IoT Orchestration in the Fog

Bruno de Moura Donassolo

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Bruno DE MOURA DONASSOLO

Thèse dirigée par **Arnaud LEGRAND**, Université Grenoble Alpes
et codirigée par **Panayotis MERTIKOPOULOS**, CNRS
et **Ilhem FAJJARI**, Orange Labs

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

L'orchestration des applications IoT dans le Fog

IoT Orchestration in the Fog

Thèse soutenue publiquement le **4 novembre 2020**,
devant le jury composé de :

Monsieur ARNAUD LEGRAND
DIRECTEUR DE RECHERCHE HDR, CNRS DELEGATION ALPES, Directeur de thèse
Monsieur ADRIEN LEBRE
PROFESSEUR, IMT ATLANTIQUE BRETAGNE-PAYS DE LA LOIRE, Rapporteur
Madame E. VERONICA BELMEGA
MAITRE DE CONFERENCES HDR, UNIVERSITE DE CERGY-PONTOISE, Rapporteure
Madame NATHALIE MITTON
DIRECTRICE DE RECHERCHE HDR, INRIA DELEGATION LILLE NORD EUROPE, Examinatrice
Monsieur FREDERIC DESPREZ
DIRECTEUR DE RECHERCHE HDR, INRIA CENTRE DE GRENOBLE RHÔNE-ALPES, Président
Monsieur PANAYOTIS MERTIKOPOULOS
CHARGE DE RECHERCHE HDR, INRIA CENTRE DE GRENOBLE RHÔNE-ALPES, Co-directeur de thèse
Monsieur Ilhem FAJJARI
Ingénieur recherche, Orange Labs, France, Co-encadrant de thèse
Monsieur OLA ANGELSMARK
DOCTEUR-INGENIEUR, ERICSSON A LUND - SUEDE, Examinateur

Abstract

Internet of Things (IoT) continues its evolution, causing a drastically growth of traffic and processing demands. Consequently, 5G players are urged to rethink their infrastructures. In this context, Fog computing bridges the gap between Cloud and edge devices, providing nearby devices with analytics and data storage capabilities, increasing considerably the capacity of the infrastructure.

However, the Fog raises several challenges which decelerate its adoption. Among them, the orchestration is crucial, handling the life-cycle management of IoT applications. In this thesis, we are mainly interested in: i) the provisioning problem, i.e., placing multi-component IoT applications on the heterogeneous Fog infrastructure; and ii) the reconfiguration problem, i.e., how to dynamically adapt the placement of applications, depending on application needs and evolution of resource usage.

To perform the orchestration studies, we first propose `FITOR`, an orchestration system for IoT applications in the Fog environment. This solution addresses the lack of practical Fog solutions, creating a realistic environment on which we can evaluate the orchestration proposals.

We study the Fog service provisioning issue in this practical environment. In this regard, we propose two novel strategies, `O-FSP` and `GO-FSP`, which optimize the placement of IoT application components while coping with their strict performance requirements. To do so, we first propose an Integer Linear Programming formulation for the IoT application provisioning problem. Based on extensive experiments, the results obtained show that the proposed strategies are able to decrease the provisioning cost while meeting the application requirements.

Finally, we tackle the reconfiguration problem, proposing and evaluating a series of reconfiguration algorithms, based on both online scheduling and online learning approaches. Through an extensive set of experiments, we demonstrate that the performance strongly depends on the quality and availability of information from Fog infrastructure and IoT applications. In addition, we show that a reactive and greedy strategy can overcome the performance of state-of-the-art online learning algorithms, as long as the strategy has access to a little extra information.

Contents

1 Introduction

Despite the enormous attention received in recent years, the Internet of Things (IoT) is not a new concept. One of the first connected devices was a Coke machine of Carnegie-Mellon Computer Science department in the early 80's [The98]. By consulting a minicomputer connected to it, members of the department could avoid unnecessary travels to an empty Coke machine. Nevertheless, the term IoT was only coined in 1999 by Kevin Ashton [KKL13], during a presentation in which he stated how Radio Frequency Identification (RFID) could revolutionize the way we manage and track products in the inventory control. In the past two decades, technology has evolved, reducing cost, size and improving performance of IoT devices.

The IoT promises to revolutionize our daily lives and the way we interact with our surrounding. It refers to the interconnection of heterogeneous end devices, especially everyday life objects, that are addressable and controllable via Internet. The number of such devices is expected to reach 64 billion by 2025 [Pet20]. This huge number of IoT devices is reshaping peoples' lives who are expected to use a myriad of applications with stringent requirements in terms of throughput, latency and reliability. The market value of IoT is expected to reach a trillion dollars in the next decade [Dme20]. This revolution has not been limited to the interconnection of objects themselves but has extended to people, process, data and things, creating what Cisco called Internet of Everything (IoE) [Eva12]. This holistic integration provided by IoE may unleash undoubted advances in many fields of science, medicine, business, among others.

The device-generated data is leveraged to power novel smart applications in various domains such as smart city, smart transportation, smart healthcare, etc. By processing the data, IoT applications can provide services which are aware of the surrounding environment, while considering the peoples' needs in terms of Quality of Service (QoS). Nevertheless, with such an exceptional evolution of demands, the network infrastructure struggles to resist to the overwhelming load and new innovative techniques are required. Therefore, operators need to make every possible effort to grow and consolidate their infrastructures making them ready for the deployment of next-generation network. A primary objective will be to place all

user-specific computation and intelligence at the edge while ensuring energy and natural resource usage efficiency.

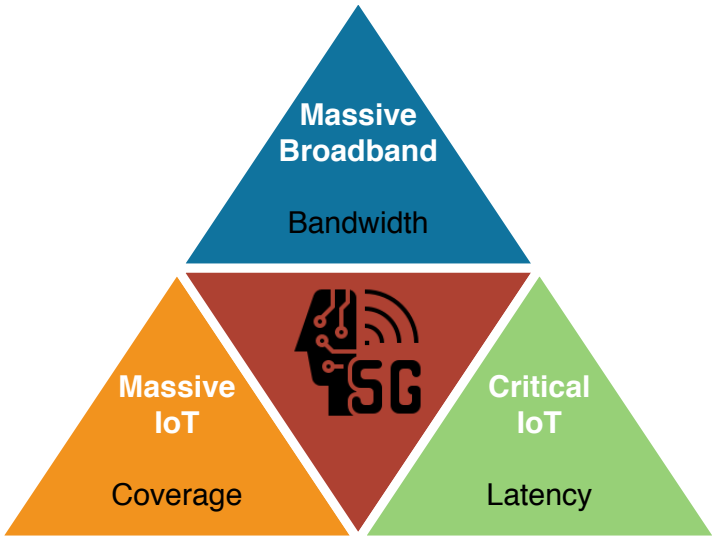


Figure 1.1: 5G and IoT. The three axes of applications and their main characteristics.

In this context, 5G is a key driver of IoT's proliferation. This fifth generation technology standard has been designed in such a way that offers a programmable and flexible infrastructure, allowing different networks (e.g., IoT, cellular, vehicular, etc.) to share the same access network. In doing so, 5G supports higher bandwidth speeds, increased reliability and availability, lower latency and better coverage, while reducing both CAPEX and OPEX. Fig. 1.1 depicts the three main categories of applications that 5G supports:

- **Massive Broadband**: refers to the applications requiring important bandwidth capabilities. According to [Fre20], mobility traffic is expected to reach 164 EB per month in 2025, 76% of it will be generated by video streaming. In this regard, 8K virtual reality applications may require up to **2.35 Gbps** of **bandwidth** [Man+17].

- **Critical IoT**: this segment includes critical applications where low latency is vital. For example, augmented reality applications require latency lower than 20 ms to ensure a satisfactory user's Quality of Experience (QoE) [YLL15]. Moreover, smart grid applications require even lower **latency**, up to **3 ms** [Sch+17].

- **Massive IoT**: refers to the applications which are less latency and bandwidth demanding but require a huge volume of devices with excellent coverage. In smart city applications for instance, the number of devices can reach **1 million per km^2** in high density areas [ITU17].

It is worth noting that these requirements will be even more stringent for the 6th generation of cellular networks. In [Tal+20], it is claimed that 6G networks can reach: i) peak data traffic greater than 1 Tb/s; ii) connectivity density in the order of 10 million devices per km^2; and iii) sub-ms latency.

For years, Cloud infrastructures have driven the IoT growth, providing increased scalability, enhanced performance and cost-effective resources. Nevertheless, the emergence of the aforementioned new IoT use cases poses several challenges that today's Cloud infrastructures, despite their agility, struggle to overcome. With its data centers, often deployed far away from end users, the Cloud cannot meet the stringent requirements in terms of network latency and bandwidth. Indeed, the expected 10 μs of latency for 6G networks [Zha+19], alongside the physical limit set by the speed of light in vacuum, will necessitate the deployment of application within a radius of approximately 3 km from end users.

In this context, Fog computing [Bon+12] has emerged as a novel concept to cope with Cloud's shortcomings. The idea behind the Fog is to extend the Cloud towards the end users, relying on geographically distributed and heterogeneous devices, such as servers, base-stations, routers, switches, etc. This augmented architecture provides nearby resources, performing analytics tasks and data storage. They are deployed between end devices and centralized services hosted by the Cloud. In doing so, a decentralized processing will be supported while taking advantage of the Cloud utilities and virtualization technology. It is straightforward to see that, the establishment of such a smart decentralized processing will ensure enhanced network performance, lower operational costs, alleviated network congestion and improved survivability.

Despite its multiple advantages, the heterogeneity and the distribution of Fog infrastructure raise new challenges in terms of resources and IoT application lifecycle management. Specifically, how to optimize the provisioning of IoT application modules on Fog devices to compose an application workflow, while meeting non-functional requirements in terms of quality of service (QoS), performance and low latency. Indeed, selecting optimal Fog resources becomes increasingly challenging when considering the uncertainty of the underlying Fog environment. In this perspective, **orchestration** is the cornerstone that will enable the exploitation of Fog infrastructures and the life-cycle management of multi-component IoT applications. It is worth noting that an IoT application can be modeled as a collection of lightweight, inter-dependent application components referred as micro-services. This building block is responsible for the design, on-boarding and delivering of application components that, put together, implement an end-to-end IoT service.

To achieve the aforementioned objectives, the orchestration in the Fog is responsible for: i) the provisioning of IoT applications, i.e., deciding where to place each micro-service in the infrastructure, while considering its requirements and the current state of the underlying environment; and ii) the reconfiguration of IoT applications, i.e., how to adapt the placement of micro-services, when the application requirements have changed or the infrastructure state has evolved. Note that the orchestration is a mature subject in Cloud environments, with many proposed systems such as Kubernetes[1] and OpenStack[2]. However, the dynamicity, diversity and uncertainty of Fog nodes and IoT applications make these solutions ill-suited for the Fog environment [JHT18].

Finally, this thesis aims to deal with these orchestration challenges, providing insights that will allow a better use of Fog infrastructures and, consequently, will leverage the deployment of innovative IoT applications.

Main contributions

The main contributions of this thesis are summarized as follows:

1. Initially, we conduct a survey of orchestration solutions in the literature. We provide an in-depth overview of existing Fog architectures, describing their characteristics and comparing their ability to manage the Fog environment. Moreover, we study state-of-the-art orchestration approaches, detailing how they deal with the provisioning and reconfiguration of IoT applications in the Fog.

2. We implement `FITOR`, an orchestration system which builds a realistic Fog environment, while offering efficient orchestration mechanisms. Our framework provides an effective representation of the Fog infrastructure in terms of topology, resources usage, circulating flows, etc. In this perspective, `FITOR` leverages this accurate view to implement powerful Fog service orchestration strategies. `FITOR` was rewarded with a "best student paper" at [Don+18] and presented as a demo in [Don+19a].

3. We study the provisioning of IoT applications in Fog infrastructures. To deal with the provisioning problem, we propose two novel strategies: `O-FSP` and `GO-FSP`.

[1] Kubernetes. Available at: https://kubernetes.io/
[2] OpenStack. Available at: https://www.openstack.org/

- `O-FSP` is a provisioning solution for IoT applications that optimizes the placement of the IoT application components while considering their requirements in terms of resources usage and QoS. `O-FSP` is a heuristic that iteratively constructs optimized solutions for the provisioning problem while minimizing the provisioning cost. The obtained results show that O-FSP enhances the resources usage and the acceptance rate of IoT applications compared with the related strategies. This work was published at IEEE CCNC 2019 [Don+19b].

- We extend and put forward `GO-FSP`, a Greedy Randomized Adaptive Search Procedure (GRASP)-based approach to the provisioning problem. `GO-FSP` bears two optimization objectives to fulfill QoS requirements of IoT applications. Indeed, it places IoT application components while jointly minimizing the provisioning cost and ensuring a satisfactory load share between Fog nodes. `GO-FSP` was presented at IEEE ICC 2019 [Don+19c].

4. Finally, we address the reconfiguration of IoT applications in the Fog. This reconfiguration study was submitted as a journal version to IEEE TPDS and is available at [Don+20].

    - We evaluate a total of twelve reconfiguration strategies based on different approaches, ranging from simple baseline strategies to sophisticated online learning and online scheduling strategies. Through an extensive analysis of the monitoring data, we identify the essential characteristics of each strategy as well as their impact on overall performance, which allowed us to propose substantial improvements to state-of-the art strategies.

    - Each of these strategies is studied in two distinct scenarios with different levels of information. In the first scenario, we show how strategies can take advantage of faithful application information provided by developers to describe their resource requirements. In the second scenario, on the other hand, we assess the performance of the strategies when this information is inaccurate.

    - We demonstrate that although off-the-shelf learning strategies are ineffective, reactive and greedy but informed strategies can achieve very good performance, even when compared to the situation where one would have access to a perfect and clairvoyant knowledge on the evolution in resource usage of each application. Surprisingly, these strategies perform well even in a scenario with inaccurate information.

Thesis outline

This thesis is composed of four main parts:

1. In Part I, we give insights into the orchestration problem studied in this thesis. Chapter 2 explains the context of this thesis, detailing the Fog environment and its main challenges. In Chapter 3, we give an overview of the existing architectures and orchestration approaches proposed in the context of Fog. Finally, Chapter 4 lays the foundations for the rest of the thesis, presenting the characteristics of the Fog infrastructure, IoT applications and the orchestration problem.

2. Part II is dedicated to the construction of the framework needed to effectively study the orchestration problem. In Chapter 5, we envision an architecture for the orchestration of IoT applications in the Fog, called `FITOR`. This architecture is built making use of the Calvin IoT framework, described in Chapter 6. In Chapter 7, we detail the components used to build `FITOR`, along with the modifications made to make Calvin Fog friendly. At the end of this part, our methodology is presented in Chapter 8.

3. We study the provisioning of IoT applications in the Fog at Part III. Chapter 9 introduces the provisioning problem along with its formulation. In Chapter 10, we propose `O-FSP`, an iterative heuristic to solve the provisioning problem which optimizes the provisioning cost, while considering the application requirements. In Chapter 11, we extend this work, introducing `GO-FSP` that not only minimizes the cost but also improves the load sharing of the infrastructure nodes.

4. Finally, Part IV is devoted to the reconfiguration problem. Chapter 12 details the reconfiguration problem addressed in this thesis. This problem is evaluated using two different scenarios. In Chapter 13, a well-informed environment is considered, in which the reconfiguration strategies have access to accurate information about applications and the environment. On the other hand, a less informed scenario is studied at in Chapter 14.

Finally, in Chapter 15, we summarize the main contributions and present our ongoing and future work in the field.

Part I

The Orchestration of IoT Application in the Fog: Definition and Challenges

2 Context

2.1 Fog Definition and Motivations

To cope with IoT applications' evolution, Fog computing has emerged as an alternative to avoid the burden of data-centers and network in Cloud infrastructures. By extending the Cloud towards the edge of the network, Fog is capable of supporting the geographically distributed, latency sensitive or bandwidth intensive IoT applications. The term Fog was first proposed by Cisco [Bon+12], and its name comes directly from nature, as the fog can be seen as clouds near the ground. Several researches have defined Fog computing in similar but complementary ways, such as:

- *"Fog computing is a scenario where a huge number of heterogeneous (wireless and sometimes autonomous) ubiquitous and decentralised communicate and potentially cooperate among them and with the network to perform storage and processing tasks without the intervention of third-parties. These tasks can be for supporting basic network functions or new services and applications that run in a sandboxed environment. Users leasing part of their devices to host these services get incentives for doing so."* [VR14]

- *"Fog Computing is a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud Computing Data Centers, typically, but not exclusively located at the edge of network."* [Bon+12]

- *"Fog computing is a layered model for enabling ubiquitous access to a shared continuum of scalable computing resources. The model the deployment of distributed, latency-aware applications and services, and consists of fog nodes (physical or virtual), residing between smart end-devices and centralized (cloud) services."* [Ior+18]

Fog computing is mainly a virtualized platform which provides computing, storage and network services anywhere along the continuum from Cloud to end users. Fig. 2.1 illustrates a conventional Fog environment, which is divided into three main layers:

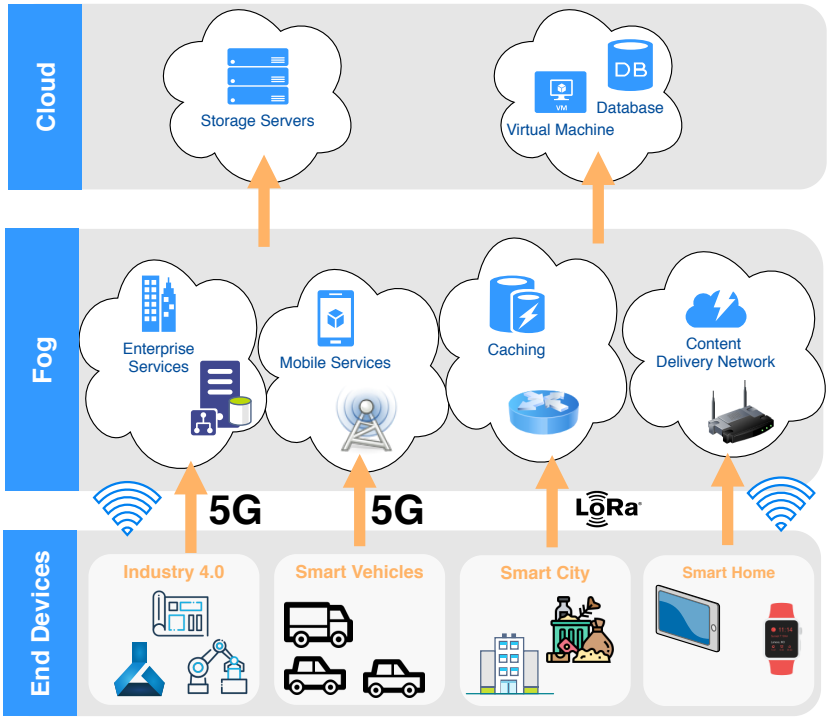


Figure 2.1: Illustration of the Fog environment where different domains share resources in the spectrum between End Devices and the Cloud to implement their business logic.

- **End Devices**: responsible for sensing and acting in the surrounding environment, they may scale to millions of components. Some examples are cited hereafter: temperature sensors, humidity sensors, smart lamps, locks, GPS, etc.

