOI: 10.1016/j.procs.2018.10.172 (see page 76).

[6] AMD. **AMD mProf**. *https://developer.amd.com/amd-uprof/*. 2020 (see page 24).

[7] Anders Andrae. **Projecting the chiaroscuro of the electricity use of communication and computing from 2018 to 2030** (Feb. 2019). DOI: 10.13140/RG.2.2.25103.02724 (see page 5).

[8] Apimirror. **docker / Swarm: Scheduling**. *http://apimirror.com/docker~1.11/Swarm:%20Scheduling*. 2019 (see page 52).

[9] VMware Cloud Native Apps. **What is a Container?** *https://www.youtube.com/watch?v=EnJ7qX9fkcU*. 2017 (see page 45).

[10] Raja Appuswamy, Matthaios Olma, and Anastasia Ailamaki. **Scaling the Memory Power Wall With DRAM-Aware Data Management**. In: DaMoN'15. Melbourne, VIC, Australia: Association for Computing Machinery, 2015. ISBN: 9781450336383. DOI: 10.1145/2771937.2771947. URL: https://doi.org/10.1145/2771937.2771947 (see pages 22, 24).

[11] AWS. **Choosing regions and availability zones**. *https://docs.aws.amazon.com/AmazonElastiCache/latest/mem-ug/RegionsAndAZs.html*. 2022 (see page 44).

[12] AWS. **Container Migration Methodology** (2020) (see page 45).

[13] Rasool Azimi and Hedieh Sajedi. **Distributed Data Clustering in Peer-to-Peer Networks: A Technical Review**. In: Aug. 2014 (see pages 60, 62, 63).

[14] Nour M. Azmy, Islam A.M. El-Maddah, and Hoda K. Mohamed. **Adaptive Power Panel of Cloud Computing Controlling Cloud Power Consumption**. In: *Proceedings of the 2Nd Africa and Middle East Conference on Software Engineering*. AMECSE '16. Cairo, Egypt: ACM, 2016, 9–14. ISBN: 978-1-4503-4293-3. DOI: 10.1145/2944165.2944167. URL: http://doi.acm.org/10.1145/2944165.2944167 (see pages 10, 48, 51, 52).

[15] Microsoft Azure. **How the Kubernetes scheduler works**. *https://www.youtube.com/watch?v=rDCWxkvPlAw*. 2019 (see pages 9, 52).

[16] Frank Bellosa. **The Benefits of Event: Driven Energy Accounting in Power-Sensitive Systems**. In: *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*. EW 9. Kolding, Denmark: Association for Computing Machinery, 2000, 37–42. ISBN: 9781450373562. DOI: 10.1145/566726.566736. URL: https://doi.org/10.1145/566726.566736 (see pages 10, 13, 22).

[17] Manjot Bhatia. **RR Based Grid Scheduling Algorithm**. In: *Proceedings of the International Conference on Advances in Computing and Artificial Intelligence*. ACAI '11. Rajpura/Punjab, India: Association for Computing Machinery, 2011, 120–123. ISBN: 9781450306355. DOI: 10.1145/2007052.2007076. URL: https://doi.org/10.1145/2007052.2007076 (see pages 59, 62, 63).

[18] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. **The Gem5 Simulator**. *SIGARCH Comput. Archit. News* 39:2 (Aug. 2011), 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: https://doi.org/10.1145/2024716.2024718 (see page 13).

[19] Emil Björnson and Erik G. Larsson. **How Energy-Efficient Can a Wireless Communication System Become?** *CoRR* abs/1812.01688 (2018). arXiv: 1812.01688. URL: http://arxiv.org/abs/1812.01688 (see page 28).

[20] Daniel Fernandez Boris Scholl Trent Swanson. **Microservices with Docker on Microsoft Azure**. Addison-Wesley Professional, 2016. ISBN: 0672337495 (see page 52).

[21] David Both. **The central processing unit (CPU): Its components and functionality**. *https://www.redhat.com/sysadmin/cpu-components-functionality*. 2020 (see page 12).

[22] Len Brown. **turbostat - Report processor frequency and idle statistics**. *https://www.linux.org/docs/man8/turbostat.html* (see page 16).

[23] M. Cai, M. Frank, J. Chen, and P. Szekely. **MAAN: a multi-attribute addressable network for grid information services**. In: *Proceedings. First Latin American Web Congress*. 2003, 184–191 (see pages 87, 91, 105).

[24] cdw. **Types of Network Protocols: The Ultimate Guide**. *https://www.cdw.com/content/cdw/en/articles/networking/2019/04/09/types-of-network-protocols.html*. 2019 (see page 27).

[25] Danilo Charântola, Alexandre C. Mestre, Rafael Zane, and Luiz F. Bittencourt. **Component-Based Scheduling for Fog Computing**. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. UCC '19 Companion. Auckland, New Zealand: Association for Computing Machinery, 2019, 3–8. ISBN: 9781450370448. DOI: 10.1145/3368235.3368829. URL: https://doi.org/10.1145/3368235.3368829 (see pages 59, 61, 64, 65).

[26] Piyush Chauhan and Nitin. **Fault Tolerant Decentralized Scheduling Algorithm for P2P Grid**. *Procedia Technology* 6 (2012). 2nd International Conference on Communication, Computing & Security [ICCCS-2012], 698–707. ISSN: 2212-0173. DOI: https://doi.org/10.1016/j.protcy.2012.10.084. URL: http://www.sciencedirect.com/science/article/pii/S2212017312006299 (see pages 59, 64, 65, 75).

[27] Piyush Chauhan and Nitin Nitin. **Decentralized Scheduling Algorithm for DAG Based Tasks on P2P Grid**. *Journal of Engineering* 2014 (Jan. 2014), 1–14. DOI: 10.1155/2014/202843 (see pages 59, 62, 63).

[28] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. **Searching in Metric Spaces**. *ACM Comput. Surv.* 33:3 (Sept. 2001), 273–321. ISSN: 0360-0300. DOI: 10.1145/502807.502808. URL: https://doi.org/10.1145/502807.502808 (see page 88).

[29] Feng Chen and Xiaodong Zhang. **Caching for Bursts (C-Burst): Let Hard Disks Sleep Well and Work Energetically**. In: *Proceedings of the 2008 International Symposium on Low Power Electronics & Design*. ISLPED '08. Bangalore, India: ACM, 2008, 141–146. ISBN: 978-1-60558-109-5. DOI: 10.1145/1393921.1393961. URL: http://doi.acm.org/10.1145/1393921.1393961 (see page 35).

[30] Qiao Chen and Jian Li. **The Balance Mechanism of Power and Performance in the Virtualization**. In: *Proceedings of the Second International Conference on Innovative Computing and Cloud Computing*. ICCC '13. Wuhan, China: ACM, 2013, 189:189–189:192. ISBN: 978-1-4503-2119-8. DOI: 10.1145/2556871.2556912. URL: http://doi.acm.org/10.1145/2556871.2556912 (see page 13).

[31] Wei-Kai Cheng, Po-Yuan Shen, and Xin-Lun Li. **Retention-Aware DRAM Auto-Refresh Scheme for Energy and Performance Efficiency**. *Micromachines* 10:9 (2019), 590. DOI: 10.3390/mi10090590 (see page 19).

[32] Prateek Chhikara, Rajkumar Tekchandani, Neeraj Kumar, and Mohammad S. Obaidat. **An Efficient Container Management Scheme for Resource-Constrained Intelligent IoT Devices**. *IEEE Internet of Things Journal* 8:16 (2021), 12597–12609. DOI: 10.1109/JIOT.2020.3037181 (see pages 61, 64, 65).

[33] Salvatore Chiaravalloti, Filip Idzikowski, and Łukasz Budzisz. **Power consumption of WLAN network elements** (Aug. 2011). DOI: 10.13140/2.1.4424.8005 (see pages 28, 32, 127).

[34] CISCO. **Cisco Packet Tracer**. *https://www.netacad.com/courses/packet-tracer*. 2020 (see page 106).

[35] Google Cloud. **What is cloud computing?** *https://cloud.google.com/learn/what-is-cloud-computing*. 2021 (see page 44).

[36] IBM Cloud. **IBM Edge Computing for Servers**. *https://www.ibm.com/docs/en/cloud-private/3.2.0?topic=edge-computing-servers*. 2017 (see page 59).

[37] cloudbus. **ContainerCloudSim: An Environment For Modeling And Simulation Of Containers In Cloud Data Centers**. *http://www.cloudbus.org/cloudsim/container.html*. 2016 (see page 107).

[38] Maxime Colmant, Romain Rouvoy, Mascha Kurpicz-Briki, Anita Sobe, Pascal Felber, and Lionel Seinturier. **The next 700 CPU power models**. *Journal of Systems and Software* 144 (July 2018). DOI: 10.1016/j.jss.2018.07.001 (see pages 9, 16, 17).

[39] B. F. Cornea, A. Orgerie, and L. Lefèvre. **Studying the energy consumption of data transfers in Clouds: the Ecofen approach**. In: *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*. 2014, 143–148 (see page 28).

[40] Lorenzo Corneo and Per Gunningberg. **Scheduling at the Edge for Assisting Cloud Real-Time Systems**. In: TOPIC '18. Egham, United Kingdom: Association for Computing Machinery, 2018, 9–14. ISBN: 9781450357760. DOI: 10.1145/3229774.3229777. URL: https://doi.org/10.1145/3229774.3229777 (see page 65).

[41] Luis Corral, Anton B. Georgiev, Andrea Janes, and Stefan Kofler. **Energy-aware Performance Evaluation of Android Custom Kernels**. In: *Proceedings of the Fourth International Workshop on Green and Sustainable Software*. GREENS '15. Florence, Italy: IEEE Press, 2015, 1–7. URL: http://dl.acm.org/citation.cfm?id=2820158.2820160 (see pages 13, 35).

[42] Luis Corral, Anton B. Georgiev, Alberto Sillitti, and Giancarlo Succi. **Method Reallocation to Reduce Energy Consumption: An Implementation in Android OS**. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC '14. Gyeongju, Republic of Korea: ACM, 2014, 1213–1218. ISBN: 978-1-4503-2469-4. DOI: 10.1145/2554850.2555064. URL: http://doi.acm.org/10.1145/2554850.2555064 (see page 13).

[43] Breno Costa, Joao Bachiega, Leonardo Rebouças de Carvalho, and Aleteia P. F. Araujo. **Orchestration in Fog Computing: A Comprehensive Survey**. *ACM Comput. Surv.* 55:2 (Jan. 2022). ISSN: 0360-0300. DOI: 10.1145/3486221. URL: https://doi.org/10.1145/3486221 (see page 42).

[44] Crucial. **Crucial P1 1TB 3D NAND NVMe PCIe M.2 SSD**. *https://www.crucial.com/ssd/p1/ct1000p1ssd8*. 2020 (see page 18).

[45] Crucial. **Different Types of RAM Explained**. *https://www.crucial.com/articles/about-memory/different-types-of-memory-explained*. 2020 (see page 18).

[46] Crucial. **How Much Power Does Memory Use?** (2019) (see page 127).

[47] Crucial. **Memory speeds and compatability**. *https://www.crucial.com/support/memory-speeds-compatability#:~:text=PC2700%20memory%20---%20the%20slowest%20DDR,%2C%20or%202.7GB%2Fs*. 2020 (see page 18).

[48] Crucial. **What Is the Difference Between DDR4, DDR3, DDR2, DDR, and SDRAM?** *https://www.crucial.com/support/articles-faq-memory/difference-between-ddr4-ddr3-ddr2-ddr-sdram*. 2019 (see page 19).

[49] CrystalDiskMark. **CrystalDiskMark**. *https://crystalmark.info/en/* (see page 124).

[50] D-link. **Technologie D-Link Green**. *https://eu.dlink.com/fr/fr/support/faq/knowledge/technologie-dlink-green*. 2020 (see page 84).

[51] Keling Da, Marc Dalmau, and Philippe Roose. **Kalimucho: Middleware for Mobile Applications**. In: SAC '14. Gyeongju, Republic of Korea: Association for Computing Machinery, 2014, 413–419. ISBN: 9781450324694. DOI: 10.1145/2554850.2554883. URL: https://doi.org/10.1145/2554850.2554883 (see pages 6, 9, 41, 76).

[52] Keling Da, Marc Dalmau, and Philippe Roose. **Kalimucho: Middleware for Mobile Applications**. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC '14. Gyeongju, Republic of Korea: Association for Computing Machinery, 2014, 413–419. ISBN: 9781450324694. DOI: 10.1145/2554850.2554883. URL: https://doi.org/10.1145/2554850.2554883 (see page 75).

[53] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. **RAPL: Memory Power Estimation and Capping**. In: *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED '10. Austin, Texas, USA: Association for Computing Machinery, 2010, 189–194. ISBN: 9781450301466. DOI: 10.1145/1840845.1840883. URL: https://doi.org/10.1145/1840845.1840883 (see pages 9, 12, 14, 16, 22, 23).

[54] KDAB David Faure. **Profiling CPU and Memory on Linux, with Opensource Graphical Tools**. *https://www.youtube.com/watch?v=HOR4LiS4uMI*. 2020 (see page 26).

[55] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. **MemScale: Active Low-Power Modes for Main Memory**. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: Association for Computing Machinery, 2011, 225–238. ISBN: 9781450302661. DOI: 10.1145/1950365.1950392. URL: https://doi.org/10.1145/1950365.1950392 (see page 20).

[56] Amit Deshpande and Nampreet Pal Singh. **Challenges and patterns for modernizing a monolithic application into microservices**. *https://developer.ibm.com/articles/challenges-and-patterns-for-modernizing-a-monolithic-application-into-microservices/*. 2021 (see page 41).

[57] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. **A Validation of DRAM RAPL Power Measurements**. In: *Proceedings of the Second International Symposium on Memory Systems*. MEMSYS '16. Alexandria, VA, USA: Association for Computing Machinery, 2016, 455–470. ISBN: 9781450343053. DOI: 10.1145/2989081.2989088. URL: https://doi.org/10.1145/2989081.2989088 (see page 23).

[58] Paolo Di Francesco. **Architecting microservices**. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE. 2017, 224–229 (see page 74).

[59] Jolene Dobbin. **Lag! Top 5 Reasons your Ping is so High**. *https://www.hp.com/us-en/shop/tech-takes/5-reasons-your-ping-is-so-high*. 2020 (see page 80).

[60] Kubernetes Docs. **Running in multiple zones**. *https://kubernetes.io/docs/setup/best-practices/multiple-zones/*. 2021 (see page 44).

[61] VMware Docs. **Migrating Virtual Machines**. *https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.vm_admin.doc/GUID-FE2B516E-7366-4978-B75C-64BF0AC676EB.html*. 2019 (see page 45).

[62] Dropbox. **Types of storage devices**. *https://experience.dropbox.com/get-organized/storage-devices*. 2021 (see page 34).

[63] N. Drost, R.V. van Nieuwpoort, and H. Bal. **Simple locality-aware co-allocation in peer-to-peer supercomputing**. In: *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*. Vol. 2. 2006, 8 pp.–14. DOI: 10.1109/CCGRID.2006.1630909 (see pages 60, 62, 63).

[64] Kevin Drumm. **Dynamic Random Access Memory (DRAM). Part 1: Memory Cell Arrays**. *https://www.youtube.com/watch?v=I-9XWtdW_Co&t=151s*. 2020 (see page 19).

[65] Kevin Drumm. **Dynamic Random Access Memory (DRAM). Part 2: Read and Write Cycles**. *https://www.youtube.com/watch?v=x3jGqOrXXc8*. 2020 (see page 19).

[66] C. du Mouza, W. Litwin, and P. Rigaux. **SD-Rtree: A Scalable Distributed Rtree**. In: *2007 IEEE 23rd International Conference on Data Engineering*. 2007, 296–305 (see page 91).