- **Fog**: it is a broad layer, including all the infrastructure nodes between end devices and the Cloud. Heterogeneity is a key attribute of the Fog which embraces a great variety of devices. In fact any equipment with some computing capability can be used, some examples are cited hereafter:

  - *Computing* devices: servers, PCs, raspberry PIs, etc.

  - *Networking* devices: gateways, routers, switches, SDN controllers, base stations, etc.

  - *Mobile* devices: vehicles, smartphones, laptops, tablets, etc.

- **Cloud**: traditional data-centers provide a last layer of processing to applications which need powerful computing and storage resources.

In Fig. 2.1, we can see some characteristics that emerge from this environment, making possible the deployment of new types of applications, such as:

- **Low latency**: many applications, belonging to the Industry 4.0 and smart vehicle domains, may require very low latency to achieve a satisfactory execution.

- **Location awareness**: the geographic location of IoT sensors is important for applications to understand the outer environment in which they are involved.

- **Geographic distribution**: is related to the distribution of IoT sensors in the environment, such as Smart City applications.

- **Mobility**: should be considered since many sensors and applications may be linked to a given mobility pattern of human beings.

- **Large scale**: Fog is known by its scale as it may aggregate a huge amount of IoT devices, geographically distributed over large areas, such as a city.

- **Wireless communications**: widely present at the edge of the network to communicate with IoT devices.

- **Heterogeneity**: is consequence of the great variety of devices present in the Fog.

- **Virtualization**: applications leverage a virtualization layer to deal with the heterogeneity of the infrastructure.

2.1.1 Motivations

The use of Cloud infrastructures by IoT applications brings several benefits, such as high amount of computing and storage resources, scalability, flexibility and availability. However, the Cloud is ill-suited to applications with stringent requirements. For instance, in the Internet of Vehicles domain, it is estimated that a single vehicle can generate approximatively 4000 GB of data every day which needs to be analyzed. Moreover, Lidar sensors and cameras may require up to 70 Mbps of bandwidth [Xu+18]. It is clear that uploading this huge amount of data to a centralized Cloud for further processing is not possible, due to the network infrastructure required. This kind of application can take advantage of a distributed environment, such as Vehicular Fog Computing [Hou+16b], keeping processing close to data generation and improving hence its performance and reliability.

In the remainder of this section, we discuss how applications can take advantage of Fog characteristics to provide better quality of service to end users.

Scalability

A major challenge to cope with in the context of IoT is related to the mobility pattern of people in their daily life. As a consequence of this movement, the use of resources by applications can vary significantly. However, the large number of nodes spread out geographically allows both horizontal (through the use of more nodes) and vertical (using more powerful Fog nodes) scalability, making the Fog capable of handling this variation in resource usage.

Heterogeneity

Another relevant feature of the Fog is its heterogeneity in terms of nodes. Likewise, the capabilities of the Fog nodes are different. Therefore, an application with specific needs, such as storage or graphics processing, can select the best nodes in the infrastructure that meet its needs.

Interaction cycles

According to [Gia+15], perception-action cycles are important for Fog applications. These cycles correspond to the information flow in the application, for example, a sensor sending data to the Cloud. Each cycle has different requirements in terms of delays. A proper placement of application components is important to meet the requirements of these interaction cycles.

Mobility

Many IoT applications, such as those involving vehicles or mobile phones, for example, are nomadic. A user who is walking around the city won't accept a degradation of his quality of experience due to connectivity issues. The geo-distribution of Fog nodes can overcome this challenge by providing nearby network and computing resources.

Security and privacy

The pervasive component of IoT devices may lead to the collection of sensitive and confidential data about users. This data needs to be correctly handled by the entities, guaranteeing the protection and privacy of data, as stated, for example, in the European Union's GDPR (General Data Protection Regulation). Despite all security challenges that the Fog environment imposes, applications can rely on nearby Fog nodes to process the data, and thus, avoid privacy issues when uploading sensible information to third-party Clouds.

Energy

The energy consumption of computing facilities induces a large carbon footprint. The use of distributed infrastructures, such as the Fog, can consume up to 14% less energy than fully centralized architectures [AOL19]. In addition, significant energy savings can be made by running an application locally. In [Jal+16], the authors identify classes of applications that can benefit, from an energy point of view, from the Fog computing. Namely, it includes applications which generate data continuously, such as video surveillance applications.

2.1.2 Initiatives

Fog computing is an active research topic that attracted both academic and industrial attention in recent years, with a corpus of literature consisting of thousands of papers. Lately, standardization initiatives have emerged aiming at defining a common language for the field. This section summarizes them.

OpenFog

On February, 2017, the OpenFog Consortium released its first version of reference architecture for Fog computing [Con17]. In their 162-pages document, they present a mid-to-high level architecture view for Fog nodes and networks. The document describes the infrastructure needed to build Fog as a Service (FaaS) platforms. FaaS includes Infrastructure, Platform and Software as a Service, providing the basic elements on which applications can be developed. The architecture relies on eight core pillars driving the implementation:

- Security: handles all security and privacy aspects, such as isolation, access control, encryption, etc.

- Scalability: addresses the dynamic aspects related to the Fog deployment. It includes the scalability of Fog nodes, software and network infrastructures handled in a demand-driven elastic environment.

- Openness: is a crucial property of the Fog. It corresponds to the composability and interoperability of Fog nodes from different vendors, enabling their automatic discovery and inclusion in the system.

- Autonomy: refers to the ability to provide the designed functionality at all levels of the infrastructure hierarchy, from the edge to the Cloud.

- Reliability, Availability and Serviceability: referred to as RAS, this building block aims to assure a continuous delivery of the designed service in face of external failures.

- Agility: analyzes data generated by end devices to make correct business and operational decisions, preferably without human interaction.

- Hierarchy: splits up the system into a logical hierarchy based on functional requirements. In doing so, each layer is responsible for the control, monitoring and operation of its nodes.

- Programmability: enables hardware and software programmability, as well as multi-tenancy and hardware abstraction through virtualization.

Furthermore, the OpenFog architecture description is presented according to three hierarchical viewpoints: node, system and software:

1. **Node view**: at the bottom layer, the node view corresponds to all aspects that must be addressed to bring a node to the system, such as security, management, network, access to sensors and actuators, etc.

2. **System view**: this viewpoint in the middle is responsible for aggregating one or more node views to create a platform and facilitate the Fog deployment.

3. **Software view**: at the top layer, this view contains application services, along with node and system views, used to address a particular customer scenario.

Beyond that, perspectives common to the three viewpoints are presented. These perspectives include: performance and scale aspects, security, manageability, data analytics and multi-vendor IT resources management.

IEEE 1934

In August 2018, IEEE COM/EdgeCloud-SC committee published the document "IEEE 1934-2018" [IEE18] which adopts the OpenFog Reference Architecture as the official standard for Fog computing. Even though the proposed document is very close to that issued by OpenFog community, seven changes are given in Annex. Among them, we highlight the adoption of a new pillar of the architecture:

- Manageability: orchestration and automation are owned by this pillar, which includes discovering nodes, provisioning services at the optimal place, managing the software life-cycle, and deploying software updates.

This new management pillar reinforces the importance of orchestration and automation in the Fog architecture, required to: i) optimize the placement of application for an enhanced resource utilization; ii) reduce traffic in the core of the network; and iii) enable data analytics for an improved decision making.

NIST - Information Technology Laboratory

A new NIST's (National Institute of Standards and Technology) standardization effort saw the light of day by March 2018. As the outcome of this work, a Special Publication 500-325, named "Fog Computing Conceptual Model" [Ior+18] was published. The report puts forward a conceptual model of Fog/Mist computing while detailing its relationship with the Cloud-based model for IoT.

In detail, the document highlights six essential points to distinguish Fog computing from other paradigms: i) contextual location awareness and low latency; ii) geographical distribution; iii) heterogeneity; iv) interoperability and federation; v) real-time interactions and vi) scalability and agility.

Moreover, the Fog's geo-distribution raises questions related to the ownership of Fog nodes. To answer these questions, NIST defines different deployment models for Fog nodes:

- Private Fog node: provisioned for exclusive use by a single organization.
- Community Fog node: provisioned for use by a specific community of consumers with shared interests.
- Public Fog node: provisioned for open use by the general public.

- Hybrid Fog node: it is a composition of two or more deployment models (private, community or public). They are bounded together to enable data and application portability.

2.1.3 Use cases

In this section, we describe some of the potential applications that can benefit from the flexibility offered by the Fog environment.

Smart building

Most cities in the world have a dense concentration of buildings, which gather a large amount of people in enclosed areas. In such environments, it is essential to rapidly and effectively address any emergency situation, such as a fire. Smart buildings can handle these situations by providing contextual data through their sensors and actuators.

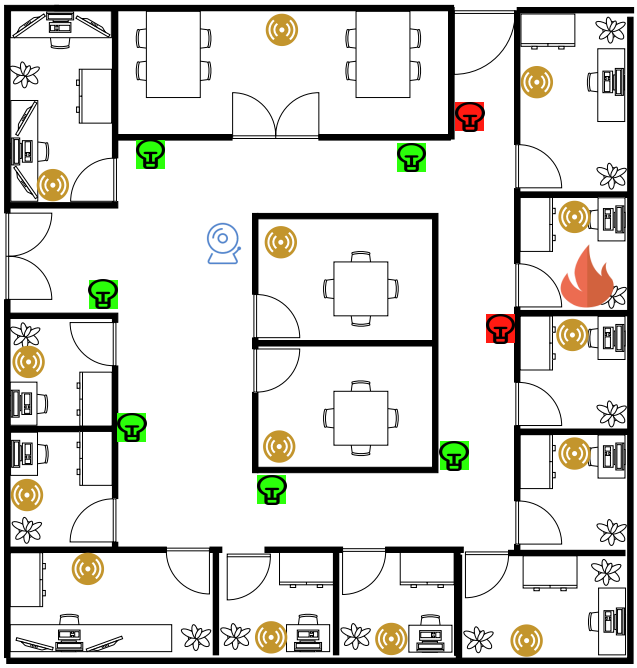


Figure 2.2: Example of a smart building application for fire combat.

To illustrate this scenario, Fig. 2.2 presents the map of an intelligent building, which contains a set of sensors, such as smart lamps, bells, cameras, screens, etc. The process of dealing with an emergency can be divided into three stages:

1. Identify the emergency: it is important to identify a potential fire as soon as possible. To achieve this objective, smoke or temperature sensors can be

used to locate the fire. In the absence of these sensors, cameras can be used to identify it.

2. Protect people: as soon the fire is detected, the first step is to ensure the safety of people. To do so, the system uses presence sensors and/or cameras to identify where people are. After that, it calculates an evacuation plan, finding the safest route to the available exits. The system then relies on smart lamps and screens to indicate the path. Additionally, it can use smart locks to avoid dangerous rooms that people may unconsciously enter. It is important to note that several requirements in terms of latency, bandwidth and processing are necessary at this stage.

3. Combat and emergency control: once the building is evacuated, the system can proceed to fight the fire, isolating it or using the available infrastructure to indicate the fire location to the firefighters.

Note that the blocks of the system must perform independently of the connection to the outside world, hence, relying on local components to run. Besides, it is straightforward to see that such a fire combat application has stringent requirements in terms of processing and memory (calculating evacuation plan, processing video from cameras, etc.), privacy (personal data from cameras), network latency (acting on sensors and actuators) and bandwidth (collecting data from cameras, sensors, etc.). In this context, Fog computing can provide the necessary resources to implement the application, considering all these critical requirements.

Smart cities

Making use of data generated by IoT devices will undeniably improve the quality of life for citizens around the world [Gha+17; Per+17]. Hereafter, some examples of applications offering new value-added services for cities.

In [Tan+17], the authors present a smart pipeline monitoring system. The monitoring system is able to detect potential hazardous events that could lead to the failure of the pipelines. Its implementation is based on a multi-layer monitoring system, which rests on a Fog environment. Edge devices are responsible for detecting disturbances in the pipeline, such as leakage or corrosion, while intermediate computing nodes can detect higher level events, such as damaged pipeline or approaching fire.

In [Brz+16] a levee monitoring use case is proposed. The idea behind this work is to distribute the analysis of the data collected. Initially, the measurement layer

reads information via sensor networks implanted in the levee. Then, the telemetry stations collect and process this data, generating flood alert warnings if necessary. By running it in a distributed manner, the system can perform a local threat level assessment even if Internet connectivity is lost.

The authors in [Per+17] propose a smart waste management system, consisting of the collection, transportation, processing, disposal and monitoring of waste. Proper waste management can bring not only financial, but also environmental benefits. To implement this system, garbage cans must be equipped with low cost passive sensors. These sensors, unable to send the data directly, will rely on nearby Fog nodes to collect, process and forward it. Subsequently, the gathered information is used to optimize the routes of the garbage trucks. Finally, this data can be used by the recycling companies to track the amount of garbage that reaches their plants.

Industry 4.0

The industry is undergoing its fourth revolution, which focuses on the end-to-end digitization of the production chain [GVS16]. The so-called Industry 4.0 is possible thanks to the use of Cyber-Physical System (CPS) and Internet of Things [Jaz14]. The CPS connects physical machines with computing and communication infrastructures, enabling hence their monitoring, control and automation. On the other hand, IoT sensors collect contextual data that can be used by algorithms to improve productivity.

Many activities belonging to the manufacturing industry can benefit from the Industry 4.0 paradigm. For example, the monitoring of cooling systems in a plant is vital to ensure its smooth operation. By using sensors in all its pipes and pumps, we can identify possible hazards, generate warning alerts and stop the production line. Yet, preventive maintenance can be performed when the cooling system deviates from its normal behavior, reducing hence machine downtime.

Nevertheless, several challenges arise in Industry 4.0 [BS15], such as: i) **mixed criticality**: the automation system may contain real and non-real time functionality; ii) **low latency**: short delays are vital for industrial processes. According to [Pul+19], smart factories may require latency ranging from 250 μs to 10 ms; iii) **ultra reliability**: the industrial environment has strict availability and reliability constraints; iv) **high safety**: functional safety is essential for many industries, such as oil and gas, nuclear plants; and v) **high security**: sensible collected data in industrial environment must be protected against external access.

In this context, Fog computing is a key enabler of Industry 4.0. Fog's flexibility, providing resources from the extreme edge until the Cloud, makes it able to deal with industrial challenges. For instance, a company can incorporate private resources into the Fog environment to have a low and predictable latency, in a safe and controllable infrastructure.

Smart transportation

The increase of traffic congestion is probably one of the most important challenges in large metropolitan areas nowadays. As the population is concentrated in big cities alongside a slowly evolving infrastructure, traffic congestion worsens, bringing new problems, such as accidents, air pollution, stress, etc.

In this context, VANET (Vehicular Ad-hoc NETworks) [HL09] offers new opportunities to implement ITS (Intelligent Transportation Systems), connecting vehicles and infrastructures together through wireless communications. Additionally, Fog computing can be used to meet the special requirements in terms of application mobility, location awareness and low latency [KCT16].

Fig. 2.3 illustrates a layered view of the transportation problem, where different amounts of information are available at each level. In the bottom layer, for example, a vehicle can use its own sensors to implement a parking assist application, without communicating with external elements. However, in the street view of the problem, a TLC (Traffic Light Control) application can be implemented, as defined by ETSI [Sys18]. Thanks to this application, vehicles and road side units can collaborate to avoid collisions, by controlling the traffic lights in the intersection area. The extreme edge can provide the resources needed to implement this application. Considering the neighborhood, hazardous event notification [Sys09] is another value-added service for smart transportation. In this case, the infrastructure in the edge is used to: i) identify the pothole; ii) calculate a possible

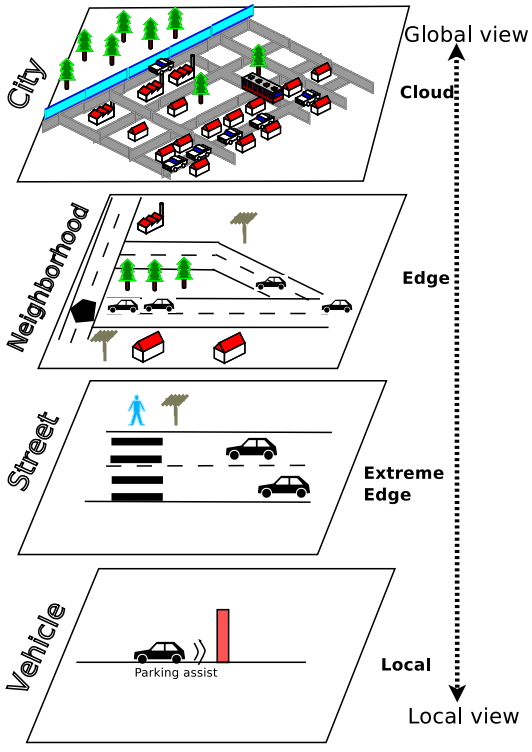


Figure 2.3: Smart transportation

alternative path; and iii) notify other vehicles about the incident. Finally, in the top layer, long-term analysis of the traffic pattern can be made to predict possible problems and thus, implement better public policies.

Smart grid

Electricity generation is responsible for emitting large amounts of CO_2 into the atmosphere. Coal, gas and oil accounted for 64% of the electricity generation mix in 2018 [IEA20]. Traditionally, energy is generated in central power plants, but alternative models are possible, such as the use of solar panels in households. In this context, Smart Grid [Fan+12] is essential, as it allows a better utilization of renewable energy generated by customers.

A Smart Grid infrastructure contains not only the power generators, but also, energy transformers, sensors and actuators. The system must support the bidirectional flow of electricity and adapt it following the usage pattern. For example, a house can supply power to the grid during work hours, but it must receive extra power during peak hours. The advantages of Smart Grids go beyond reducing greenhouse gas emissions, including: i) increase reliability and resilience to disruptions; ii) optimize the use of power plants and avoid the construction of new facilities; iii) automate maintenance and operation; and iv) reduce customers' energy consumption.

Despite its numerous benefits, many challenges arise from using Smart Grids. Specifically, autonomy and resilience are crucial to avoid power outages in critical environments, such as hospitals. However, its large scale and geo-distributed infrastructure makes the problem even harder. In addition, privacy is relevant in this context, since customer data is collected to optimize energy consumption. Therefore, the use of a centralized control unity in the Cloud is not convenient, due to the stringent requirement in terms of latency (3 ms to 20 ms) [Sch+17]. In this context, Fog computing is a prominent solution which can meet the requirements necessary for the implementation of Smart Grids, in terms of reduced latency, increased privacy and locality [OO16; AV16].

2.2 Fog Architecture Overview

As detailed in previous sections, Fog computing offers new opportunities for next generation networks. However, despite its multiple advantages, it brings new challenges to deal with. In this section, we give insights into a high level Fog architecture based on [Bon+14], responsible for managing this complex and distributed infrastructure.