[67] IBM Cloud Education. **Hypervisors**. *https://www.ibm.com/cloud/learn/hypervisors*. 2019 (see page 45).

[68] Mohammed S Elbamby, Mehdi Bennis, Walid Saad, Matti Latva-Aho, and Choong Seon Hong. **Proactive edge computing in fog networks with latency and reliability guarantees**. *EURASIP Journal on Wireless Communications and Networking* 2018:1 (2018), 1–13 (see pages 61, 64, 65).

[69] Utmel Electronic. **How many Transistors in a CPU?** *https://www.utmel.com/blog/categories/transistors/how-many-transistors-in-a-cpu*. 2020 (see page 11).

[70] Niloufar Piroozi Esfahani and Alberto E. Cerpa. **Poster: Energy Optimization Framework in Wireless Sensor Network**. In: *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. SenSys '15. Seoul, South Korea: ACM, 2015, 441–442. ISBN: 978-1-4503-3631-4. DOI: 10.1145/2809695.2817904. URL: http://doi.acm.org/10.1145/2809695.2817904 (see page 29).

[71] Arthur De M. Del Esposte, Eduardo F.z. Santana, Lucas Kanashiro, Fabio M. Costa, Kelly R. Braghetto, Nelson Lago, and Fabio Kon. **Design and evaluation of a scalable smart city software platform with large-scale simulations**. *Future Generation Computer Systems* 93 (2019), 427–441. DOI: 10.1016/j.future.2018.10.026 (see page 107).

[72] Olamilekan Fadahunsi and Muthucumaru Maheswaran. **Locality Sensitive Request Distribution for Fog and Cloud Servers**. *Serv. Oriented Comput. Appl.* 13:2 (June 2019), 127–140. ISSN: 1863-2386. DOI: 10.1007/s11761-019-00260-2. URL: https://doi.org/10.1007/s11761-019-00260-2 (see pages 60, 64, 65).

[73] L.M. Feeney and M. Nilsson. **Investigating the energy consumption of a wireless network interface in an ad hoc networking environment**. In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*. Vol. 3. 2001, 1548–1557 vol.3. DOI: 10.1109/INFCOM.2001.916651 (see pages 28, 29, 33).

[74] Martin Fowler. **Microservices a definition of this new architectural term**. *https://martinfowler.com/articles/microservices.html*. 2014 (see page 74).

[75] Apache Software Fundation. **Apache Aurora: Scheduler Configuration**. *https://aurora.apache.org/documentation/latest/operations/configuration/*. 2020 (see page 54).

[76] Apache Software Fundation. **CapacityScheduler Guide**. *https://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html*. 2020 (see pages 47, 48, 50).

[77] Apache Software Fundation. **Chronos: A fault tolerant job scheduler for Mesos**. *https://mesos.github.io/chronos/*. 2020 (see page 54).

[78] Apache Software Fundation. **Fair Scheduler**. *https://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html*. 2020 (see pages 47, 48, 50).

[79] Apache Software Fundation. **Marathon: A container orchestration platform for Mesos and DC/OS**. *https://mesosphere.github.io/marathon/*. 2018 (see page 54).

[80] Apache Software Fundation. **What is Mesos? A distributed systems kernel**. *https://mesos.apache.org*. 2020 (see page 53).

[81] Volker Gaede and Oliver Günther. **Multidimensional Access Methods**. *ACM Comput. Surv.* 30:2 (June 1998), 170–231. ISSN: 0360-0300. DOI: 10.1145/280277.280279. URL: https://doi.org/10.1145/280277.280279 (see page 88).

[82] Abolfazl Gandomi, Midia Reshadi, Ali Movaghar, and Ahmad Khademzadeh. **HybSMRP: a hybrid scheduling algorithm in Hadoop MapReduce framework**. *Journal of Big Data* 6 (2019), 1–16 (see pages 47, 48, 50).

[83] Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. **One Torus to Rule Them All: Multi-Dimensional Queries in P2P Systems**. In: *Proceedings of the 7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004*. WebDB '04. Paris, France: Association for Computing Machinery, 2004, 19–24. ISBN: 9781450377881. DOI: 10.1145/1017074.1017081. URL: https://doi.org/10.1145/1017074.1017081 (see page 91).

[84] Ankita Garg and Vaidyanathan Srinivasan. **Linux VM Infrastructure for memory power management**. *http://linuxplumbersconf.net/2011/ocw//system/presentations/417/original/LinuxMemoryPowerSaving.pdf*. 2011 (see page 22).

[85] William Gayde. **Anatomy of a CPU**. *https://www.techspot.com/article/2000-anatomy-cpu/#allcomments*. 2020 (see pages 11, 12).

[86] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. **Dominant Resource Fairness: Fair Allocation of Multiple Resource Types**. In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, 323–336 (see pages 49, 50, 53).

[87] David Greenberg. **Mesos: The Operating System for your Cluster**. *https://www.youtube.com/watch?v=gVGZHzRjvo0&t=1049s*. 2014 (see page 53).

[88] Greencloud. **Greencloud - The green cloud Simulator**. *https://greencloud.gforge.uni.lu/*. 2017 (see page 106).

[89] Beenish Gul, Imran Ali Khan, Saad Mustafa, Osman Khalid, Syed Sajid Hussain, Darren Dancey, and Raheel Nawaz. **CPU and RAM Energy-Based SLA-Aware Workload Consolidation Techniques for Clouds**. *IEEE Access* 8 (2020), 62990–63003. DOI: 10.1109/ACCESS.2020.2985234 (see pages 46–48, 51, 52).

[90] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. **iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments**. *Software: Practice and Experience* 47:9 (2017), 1275–1296. DOI: 10.1002/spe.2509 (see pages 28, 107).

[91] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. **Estimating Android Applications' CPU Energy Usage via Bytecode Profiling**. In: *Proceedings of the First International Workshop on Green and Sustainable Software*. GREENS '12. Zurich, Switzerland: IEEE Press, 2012, 1–7. ISBN: 978-1-4673-1832-7. URL: http://dl.acm.org/citation.cfm?id=2663779.2663780 (see page 13).

[92] Red Hat. **How containers use PID namespaces to provide process isolation**. *https://www.youtube.com/watch?v=J17rXQ5XkDE*. 2020 (see pages 45, 74).

[93] HP. **Double Data Rate SDRAM: fast performance at an economical price** (Oct. 2004) (see page 19).

[94] https://linux.die.net. **iftop - display bandwidth usage on an interface by host**. *https://linux.die.net/man/8/iftop*. 2021 (see page 32).

[95] IBM. **Networking**. *https://www.ibm.com/cloud/learn/networking-a-complete-guide*. 2021 (see page 27).

[96] Ahmed Banafa - IBM. **What is fog computing?** *https://www.ibm.com/blogs/cloud-computing/2014/08/25/fog-computing/*. 2014 (see page 58).

[97] IEA. **Data Centres and Data Transmission Networks**. *https://www.iea.org/reports/data-centres-and-data-transmission-networks*. 2021 (see page 5).

[98] IEA. **Electricity Market Report - January 2022**. *https://www.iea.org/reports/electricity-market-report-january-2022*. 2022 (see page 5).

[99] IEA. **Global changes in electricity generation, 2015-2024**. *https://www.iea.org/data-and-statistics/charts/global-changes-in-electricity-generation-2015-2024*. 2022 (see page 5).

[100] GL Communications Inc. **IP WAN Emulator**. *https://www.gl.com/wan-link-emulation-ipnetsim.html*. 2020 (see page 106).

[101] EPSI - L'école de ingénierie informatique. **Edge computing et Fog computing, quelles différences ?** *https://www.epsi.fr/edge-et-fog-comptuting/*. 2019 (see page 59).

[102] IEEE Innovation. **Why Does Edge Computing Matter?** *https://innovationatwork.ieee.org/why-does-edge-computing-matter/*. 2019 (see page 58).

[103] Takuro Inoue, Makoto Ikeda, Tomoya Enokido, Ailixier Aikebaier, and Makoto Takizawa. **A Power Consumption Model for Storage-based Applications**. In: *2011 International Conference on Complex, Intelligent, and Software Intensive Systems*. 2011, 612–617. DOI: 10.1109/CISIS.2011.101 (see page 35).