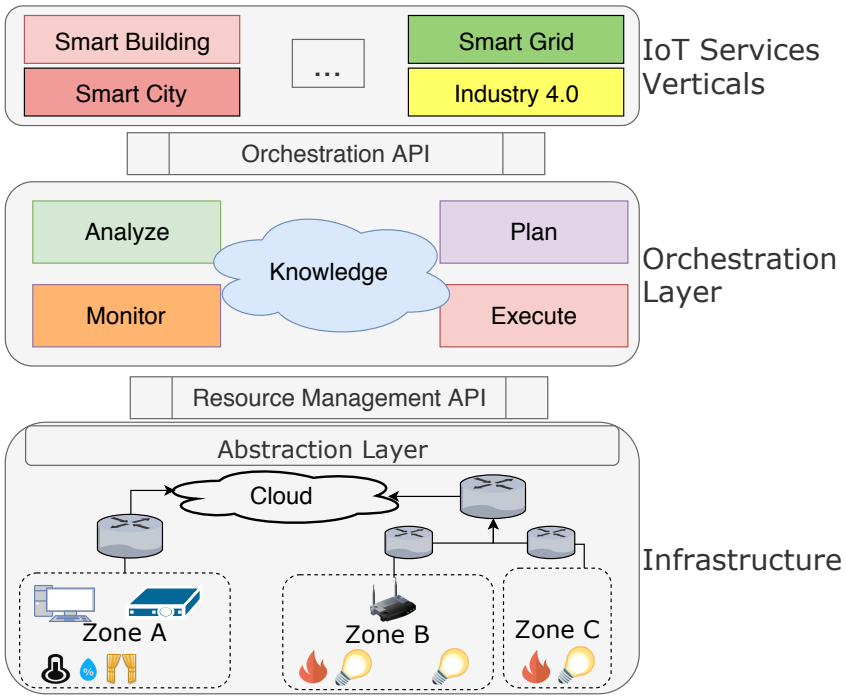


Figure 2.4: Fog architecture

Fig. 2.4 depicts the main layers of the architecture, together with the interfaces between them. The upper layer encompasses the different **IoT services verticals**. These verticals will potentially share the same underlying infrastructure; for this reason, it is important to provide efficient **resource management**, isolating and preventing applications from interfering in the execution of others. Moreover, the orchestration layer needs to expose a uniform and generic interface to the verticals, facilitating the use of the Fog landscape.

The **orchestration layer** is responsible for providing a dynamic, policy-based life-cycle management of IoT services. The orchestrator receives incoming IoT services requests from verticals; each service is characterized by a set of requirements in terms of resources, location and QoS necessary for its execution. Together with the information about the infrastructure, the orchestrator must provide the appropriate resources to comply with the requested policy. Note that, despite being illustrated as a single entity, the orchestration layer may be as distributed as the Fog infrastructure.

The complexity induced by the large number of pervasive IoT devices rapidly surpasses the human capacity to handle it. Therefore, autonomy is an important characteristic of the orchestration layer. An autonomic system is discerned by its capacity of self-configuration, self-optimization, self-healing and self-protection [KC03]. In Fog architecture, the MAPE-K loop concept (Map, Analyze, Plan, Execute and Knowledge) should be adopted to implement an autonomic orchestrator. The

MAPE-K loop is presented in Fig. 2.4, and each phase, adapted to the Fog context, is detailed hereafter:

- **Monitor**: collects all information about the Fog infrastructure and the performance of applications.

- **Analyze**: uses the data from the monitor phase to extract relevant statistics about the state of resources and applications.

- **Plan**: the output of the analysis phase and the requirements for IoT applications are used jointly by the planning phase to guarantee the allocation of the required resources for each vertical.

- **Execute**: this phase is responsible for reinforcing the implementation of the plan created in the previous phase.

Finally, the bottom layer in Fig. 2.4 presents the **Fog infrastructure**, which is distributed and heterogeneous by nature. The nodes range from simple and low-cost devices, such as Arduino [Ard] or Raspberry PI [Ras] boards, to set-top boxes, access points, edge routers, high-end servers, etc. These devices contain a varying amount of memory and storage, as well as different processors, operational systems and software stacks. Consequently, the resulting environment is composed of a vast hardware and software diversity. In this context, an **abstraction layer** is vital to make the Fog environment easily exploitable.

This high-level architecture can be translated into several implementations, depending on the physical constraints, applications and infrastructure. More details can be found in the related work and `FITOR` chapters (Chapters 3 and 5, respectively).

2.3 Challenges in Fog Architectures

Several studies highlight the challenges of Fog environments [VR14; Mou+18; MSW18; Nah+18; Pul+19]. The latter are related to various aspects, such as infrastructure, application, algorithms and platforms. In this section, we summarize the most relevant challenges that need to be addressed to achieve an efficient exploitation of the Fog.

2.3.1 Heterogeneity

The diversity of components is one of the main characteristics of the Fog infrastructure. Heterogeneity is a characteristic of the endpoints and Fog nodes. For the endpoints, different types of sensors and actuators are available, such as cameras, smart locks, temperature sensors, humidity meters and motion detectors. Likewise, Fog nodes may be heterogeneous, including servers, base stations, switches, routers, smart phones, etc. Consequently, applications running on the infrastructure must deal with such a diversity.

To deal with the endpoints' heterogeneity, semantic ontology is an important tool. Ontology describes the category, property and relationship between endpoints, facilitating their use by application developers. For example, given a security application aiming to take photos of a specific room, instead of directly accessing the camera in the room, the latter may require any recording-capable device (webcam, smartphones, tablets, etc.) available. This type of request can significantly improve the robustness of the application, but its implementation can be more complex, due to the heterogeneity and the different protocols used in the communication between the end-devices.

Interoperability is another important challenge posed by heterogeneity in the Fog. The variety of technologies used by the endpoints, such as Bluetooth, Wi-Fi, Zigbee, etc., makes the communication even more difficult. Therefore, an abstraction layer is necessary, capable of standardizing the data exchange between applications and endpoints. Moreover, the hardware and software heterogeneity of Fog nodes requires an additional virtualization layer to enable the application execution. In this context, technologies such as virtual machines or containers can encapsulate all the necessary libraries and dependencies to run applications.

Finally, the limited capacity of the devices in terms of compute, memory and storage poses challenges. Since resources are shared, often only a partial amount of the total resources is available to run the applications. Consider the case of a network switch or router at the edge of the network. Its main function is to provide connectivity to the rest of the network. Although a percentage of its spare resources can be used for other purposes, applications running on it should not impact its main functionality. To avoid such an impact, it is important to implement an adequate orchestration mechanism, capable of efficiently placing and managing the application components.

2.3.2 Mobility and scale

Part of the Fog environment is mobile, as it depends on the movement pattern of the humans using the devices. Some examples of these Fog nodes include: smart phones, wearable devices, connected cars or laptops. In addition, these devices could be unavailable due to power outage or battery depletion. As a result, the management entity that controls the Fog network must also take into account the availability, battery and movement patterns of nodes.

Moreover, the mobility, coupled with the scalability of the Fog environment, raises questions about the discovery of nodes and their federation. The former is related to the ability to locate and automatically insert new nodes in the environment. The latter is about the composition of nodes in self-organized communities to facilitate the management of the platform. In this context, good organization and discoverability are important to keep the complexity of the system under control.

Some of these challenges have already been addressed in the context of MANET (Mobile Ad-hoc Networks) and VANET (Vehicular Ad-hoc Networks) networks. Some authors [ElS+18; Fan+17; KCT16] propose the union of MANET and VANET with Fog and Edge computing, in order to benefit from Fog and mobile networks. Despite these works, the mobility and scale remain open challenges in the field.

2.3.3 Orchestration

The orchestration is responsible for the life-cycle management of applications running on the Fog. The challenge lies in discovering, deploying, configuring and maintaining the Fog nodes, which are generally managed by different entities. Furthermore, the orchestration must also consider the application perspective, meeting both functional and non-functional requirements to provide a good service experience for end users.

In this direction, the locality of Fog nodes can be explored to implement a more reliable system, which adapts and eliminates the need for a central management entity. P2P (peer-to-peer) systems, for example, can provide interesting insights on how to build, manage and integrate such environments in the Fog.

Another relevant issue that the orchestration must address is fault management. In this large scale environment, the probability of failure of a single node is quite high. Various reasons can cause failures, such as hardware problems, software incompatibility, user activity, mobility, connectivity, battery, etc. Therefore, monitoring

the Fog nodes is vital, not only to keep track of active nodes, but also to know the availability of resources (CPU, RAM, etc.) on Fog nodes.

The orchestration is also responsible for ensuring a high quality of service for end users. To do so, it is mandatory to establish an SLA (Service Level Agreement) between the infrastructure owner and the application developer. Defining the parameters to be considered when describing an SLA is yet another issue. Beyond the definition of a guideline that developers can use to describe the desired QoS for their applications, the orchestrator must be able to guarantee this SLA through the multi-layer, multi-owner, distributed and heterogeneous Fog environment.

Finally, we identify two main phases in the application life-cycle management, the **provisioning** and the **reconfiguration**. The **provisioning** is responsible for selecting the best nodes in the Fog infrastructure to run the applications. This procedure must address the application requirements and the objectives of the infrastructure owner. Despite the latest research efforts, scenarios that take into account several objectives (e.g. cost, latency, bandwidth, etc.) are not sufficiently investigated in the literature. In addition, the **reconfiguration** problem aims to adapt the initial placement of applications to changes in infrastructure or application needs. These changes include, for example, nodes entering/leaving the system or peaks load caused by more end users using the application. To solve these challenges, the orchestration system must be able to scale or reconfigure the applications. Although many studies address these issues in other areas, a comprehensive approach to the Fog is still needed.

2.3.4 Security and privacy

In [KS18; YQL15], authors point out the security and privacy concerns in IoT environments. These challenges are even greater in this multi-ownership environment, which can include: i) end users who provide their private nodes to reduce cost or improve privacy; ii) Internet service providers which control home and cellular bases stations; iii) Cloud providers which supply sufficient and powerful resources for applications. This joint ownership and flexibility of the Fog complicate the security and privacy aspects of the system.

Security is required in each component of a Fog infrastructure. For instance, Fog nodes at the edge of the network usually use wireless communications, which may be exposed to sniffing or jamming attacks. Applications must use trusted nodes that do not compromise the implemented functionality. Additionally, users must be authenticated before accessing the application. All of these problems must

be addressed considering the heterogeneity, scale and constraints in terms of IT resources at different Fog nodes.

The widespread presence of sensors that collect user data, potentially sensitive, raises major privacy concerns: i) **Data privacy**: applications can offload user data processing, which must be done at trusted nodes and with appropriate encryption; and ii) **Location privacy**: the location is a sensitive information, widely available on modern devices, such as smartphones and smart-watches. This data must be protected to avoid putting people in dangerous situations, such as robbery or kidnapping.

In such scenarios, it is clear that, to ensure user privacy, intelligent privacy mechanisms must be implemented, considering all the characteristics of the IoT and Fog environments.

2.3.5 Application modeling

IoT and Fog infrastructures offer opportunities for developers to implement new applications to end users. However, to take full advantage of the environment, new programming models must be developed to ease the modeling of applications in the Fog.

In this sense, micro-service architecture is a promising technique for the development of IoT applications. By dividing into small, self-contained and independent modules, applications can be easily scaled to handle a large number of end users. Moreover, availability can be guaranteed by deploying the same micro-service on different infrastructure nodes.

Despite the potential benefits of the micro-service architecture, its adoption raises several challenges. Ideally, micro-services are stateless to facilitate their deployment and scalability. However, many services require saving the state, whose administration in a highly unstable and volatile environment, such as the Fog, may be complicated.

Finally, as the Fog environment is extremely heterogeneous, we need a unified language to describe the application requirements and SLAs. Developers must be capable of describing their needs in terms of IT resources (CPU, RAM, storage, libraries, etc.) and network (latency, bandwidth, error rate, etc.). Furthermore, the application must be able to provide a feedback on its current performance, and possibly, require new resources to respond to some peak usage in the application.

2.4 Scope of this Thesis

In Section 2.3, we gave an overview of the main challenges related to the use of Fog environments. In this thesis, we focus our research on the study of the orchestration of IoT applications in a Fog environment. Nevertheless, the term "orchestration" is generic and can include many aspects of the lifecycle management of IoT applications, such as testing, version or release management. In this work, we address two aspects of the orchestration: provisioning and reconfiguration of IoT applications.

The first aspect is the provisioning of applications in the environment. In this context, we intend to answer the following question: "*Where to efficiently place each component of an IoT application on the available Fog infrastructure, while considering application's requirements and the capacity of the infrastructure?*" Answering this question requires defining which parameters are important to consider when describing IoT applications, along with characteristics and objectives on the infrastructure side.

The reconfiguration problem arises once applications are provisioned and are running on the infrastructure. We must adapt the placement to guarantee a good quality of service, even when the application or infrastructure changes. This leads to new challenges which we address in this thesis, such as: i) defining the behavior of applications, i.e., the evolution of application load; ii) defining the satisfaction (utility) of running applications; and iii) choosing the reconfiguration mechanism to be adopted (migration, scale in/out, up/down).

3 Related Work

In this chapter, we give an in-depth overview of existing Fog architectures, describing their characteristics and comparing their ability to manage the Fog environment. Moreover, we study orchestration approaches proposed in the literature, detailing how they deal with the provisioning and reconfiguration of IoT applications in the Fog.

3.1 Overview of Fog Architectures

In this section, we give an overview of the existing Fog architectures and platforms. Initially, we describe the criteria used to classify the available platforms, while briefly describing each proposal.

3.1.1 Classification

The following parameters are used to classify and compare the Fog architectures proposed in the literature, detailed in Table 3.1:

- **Architecture**: describes the number of tiers into which the Fog environment is divided. Each layer groups nodes with similar characteristics. Also, this item details the architecture target: general, for any type of application, or application specific, if a domain is specified (medical, smart city, etc.).

- **Prototype**: whether the architecture is implemented, or at least, a prototype is provided to validate the proposed architecture.

- **Heterogeneity**: describes whether the proposal is able to handle the infrastructure. More precisely, each proposal describes its solution with different details, e.g. containers, VMs, or simply an abstraction layer.

- **Monitoring**: defines whether a monitoring solution is supported by the architecture to give a deep insight into the infrastructure.

- **QoS**: sets out whether the desired QoS is specified, in some level of detail, for IoT applications. Note that we made no distinction in the level of detail available to describe the QoS.

- **Mobility**: describes the capacity of the proposed architecture to handle user mobility.

- **Reconfiguration**: describes the system's ability to evolve over time, either by scaling resources or migrating applications.

3.1.2 Comparison

In [Yi+15], a 3-layer architecture is presented, composed of user, Fog and Cloud layers. The characteristics of the main components of the proposed Fog architecture are discussed, such as System Monitor, Resource/Offloading Management, Authentication and Authorization. However, no details on the actual implementation are given. Unlike most works in the field [Bri+17; Yan+16; Wan+18; Sau+16; Hon+13; Ran], the authors claim that VMs are more adapted than containers as virtualization technology, because they can host different types of guest operating systems. A small scale prototype based on OpenStack and Amazon EC2 is implemented. As an evaluation, the response time of a simple Face Recognition application is measured, considering the case of the application running on the Fog or the Cloud environments.

The authors in [SA16] propose a 4-layer Fog architecture composed of: i) end devices, ii) Cloud, iii) SDN control plane; and iv) SDN data plane. The paper introduces the concept of proxy VMs, responsible for processing the data received from a given IoT device. The system is able to reconfigure itself and deal with the user's mobility. However, it does not include requirements of applications in terms of IT or network resources. More specifically, mobility is managed by the decomposition and/or migration of proxy VMs between different nodes in the SDN cellular core (data plane). Simulations are performed using mobility traces of real users, showing that the total traffic on the network core can be reduced through dynamic VM allocation.

In [TLG16], the authors propose a hierarchical edge-cloud architecture to handle workloads from mobile devices, where each tier aggregates the peak load from lower layers. The 3-layer architecture contains: i) the mobile devices, ii) a multi-layer, hierarchical edge; and iii) the Cloud. The proposal handles the users' workload by: i) deciding the amount of server capacity available for each workload and ii) selecting on which server each task will run. The results, which are based

Table 3.1: Comparison of Fog architectures

	Architecture	Heterogeneity	Prototype	Monitoring	QoS	Mobility	Reconfig.
[Yi+15]	3-tier, general	✓ VMs	✓	✓	✗	✗	✗
[SA16]	4-tier, general	✓ proxy VMs	✗	✓	✗	✓	✓
[TLG16]	3-tier, general	✗	✓	✗	✗	✗	✗
[Den+16; Den+15]	5-tier, general	✗	✗	✗	✓	✗	✗
[SM16]	3-tier, general	✗	✗	✗	✓	✓	✗
[Tan+15; Tan+17]	4-tier, smart city	✗	✓	✗	✗	✗	✗
[SCM18]	3-tier, general	✓ abstraction layer	✗	✓	✓	✓	✗
[Mas+16]	4-tier, general	✓ no details	✗	✗	✓	✓	✗
[Kap+17]	4-tier, general	✓ abstraction layer	✗	✓	✓	✓	✗
[Yan+17]	3-tier, general	✓ abstraction layer	✗	✓	✓	✓	✓
[Bri+17]	2-tier, general	✓ containers	✓	✓	✗	✗	✗
[Yan+16]	3-tier, general	✓ containers	✓	✓	✓	✗	✓
[Tom+17]	3-tier, general	✓ VMs or containers	✗	✓	✓	✓	✓
[Wan+18]	3-tier, general	✓ containers	✓	✓	✓	✓	✓
[Hou+16a]	3-tier, vehicular	✗	✗	✗	✗	✓	✗
[Wen+17]	3-tier, general	✗	✗	✗	✓	✗	✗
[Sau+16; Hon+13]	3-tier, general	✓ containers	✓	✓	✓	✓	✓
[Rah+18]	3-tier, e-health	✓ abstraction layer	✓	✓	✗	✓	✓
[Ran; Xio+18]	2-tier, general	✓ containers	✓	✓	✗	✗	✓

on a small scale prototype and a trace-based simulation, show that a hierarchical architecture improves the completion time for tasks on heavy workloads.

In [Den+16; Den+15], the authors study the tradeoff between delay and power consumption in a Fog environment. The proposed architecture is composed of end users, Local Area Networks (LAN), Fog nodes, Wide Area Networks (WAN) and the Cloud. In this system, delays are calculated following a queuing system, which considers the traffic arrival and the service rate for each node. QoS is defined as the maximum time allowed running each service. On the other hand, energy consumption is calculated depending on the device: i) for Fog nodes, by a quadratic function which takes into account the assigned workload, and ii) for the Cloud, more energy-efficient, by a linear function which considers the CPU frequency. Finally, they calculate the optimal allocation of services for the Fog and Cloud, which minimizes the power consumption while guaranteeing a certain delay constraint.

A mathematical model for a 3-tier Fog architecture is presented in [SM16]. The model characterizes the main components in a Fog environment, such as terminal nodes, Fog devices and the Cloud. Also, service metrics are detailed in terms of latency and energy consumption. At the end, a theoretical performance evaluation is carried out for latency and energy. The evaluation considers different loads in the Fog and Cloud. The results highlight the possible gain of using the resources available in the Fog, in both service latency and energy consumption.

The papers [Tan+15; Tan+17] propose a hierarchical Fog architecture for Smart Cities, whose focus is on smart pipeline monitoring. In this context, the layers of Fog nodes are used to monitor different levels of the infrastructure, depending on the latency requirements. A real prototype of pipeline monitoring system is implemented to perform studies on response time and data traffic. Nevertheless, being a specific architecture, the general details are not thoroughly presented.

The suitability of a Fog environment is studied in [SCM18]. In this work, the environment is divided into three layers: terminal nodes, Fog and Cloud. The study varies the number of terminal nodes and analyzes different metrics: i) energy consumption, ii) latency, iii) CO_2 emission; and iv) cost. In addition, the authors discuss some aspects of Fog environments, such as monitoring, hardware abstraction, QoS and mobility. However, no prototype or details are provided on each of these topics.

In [Mas+16], the Fog-to-cloud computing paradigm is proposed. The idea is to have a vertical for each application that uses the Fog environment, such as vehicular, communities, etc. In this scenario, coordinated management is necessary to organize the different elements used by a service. This coordinated management must

assure the desired QoS while dealing with the heterogeneity and mobility of nodes. Although no prototype is presented, a medical emergency scenario is analyzed, showing the potential speedup provided by the Fog-to-cloud paradigm.

[Kap+17] presents a cooperative model for Fog systems, where edge nodes can register themselves in the Fog platform. The architecture consists in 4-layers: device, hub, Fog and the Cloud. The communication follows a publish/subscribe model and a standard format for Fog messages is defined. In this architecture, the hub layer is responsible for translating and abstracting different device technologies. Moreover, the message contains headers to describe the desired QoS. For example, a time sensitivity flag is used for urgent tasks, and a RAM field in the header is used to describe the amount of memory needed to process the message.

By bringing together the OpenFog and ETSI MANO architectures, the authors in [Yan+17] propose a theoretical architecture for Fog computing. The ETSI MANO is extended to manage instances in the Fog domain and not just in the backend/network domain. To clarify the differences between the two environments, the concept of VF (Virtual Function) is defined. VF extends and replaces VNFs (Virtual Network Functions), by running also user applications. In addition, extensions to the ETSI MANO data modeling language are proposed, which capture the business requirements needed in the Fog. The OpenFog architecture, on the other hand, provides vertical perspectives, such as manageability and performance/scale. These perspectives provide monitoring, reconfiguration and mobility capabilities to the system.

Similarly, inspired by ETSI NFV MANO reference architecture, the authors in [Bri+17] present a 2-layer service orchestration architecture, composed of end devices and Fog nodes. This architecture contains two main components: i) a Fog Orchestration Agent (FOA), which runs on Fog nodes and is responsible for managing the containerized services running locally; and ii) a Fog Orchestrator (FO) which uses a centralized view of the environment to control IoT services and Fog nodes. To maintain an updated view of the environment, each FOA monitors the resources available on its Fog node. This information can be collected by the FO, if necessary, during the orchestration phase. A prototype based on Docker Swarm and OpenMTC technologies is implemented as a proof-of-concept.

Based on the Cloud Foundry architecture, the authors in [Yan+16] propose extensions to support the orchestration of IoT applications, forming a 3-tier architecture composed of IoT, Gateways and the Cloud. These extensions include new modules to: i) develop and deploy IoT applications; ii) manage and monitor applications and infrastructure; and iii) ensuring a certain QoS level, by scaling up/down a service

container. A prototype is implemented to analyze the total delay of a fire detection application running in a distributed environment.

Another architecture based on SDN is presented in [Tom+17]. In this work, the environment is decomposed into several Fog regions which are managed by an SDN controller. The latter orchestrates the running IoT applications, keeping an updated view of the available Fog nodes, their location and their capabilities (IT and network resources). By having the complete information about the environment, the SDN controller can handle the user mobility and reconfigure the applications by replicating and/or migrating them. Although the architecture for the SDN controller is depicted, detailing its functional blocks, no prototype is implemented to prove its feasibility.

The Enorm framework [Wan+18] is a 3-tier architecture composed of Cloud, edge and end devices. The environment is divided into regions and the management is centralized in the Cloud. End devices offload tasks to edge nodes when necessary, either due to user mobility or QoS (e.g. violation of latency constraint). Besides that, scalability is handled by an auto-scaling mechanism which scales up/down the resources, considering the network latency and task execution time.

The survey in [Hou+16a] puts forward the use of vehicles to improve current vehicular infrastructures, and thus, envision a VFC (Vehicular Fog Computing) environment. By performing an extensive study on mobility and connectivity patterns on vehicular traces from large-scale urban areas, the authors claim that the QoS of applications can be enhanced by using the available vehicles as support for running applications. Moreover, the authors discuss the challenges in the VFC environment without, however, detailing how the architecture could be implemented.

The authors in [Wen+17] present a Fog orchestrator concept based on a Planning-Execution-Optimization control loop. The orchestrator has a typical 3-tier architecture: sensors, Fog and Cloud. In their prototype, applications are provisioned and the QoS is enhanced thanks to the use of a genetic algorithm.

In [Sau+16; Hon+13], the authors propound a programming infrastructure for Fog environments called Foglet. In the Foglet, the environment is divided in geographic regions, where each Fog node is responsible for a set of end devices. Relying on the description provided by the developer, such as location, capacity and QoS requirements (e.g. class of computing such as CPU, storage, etc.), the runtime system manages the application execution. Furthermore, the applications are continuously monitored by the Foglet and a migration mechanism is implemented to handle user mobility and to guarantee a certain QoS level (e.g. communication

latency). In regard to the virtualization layer, the authors claim that containers are well-adapted thanks to their smaller memory footprint compared to VMs.

The suitability of Fog computing for health care systems is studied in [Rah+18]. The authors propose a 3-tier architecture for e-health systems, composed of a sensors/actuators network, smart e-health gateways and back-end system. The specific properties needed by the e-health system include: i) interoperability: smart gateways abstract the communication to heterogeneous sensors; ii) adaptability: a module in the gateway is responsible for adapting and reconfiguring the Fog layer when critical events arrive; and iii) mobility: handover and roaming are described to avoid service interruptions and data loss. An EWS (Early Warning Score) health application is implemented as a proof-of-concept for the use of Fog computing in this domain. The application uses a set of smart gateways to monitor patients' vital signals and warns the medical staff when they deviate from the expected standard.

In 2019, Rancher[1] released a lightweight Kubernetes distribution[2], called k3s [Ran], which focuses on running Kubernetes at the edge of the network. By removing non vital components and simplifying the installation process, k3s offers a certified Kubernetes version, capable of running on small ARM devices, with 512MB of RAM and 200MB of disk space. K3s is based on a server/agent model, where the server manages a set of agent nodes that run the application containers. In this way, heterogeneity is treated with the use of containers, while an autoscaler module guarantees the horizontal scale of the applications. Although k3s is a promising project that enables the use of a Kubernetes orchestration system at the edge of the network, some challenges remain open, especially in managing the mobility and the geo-distribution of Fog nodes.

Similarly to k3s, KubeEdge [Xio+18] aims to provide a Kubernetes-based infrastructure for the network edge. While k3s enables all components (server and agents) of the Kubernetes architecture to run on the edge, KubeEdge relies on the Cloud to run the control plane server which manages the infrastructure. Thanks to two novel components (EdgeController and MetadataSyncService), KubeEdge is able to continue operating even when the connectivity between Cloud and edge is lost.

[1] Rancher. URL: https://rancher.com/.
[2] Kubernetes. URL: https://kubernetes.io/.

3.2 Orchestration Approaches

Several strategies are proposed in literature to deal with the orchestration of IoT applications in Fog environments. In this section, an overview of these solutions is presented, describing: i) how the placement problem is modeled, solved and evaluated; ii) which QoS parameters are considered; and iii) whether the proposed approach deals with the configuration of the already running applications to cope with the evolution of the system and the applications.

3.2.1 Classification

The criteria considered to classify the related orchestration strategies are detailed below:

- **Application requirements**: what information the developer uses to describe the application requirements:
  - **Node**: parameters related to the node, typically: CPU, RAM and storage.
  - **Network**: network-related information, such as bandwidth and latency.
  - **Other**: includes everything that is not directly related to the node or the network. For example, deadline for task execution, SLAs (Service Level Agreements), etc.
- **System modeling and problem formulation**: this criterion describes the theoretical framework used to model the system and orchestration problem.
- **Objective**: emphasizes the objective of the solution, e.g. minimize cost, maximize infrastructure utilization, etc.
- **Solution**: describes the resolution approach adopted to solve the orchestration problem. Authors can solve it by using an exact method or proposing some approximation heuristic.
- **Reconfiguration**: reflects the capacity of the system to reconfigure itself face to changes in the infrastructure or applications.
- **Evaluation methodology**: describes how the proposed solution is evaluated. Usually, it includes the following options: i) numerically: the evaluation is

performed by solving the mathematical model of the problem, ii) simulation: relies on the use of some simulation to carry out the evaluation; and iii) testbed: an implementation is done and tested in a real environment.

Table 3.3: Comparison of orchestration approaches

	Application Requirements			Model	Objective	Solution	Reconfig.	Eval.
	Node	Network	Other					
[Ska+17a]	CPU, RAM, storage	-	Deadline	ILP	Max. Fog resources utilization	Exact	Solve problem periodically	Simulation
[Ska+17b]	CPU, RAM, storage	-	Deadline	ILP	Max. Fog resource utilization	Heuristic	Solve problem periodically	Simulation
[You+18]	CPU, RAM, storage	-	Deadline, SLA violation	INLP	Min. cost	Heuristic	Solve problem periodically	Simulation
[Xia+18]	CPU, RAM, storage, locality	Bandwidth, latency	-	ILP	Min. average response time	Heuristic	-	Simulation
[Hon+17]	CPU, RAM, locality	Bandwidth	-	ILP	Max. number of satisfied requests	Heuristic	-	Simulation and testbed
[BG17]	Processing	-	-	MILP	Min. cost	Heuristic	Migration	Simulation
[Sau+16]	Locality, type	Latency	-	-	-	-	Migration	Testbed
[Den+16]	-	Bandwidth	Delay, service rate	MINLP	Min. power consumption	Optimal approxi-mation	-	Numerically

Table 3.3: Comparison of orchestration approaches (continued)

	Application Requirements			Model	Objective	Solution	Reconfig.	Eval.
	Node	Network	Other					
[AH15a; AH15b]	CPU, RAM, storage	Bandwidth	-	Probabilistic	Resource estimation	Exact	-	Simulation
[Gia+15]	Storage, locality, type	Bandwidth	User-defined	-	-	-	-	Testbed
[Nis+13]	Processing	Bandwidth	-	Convex optimization	Max. utility	Exact	-	Numerically
[Zen+16]	Storage	-	Nº replicas/IO interrupts, service rate	MINLP	Min. task completion time	Heuristic	-	Simulation
[Ott+13]	-	-	Total delay	Time-graph	Min. bandwidth utilization	Exact	Migration	Simulation
[Gu+17]	Processing, storage	-	Total delay	MILP	Min. cost	Heuristic	-	Simulation
[OSB15]	Processing	-	Delay	Combinato-rial	Min. power consumption	Heuristic	-	Simulation
[BF17]	CPU, RAM, storage, SW, locality, type	Bandwidth, latency	-	-	-	Heuristic	-	Simulation

Table 3.3: Comparison of orchestration approaches (continued)

	Application Requirements			Model	Objective	Solution	Reconfig.	Eval.
	Node	Network	Other					
[TLG16]	Processing	-	-	MINLP	Min. total delay	Heuristic	-	Numerically, simulation, testbed
[CG19]	-	-	-	OCO	Min. delay	Gradient descent	Learning	Numerically
[Ait+19]	CPU, RAM, locality	Bandwidth, latency	-	CSP	Min. weighted average latency	Exact	-	Numerically

3.2.2 Comparison

In [Ska+17a], the authors study the Fog service placement problem (FSPP). The concept of Fog colonies is introduced to divide the Fog landscape into smaller, manageable areas. FSPP models the problem as an Integer Linear Programming (ILP), whose objective consists in maximizing the Fog utilization while taking into account all constraints specified by the user. These constraints include: i) CPU, RAM and storage and ii) deadline for tasks. However, no description of network requirements is allowed. The problem is then solved using the IBM CPLEX solver. The reconfiguration problem is handled by solving the ILP problem periodically, but no further details are given. Finally, an extension for the iFogSim is implemented to evaluate the proposal in a simulated environment.

Due to the complexity of the service placement problem, [Ska+17b] extends the previous work [Ska+17a] by implementing a heuristic to solve it. The heuristic is based on a genetic algorithm that iteratively tries to improve the solution by performing operations in the placement, such as selection, crossover and mutation.

In [You+18], the provisioning problem is modeled as an INLP (Integer Nonlinear Programming). In terms of IT resources, the model accepts the description of CPU, RAM and storage. Once again, no network requirements are considered, although a maximum delay for each task can be configured. Moreover, users can describe a percentage of acceptable SLA violation and a penalty cost (if the SLA violation threshold is reached). Together with other costs related to the use of the infrastructure, a total cost is calculated. The objective function consists in minimizing this cost. To solve it, two greedy algorithms are implemented and evaluated using simulation. Finally, the authors claim that the reconfiguration can be made by solving the INLP problem periodically.

The model proposed in [Xia+18] considers also the requirements in terms of network resources and node location. The concept of Dedicated Zone is introduced, i.e., a geographical area where the application component can be placed. Moreover, it is possible to describe bandwidth and latency requirements for application links. By minimizing the communication time, the proposed approach aims to decrease the average response time of applications running on a Fog infrastructure. Several solutions are proposed to address the problem. Initially, a backtrack search procedure generates all possible solutions. Due to its high cost, two heuristics are proposed, based on the location characteristic of Fog nodes and applications. The proposed approach is evaluated using a home made simulator on top of SimGrid. However, the reconfiguration aspect of the orchestration is not considered in this paper.

In [Hon+17], the authors propound a heuristic that aims to maximize the number of satisfied IoT requests, while respecting targeted QoS levels. The proposed ILP model includes constraints in terms of CPU, RAM and locality for IT resources, and bandwidth for network links. However, the latency is not considered in this work. To maximize the number of accepted requests, a greedy algorithm ranks the demands for the scarcest resources first, as they are likely to be the most common cause for request rejection. The evaluation comprises an implementation in a real testbed and an extensive test in a simulator.

The MCAPP (Multi-component Application Placement Problem) is presented in [BG17] and it is modeled as a MILP (Mixed Integer Linear Program) problem. The applications describe only their processing requirements and no other criterion is considered (memory, bandwidth, etc.). Moreover, the applications have associated costs to run, communicate and relocate their components. Thus, the goal is to minimize the total execution cost. In the MCAPP, the system is divided into T time-slots and a heuristic is adopted to calculate the placement for each time-slot. First, the heuristic places the application's components without considering the communication cost using the Hungarian algorithm. Afterwards, a search phase is used to improve the initial placement considering the communication costs.

A developer-centric approach is presented in [Sau+16], where the Foglet programming interface is proposed. In this work, the authors do not aim to optimize the use of the environment, but rather to provide an interface that allows the description of application requirements, including: i) locality: geo-spatial region where the application may run; ii) type: computational class required for the application, e.g. processing; and iii) latency: communication delay between each application level. Although no optimization is performed during the placement of applications, a migration mechanism is proposed to redeploy the applications when the latency threshold is reached. A proof-of-concept is implemented using C++, ZeroMQ and Protobuf.

In [Den+16], the authors study the workload allocation problem in a Cloud-Fog environment. Their objective is to minimize the power consumption, taking into account the delay agreed in the SLA. The workload is characterized by a service rate dispatched to the Fog or Cloud for processing. Computation delays are calculated according to the queuing theory, while the communication delay is considered when the requests are dispatched from the Fog to the Cloud. Finally, the power consumption minimization problem is decomposed into three parts, which are solved independently: i) energy consumption vs. delay in the Fog; ii) energy consumption vs. delay in the Cloud; and iii) minimizing the communication delay. Therefore, an approximate optimal solution for the problem is proposed. Finally,

the system is assessed through a numerical evaluation using MATLAB in a small scale setup with five Fog devices and three Cloud servers.

The authors in [AH15a] propose a method to calculate the amount of resources (CPU, memory, storage and bandwidth) that a service provider needs to allocate to satisfy user requests. The model considers the service price agreed in the contract and the probability that users will relinquish the demanded resources. Users are classified into two categories: i) high: users who have more than 50% chance of giving up the resources; and ii) low: when the probability is less than 50%. This probability is calculated considering the user's historical information. A simulation, using CloudSim, shows that the proposal can help the service provider to estimate the amount of resources. The work in [AH15b] extends [AH15a] by taking into account the users' mobility pattern to calculate the amount of resources needed. Consequently, it gives more resources to highly mobile users to ensure good quality of service. Nevertheless, both papers estimate only the amount of resources needed and are not interested in placing the applications in the infrastructure.

A distributed dataflow programming model for the Fog is propounded in [Gia+15], in which IoT applications are described as directed graphs. The developer can annotate the nodes in the graph with the characteristics of the application, such as location, available storage, computation class type, bandwidth and user-defined properties. These annotations are matched against the physical nodes in the infrastructure when the application is deployed. However, no optimization is performed when provisioning the IoT applications in the environment. A proof-of-concept is implemented on top of IBM's Node-Red dataflow programming framework [JS 13].

In [Nis+13], a mathematical framework is used to analyze the service provisioning problem in a mobile cloud environment. A service is characterized by a set of tasks that can be offloaded to other nodes to reduce completion times. Furthermore, the proposed framework is generic, abstracting the different resources (processing, communication, etc.) in utility functions. Therefore, the objective is to maximize the sum of utility functions. When the system has good properties, i.e. convex and continuous functions, the problem is solved through convex optimization. This is the case of the evaluation scenario, where a small scale set-up, considering two nodes and two services, is solved numerically.

The authors in [Zen+16] address two related problems that arise when we execute tasks on a Fog environment: task scheduling and data placement. In their model, a set of clients can whether run tasks locally or dispatch them to computing servers. To be executed, each task loads part of the data from the storage servers. The model includes the service rate, the size of the image to be stored, the average number

of I/O interrupts and the number of replicas for each data image. These two problems are modeled as a MINLP (Mixed-Integer Non-Linear Problem) and solved through a heuristic. The algorithm minimizes the storage and computation times independently and then, couples the solution considering the transmission time. The evaluation is carried out through simulation, comparing the proposed heuristic with two greedy approaches which run tasks only on clients or on servers.

In [Ott+13], the authors propose a placement method to deal with the mobility of end users in a Fog environment. The applications are composed of operators, which process data streams from mobile sources. The placement method relies on a time-graph model, which decides where to place and when to migrate each application operator. In this time-graph model, the vertex represents the physical node hosting the operator, while the edge represents the bandwidth used to deploy, execute and migrate (if necessary) the operator. Finally, the model is solved by finding the shortest-path in the graph.

[Gu+17] focuses on the placement of Virtual Medical Devices (VMDs) in a Fog-based medical cyber-physical system. The applications are composed of a set of VMDs, which must be placed closer to the end users to meet the QoS constraints. The problem is modeled as a MILP (Mixed Integer Linear problem), whose objective is to minimize the total cost of communication and VMD deployment. Due to its intractability, the problem is solved through a 2-phase heuristic: i) initially, the communication cost is minimized by solving a relaxed version (LP) of the MILP problem and, subsequently, approximating the solution to valid integer values; and ii) the deployment cost is minimized by migrating small VMDs to other resources.