[104] Texas Instruments. **Network interface card (NIC)**. *https://www.ti.com/solution/network-interface-card-nic?variantid=34385&subsystemid=25606*. 2016 (see page 27).

[105] Intel. **C-State**. *https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/energy-analysis-metrics-reference/c-state.html*. 2021 (see pages 13, 14).

[106] Intel. **Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor** (2004) (see page 14).

[107] Intel. **How to Overclock Your Unlocked Intel Core Processor**. *https://www.intel.com/content/www/us/en/gaming/resources/how-to-overclock.html*. 2021 (see pages 11, 12, 127).

[108] Intel. **Intel - msrTools**. *https://github.com/intel/msr-tools* (see page 16).

[109] Intel. **Intel Core i5-8400 Processor**. *https://ark.intel.com/content/www/us/en/ark/products/126687/intel-core-i58400-processor-9m-cache-up-to-4-00-ghz.html* (see page 127).

[110] Intel. **Intel VTune Profiler**. *https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html*. 2021 (see page 14).

[111] Intel. **Potencia de diseño térmico (TDP) en procesadores Intel**. *https://www.intel.la/content/www/xl/es/support/articles/000055611/processors.html*. 2019 (see page 11).

[112] Intel. **powertop**. *https://01.org/powertop*. 2020 (see page 16).

[113] Intel. **Understanding Load Capacitance and Access Time**. *https://www.cypress.com/file/202411/download* (see page 127).

[114] Intel. **What exactly is a P-state?** *https://software.intel.com/content/www/us/en/develop/blogs/what-exactly-is-a-p-state-pt-1.html?language=en*. 2015 (see page 13).

[115] Intel. **White paper:Energy-Efficient PlatformsConsiderations for Application Software and Services**. Tech. rep. 325085-001. Intel, Mar. 2011 (see page 13).

[116] intel. **Intel Ethernet Connection I219-LM**. *http://www.intel.com/content/www/us/en/embedded/products/networking/ethernet-connection-i219-datasheet.html*. 2015 (see pages 29, 32).

[117] Digital Guide IONOS. **¿Qué es el Grid Computing?** *https://www.ionos.es/digitalguide/servidores/know-how/grid-computing/*. 2022 (see page 59).

[118] George Iordache, Marcela S. Boboila, Florin Pop, Corina Stratan, and Valentin Cristea. **A Decentralized Strategy for Genetic Scheduling in Heterogeneous Environments**. In: vol. 3. Jan. 2007, 355–367. DOI: 10.1007/11914952_13 (see pages 60, 62, 63).

[119] iso.org. **OPEN SYSTEMS INTERCONNECTION (OSI)**. *https://www.iso.org/ics/35.100/x/* (see page 27).

[120] Abhishek Jaiantilal, Yifei Jiang, and Shivakant Mishra. **Modeling CPU Energy Consumption for Energy Efficient Scheduling**. In: *Proceedings of the 1st Workshop on Green Computing*. GCM '10. Bangalore, India: ACM, 2010, 10–15. ISBN: 978-1-4503-0450-4. DOI: 10.1145/1925013.1925015. URL: http://doi.acm.org/10.1145/1925013.1925015 (see pages 10, 13, 14).

[121] Juyeon Jo, Yoohwan Kim, Kyu Hwan Lee, Sung Hyun Cho, and Jae Hyun Kim. **The Energy Saving Strategy Using the Network Coding in the Wireless Mesh Network**. In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. ICUIMC '12. Kuala Lumpur, Malaysia: ACM, 2012, 127:1–127:4. ISBN: 978-1-4503-1172-4. DOI: 10.1145/2184751.2184895. URL: http://doi.acm.org/10.1145/2184751.2184895 (see page 29).

[122] JPY. **Calculating network data transfer speeds**. *https://news.jpy.com/kbase/support-articles/2015/9/7/calculating-network-data-transfer-speeds* (see page 127).

[123] SeungGu Kang, Hong Jun Choi, Cheol Hong Kim, Sung Woo Chung, DongSeop Kwon, and Joong Chae Na. **Exploration of CPU/GPU Co-execution: From the Perspective of Performance, Energy, and Temperature**. In: *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*. RACS '11. Miami, Florida: ACM, 2011, 38–43. ISBN: 978-1-4503-1087-1. DOI: 10.1145/2103380.2103388. URL: http://doi.acm.org/10.1145/2103380.2103388 (see page 13).

[124] Alexey Karyakin and Kenneth Salem. **An Analysis of Memory Power Consumption in Database Systems**. In: *Proceedings of the 13th International Workshop on Data Management on New Hardware*. DAMON '17. Chicago, Illinois: Association for Computing Machinery, 2017. ISBN: 9781450350259. DOI: 10.1145/3076113.3076117. URL: https://doi.org/10.1145/3076113.3076117 (see pages 16, 19, 22, 23, 25, 26).

[125] KDAB. **First stable release of the fast Linux heap memory profiler**. *https://www.kdab.com/heaptrack-v1-0-0-release/*. 2017 (see page 22).

[126] KDAB. **Hotspot A GUI for perf report**. *https://www.kdab.com/hotspot-video/*. 2018 (see page 22).

[127] The Linux Kernel. **ethtool - utility for controlling network drivers and hardware**. *https://mirrors.edge.kernel.org/pub/software/network/ethtool/*. 2021 (see page 32).

[128] The Linux kernel. **CFS Scheduler**. *https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html* (see page 110).

[129] kernel.org. **Device Power Management Basics**. *https://www.kernel.org/doc/html/v4.14/driver-api/pm/devices.html*. 2021 (see page 28).

[130] kernel.org. **Energy Aware Scheduling**. *https://www.kernel.org/doc/html/latest/scheduler/sched-energy.html#:~:text=1.-,Introduction,a%20minimal%20impact%20on%20throughput*. 2021 (see page 13).

[131] Johannes Kirschnick, Jose M. Alcaraz Calero, Patrick Goldsack, Andrew Farrell, Julio Guijarro, Steve Loughran, Nigel Edwards, and Lawrence Wilcock. **Towards an Architecture for Deploying Elastic Services in the Cloud**. *Softw. Pract. Exper.* 42:4 (Apr. 2012), 395–408. ISSN: 0038-0644. DOI: 10.1002/spe.1090. URL: https://doi.org/10.1002/spe.1090 (see pages 45, 49, 50).

[132] Luis Augusto Dias Knob, Carlos Henrique Kayser, and Tiago Ferreto. **Improving Container Deployment in Edge Computing Using the Infrastructure Aware Scheduling Algorithm**. In: *IEEE Symposium on Computers and Communications, ISCC 2021, Athens, Greece, September 5-8, 2021*. IEEE, 2021, 1–6. DOI: 10.1109/ISCC53001.2021.9631490. URL: https://doi.org/10.1109/ISCC53001.2021.9631490 (see pages 59, 62, 63).

[133] Sushant Kondguli and Michael Huang. **A Case for a More Effective, Power-Efficient Turbo Boosting**. *ACM Trans. Archit. Code Optim.* 15:1 (Mar. 2018), 5:1–5:22. ISSN: 1544-3566. DOI: 10.1145/3170433. URL: http://doi.acm.org/10.1145/3170433 (see page 13).

[134] Jan Krajewski, José Lozano, Julian Driver, Emmanuel Escandon, Sumith Kumar, Silvia Malatini, and Jose Lozano Hinojosa. **PASTRY: the third generation of peer -to -peer networks**. PhD thesis. Jan. 2006 (see page 91).

[135] Kubernetes. **Kubernetes Labels and Selectors**. *https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/*. 2021 (see page 53).

[136] Kubernetes. **Kubernetes Scheduler**. *https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/*. 2021 (see pages 6, 49–52, 75).

[137] Lukasz Kuczera. **6 COMPANIES PIONEERING THE USE OF DISTRIBUTED SYSTEMS**. *https://scalac.io/blog/6-companies-using-distributed-systems/*. 2020 (see page 41).