In the context of cellular networks, the authors in [OSB15] study offloading of tasks to Fog nodes. The environment is composed of Small Cells (SCs), each containing computation and storage capabilities. These SCs are grouped together to form Small Cell Clouds (SCCs). A model for the generation of SCCs is presented, minimizing the overall communication power consumption, while respecting the delay of tasks. This combinatorial optimization problem is solved by a 2-phase algorithm: i) in the first phase, each SC sorts its tasks according to some metric (latency, energy, etc.); and ii) each SC informs its capacity and exceeding tasks, i.e. tasks that cannot be executed by the SC. These tasks are re-sorted and the clusters needed to process them are formed. In the end, different metrics (latency, power consumption, etc.) are tested in a simulated urban environment.

Another approach is proposed in [BF17]. Instead of optimizing the application deployment, a set of possible deployments is generated. For this, the authors propose a model for describing IoT applications and the Fog infrastructure. The model specifies the properties of applications and devices in terms of: i) hardware

and software, such as RAM, storage, operational system or installed libraries; ii) location and type; and iii) network bandwidth and latency. Afterwards, the deployment is defined as a mapping function between application's components (alongside their requirements) and the infrastructure. Finally, heuristics generate eligible deployments, which respect the requirements described by the user, but do not optimize the infrastructure utilization.

In [TLG16], a hierarchical edge architecture is presented, in which the application can be run in any level of the infrastructure between the end-user and the Cloud. The studied placement problem aims to minimize the total delay to execute the application, while considering individual computation and communication delays. However, users cannot describe individual requirements in terms of network and memory capacities. To achieve this objective, a heuristic based on a simulated annealing algorithm is proposed to solve the problem. Two outputs are generated: where to place the application and the amount of computation resources provided to it. The extensive evaluation includes a numerical evaluation of lower bounds on delays, a small scale testbed and a large scale simulation based on Wikipedia's network trace.

A different model for offloading of tasks in a Fog environment is presented in [CG19]. In this model, users are not able to describe requirements for the IoT applications. Instead, the model accepts unknown constraint functions which are revealed after the offloaded is performed. Their objective is to minimize a cost function (e.g. average delay) while guaranteeing a long-term satisfaction of the constraint function. At each time step, the feedback of objective and constraint functions are available. To solve the problem, the authors propose the use of a gradient descent method with bandit feedback, which learns the best task offloading and configures the system accordingly. At last, the proposed approach is assessed numerically in a smart home scenario with 10 Fog nodes and one Cloud center, evaluating the total cost and the constraint violations.

Finally, the authors in [Ait+19] tackle the problem in a similar but innovative manner. They model the provisioning problem as a CSP (Constraint Satisfaction Problem), which considers the main characteristics of the Fog, such as CPU, RAM, network bandwidth and latency. Instead of proposing a heuristic for solving the problem, Choco constraint solver [JRL08] is used to obtain an exact solution. To assess the scalability of the proposal, different scenarios are evaluated, varying the number of applications, the number of components per application and the infrastructure size. However, no evaluation is performed to assess the application performances in the Fog environment.

3.3 Conclusion

Despite being a relatively new concept, Fog has attracted a lot of attention. As detailed in the previous sections, several works proposed architectures and orchestration systems for the Fog, covering many relevant challenges in the field. Nevertheless, we could identify some limitations in these works. First, we lack real implementations of Fog systems, while some papers discuss only the challenges and possible solutions, some rely on small scale homemade prototypes as proof of concept for their proposals.

Regarding the orchestration, the provisioning problem is certainly the one which received the most attention. Several models were proposed, considering different application requirements and objective functions. Due to the hardness and size of the problem, most authors propose heuristics to solve it, the exception of few exact solutions. A common characteristic of these works is that they consider a single optimization objective (e.g., minimize delay or maximize utility). Therefore, an interesting research direction is to extend them to account for multiple, and sometimes, conflicting objectives.

On the other hand, the reconfiguration aspect of the orchestration has not been sufficiently investigated, with a handful papers studying this problem. Most of them deal with the reconfiguration by solving the provisioning problem periodically. In this approach, a snapshot of applications and the infrastructure is taken at the end of each time step, considering all the changes that occurred in the last period. With this snapshot, the provisioning model is used to optimize the placement of applications, assuming that they will not change during the next period. Hence, it is straightforward to see that comprehensive studies on reconfiguration approaches are necessary for the Fog.

Finally, the evaluation of proposed approaches is mainly based on simulation. Despite all advantages of using simulation, such as the adaptability and scalability of the experiments, the validation of the simulator is delicate. Many simulators are built specifically for a study and their resource models cannot be validated. In a highly heterogeneous and scalable environment such as the Fog, this limitation can lead to misleading conclusions. Consequently, the use of real environments can mitigate this problem and provide reliable and trustworthy results.

4 Components and Characteristics of a Fog-IoT Environment

In this chapter, we lay the foundations for the rest of the thesis, detailing the components present in our environment. It includes: i) the model and elements of the Fog infrastructure; ii) IoT applications and their requirements and iii) the components and characteristics of the orchestration problem.

4.1 Fog Infrastructure

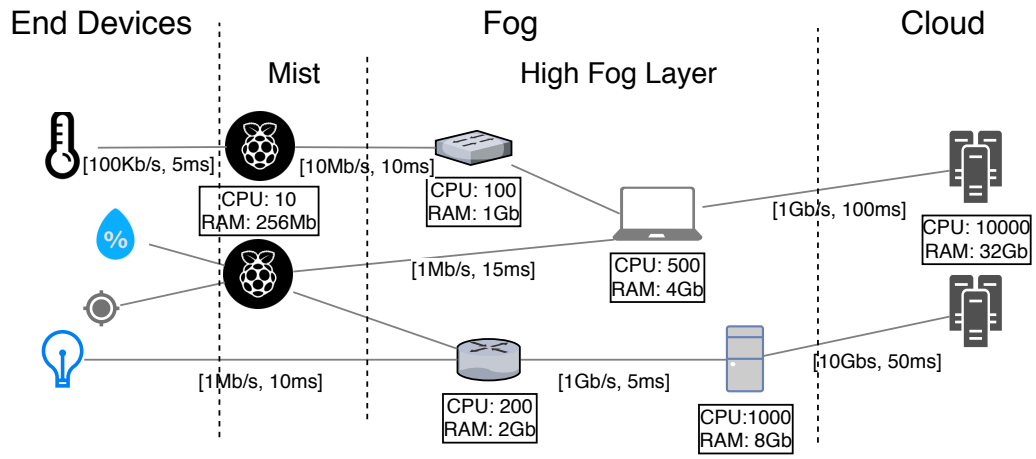


Figure 4.1: Graph model for a Fog infrastructure. This figure illustrates the typical Fog infrastructure used in this thesis.

Fig. 4.1 depicts the adopted Fog infrastructure, from end-devices to the Cloud. According to [ATT17], millions of devices are present in the edge of the network. The scale gradually decreases as we reach the Cloud in the top of the infrastructure, which encompasses only dozens of powerful nodes. We emphasize that the Fog layer can be divided into two sub-layers:

- **Mist**: in the extreme edge of the network, this first layer of nodes allows some processing with a very low latency, typically less than 5ms. In contrast, they have very limited processing, memory and storage capacities.

- **High Fog**: contains everything else until the Cloud. Their devices have greater resource capacities and, consequently, can support complex and resource-intensive applications.

The Fog infrastructure can be modeled as a graph, where the vertices represent the devices on which applications can run and the edges are the network links. Nodes and edges are labeled with their characteristics, such as CPU, RAM or location for nodes, and network latency and bandwidth for edges. Note that the edges are undirected, which denotes a bi-directional path between infrastructure nodes. Moreover, Fig. 4.1 presents the typical components and their characteristics which are considered throughout this thesis.

4.2 IoT Applications

IoT applications (or services)[1] are usually long-term processes, which interact with sensors and actuators in the environment to implement a service to end users. Applications collect and analyze the data from sensors, working on the environment through actuators, if necessary. Furthermore, IoT applications running in a Fog environment face several challenges, such as heterogeneity, geo-distribution and limited resources.

In this context, micro-services arise as a promising architectural style to cope with these challenges [BGT16]. An IoT application is composed of a set of these building blocks, called micro-services, which communicate together. Each micro-service is responsible for implementing a tiny part of the business logic, and by putting many micro-services together, a complete end-to-end IoT service can be easily implemented. According to [Fow15], micro-services are small, modular, loosely-coupled, independent deployable and with diversified technology. Consequently, they are easily adaptable for the constraint devices present in the Fog environment.

The variety of micro-services and the heterogeneity of the Fog pose new communication challenges. Application components must be able to communicate with each other and IoT devices, independently of the infrastructure. Regarding the access to IoT devices, [Diz+19] highlights two main communication patterns: i) publish/subscription for sensing the data; and ii) request-response for acting in the environment. Moreover, Ports and Adapters pattern [Coc05] offers an interesting concept to deal with the heterogeneity. The principle is simple, the communication of the components with the external world is performed via well-defined ports,

[1] Note that we use the terms "IoT applications" and "IoT services" interchangeably throughout the text.

on which a technology-specific adapter converts the message according to the underlying hardware.

The use of micro-services, communicating via well-defined interfaces, provides the adequate abstraction necessary in a Fog environment. Hence, the application can be seen as a DAG (Directed Acyclic Graph), where the nodes are the micro-services and the edges are the links connecting two ports. This is enough to describe the functional behavior of applications; however, we still need to describe their non-functional requirements.

4.2.1 Requirements

The ability to describe non-functional requirements is vital for IoT application running on a Fog environment. Nevertheless, the list of possible parameters can be quite long, depending on the characteristics of applications and the Fog. In Fig. 4.2, we list some of these requirements, which range from device-oriented requirements, such as CPU and RAM, through network bandwidth and latency, to more specific ones, such as task deadline or location.

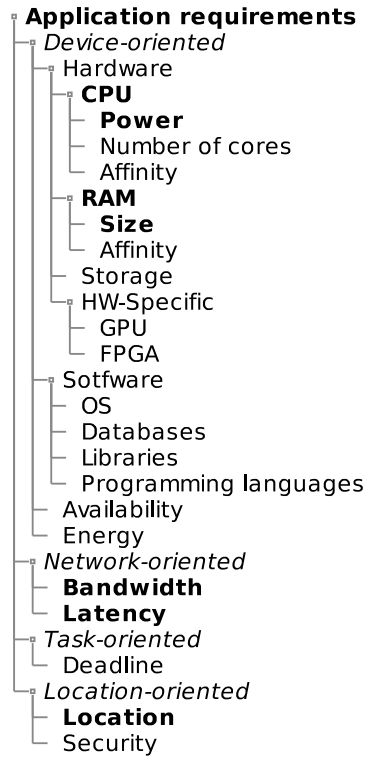


Figure 4.2: List of possible application requirements (non-exhaustive).

Given this complexity, in our work, we focus on a subset of these requirements that we believe are representative for most applications running on the Fog, including: **CPU** power, **RAM** capacity, **location**, network **bandwidth** and **latency**.

Fig. 4.3 typifies the graph of an IoT application. In this application, two data sources generate data that are processed by components in the application logic layer. Edges in the graph are annotated with the network requirements, while nodes have requirements in terms of CPU, RAM and location.

4.3 Orchestration

Orchestration is the third pillar of our study, enabling IoT applications to leverage the Fog infrastructure. As discussed in Section 2.4, we are mainly interested in

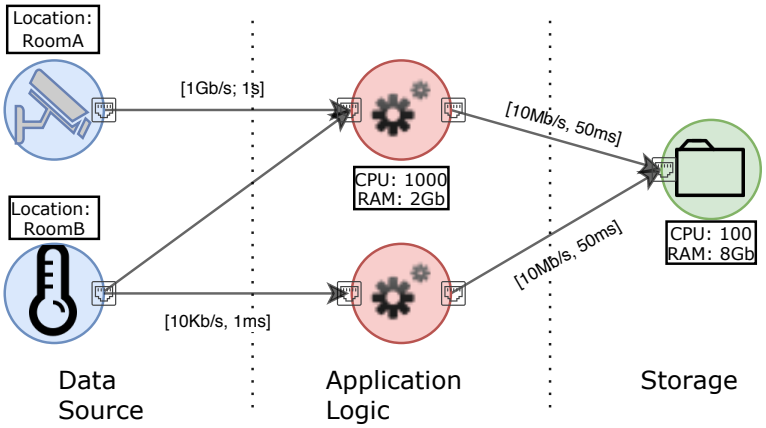


Figure 4.3: Graph model for a Fog-IoT application. This model represents a typical application studied in this work.

two aspects of the orchestration problem: provisioning and reconfiguration of IoT applications. Nevertheless, the context in which these problems are involved brings specific characteristics to the orchestration (online, inexact, multi-objective, distributed and delayed), which we will discuss in this section.

Despite the advantages of the micro-services architecture, it leads to an increased operational complexity, due to the management of a large number of tiny services [Fow15]. These services enter and leave the system and their use of resources changes over time. This evolution leads to our first specific characteristic: the **online** aspect of the orchestration problem.

Regarding the IoT applications, we have seen that they may have different demands in terms of software and hardware capabilities, which are often difficult to be described accurately. Hence, we have the **inexact** aspect that the orchestrator must consider. Moreover, different metrics can be used to evaluate the satisfaction of the application with the current resources, such as elapsed time or throughput.

On the other hand, infrastructure owners have their objectives which can be conflicting with application's metrics. These objectives may include: minimize infrastructure cost, minimize congestion, maximize resource utilization, etc. This **multi-objective** scenario brings forward extra challenges to the orchestration system.

In addition, the large scale and distribution of the Fog environment make the use of a centralized orchestrator poorly adapted. Although it is not mandatory, a **distributed** orchestration system is therefore appropriate.

Finally, large scale and distribution are also responsible for the **delay** in obtaining information about the infrastructure. Despite the effort to keep an updated view of the available resources, it is impossible to have a real-time information about all resources available in the infrastructure.

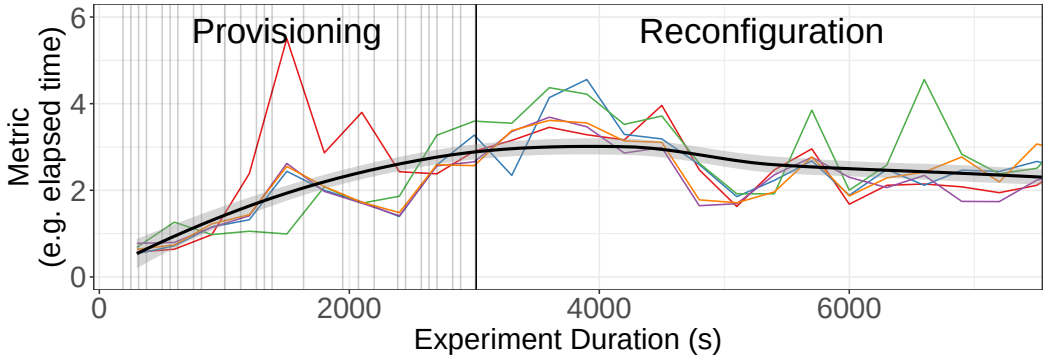


Figure 4.4: Studying the orchestration phases. Each colored line represents the metric for one test run. The black line represents the average performance of all runs. During the provisioning phase, each vertical gray line represents the deployment of an application.

In Fig. 4.4, we present the appearance of the orchestration experiments that we performed during this thesis. Each colored line corresponds to the execution of a single test, measuring and presenting in y-axis some metric (e.g. average elapsed time, aggregating all applications over a five minutes interval).

During the first phase, the initial deployment of IoT applications is performed. The **provisioning** can be seen as a mapping problem, where we map the application graph over the infrastructure graph, respecting the application requirements and the infrastructure resources. Each vertical gray line represents an application entering the system that must be provisioned. In Part III, we study this problem, analyzing different metrics (e.g. cost, acceptance rate, etc.) until all applications are deployed (black vertical line in the figure).

Once all applications are deployed and running, we have the **reconfiguration** phase, which is studied in Part IV. During this phase, applications are sharing and competing for resources. Despite the requirements provided in the provisioning phase, applications evolve and, consequently, change their use of resources. Therefore, we study different manners to reconfigure the placement of applications to keep them satisfied with the available resources.

Nevertheless, before getting to the heart of this thesis, with the study of the provisioning and reconfiguration, we have to build the foundations of our work. Hence, the next chapters in Part II are devoted to detail the research methodology and the orchestration system used during our experiments.

Part II

From Theory to Practice: Research Methodology and An Orchestrator for IoT Applications in the Fog

5 An Architecture for IoT Orchestration in the Fog

The first step consists in implementing a framework for the orchestration of IoT applications in the Fog. To achieve our objective, we analyze what are the fundamental components needed to mimic a Fog environment and also, to orchestrate IoT applications. It includes from infrastructure aspects, such as the characteristics of nodes, up to the software components to run and analyze the platform. As result of this reflection, we propose the `FITOR` (**Fog IoT OR**chestrator) architecture, which contains all necessary parts to perform the orchestration of IoT applications in the Fog. Although it is not a complete architecture for the Fog (many aspects, such as security and resilience, are not considered), `FITOR` is able to provide all basic blocks for the study carried out within the scope of this thesis.

Fig. 5.1 provides an overview of the `FITOR` architecture. Our proposed solution is device-aware, and hence, handles the heterogeneity of the Fog environment. Application components can be easily deployed on end devices and Fog nodes. Hereinafter, we detail each component of the proposed architecture.

The Service Descriptor

This component aims to describe the IoT application, its building components and its requirements. As discussed in Section 4.2, an IoT application is usually composed of a set of micro-services which interact to implement the business logic. In this context, the actor model [Hew10] emerges naturally as an option to describe an IoT application which will be later implemented as micro-services. In short, an actor is a unit logic that performs simple tasks and sends messages to other actors (more details about the actor model are presented in Section 6.3).

Indeed, the developer needs to describe the actors, their requirements in terms of both location and computational effort, and how data should circulate between them. It is worth noting that, in order to ensure a guaranteed QoS, network related requirements could be specified during the description of links between the actors. Specifically, CPU/RAM affinity and capacity can be specified for the actors, while latency and bandwidth can be defined for the links.

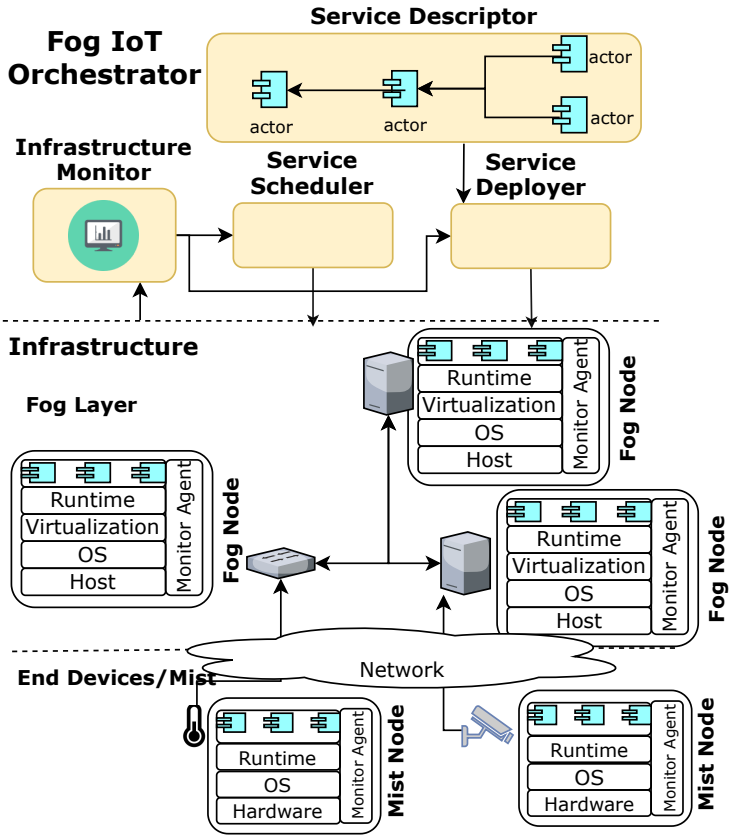


Figure 5.1: `FITOR` architecture

The Service Deployer

Once the description is submitted, this component handles the mapping between the application components (i.e., actors, communications) and the nodes hosting the latter. Its main objective consists in optimizing the placement of actors and their links while considering their location, computational and network requirements. To do so, it makes use of the service descriptor and the collected infrastructure related metrics, to find the best placement of the application components.

The Service Scheduler

A running application is exposed to the uncertainty of the Fog environment. In addition, its components may change in response to the variation of data sources or internal transformation. To deal with this dynamicity, the Service Scheduler uses the monitored key performance indicators from the applications and from the Infrastructure Monitor to trigger scale out/in actions to allocate or de-allocate resources. Besides, migration actions could be triggered when necessary.

The Infrastructure Monitor

This is responsible for sketching out the telemetry information by extracting several resource metrics from the Fog nodes and links. To do so, it makes use of various probes to get real-time information about both physical resources and their running containers. In our context, we consider two main categories of metrics to observe:

- **Host-related metrics**: concerning all information about the host and container in which the application is running. This includes: CPU, RAM, disk, etc.

- **Network related metrics**: corresponding to the end-to-end latency and bandwidth, which are crucial and thus, mandatory to be collected during the application execution.

It is important to note that two views of resources are provided: i) potentially available resources, which is a global and static view of the total capacity of nodes in terms of compute, storage and RAM; ii) the real-time availability of resources, which considers the current load induced by the applications.

Fog node

Fog nodes are the main workforce in the environment, on which the IoT applications will run. They can be hosted by servers, network equipment, or even end devices. Due to its heterogeneity, they have a **virtualization** layer and are capable of running containerized (e.g., docker) micro-services. On the top of the virtualization layer, the so-called **runtimes** are responsible for the execution of actors. They handle, for example, the actor scheduling and the data transport message parsing. It is worth noting that one runtime may concurrently run multiple actor instances belonging to different applications. A runtime is characterized by a set of capabilities (e.g., access to a camera, access to a disk, etc.) and performances metrics (e.g., CPU, memory) which are monitored by the **monitor agent**. The latter collects, aggregates, processes and exports information about all components in the Fog node, including the physical host, running containers and network bandwidth and latency.

Mist node

On the other hand, mist nodes are the first processing layer, with nodes very close to the sensors and actuators. They are very similar to regular Fog nodes (precisely, mist nodes are a subclass of Fog nodes) and are able to execute IoT applications.

However, due to their constrained resources, the virtualization layer is removed and, consequently, the runtime layer runs directly in bare-metal.

6 The Calvin Framework

In this chapter, we present the Calvin framework, a key element in the implementation of the architecture proposed in Chapter 5. We describe the reasons why we have chosen Calvin to be our IoT framework, its architecture and how it describes and deploys the IoT applications. An in-depth look at Calvin is important to understand how FITOR is built.

6.1 Why Calvin?

Table 6.1: Comparison of IoT frameworks

	Concept	Available	Language	App. requirements
Calvin [PA15]	Actor, flow based	✓	Python	✓json file
D-NR [Gia+15]	Distributed dataflow	✓	JavaScript	✓parameters in GUI tool
Patricia [Nas+13]	Intent, IntentScope	✗	Java	✓scope properties
PyoT [Azz+14]	T-Res/CoAP, REST	✓last commit 2015	Python	✗
Compose API [Ord+14]	REST, event-driven, stream-oriented	✓last commit 2016	JavaScript	✗
Simurgh [KDB15]	REST	✗	-	✓json file
Dripcast [NHE14]	RPC	✗	Java	✗
IoTSuite [Cha+16]	MQTT, Pub/Sub, dataflow	✓last commit 2017	Java	✓SDL (srijan deployment language)
OpenIoT [KL14]	REST	✓last commit 2015	Java	✗

The reasons behind choosing Calvin as an important part of our architecture come from 2017, when we gathered the tools to create `FITOR`. At that time, we did

an analysis of the most relevant IoT frameworks, comparing their availability, extensibility and their capacity to describe the application requirements.

Table 6.1 summarizes our conducted analysis of the frameworks. It is worth noting that most of them are short term projects, which are not updated frequently. For this reason, we restricted our candidate list to the two most active frameworks: Calvin and D-NR. The main motivations behind our selection of Calvin are the following: i) good documentation and active community, Calvin has a verbose wiki describing the project and its internals; ii) the description of application requirements in a programmatically (json vs. GUI); and iii) programming language, as I feel more comfortable using Python instead of JavaScript.

6.2 Overview

The Calvin framework is the corner stone of `FITOR`'s architecture, as we will see in Chapter 7. A customized version of Calvin is responsible for the implementation of the three main components that allow the execution of IoT applications: Service Descriptor, Service Deployer and Service Scheduler. Due to its key role in `FITOR`'s architecture, we dedicate this chapter to explain it in detail, presenting its strong aspects and limitations that hinder its use in the Fog environment.

Calvin [PA15] is a community project, started by Ericsson, which proposes a framework for the development of IoT applications. Calvin borrows concepts from actor and dataflow models to create a high-level abstraction, and consequently, hides the complexity and heterogeneity of the IoT world. As the authors claim, applications can then be created by simply putting together small building blocks, like a LEGO game.

The abstraction layer proposed by Calvin is possible thanks to four separate concepts which are part of an application:

- Describe: these are the building blocks of an application, small and reusable components, called actors which are responsible for implementing the business logic.

- Connect: describes the connections between the components of an application while indicating how the data circulate among them. The actors alongside their connections create the application graph.

- Deploy: it is the instantiation of the application in the platform, deciding where to place each application component in the infrastructure. It is the role of the Service Deployer in our architecture.

- Manage: once the application is running, it enters in the management phase. This aspect is responsible for the migration of running applications, scaling and error recovery. Precisely, thanks to our Service Scheduler, we can handle the reconfiguration and migration of applications.

6.3 The Actor Model

The actor model [Hew10] is an interesting conceptual model to describe IoT applications independently of the underlying infrastructure. These actors may be implemented later using any technology, such as micro-services. Calvin uses the actor model to describe the functional behavior of applications, which are implemented through Python code.

Hewitt first proposed the actor model back in 1973 [HBS73], as a universal primitive for concurrent computation. In this model, everything is an actor and every computation happens within the actor. This implies that each actor has a private and non-shareable state, communicating with each other through asynchronous messages.

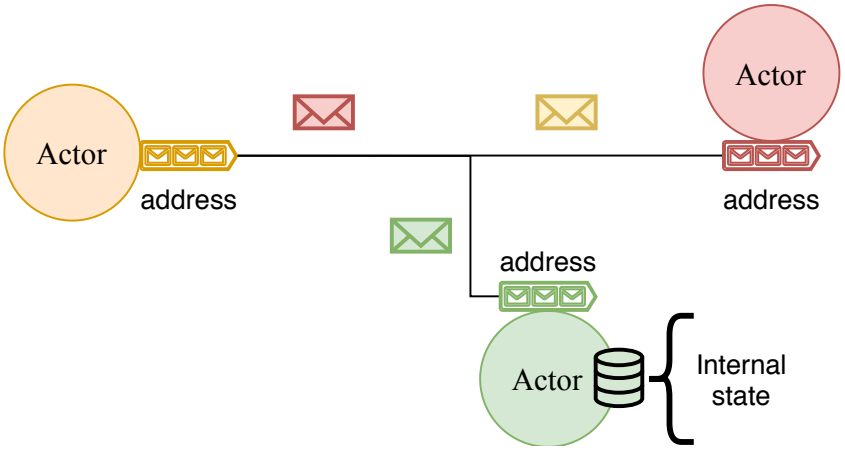


Figure 6.1: The actor model

Fig. 6.1 shows the interaction between three actors in the system. Actors are independent computing units that send asynchronous messages to other actors through known addresses. The address may be implemented in several ways, such as network, e-mail, memory or disk addresses, depending on the context on which actors are running.

When receiving a message, an actor may concurrently perform three actions: i) handle the message and change its internal state; ii) send messages to other actors and iii) create new actors [1]. By decoupling actors from communications, the actor model allows the construction of concurrent systems in a flexible and extensible way. Such a design is particularly useful when programming in a large, distributed and asynchronous Fog environment.

6.4 Architecture

The Calvin's architecture is shown in Fig. 6.2. In the bottom half of the picture, we have the platform dependent part, starting with the host's hardware and operational system (OS). The Calvin process, called runtime, runs directly on top of the OS. The runtime layer is responsible for providing the abstraction needed for running applications. The platform dependent part of the runtime will establish the communication between different runtimes. Note that in our architecture, we add a virtualization layer between the OS and the runtime to make the environment uniform and hence, alleviate the process of installing Calvin's dependencies (e.g., libraries).

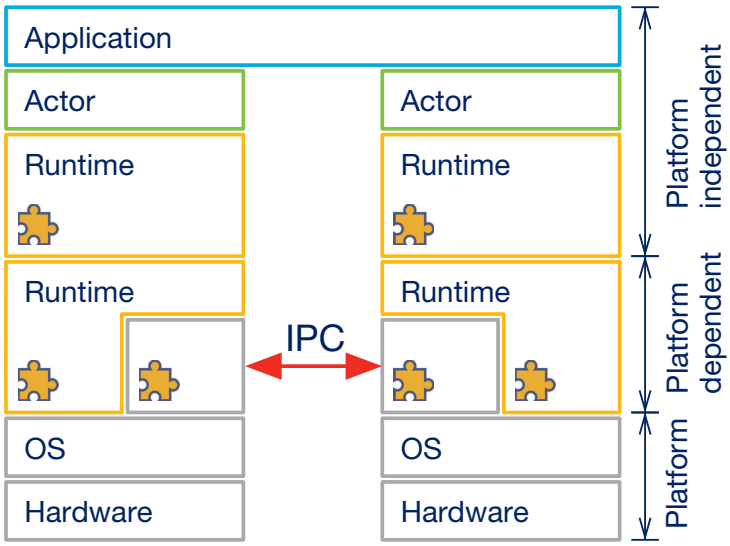


Figure 6.2: Calvin's architecture [[PA15], Figure 1 (right part)].

We focus, in our work, on the top part (and platform independent) of the architecture. The runtime is responsible for controlling the actors, and consequently, the applications running on the host. Both provisioning and reconfiguration of the applications running in the environment are handled by the runtime. Finally, we have the set of applications and their actors. The actors are shared among the differ-

[1] Although envisioned in the actor model, our Calvin's applications do not create new actors when receiving messages.

ent Calvin's runtimes available in the environment, according to the provisioning policy implemented. To enable the targeted orchestration of the IoT applications addressed within the framework of this thesis, we had to create our IoT applications using the Calvin model, implement the necessary actors and modify the runtime to put into action our provisioning and reconfiguration strategies.

6.5 Describing an Application

As previously discussed, Calvin allows the development of IoT application by bringing together the building blocks, called actors. However, it is necessary to implement formerly the actors that will compose the application. In general, describing one application involves three steps: i) the implementation of actors, ii) building the application and iii) describing its requirements. In the following sections, we describe each of these steps.

6.5.1 Actor development

An actor is simply a piece of python code which reacts to some event producing some output. By keeping it simple, we increase its reusability in different applications. To illustrate the creation process of an actor, we present the snippet of code used to implement one of the actors that we will use in our applications later: a Burn actor.

The idea of Burn is simple, as illustrated in Fig. 6.3, it receives one message (or token), perform some processing over it and forward it to next actor through the output port.

Figure 6.3: Burn actor. It receives messages through the input token port in the left, process it, and send it unchanged through the output token port in the right.

This high-level description of the actor is then implemented via python code, as presented in 6.1. All actors in Calvin inherit from the Actor class which defines the methods that can be implemented. These methods control the initialization, migration, replication and finalization of an actor[2].

[2]The reader is invited to access the wiki of Calvin project for a more complete documentation about actor interface. Available at: https://github.com/EricssonResearch/calvin-base/wiki/Actors.

```
1  from calvin.actor.actor import Actor, condition

3  class Burn(Actor):
       def init(self):
5          pass

7      @condition(action_input=['token'], action_output=['token'])
       def processMsg(self, input):
9          # Compute something
           return (input, )
11
       action_priority = (processMsg, )
```

Listing 6.1: Python code implementing a simplified version of Burn actor.

For our Burn actor, we define only two methods: `init` and `processMsg`. As the name suggests, `init` is called to perform the initialization of the actor, setting internal variables if necessary. On the other hand, `processMsg` is called when one token needs to be processed. It is the responsibility of Calvin's runtime to call the appropriate method when a message is received in the input port (named *token)*. Calvin relies on decorators on python code to configure this behavior. In our example, the decorator *@condition*, at line 7, indicates that the method `processMsg` must be called when the input port has received a message and the output port has space to store the output message.

6.5.2 Building an application

Once all the necessary actors for an application are implemented, we can proceed with the creation of the application itself. To encourage the sharing of actors between applications, Calvin has the concept of "Actor store", where actors are stored and shared. The store categorizes the actors into modules, such as media, math, IO, etc., to make easier their retrieval.

Building an application is easy; we just need to connect the output ports to the input ports of different actors. To do so, Calvin offers two possibilities: graphical or textual.

Fig. 6.4 presents the graphical interface to create applications. On the left, we have the Calvin's actor store, from where we can select and drag-in the actor to the main screen. In the figure, we have implemented a 3-level application, in which the Trigger actor periodically sends messages to be processed by the Burn. The latter, after finished its computation, forwards the message to be stored by the Sink actor[3].

[3]Note that this same application structure, with some variations, is used throughout the thesis.

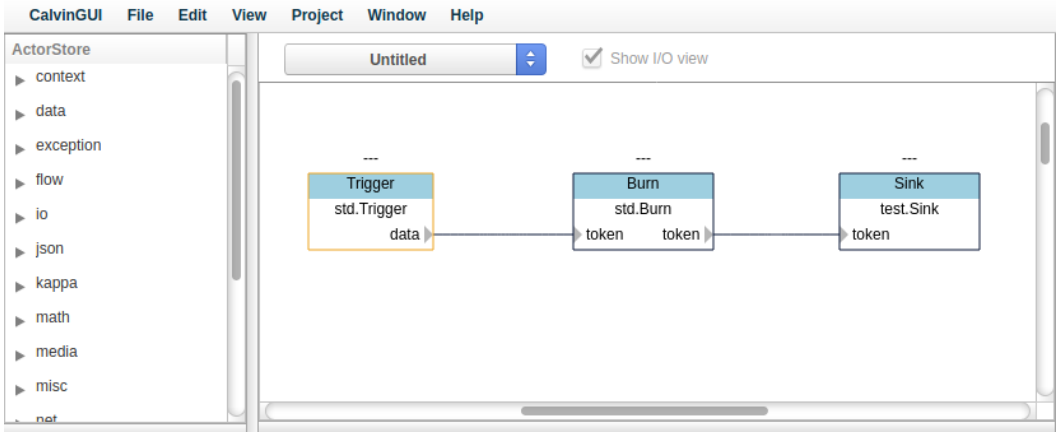


Figure 6.4: Snapshot of Calvin's graphical user interface used to implement an application example.

The graphical interface is nice for creating and visualizing small applications. However, as the number of actors and applications increases, we need a more programmatic manner to describe the applications. In this context, Calvin uses a proprietary language, called CalvinScript, to describe the applications[4].

```
Sink : test.Sink(active=true, quiet=false, store_tokens=false, threshold=5)
Trigger : std.Trigger(data="test", tick=1)
Burn : std.Burn(dump=false, duration=0.1)

Trigger.data > Burn.token
Burn.token > Sink.token
```

Listing 6.2: 3-level application using the Calvin's syntax. It implements the same application presented in Fig. 6.4

In Listing 6.2, we implement the same 3-level application presented in Fig. 6.4. The first three lines describe the actors and their parameters (this information is necessary to correctly initialize the actors). The last two lines use the ">" connector to link the respective input and output ports of the actors.

6.5.3 Requirements

Finally, the last step, but vital for IoT applications, corresponds to the description of the requirements for the actors in an application. In [AP17], the authors propose the mechanisms to describe the requirements for IoT applications, which are implemented in the Calvin framework. The objective is to handle the complexity and heterogeneity of large IoT systems automatically, delegating to the runtimes the task of managing and selecting the nodes to run the actors. To do so, applications

[4]We refer to wiki for a complete description of the CalvinScript's syntax. Available at: https://github.com/EricssonResearch/calvin-base/wiki/Applications.

supply the set of necessary requirements, which are matched against the set of capabilities the runtimes have.

The requirements described by developers are hard, i.e., applications cannot run if they cannot be met. These requirements are divided into two main groups: actor and application related. The former group, **actor-related prerequisites**, is composed by the requirements inherent to the actors themselves, such as the access to a camera, the presence of a timer or the ability to measure the temperature. Without these minimum capabilities, actors cannot execute and perform their actions. The latter group, **application-related prerequisites**, is not directly linked to a single actor and may vary from one application to another. These **requirements** include properties such as **geographical location**, **node name** and **ownership**. Note that we have little room for improvements with the actor-related requirements, since they depend on the physical access to sensors and actuators. Therefore, within the framework of this thesis, we will focus on the application-related requirements, considering the specific needs of IoT applications running in the Fog environment.

Calvin uses a json-based syntax to describe the requirements, as illustrated in 6.3. The language, although somewhat verbose, allows the specification of requirements for each actor. In our example, the Sink may be located on any node in Paris, France. Note that the attributes that describe the requirements, the address in this case, are hierarchical, i.e., specifying only the *country* argument will refer to all nodes within this country (France in our example), regardless of the city. Moreover, it is possible to be more specific when describing an address, adding, for example the street, street number or building [5].

```
{
    "requirements": {
        "Sink": [{"op": "node_attr_match",
                  "kwargs": {"index": ["address", {"country": "FR",
                                                   "locality": "Paris"}]},
                  "type": "+"
                 }]
    }
}
```

Listing 6.3: Describing the requirements for a Calvin application.

[5] Once again, we refer to wiki for a complete description of the possible requirements and their sub-levels. Available at: https://github.com/EricssonResearch/calvin-base/wiki/Application-Deployment-Requirement.