[138] Peter de Lange, Bernhard Göschlberger, Tracie Farrell, and Ralf Klamma, 172–186. In: Jan. 2018. ISBN: 978-3-319-98571-8. DOI: 10.1007/978-3-319-98572-5_14 (see page 75).

[139] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. **Power Aware Page Allocation**. *SIGARCH Comput. Archit. News* 28:5 (Nov. 2000), 105–116. ISSN: 0163-5964. DOI: 10.1145/378995.379007. URL: https://doi.org/10.1145/378995.379007 (see page 22).

[140] Yena Lee, Jae-Hoon An, and Younghwan Kim. **Scheduler for Distributed and Collaborative Container Clusters Based on Multi-Resource Metric**. In: RACS '20. Gwangju, Republic of Korea: Association for Computing Machinery, 2020, 279–281. ISBN: 9781450380256. DOI: 10.1145/3400286.3418281. URL: https://doi.org/10.1145/3400286.3418281 (see pages 49, 50, 53).

[141] Jacob Leverich, Matteo Monchiero, Vanish Talwar, Parthasarathy Ranganathan, and Christos Kozyrakis. **Power Management of Datacenter Workloads Using Per-Core Power Gating**. *Computer Architecture Letters* 8 (Feb. 2009), 48–51. DOI: 10.1109/L-CA.2009.46 (see page 13).

[142] Mei Li, Guanling Lee, Wang-Chien Lee, and Anand Sivasubramaniam. **PENS: An Algorithm for Density-Based Clustering in Peer-to-Peer Systems**. In: *Proceedings of the 1st International Conference on Scalable Information Systems*. InfoScale '06. Hong Kong: Association for Computing Machinery, 2006, 39–es. ISBN: 1595934286. DOI: 10.1145/1146847.1146886. URL: https://doi.org/10.1145/1146847.1146886 (see pages 60, 62, 63).

[143] Jenn-Wei Lin, Chien-Hung Chen, and Chi-Yi Lin. **Integrating QoS awareness with virtualization in cloud computing systems for delay-sensitive applications**. *Future Generation Computer Systems* 37 (2014), 478–487. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2013.12.034. URL: https://www.sciencedirect.com/science/article/pii/S0167739X13002987 (see pages 45, 47, 48).

[144] Arch Linux. **CPU frequency scaling**. *https://wiki.archlinux.org/index.php/CPU_frequency_scaling*. 2021 (see page 13).

[145] Arch Linux. **Undervolting CPU**. *https://wiki.archlinux.org/index.php/Undervolting_CPU*. 2020 (see page 13).

[146] Suse Linux. **Definition Computer Cluster**. *https://www.suse.com/suse-defines/definition/computer-cluster/*. 2021 (see page 44).

[147] linux-intel-undervolt. **linux-intel-undervolt**. *https://github.com/georgewhewell/undervolt*. 2018 (see page 17).

[148] linux.die.net. **dmidecode(8) - Linux man page**. *https://linux.die.net/man/8/dmidecode* (see pages 20, 25).

[149] Yung-Feng Lu, Jun Wu, and Chin-Fu Kuo. **A Path Generation Scheme for Real-time Green Internet of Things**. *SIGAPP Appl. Comput. Rev.* 14:2 (June 2014), 45–58. ISSN: 1559-6915. DOI: 10.1145/2656864.2656868. URL: http://doi.acm.org/10.1145/2656864.2656868 (see page 29).

[150] LucaCanali. **Notes and tools for measuring CPU-to-memory throughput in Linux**. *https://github.com/LucaCanali/Miscellaneous/blob/master/Spark_Notes/Tools_Linux_Memory_Perf_Measure.md*. 2019 (see page 24).

[151] Niti Madan, Alper Buyuktosunoglu, Pradip Bose, and Murali Annavaram. **Guarded Power Gating in a Multi-core Setting**. In: *Proceedings of the 2010 International Conference on Computer Architecture*. ISCA'10. Saint-Malo, France: Springer-Verlag, 2012, 198–210. ISBN: 978-3-642-24321-9. DOI: 10.1007/978-3-642-24322-6_17. URL: http://dx.doi.org/10.1007/978-3-642-24322-6_17 (see page 13).

[152] Redowan Mahmud, Kotagiri Ramamohanarao, and Rajkumar Buyya. **Application Management in Fog Computing Environments: A Taxonomy, Review and Future Directions**. *ACM Comput. Surv.* 53:4 (July 2020). ISSN: 0360-0300. DOI: 10.1145/3403955. URL: https://doi.org/10.1145/3403955 (see page 42).

[153] Vincenzo De Maio, Vlad Nae, and Radu Prodan. **Evaluating Energy Efficiency of Gigabit Ethernet and Infiniband Software Stacks in Data Centres**. In: *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. UCC '14. Washington, DC, USA: IEEE Computer Society, 2014, 21–28. ISBN: 978-1-4799-7881-6. DOI: 10.1109/UCC.2014.10. URL: http://dx.doi.org/10.1109/UCC.2014.10 (see pages 10, 29, 32, 84).

[154] Tanu Malik, Ligia Nistor, and Ashish Gehani. **Middleware for Managing Provenance Metadata**. In: *Middleware '10 Posters and Demos Track*. Middleware Posters '10. Bangalore, India: ACM, 2010, 5:1–5:2. ISBN: 978-1-4503-0601-0. DOI: 10.1145/1930028.1930033. URL: http://doi.acm.org/10.1145/1930028.1930033 (see page 29).

[155] Krishna T. Malladi, Ian Shaeffer, Liji Gopalakrishnan, David Lo, Benjamin C. Lee, and Mark Horowitz. **Rethinking DRAM Power Modes for Energy Proportionality**. In: USA: IEEE Computer Society, 2012. ISBN: 9780769549248. DOI: 10.1109/MICRO.2012.21. URL: https://doi.org/10.1109/MICRO.2012.21 (see pages 10, 19, 20).

[156] S. Mandal, S. Chakraborty, and S. Karmakar. **Deterministic 1–2 skip list in distributed system**. In: *2012 2nd IEEE International Conference on Parallel, Distributed and Grid Computing*. 2012, 296–301 (see page 91).

[157] Petar Maymounkov and David Eres. **Kademlia: A Peer-to-peer Information System Based on the XOR Metric**. In: vol. 2429. Apr. 2002. DOI: 10.1007/3-540-45748-8_5 (see pages 91, 105).

[158] Philippe Roose Merzoug Soltane and Kazar Okba. **Prediction & Modeling energy consumption for IT Data Center Infrastructure**. In: *Advanced Intelligent Systems for Sustainable Development*. 2018. DOI: 978-3-030-12065-8_1 (see page 35).

[159] Apache Mesos. **Mesos Allocation Modules**. *https://mesos.apache.org/documentation/latest/allocation-module/*. 2020 (see page 54).

[160] Micron. **Calculating Memory Power for DDR4 SDRAM** (2017) (see pages 19, 22).

[161] Micron. **DDR4 SDRAM** (2014) (see pages 19, 22).

[162] Microsoft. **What is IaaS?** *https://azure.microsoft.com/en-us/overview/what-is-iaas/#overview*. 2021 (see page 44).

[163] Microsoft. **What is PaaS?** *https://azure.microsoft.com/en-us/overview/what-is-paas/*. 2021 (see page 44).

[164] Microsoft. **What is SaaS?** *https://azure.microsoft.com/en-us/overview/what-is-saas/*. 2021 (see page 44).

[165] moderndiplomacy. **Surging electricity demand is putting power systems under strain around the world**. *https://moderndiplomacy.eu/2022/01/17/surging-electricity-demand-is-putting-power-systems-under-strain-around-the-world/*. 2022 (see page 5).

[166] Aykhan Nazimzada. **Distributed Systems & Distributed Computing**. *https://anazimzada2020.medium.com/distributed-systems-distributed-computing-part-ec6b0858b52b*. 2020 (see page 41).