6.6 Deploying an application

With the description of the application and its requirements available, a user can request to deploy her application. Fig. 6.5 presents a simplified scenario that contains two users deploying their applications in an environment composed of three runtimes. Some points are important to highlight: i) **any runtime can deploy** applications in the system: in this case, user 1 deploys her application through runtime 2, while user 2 uses runtime 3; ii) shared database, containing information about available runtimes and their capabilities.

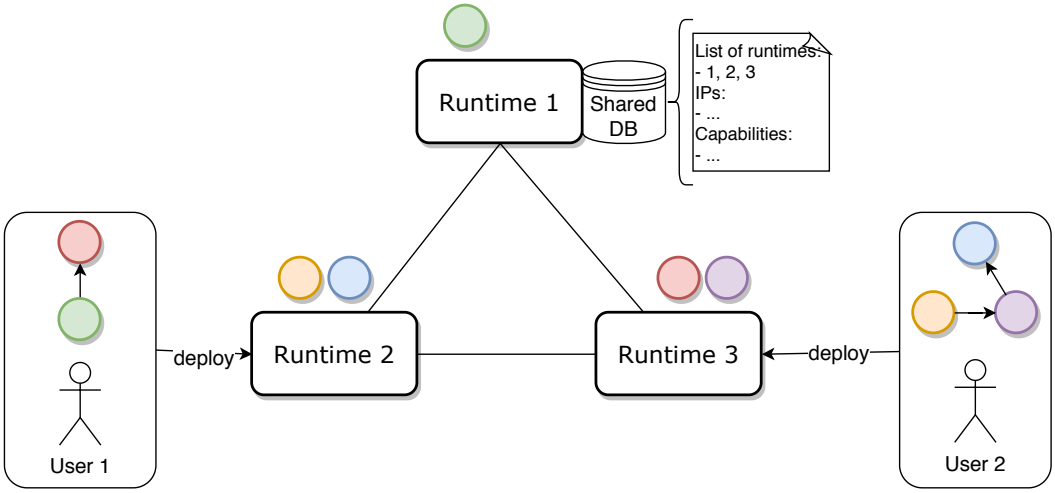


Figure 6.5: Calvin's deployment ecosystem. Runtimes deploy applications from different users concurrently. A database is used to share information about runtimes and their capabilities.

Each runtime deploys applications by itself, regardless of other applications and runtimes on the system. The deployment algorithm filters the available runtimes based on the requirements of the application. Calvin's current deployment algorithm has two main phases:

1. **Collecting** actor's placement: by considering the current information in the shared database, Calvin creates, for each actor, a list of possible runtimes to host it. The list contains only hosts capable of meeting the actor's requirements.

2. **Deciding** runtime: one runtime from the list is chosen to host each actor.

Note that Calvin arbitrarily chooses the runtime to host the actor and no optimization is performed at step 2. After that, the runtime, responsible for the deployment, forwards the actor to the appropriate runtime.

6.7 Limitations for the Fog

We have seen in this chapter how Calvin leverages the development of IoT applications, using the actor model while creating an abstraction layer on which the actors execute. Despite its advantages and interesting concepts, some limitations prevent its direct use in a Fog environment. It is important to remember that Calvin was created with a Cloud environment in mind, where, in general, all nodes are powerful, uniform and connected through a dedicated, high-performance network. In this context, the capacity of the resources, such as CPU, RAM or network bandwidth, is not a limiting factor. However, on a Fog platform, the capacity of the infrastructure nodes greatly varies due to their heterogeneity, and therefore, these characteristics cannot be ignored when describing and deploying an application.

We summarize hereinafter the three main classes of limitations identified:

- **Application description**: as we saw above, the requirements available are related to fixed properties of nodes, such as geographical location. Thus, a developer is unable to describe the resources that the application needs, neither in terms of node capacity, such as CPU and RAM, nor network capacity, such as bandwidth and latency.

- **Monitoring**: to provide these dynamic requirements, Calvin needs to be aware of the capacity and availability of the resources on which it is running. So far, Calvin has no information about the host and its resources. Therefore, a monitoring mechanism is necessary to know the platform and to deploy the applications properly.

- **Deployment**: finally, the deployment algorithm needs to be adjusted to take into account these dynamic requirements when placing applications. Moreover, due to the constrained resources of the Mist layer, it is important to have an optimized use of the resources to avoid possible bottlenecks for the applications.

7 FITOR: A Platform for IoT Orchestration in the Fog

Our main goal behind the implementation of FITOR is to create a proper platform to run, monitor and collect data about IoT applications executing in a Fog environment. Three objectives drive the platform we propose: i) perform studies about the provisioning and the reconfiguration of IoT applications in the Fog; ii) not reinventing the wheel, since many of the software needed to create the platform already exist; and iii) similar to real environments, mimicking the main characteristics of the Fog infrastructure.

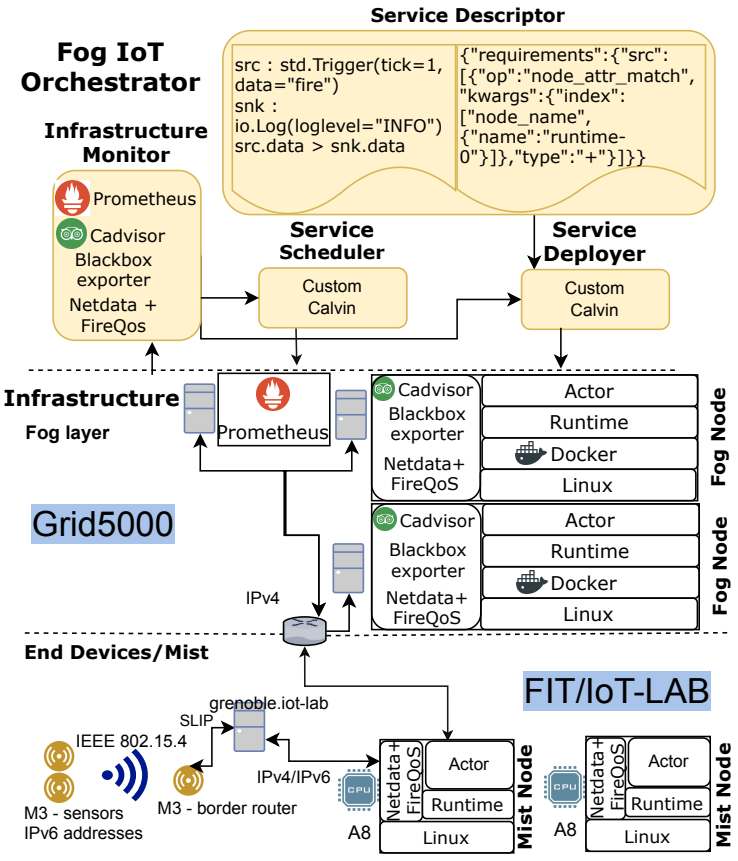


Figure 7.1: FITOR platform. This figure depicts the components used to build the architecture proposed in Fig. 5.1.

Fig. 7.1 takes up the architecture proposed in Chapter 5, presenting the components used to create the platform. In the following sections, we detail each one of these components, explaining the function they cover in the environment.

7.1 Software Components

7.1.1 Calvin

In Chapter 6, we have seen how Calvin can be used to run IoT applications and its limitations in a Fog environment. To overcome these limitations, we implemented a set of extensions in the original Calvin framework, creating the fog-friendly version used in FITOR.

Extensions in application description

Calvin's language, described in Section 6.5.2, is quite powerful and provides almost everything needed to describe a Fog application. The exception is a way to identify application links. The original language proposes the use of the ">" identifier to link two ports from different actors. However, in order to be able to define network requirements for application links, we need to give the links a specific name, as it is done for actors (e.g., Sink is the name of `test.Sink` actor in 7.1).

To overcome this limitation, we extended the Calvin's grammar to include the link name before its definition, as:
link <link name> : <actor1>.<port> > <actor2>.<port>.
The syntax is similar to the old one, we only inserted the keyword `link` before the name and the delimiter ":" after it. In 7.1, we show how to give names for the links in our application example.

```
1 Sink : test.Sink(active=true, quiet=false, store_tokens=false, threshold=5)
  Trigger : std.Trigger(data="test", tick=1)
3 Burn : std.Burn(dump=false, duration=0.1)

5 link linkA : Trigger.data > Burn.token
  link linkB : Burn.token > Sink.token
```

Listing 7.1: Grammar extension to name links in Calvin applications. It implements the same application presented in Fig. 6.4

Extensions in requirements

Requirements are the Achilles' heel when using Calvin to describe Fog-enabled IoT applications. The lack of support for dynamic parameters, such as IT and network resources, hinders its use in the Fog environment. For this reason, we have extended the model to consider nodes and links requirements. We recall that,

internally, Calvin uses a hierarchical structure to organize the nodes that meet the requirement. We will use this organization in some of the requirements below to sort nodes from most to least restrictive.

```
{
    "requirements": {
        "Sink": [{"op": "node_attr_match",
                  "kwargs": {"index": {"cpuRaw": 4, "ramRaw": 1000 }},
                  "type": "+"
                }],
        "linkA": [{"op": "link_attr_match",
                  "kwargs": {"index": {"latency": "100ms", "bandwidth": "10M"}},
                  "type": "+"
                }],

    }
}
```

Listing 7.2: Adding CPU, RAM and network-related requirements in a Calvin application.

- **Node** requirements: within the `node_attr_match` operation, which matches nodes that respect certain characteristics, we added:
  - *cpuTotal*: filters nodes with at least a certain CPU power, independent of their current load. The nodes are categorized in groups to facilitate their retrieval. The acceptable values are: 1 MIPS, 1000 MIPS, 100000 MIPS, 1 GIPS, 10 GIPS.
  - *cpuRaw*: considers the available residual CPU power (in MIPS), i.e., node's total power minus its current load.
  - *memTotal*: similarly to *cpuTotal*, but regarding the total RAM in node. The acceptable values are: 1 Kb, 100 KB, 1 MB, 100 MB, 1 GB and 10 GB.
  - *ramRaw*: the minimum residual RAM memory (in bytes) necessary to run the application.
- **Link** requirements: a new operator, `link_attr_match`, was created specifically to match the links. The following requirements are possible:
  - *bandwidth*: represents the minimum bandwidth acceptable for the link. It follows the hierarchical organization used for *cpuTotal* and *memTotal*, the acceptable values are: 100 Kb/s, 1 Mb/s, 10 Mb/s, 100 Mb/s, 1 Gb/s.
  - *latency*: as expected, determines the maximum acceptable delay between two actors. The possible values are: 1 s, 50 ms, 10 ms, 1 ms, 100 us.

Monitoring

Unfortunately, the description of applications with dynamic requirements is only the visible part of the iceberg. To filter out nodes following the residual resources, Calvin needs to have this information up-to-dated. We opt for separating the retrieval of the information from its update. The monitoring is handled by external tools, as explained in Section 7.1.2, while in Calvin, we implemented a REST API that external tools can use to update Calvin's internal database.

- **Node**-related parameters: API used to set CPU and RAM available for the node hosting the Calvin instance. Both parameters are in percentage (%) of the total resource.

```
POST http://<IP>:<PORT>/node/resource/cpuAvail
Body: {"value": <CPU available (in %)>}

POST http://<IP>:<PORT>/node/resource/memAvail
Body: {"value": <RAM available (in %)>}
```

Listing 7.3: Rest API to set the host's CPU and RAM availability.

- **Link**-related parameters: bandwidth and latency of links are related to the end-to-end measurements between two nodes. So, the API requires the identifier of the two nodes, as well as the new bandwidth or latency values.

```
POST http://<IP>:<PORT>/link/resource/bandwitdth/{node1}/{node2}
Body: {"value": <bandwidth available>}

POST http://<IP>:<PORT>/link/resource/latency/{node1}/{node2}
Body: {"value": <latency>}
```

Listing 7.4: Rest API for setting the bandwidth and latency of a link.

Provisioning and reconfiguration

Finally, the latest improvements brought to Calvin are intrinsic to our target, namely the provisioning and reconfiguration algorithms for IoT applications. These algorithms take into account the new requirements described in this chapter and will be detailed in Part III and IV.

7.1.2 Monitoring

As presented in Fig. 7.1, our platform is composed of several components with specific characteristics and, consequently, different manners to be monitored. To deal with this heterogeneity, we use different tools to monitor each component of the platform. Our goal herein is to collect as much information as possible with the least effort, even though we may not use all the data collected. Note that, consequently, we have not evaluated the possible impact of these monitoring tools on the system, which is not negligible, but it is outside the scope of this thesis.

cAdvisor

CAdvisor [Goo14] provides a complete monitoring of performance and resource usage of containers. Although cAdvisor's primary goal is the monitoring of containers, it also collects metrics system-wide. Among all the resources monitored, we are mostly interested in the CPU and RAM utilized by containers running Calvin runtimes.

In our platform, cAdvisor runs as a separate container in all Fog nodes. It monitors and exports the relevant metrics through a REST API interface which are collected by our Prometheus server.

Blackbox Exporter

Blackbox exporter [Pro15] is a tool developed by the Prometheus team to monitors different endpoints. In our context, we use it to measure the end-to-end latency between the nodes in the system. However, its scope is much broader, as it allows the probing of endpoints via many protocols, such as HTTP, HTTPS, DNS, TCP and ICMP.

We deploy and configure the Blackbox exporter probes in all Fog and Mist nodes of our infrastructure. On each node, we set all other nodes as the destination for ICMP requests, so that we can measure the complete end-to-end delay. The results are exported via a known address, which are read by Prometheus and forwarded to Calvin runtimes.

Netdata and FireQoS

Netdata [Net13] is a versatile monitoring and visualizing tool for a variety of systems. It is designed to be a fully integrated tool, including the monitoring, storage, verification and visualization of the collected metrics. Among its features, it allows the export of collected metrics to an external server, such as Prometheus. On the other hand, FireQoS [TW02] is a helper tool to configure traffic shaping on Linux machines. It provides a user-friendly abstraction for setting the Traffic Control (TC) module in the Linux kernel.

In our infrastructure, Netdata is responsible for collecting all metrics related to physical nodes. Tons of metrics are collected, but we are mainly interested on the CPU and RAM consumption of the nodes. Thanks to its low memory footprint, Netdata fits into our Mist nodes and can be used to monitor their resource consumption.

Moreover, these tools, Netdata and FireQoS, are used to monitor the bandwidth utilization without the support of network equipment. In our setup, these tools measure the bandwidth utilization of each Calvin's flow. Note that the flow is characterized by a TCP connection between two nodes. Using FireQoS, we configure the TC to identify each flow and to keep statistics about them. Then, Netdata collects the information and sends it to our infrastructure monitor.

Prometheus

Finally, Prometheus [Pro12] is an open-source monitoring and alerting toolkit. However, the Prometheus server itself does not directly monitor any system, relying on different exporting modules to generate and export these metrics. Prometheus subsequently collects and processes these metrics, providing a flexible query language to filter and organize the data.

One node in our infrastructure hosts the Prometheus server, which scraps all other monitoring tools (Netdata, cAdvisor, blackbox exporter) to retrieve and store metrics during our experiments. By saving the Prometheus database in the end of the tests, we are able to perform post-mortem analyzes of the system's performance.

7.1.3 Docker

Docker [Doc13] is probably the most adopted product for packing software, using the OS-level virtualization called containers. The use of containers versus virtual

machines as virtualization technique in the Fog is an open question, which depends on the environment (size, heterogeneity) and application requirements (security, scalability). However, we believe that containers, due to their lightweight, are better adapted to isolate micro-services running in a heterogeneous and sometimes constrained environment, such as the Fog. Thus, in our platform, the Docker engine copes with hardware and software heterogeneity. Docker containers are used to create an image which encapsulates the software necessary in our setup.

7.2 Infrastructure

7.2.1 Grid'5000

Grid'5000 [Bal+13] is an experimentation testbed whose focus is on parallel and distributed computing, such as Cloud and Big Data. Grid5000 contains a large amount of powerful resources, 841 nodes grouped in 37 homogeneous clusters and spread in 8 sites [Gri]. It is a highly controllable and configurable platform, allowing the installation of a fully customized environment, from the OS to the applications.

We make use of Grid5000 servers to emulate the Fog nodes and run our Prometheus server. We can distinguish three types of nodes in Grid'5000:

- 1 node for the Prometheus server: this node is the entry point for our tests and where we install our monitoring system.
- 1 node for the master Calvin runtime: the master node that contains the Calvin database.
- N Fog nodes running slave Calvin runtimes: where the IoT applications are executed.

7.2.2 FIT/IoT-LAB

FIT/IoT-LAB [Adj+15] is a large scale open platform to perform IoT experiments. It provides more than 1786 heterogeneous sensor nodes spread in 6 different sites in France [FIT]. IoT-LAB has a set of official boards (WSN430, M3 and A8), specially designed for the platform, but users can bring new off-the-shelf boards. Although limited, some robots are available to mimic mobility, by running a closed circuit route.

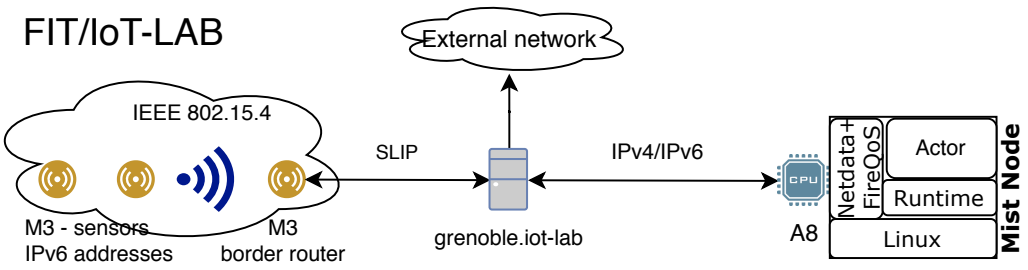


Figure 7.2: Zooming in on the FIT/IoT-LAB part of the `FITOR` infrastructure.

In our prototype, we use M3 and A8 boards to set up the end device layer and the Mist sublayer, encompassing the sensors and Mist nodes. Each M3 board is composed of an ARM micro-controller, 64KB of RAM, various sensors (e.g., ambient sensor light, atmospheric pressure and temperature, etc.) and an IEEE 802.15.4 radio interface. As we can see in the left part of Fig. 7.2, M3 nodes communicate via the radio interface and an M3 border router encapsulates and sends the IP packets to the frontend node over the serial interface, using the SLIP (Serial Line Internet Protocol) protocol. Regarding the software stack, we use the Contiki-NG operation system on M3 nodes. A version for M3 is provided by FIT/IoT-LAB maintainers[1] and allows the access to the sensor data using its IPv6 address and the REST-inspired, CoAP (Constrained Application Protocol) protocol.

On the other hand, A8 nodes are more potent. Thanks to its ARM A8 microprocessor and 256MB of RAM, it is capable of running user's applications and hence, can be considered as a Mist node. It has a wired connection and a wireless via a built-in M3 board, allowing both IPv4 and IPv6 access to nodes.