[167] Sergiu Nedevschi, Lucian Popa, Gianluca Iannaccone, Sylvia Ratnasamy, and David Wetherall. **Reducing Network Energy Consumption via Sleeping and Rate-Adaptation**. In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'08. San Francisco, California: USENIX Association, 2008, 323–336. ISBN: 1119995555221 (see pages 28, 29, 31).

[168] Netflix. **Netflix Conductor: A microservices orchestrator**. *https://netflixtechblog.com/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40*. 2016 (see pages 55, 75).

[169] Dirk Neumann, Christian Bodenstein, Omer F. Rana, and Ruby Krishnaswamy. **STACEE: Enhancing Storage Clouds Using Edge Devices**. In: *Proceedings of the 1st ACM/IEEE Workshop on Autonomic Computing in Economics*. ACE '11. Karlsruhe, Germany: ACM, 2011, 19–26. ISBN: 978-1-4503-0734-5. DOI: 10.1145/1998561.1998567. URL: http://doi.acm.org/10.1145/1998561.1998567 (see pages 60, 62–65).

[170] Linux Weekly News. **mm: Memory Power Management**. *https://lwn.net/Articles/546696/*. 2013 (see page 22).

[171] Z. Nikdel, B. Gao, and S. W. Neville. **DockerSim: Full-stack simulation of container-based Software-as-a-Service (SaaS) cloud deployments and environments**. In: *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. Aug. 2017, 1–6. DOI: 10.1109/PACRIM.2017.8121898 (see page 107).

[172] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. **Monitoring Energy Hotspots in Software**. *Automated Software Engg.* 22:3 (Sept. 2015), 291–332. ISSN: 0928-8910. DOI: 10.1007/s10515-014-0171-1. URL: https://doi.org/10.1007/s10515-014-0171-1 (see pages 13, 14, 16, 28, 29).

[173] Petr Novotny and Alexander Wolf. **Simulating Services-Based Systems Hosted in Networks with Dynamic Topology** (Feb. 2016). DOI: 10.13140/RG.2.1.1894.5686 (see page 106).

[174] nperf. **Speed test**. *https://www.nperf.com/es/* (see page 125).

[175] Omogbai Oleghe. **Container Placement and Migration in Edge Computing: Concept and Scheduling Models**. *IEEE Access* 9 (2021), 68028–68043. DOI: 10.1109/ACCESS.2021.3077550 (see pages 42, 58).

[176] T Kwame Opam. **Why I love Linux — even if I no longer use it**. *https://www.theverge.com/2014/9/21/6661393/tell-me-why-you-use-linux-everyday*. 2014 (see page 77).

[177] A. Orgerie, L. Lefèvre, I. Guérin-Lassous, and D. M. Lopez Pacheco. **ECOFEN: An End-to-end energy Cost mOdel and simulator For Evaluating power consumption in large-scale Networks**. In: *2011 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*. 2011, 1–6 (see page 28).

[178] packetstorm. **Network Simulation**. *https://packetstorm.com/network-simulation/*. 2018 (see page 106).

[179] Linux manual page. **iotop(8) — Linux manual page**. *https://man7.org/linux/man-pages/man8/iotop.8.html*. 2009 (see page 37).

[180] Linux manual page. **perf(1) — Linux manual page**. *https://man7.org/linux/man-pages/man1/perf.1.html*. 2020 (see pages 14, 17, 22).

[181] Linux manual page. **ps(1) - Linux manual page**. *https://man7.org/linux/man-pages/man1/ps.1.html*. 2020 (see page 17).

[182] Michael Pawlish, Aparna S. Varde, Stefan A. Robila, and Anand Ranganathan. **A Call for Energy Efficiency in Data Centers**. *SIGMOD Rec.* 43:1 (May 2014), 45–51. ISSN: 0163-5808. DOI: 10.1145/2627692.2627703. URL: http://doi.acm.org/10.1145/2627692.2627703 (see page 48).

[183] Benoit Petit. **Scaphandre v0.1.1: measuring the energy consumption of the tech industry (backstages)**. *https://bpetit.nce.re/2021/01/scaphandre-v0.1.1-measuring-the-energy-consumption-of-the-tech-industry-backstages/*. 2020 (see pages 16, 28).

[184] Virginia Pilloni and Luigi Atzori. **Deployment of Distributed Applications in Wireless Sensor Networks**. *Sensors (Basel, Switzerland)* 11 (2011), 7395–7419 (see page 41).

[185] Florin Pop, Valentin Cristea, Nik Bessis, and Stelios Sotiriadis. **Reputation Guided Genetic Scheduling Algorithm for Independent Tasks in Inter-clouds Environments**. In: *2013 27th International Conference on Advanced Information Networking and Applications Workshops*. 2013, 772–776. DOI: 10.1109/WAINA.2013.206 (see pages 45, 47, 48).

[186] PowerTutor. **PowerTutor**. *http://ziyang.eecs.umich.edu/projects/powertutor/*. 2011 (see page 14).

[187] Primeline-solutions. **datasheet**. *https://www.primeline-solutions.com/datasheet/18868/wd-480-gb-wd-green-sata-ssd-wds480g2g0a.pdf* (see page 125).

[188] Carlo Puliafito, Enzo Mingozzi, Carlo Vallati, Francesco Longo, and Giovanni Merlino. **Companion Fog Computing: Supporting Things Mobility Through Container Migration at the Edge**. In: *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*. 2018, 97–105. DOI: 10.1109/SMARTCOMP.2018.00079 (see pages 63–65).

[189] Deepak Puthal, Mohammad S. Obaidat, Priyadarsi Nanda, Mukesh Prasad, Saraju P. Mohanty, and Albert Y. Zomaya. **Secure and Sustainable Load Balancing of Edge Data Centers in Fog Computing**. *IEEE Communications Magazine* 56:5 (2018), 60–65. DOI: 10.1109/MCOM.2018.1700795 (see pages 61, 64, 65).

[190] Ju Ren, Deyu Zhang, Shiwen He, Yaoxue Zhang, and Tao Li. **A Survey on End-Edge-Cloud Orchestrated Network Computing Paradigms: Transparent Computing, Mobile Edge Computing, Fog Computing, and Cloudlet**. *ACM Comput. Surv.* 52:6 (Oct. 2019). ISSN: 0360-0300. DOI: 10.1145/3362031. URL: https://doi.org/10.1145/3362031 (see page 42).

[191] Leonardo R. Rodrigues, Marcelo Pasin, Omir C. Alves, Charles C. Miers, Mauricio A. Pillon, Pascal Felber, and Guilherme P. Koslovski. **Network-Aware Container Scheduling in Multi-Tenant Data Center**. In: *2019 IEEE Global Communications Conference (GLOBECOM)*. Waikoloa, HI, USA: IEEE Press, 2019, 1–6. DOI: 10.1109/GLOBECOM38437.2019.9013128. URL: https://doi.org/10.1109/GLOBECOM38437.2019.9013128 (see pages 49, 50, 54).

[192] Maria Alejandra Rodriguez and Rajkumar Buyya. **Deadline Based Resource Provisioning and Scheduling Algorithm for Scientific Workflows on Clouds**. *IEEE Transactions on Cloud Computing* 2:2 (2014), 222–235. DOI: 10.1109/TCC.2014.2314655 (see pages 45, 47, 48).

[193] Samsung. **Samsung V-NAND SSD 860 EVO**. *https://www.samsung.com/semiconductor/global.semi.static/Samsung_SSD_860_EVO_Data_Sheet_Rev1.pdf*. Dec. 2017 (see pages 37, 127).

[194] sata-io. **SATA Power Management: It's Good to Be Green**. In: 2013 (see page 34).

[195] Segate. **Desktop HDD Product Manual**. *https://www.seagate.com/www-content/product-content/barracuda-fam/desktop-hdd/barracuda-7200-14/en-us/docs/100686584v.pdf*. Sept. 2016 (see pages 37, 127).

[196] Yanfeng Shu, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. **Supporting multi-dimensional range queries in peer-to-peer systems**. In: *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*. 2005, 173–180. DOI: 10.1109/P2P.2005.35 (see pages 91, 92).

[197] Junaid Shuja, Kashif Bilal, Waleed Alasmary, Hassan Sinky, and Eisa Alanazi. **Applying machine learning techniques for caching in next-generation edge networks: A comprehensive survey**. *Journal of Network and Computer Applications* 181 (2021), 103005. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2021.103005 (see pages 61–63).

[198] I. Siddavatam, E. Johri, and D. Patole. **Optimization of Load Balancing Algorithm for Green IT**. In: *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*. ICWET '11. Mumbai, Maharashtra, India: ACM, 2011, 1344–1346. ISBN: 978-1-4503-0449-8. DOI: 10.1145/1980022.1980321. URL: http://doi.acm.org/10.1145/1980022.1980321 (see pages 48, 51, 52).

[199] Sisoftware. **SiSoftware Sandra**. *https://www.sisoftware.co.uk* (see page 124).

[200] AARON SMITH. **Record shares of Americans now own smartphones, have home broadband**. *https://www.pewresearch.org/fact-tank/2017/01/12/evolution-of-technology/*. 2017 (see page 77).

[201] PassMark Software. **Memory Benchmarks**. *https://www.memorybenchmark.net/amount-of-ram-installed.html* (see page 127).

[202] Intel Open Source. **RUNNING AVERAGE POWER LIMIT – RAPL**. *https://01.org/blogs/2014/running-average-power-limit----rapl*. 2014 (see page 14).

[203] Open Source. **What is Docker?** *https://opensource.com/resources/what-docker*. 2019 (see pages 49–52).

[204] speedtest. **Speedtest**. *https://www.speedtest.net/es* (see page 125).

[205] Maarten Steen and Andrew S. Tanenbaum. **A Brief Introduction to Distributed Systems**. *Computing* 98:10 (Oct. 2016), 967–1009. ISSN: 0010-485X. DOI: 10.1007/s00607-016-0508-7. URL: https://doi.org/10.1007/s00607-016-0508-7 (see page 41).

[206] Ion Stoica, Robert Morris, David Karger, M. Kaashoek, and Hari Balakrishnan. **Chord: A scalable peer-to-peer lookup service for internet applications**. In: vol. 149-160. Jan. 2001, 149–160. DOI: 10.1145/383059.383071 (see pages 91, 105).

[207] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. **Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications**. *IEEE/ACM Trans. Netw.* 11:1 (Feb. 2003), 17–32. ISSN: 1063-6692. DOI: 10.1109/TNET.2002.808407. URL: https://doi.org/10.1109/TNET.2002.808407 (see pages 92, 93).

[208] Peter Strempel. **EIT Digital presents report on Digital Technologies and the Green Economy**. *https://www.eitdigital.eu/newsroom/news/2022/eit-digital-presents-report-on-digital-technologies-and-the-green-economy/*. 2022 (see page 5).

[209] S. Sundresh, Wooyoung Kim, and G. Agha. **SENS: a sensor, environment and network simulator**. In: *37th Annual Simulation Symposium, 2004. Proceedings.* Apr. 2004, 221–228. DOI: 10.1109/SIMSYM.2004.1299486 (see page 106).

[210] Andrew S. Tanenbaum and Herbert Bos. **Modern Operating Systems**. 4th. USA: Prentice Hall Press, 2014. ISBN: 013359162X (see pages 12, 17, 34, 41).

[211] Shanjiang Tang, BingSheng He, Shuhao Zhang, and Zhaojie Niu. **Elastic Multi-resource Fairness: Balancing Fairness and Efficiency in Coupled CPU-GPU Architectures**. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '16. Salt Lake City, Utah: IEEE Press, 2016, 75:1–75:12. ISBN: 978-1-4673-8815-3. URL: http://dl.acm.org/citation.cfm?id=3014904.3015005 (see page 13).

[212] tanscend. **What is the difference between SDRAM, DDR1, DDR2, DDR3 and DDR4?** *https://www.transcend-info.com/Support/FAQ-296#:~:text=DDR3%20memory%20reduces%2040%25%20power,V%20or%20DDR%27s%202.5%20V).&text=DDR4%20SDRAM%20provides%20the%20lower,2133~3200%20MT%2Fs* (see page 20).

[213] The TechCave. **What is Containerization? What is Docker? | Containerization vs Virtualization**. *https://www.youtube.com/watch?v=yLNMMdyak6k&lc=z22hxrsrmteqff5av04t1aokgqzxl54rqy3ng3303szpbk0h00410*. 2020 (see page 45).

[214] IBM Technology. **Containers vs VM, what is the difference?** *https://www.youtube.com/watch?v=cjXI-yxqGTI&t=226s*. 2020 (see page 45).

[215] IBM Technology. **What are Microservices?** *https://www.youtube.com/watch?v=CdBtNQZH8a4&t=159s*. 2019 (see page 74).

[216] Viking Technology. **DRAM MEMORY MODULE RANK CALCULATION** (July 2018) (see page 18).

[217] Techspot. **AS SSD Benchmark**. *https://www.techspot.com/downloads/6014-as-ssd-benchmark.html* (see page 124).

[218] MIK Telecom. **Network Interface Cards - Cisco Documentation**. *https://docstore.mik.ua/univercd/cc/td/doc/product/tel_pswt/vco_prod/card_tds/tds03.htm#xtocid50410*. 2001 (see page 27).

[219] László Toka, Gergely Dobreff, Balázs Fodor, and Balázs Sonkoly. **Machine Learning-Based Scaling Management for Kubernetes Edge Clusters**. *IEEE Trans. on Netw. and Serv. Manag.* 18:1 (Mar. 2021), 958–972. ISSN: 1932-4537. DOI: 10.1109/TNSM.2021.3052837. URL: https://doi.org/10.1109/TNSM.2021.3052837 (see pages 60, 64, 65).

[220] Tomshardware. **SSD vs HDD Tested: What's the Difference and Which Is Better?** *https://www.tomshardware.com/features/ssd-vs-hdd-hard-drive-difference/2* (see page 127).

[221] tuxera. **CPU frequency**. *https://a1dev.com/sd-bench/stats/cpu-frequency/* (see page 127).

[222] Pongsakorn U-Chupala, Yasuhiro Watashiba, Kohei Ichikawa, Susumu Date, and Hajimu Iida. **Container Rebalancing: Towards Proactive Linux Containers Placement Optimization in a Data Center**. In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. 2017, 788–795. DOI: 10.1109/COMPSAC.2017.94 (see page 54).

[223] IDC Corporate USA. **Worldwide Smartphone Shipment OS Market Share Forecast**. *https://www.idc.com/promo/smartphone-market-share*. 2021 (see page 77).

[224] Ward Van Heddeghem, Sofie Lambert, Bart Lannoo, Didier Colle, Mario Pickavet, and Piet Demeester. **Trends in worldwide ICT electricity consumption from 2007 to 2012**. *Computer Communications* 50 (2014). Green Networking, 64–76. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2014.02.008. URL: https://www.sciencedirect.com/science/article/pii/S0140366414000619 (see page 5).

[225] Tom Warren. **Apple reveals Windows 10 is four times more popular than the Mac**. *https://www.theverge.com/2017/4/4/15176766/apple-microsoft-windows-10-vs-mac-users-figures-stats*. 2017 (see page 77).

[226] WatElectronics.com. **What is a Network Interface Card : Types and Its Working**. *https://www.watelectronics.com/what-is-a-network-interface-card-types-and-its-working/*. 2020 (see page 27).

[227] WAVEMON. **wavemon**. *https://github.com/uoaerg/wavemon*. 2021 (see page 28).

[228] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, Quan Z. Sheng, and Rajiv Ranjan. **A Taxonomy and Survey of Cloud Resource Orchestration Techniques**. *ACM Comput. Surv.* 50:2 (May 2017). ISSN: 0360-0300. DOI: 10.1145/3054177. URL: https://doi.org/10.1145/3054177 (see page 42).

[229] HWMon Wiki. **Lm_sensors - Linux hardware monitoring**. *https://hwmon.wiki.kernel.org/lm_sensors*. 2019 (see page 17).

[230] Thomas Willhalm and Roman Dementiev. **Intel® Performance Counter Monitor - A Better Way to Measure CPU Utilization**. *https://software.intel.com/content/www/us/en/develop/articles/intel-performance-counter-monitor.html*. 2017 (see pages 14, 22).

[231] Caesar Wu and Rajkumar Buyya. "Chapter 11 - Cloud Infrastructure Servers: CISC, RISC, Rack-Mounted, and Blade Servers." In: *Cloud Data Centers and Cost Modeling*. Ed. by Caesar Wu and Rajkumar Buyya. Morgan Kaufmann, 2015, 371–424. ISBN: 978-0-12-801413-4. DOI: https://doi.org/10.1016/B978-0-12-801413-4.00011-8. URL: https://www.sciencedirect.com/science/article/pii/B9780128014134000118 (see page 11).

[232] Kaiyue Wu, Ping Lu, and Zuqing Zhu. **Distributed Online Scheduling and Routing of Multicast-Oriented Tasks for Profit-Driven Cloud Computing**. *IEEE Communications Letters* 20:4 (2016), 684–687. DOI: 10.1109/LCOMM.2016.2526001 (see page 54).

[233] Rafael J. Wysocki. **CPU Performance Scaling**. *https://www.kernel.org/doc/html/v4.12/admin-guide/pm/cpufreq.html*. 2017 (see page 13).

[234] Rafael J. Wysocki. **Power Management Interface for System Sleep**. *https://www.kernel.org/doc/Documentation/power/interface.txt*. 2016 (see page 24).

[235] Rafael J. Wysocki. **System Sleep States**. *https://www.kernel.org/doc/html/latest/admin-guide/pm/sleep-states.html?highlight=self%20refresh*. 2017 (see page 24).

[236] Minxian Xu and Rajkumar Buyya. **Brownout Approach for Adaptive Management of Resources and Applications in Cloud Computing Systems: A Taxonomy and Future Directions**. *ACM Comput. Surv.* 52:1 (Jan. 2019). ISSN: 0360-0300. DOI: 10.1145/3234151. URL: https://doi.org/10.1145/3234151 (see page 42).

[237] Elias Yaacoub, Abdullah Kadri, and Adnan Abu-Dayya. **Cooperative Wireless Sensor Networks for Green Internet of Things**. In: *Proceedings of the 8H ACM Symposium on QoS and Security for Wireless and Mobile Networks*. Q2SWinet '12. Paphos, Cyprus: ACM, 2012, 79–80. ISBN: 978-1-4503-1619-4. DOI: 10.1145/2387218.2387235. URL: http://doi.acm.org/10.1145/2387218.2387235 (see page 29).

[238] Ming Yan, Chien Aun Chan, André F. Gygax, Jinyao Yan, Leith Campbell, Ampalavanapillai Nirmalathas, and Christopher Leckie. **Modeling the Total Energy Consumption of Mobile Network Services and Applications**. *Energies* 12:1 (2019). ISSN: 1996-1073. DOI: 10.3390/en12010184. URL: http://www.mdpi.com/1996-1073/12/1/184 (see pages 10, 28).

[239] Lei Ye, Gen Lu, Sushanth Kumar, Chris Gniady, and John H. Hartman. **Energy-efficient Storage in Virtual Machine Environments**. In: *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '10. Pittsburgh, Pennsylvania, USA: ACM, 2010, 75–84. ISBN: 978-1-60558-910-7. DOI: 10.1145/1735997.1736009. URL: http://doi.acm.org/10.1145/1735997.1736009 (see page 35).

[240] Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. **KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud**. In: HPDC '20. Stockholm, Sweden: Association for Computing Machinery, 2020, 173–184. ISBN: 9781450370523. DOI: 10.1145/3369583.3392679. URL: https://doi.org/10.1145/3369583.3392679 (see pages 49, 50, 53).

[241] Haitao Yuan, Jing Bi, Wei Tan, MengChu Zhou, Bo Hu Li, and Jianqiang Li. **TTSA: An Effective Scheduling Approach for Delay Bounded Tasks in Hybrid Clouds**. *IEEE Transactions on Cybernetics* 47:11 (2017), 3658–3668. DOI: 10.1109/TCYB.2016.2574766 (see pages 45, 47, 48).

[242] John Zedlewski, Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Arvind Krishnamurthy, and Randolph Wang. **Modeling Hard-Disk Power Consumption**. In: FAST '03. San Francisco, CA: USENIX Association, 2003, 217–230 (see page 35).

[243] Guowei Zhang, Fei Shen, Nanxi Chen, Pengcheng Zhu, Xuewu Dai, and Yang Yang. **DOTS: Delay-Optimal Task Scheduling Among Voluntary Nodes in Fog Networks**. *IEEE Internet of Things Journal* 6:2 (2019), 3533–3544. DOI: 10.1109/JIOT.2018.2887264 (see pages 63–65).

[244] Haitao Zhang, Huadong Ma, Guangping Fu, Xianda Yang, Zhe Jiang, and Yangyang Gao. **Container Based Video Surveillance Cloud Service with Fine-Grained Resource Provisioning**. In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 2016, 758–765. DOI: 10.1109/CLOUD.2016.0105 (see pages 48–52).

[245] PeiYun Zhang and MengChu Zhou. **Dynamic Cloud Task Scheduling Based on a Two-Stage Strategy**. *IEEE Transactions on Automation Science and Engineering* 15:2 (2018), 772–783. DOI: 10.1109/TASE.2017.2693688 (see pages 45, 47, 48).

[246] Yanqi Zhang, Yu Gan, and Christina Delimitrou. **mqSim: Enabling Accurate and Scalable Simulation for Interactive Microservices**. *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2019). DOI: 10.1109/ispass.2019.00034 (see pages 28, 35, 107).

[247] Cai Zhiyong and Xie Xiaolan. **An Improved Container Cloud Resource Scheduling Strategy**. In: *Proceedings of the 2019 4th International Conference on Intelligent Information Processing*. ICIIP 2019. China, China: Association for Computing Machinery, 2019, 383–387. ISBN: 9781450361910. DOI: 10.1145/3378065.3378138. URL: https://doi.org/10.1145/3378065.3378138 (see pages 49, 50, 53).

[248] Zhiheng Zhong and Rajkumar Buyya. **A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources**. *ACM Trans. Internet Technol.* 20:2 (Apr. 2020). ISSN: 1533-5399. DOI: 10.1145/3378447. URL: https://doi.org/10.1145/3378447 (see pages 49–53).

[249] Zhiheng Zhong, Jiabo He, Maria A. Rodriguez, Sarah Erfani, Ramamohanarao Kotagiri, and Rajkumar Buyya. **Heterogeneous Task Co-location in Containerized Cloud Computing Environments**. In: *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*. 2020, 79–88. DOI: 10.1109/ISORC49007.2020.00021 (see pages 46, 49–52).

[250] Zhiheng Zhong, Minxian Xu, Maria Alejandra Rodriguez, Chengzhong Xu, and Rajkumar Buyya. **Machine Learning-Based Orchestration of Containers: A Taxonomy and Future Directions**. *ACM Comput. Surv.* (Jan. 2022). Just Accepted. ISSN: 0360-0300. DOI: 10.1145/3510415. URL: https://doi.org/10.1145/3510415 (see pages 42, 46, 49–52).

[251] Xiaomin Zhu, Laurence T. Yang, Huangke Chen, Ji Wang, Shu Yin, and Xiaocheng Liu. **Real-Time Tasks Oriented Energy-Aware Scheduling in Virtualized Clouds**. *IEEE Transactions on Cloud Computing* 2:2 (2014), 168–180. DOI: 10.1109/TCC.2014.2310452 (see pages 46–48, 51, 52).

[252] Xingquan Zuo, Guoxiang Zhang, and Wei Tan. **Self-Adaptive Learning PSO-Based Deadline Constrained Task Scheduling for Hybrid IaaS Cloud**. *IEEE Transactions on Automation Science and Engineering* 11:2 (2014), 564–573. DOI: 10.1109/TASE.2013.2272758 (see pages 45, 47, 48).