7.2.3 Connectivity

Connectivity is the main limitation in the use of Grid'5000 and FIT/IoT-LAB. As two separate testbeds, the nodes are part of private networks and cannot communicate directly. Fortunately, Grid'5000 has a VPN solution to connect your PC to the Grid'5000 network. Thus, we use this feature to overcome this connectivity limitation by creating L3 VPNs between the A8 and Grid5000 nodes. For this, we install the OpenVPN [Ope01] client with the appropriate configuration file on each A8 node on the FIT/IoT-LAB platform.

7.3 Limitations

[1] More details about Contiki support on IoT-LAB is available at: https://github.com/iot-lab/iot-lab/wiki/Contiki-support.

7.3.1 Applications

The improvements we have made to Calvin to describe the application requirements serve as a proof of concept of its use in a Fog environment. However, it is straightforward to see that these requirements (CPU, RAM and network), although relevant, are not exhaustive for both application actors and links.

For the actors, we can consider other resources in the description, such as storage, specific hardware (GPU or FPGA for example) or software (e.g., libraries). Additionally, applications may have requirements related to energy, security or performance, restricting the nodes that could run these components.

Following the same idea, link requirements can be extended to consider jitter or congestion in a network, for example. Also, monetary aspects can be important to avoid extra cost incurred by some technologies (e.g., cellular networks).

7.3.2 Monitoring

We use a variety of monitoring tools in our environment. As discussed earlier, these tools have an impact on system performance, which can be important. Although we do not take this impact into account, we expect that the latter will the similar for all the orchestration strategies studied and, hence, our conclusions are still valid.

More relevant, however, are the consequences that these monitoring tools bring to the system: delayed and imprecise information. The data depends on the frequency of the monitoring and the updates we receive. These parameters can be configured but they also consume resources and may further impact application performance. In addition, precise information depends on the accuracy of the monitoring tools. This is particularly problematic for the bandwidth on our shared network, because we monitor the bandwidth used by flows and not the residual network bandwidth.

We consider these "limitations" as part of the game, since in a real environment it is impossible to have accurate and up-to-dated monitoring information for a large, distributed system, such as the Fog.

7.3.3 Infrastructure

The platform we propose is composed of Fog, Mist and end-devices layers. However, it is worth noting that, for sake of simplicity, we don't emulate the Cloud layer. However, such a layer could be instantiated using other data centers of Grid'5000.

The Fog infrastructure is also characterized by its heterogeneity and geo-distribution, using all available resources to run applications. However, our Fog nodes are generally composed of machines from homogeneous clusters, which reduce heterogeneity since they are located close to each other. A more diverse set of resources is possible by creating different profiles and limiting the resources available for applications. However, the geographic distribution is slightly more complicated, depending on the use of different sites and/or the fine-tuning the network parameters to emulate this characteristic.

Finally, it is important to note the limitations in terms of network resources in `FITOR`, especially the use of a VPN to connect the FIT/IoT-LAB and Grid'5000 testbeds. In this regard, the SILECS project [SIL] was created to bring together several testbeds (including Grid'5000 and FIT/IoT-LAB) and create a whole infrastructure, from connected objects to large data centers. However, up to this date, the project is under development and is not yet ready for use.

8 Experimental Methodology

In this chapter we detail the experimental methodology used in our tests during this thesis. A high level overview of all factors, which can affect the results and must be carefully configured, is provided in Fig. 8.1. These parameters are discussed in the following sections and their values are defined later, when provisioning and reconfiguration studies are presented.

8.1 Scenario

8.1.1 Platform

The platform describes the physical infrastructure used in the experiments. It is characterized by its:

- **Size**: it is the total number of nodes present in the platform.
- **Heterogeneity**: represents the diversity (sensors, Mist and Fog nodes) present in the Fog infrastructure.

In practice, these parameters are translated to a given number of nodes from both testbeds: Grid'5000 and FIT/IoT-LAB. For example, a typical scenario contains 50 nodes from FIT/IoT-LAB to represent the sensors/Mist layers, and 17 nodes from Grid'5000, forming the Fog layer.

8.1.2 Workload

The workload describes the applications running in our experimental Fog environment, their characteristics and the load they incur in the system. For clarity purposes, we start by seeing the high level characteristics of the workload which are enough to understand the behavior and load of the applications. Afterwards, some specific, low level parameters necessary for a complete application description are presented.

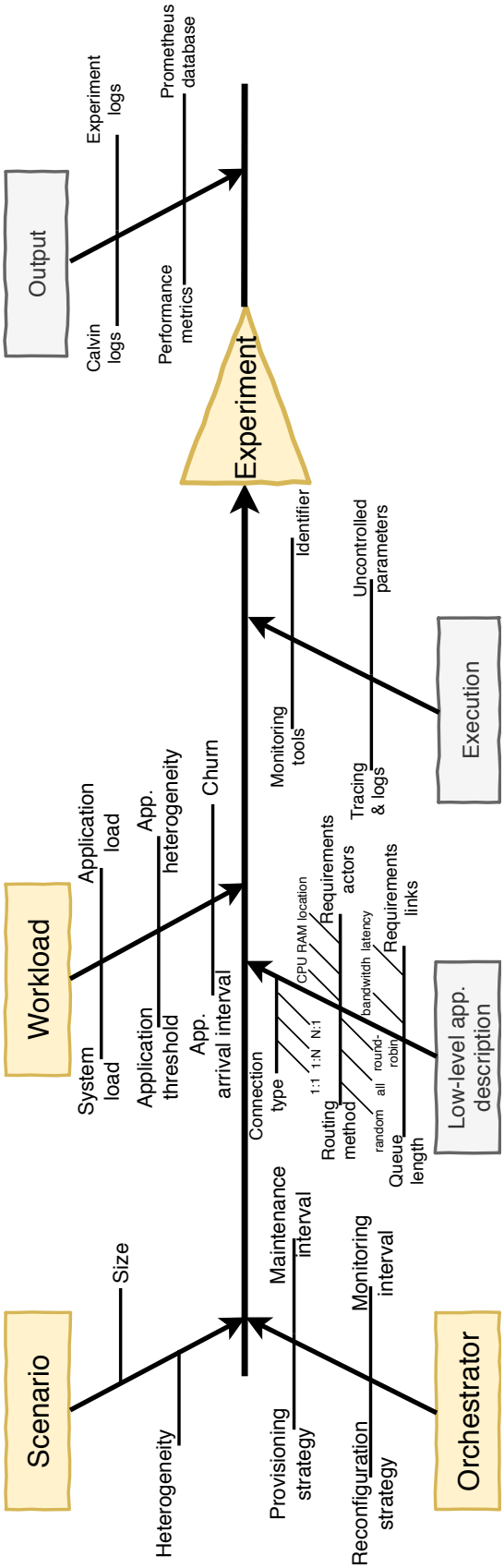


Figure 8.1: Fishbone diagram showing the factors that can impact our experiments. We highlight in orange the main parameters that drive the test execution. The gray parameters may impact, but they are not the focus of our study.

The first step when defining the workload for our experiments is deciding the application we will use. We opt for a 3-level application, as illustrated in Fig. 8.2. In the first level, sensors, or Trigger agents, generate the data for the remaining actors of the application. The data is generated respecting the workload description detailed below. Second, the Burn is responsible for processing the received data, consuming a high amount of resources, mainly CPU. In the end, the Sink is responsible for the long-term data analytics and for measuring the end-to-end delay to process the messages. This type of model, albeit simplified, reflects the main characteristics of an IoT application and facilitates the workload customization. We note that defining a realistic workload is probably the most delicate and debatable part of such study. With the ongoing evolution of Fog and IoT applications, we are unaware of a comprehensive use case, containing a well detailed description of the behavior of IoT applications running on the Fog. Consequently, some parameters cited hereafter are cautiously configured, considering our infrastructure, to obtain a heterogeneous, complex but still manageable workload.

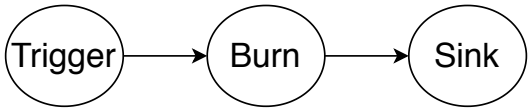


Figure 8.2: A typical 3-level application used in the experiments. The number of actors in each level and their characteristics vary according to the test, but the structure is kept.

Fig. 8.2 illustrates the components of a single application. The workload is composed of a set of application and is characterized by:

- **Application load**: different profiles are created to describe the application's resource consumption. For example, by controlling the message sending rate of the Trigger and the processing load of the Burn agent for each message, we can adjust the behavior, in terms of CPU consumption, of the applications in the workload.

- **Application heterogeneity**: is the mix of applications present in the workload, according to the application load defined before.

- **System Load**: represents the charge induced by applications to the platform. This parameter is correlated with the platform size, application load and heterogeneity. In practical terms, it determines the number of applications in the experiment.

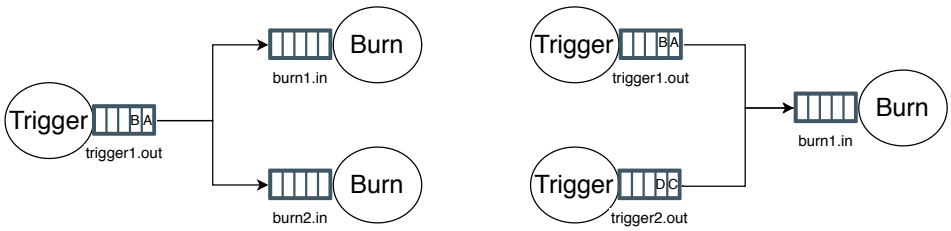
- Application **arrival interval**: defines how the applications enter in the system during the provisioning phase.

- **Application threshold**: the acceptable end-to-end delay to process application messages, i.e., the time taken by the message to leave the Trigger and to arrive at Sink component. This parameter is important to determine the satisfaction of applications with their current placement.

- **Churn**: defines the evolution over time of applications and consequently their resource usage. This parameter is important to study the reconfiguration problem.

Low-level application description

In addition to the high-level parameters described in the previous sections, several other attributes need to be fixed to have a proper experimental environment. These parameters do not have a direct impact on the comparison of different provisioning and reconfiguration strategies but they may impact on the overall performance of the system.



(a) 1-N connection. **(b)** N-1 connection.

Figure 8.3: Internal representation of ports in actors. Each port has a queue of limited size associated. Extra configurations are available to dictate how messages are sent between source and target actors.

Fig. 8.3 illustrates the parameters related to the actors and their connections:

- **Connection type**: describes how actors from different layers are physically connected. In our case Trigger $\rightarrow$ Burn and Burn $\rightarrow$ Sink.

  - 1:1: each actor is connected to one in top layer, as in Fig. 8.2.

  - 1:N: one actor is connected to all actors in top layer (Fig. 8.3a).

  - N:1: one actor aggregates token from many input actors (Fig. 8.3b).

- **Queue length**: it is the maximum number of messages that can be stored in each queue. In all our experiments, the queue has *10 slots* available.

- **Routing method**: defines how messages are forwarded from one actor to another, such as:

  - Round-robin: route tokens in a round-robin schedule. For the case described in 8.3a, it would result in the token "A" going to burn1.in and "B" to burn2.in.

  - *Default/all*: sends all tokens to all actors. This is the default parameter chosen in our experiments. So, both "A" and "B" would be sent to burn1.in and burn2.in.

  - Random: as the name implies, it forwards messages at random to the actors.

Finally, as we have seen before, IoT applications running in the Fog may have specific requirements in terms of IT resources and network. The following parameters can be described:

- Requirements for **links**: for each link in the application, requirements in terms of maximum allowed *latency* and minimum *bandwidth* can be described.

- Requirements for **actors**: describes the necessary resources of the host on which the actor is running, such as CPU and RAM availability, location, etc.

8.1.3 Orchestrator parameters

Besides the description of platform and workload made above, the orchestrator itself has several parameters that may impact in the experiments:

- **Provisioning strategy**: corresponds to the algorithm used in the provisioning phase to decide the initial deployment of applications. Several provisioning strategies are studied in Chapters 10 and 11.

- **Reconfiguration strategy**: describes the algorithm used in the reconfiguration phase to adapt the placement of applications when necessary. This parameter is analyzed in Chapters 13 and 14.

- **Maintenance interval**: defines the time frequency at which the orchestrator will verify the satisfaction of applications and will run the reconfiguration algorithm. The responsiveness of algorithms depends on this parameter. By default, we choose a *5 seconds* maintenance interval.

- **Monitoring interval**: controls the update of host information about resource usage, such as CPU utilization. In our experiments, this parameter is configured to *60 seconds* interval, which means that load information may be up to 1 minute out-dated. This is left intentionally high to obtain a difficult setup.

- **Orchestrator version**: represents the Calvin's version used in the tests.

8.1.4 Uncontrolled factors

For the sake of completeness, we cite some uncontrolled factors which may result in instabilities during the tests. Since we are using a real platform for our tests, the list of uncontrolled factors is probably much longer. However, we believe that these examples can give a good understanding of the different challenges we faced while running tests in the proposed environment.

The first source of instabilities is the software stack running on machines. In Grid'5000 nodes, this potential effect is minimized by reinstalling all software, from the operational system to the tools used during the experiments. In FIT/IoT-LAB, as the customization is more complicated and time-consuming, we chose to install only the necessary software on top of the provided operational system. However, note that the same image is used on all FIT/IoT-LAB nodes.

More specifically, the script responsible for monitoring the node's resource consumption and updating it in Calvin has an impact in applications performance. Each time this information is sent to Calvin, applications stop running for a short amount of time (Calvin is a mono-thread process). The effect can be somehow controlled through the *monitoring interval* parameter, but the effect still exists. For instance, we noticed that a monitoring interval of 5 seconds degrades the applications performance, mainly because of the weak CPU in Mist nodes (A8 boards in FIT/IoT-LAB).

The machines themselves can be a source of uncertainty. Both platforms, Grid'5000 and FIT/IoT-LAB, are shared between different users. Consequently, the nodes used in different test runs are not same. However, we always assured to have similar machines in order to obtain a fair comparison among the strategies.

Finally, the network behavior is unpredictable and can lead to connectivity issues and a delayed communication between nodes on the platform. In this context, the VPN used to communicate FIT/IoT-LAB and Grid'5000 nodes may have a considerable impact which is very hard to control.

8.2 Setup

The setup of our experimental platform is basically the deployment of all components in FITOR (as seen in Chapter 7). For that, we use nodes from Grid'5000 and FIT/IoT-LAB platforms. The location and number of nodes used differ in each section and are commented on the respective chapters.

It is evident that a manual installation and configuration of FITOR in all nodes in the platform is impracticable. Hence, we use a set of automated processes to deploy the experimental environment. Even with this automated process, the setup of an environment composed of about 65 nodes can take up to 2 hours.

The test's control flow is done by python and bash scripts in the front-end of the Grid'5000 platform. They are responsible for generating the test cases, installing the nodes in the Grid'5000 and FIT/IoT-LAB platform, running the test and collecting the results. For Grid'5000 nodes, we use Ansible [Red12] scripts to install and configure all nodes in parallel. Ansible is an IT automation platform which install and configures the nodes through ssh connections. On the other hand, due to the lack of support for Ansible in FIT/IoT-LAB platform, the deployment of nodes is done using bash scripts along with the specific tools provided by the platform.

8.3 Experiments

Given the specified scenario and after the setup of the platform done, we are able to execute the experiments, collect and analyze the results.

8.3.1 Execution

All the parameters described in the sections above are then translated to a set of csv (comma-separated values) files which are read by the script that controls the test execution. In Listing 8.1, we list the files used in a reconfiguration experiment, along with an example of a main configuration file and its parameters. Using this information, the execution script is capable of generating the applications, deploying them and collecting all the relevant logs. The test run and the collected data are identified by a unique 32 character hexadecimal string.

```
tree exp20_ef/
|-- [  95]  ew_param.csv
|-- [ 194]  experiments_list.csv
|-- [ 266]  wl_app_char.csv
|-- [ 118]  wl_churn.csv
|-- [  99]  wl_heter.csv
`-- [  50]  wl_load.csv

cat exp20_ef/experiments_list.csv
Exp_duration,A8_nodes,G5K_nodes,WL_load,WL_heter,WL_churn,Calvin_version,
    Calvin_redeploy,Calvin_deploy,Calvin_maint_interval,Calvin_mig_cooldown
7200,50,17,H,I,S,c1e03e4,app_learn_v2,fixed,10,0
```

Listing 8.1: Structure of an experiment folder. Each `csv` file describes some characteristics of the test. In "experiments_list" file presents the main parameters for the experiment, such as its duration (2 hours in this example).

8.3.2 Output

In order to have a good understanding of our system's behavior in each experiment, a large amount of logs is collected. Listing 8.2 shows the typical output directory tree generated by an experiment. Between the files, a human-readable output file is created, centralizing all important logs (output*.org below). Note that, depending on the setup configuration, these logs can reach 1.5GB in size for each test run.

```
tree 8c4e57ed-ce23-4a06-9d35-bfadbfc75f6f/
|-- [155K]  calvin_files/
|-- [4.8K]  original_files/
|-- [142M]  snapshot/
|-- [ 304]  generated_experiments_list.csv
|-- [ 50M]  logs.tar.gz
|-- [1.4G]  output_exp_8c4e57ed-ce23-4a06-9d35-bfadbfc75f6f.org
|-- [ 16K]  stderr.txt
`-- [252K]  stdout.txt
3 directories, 5 files
```

Listing 8.2: Example of output for an experiment. The output file in bold centralizes all logs in a single and human-readable file.

These logs include a variety of information, such as: i) *original files* with the test description; ii) Calvin's files describing the applications and their characteristics; iii) *Calvin's logs* (for all nodes in the platform), including information about applications, actors and nodes; iv) general information, such as stdout and stderr outputs, CPU and RAM information, Linux version, etc.; and v) *Prometheus database* with all metrics collected from the platform.

8.3.3 Data analysis

Finally, the last step is the analysis of obtained results. This step is as important as any previously done and must be performed meticulously. Many misleading conclusions in scientific works are consequence of an incorrect analysis of the available data. In this direction, the reproducible research comes to close this gap, providing the tools to reproduce the results from the original data.

In my thesis, the data analysis is done in two steps. First, the logs are parsed by a set of scripts to recover the relevant information. This data is stored in a `csv` file in a proper repository. In a second time, we use the `org-mode` plugin [Sch+12] for `emacs` to create a research document describing the steps done to obtain the figures used in the papers. The analysis is mostly done through excerpts of R scripts in the `org-mode` document. With access to the data and the corresponding research document, other researchers are able to reproduce and verify the results.

Part III

The Provisioning of IoT Applications in the Fog

9 Problem Statement

We introduce in this chapter the provisioning problem, first component of the IoT orchestration in the Fog. A formulation for the problem is described, along with a presentation of the GRASP method subsequently used to solve it.

9.1 Introduction

The provisioning of IoT Applications in the Fog is an important step to implement a comprehensive orchestration system. It is responsible for placing the application's components in the infrastructure, with two-fold objective. In one side, the orchestrator must satisfy the IoT applications, selecting the best possible resources so applications can offer a good quality of service to the end users. On the other hand, it needs to optimize the use of the infrastructure, minimizing its cost and serving as many applications as possible.

The provisioning can be seen as a graph matching problem. The Service Deployer, which is the component responsible for the provisioning, receives two graphs as input. The first is the graph representing the IoT application, its components and the requirements in terms of IT resources and network. The second is the infrastructure graph, with a description of the nodes, links and their capabilities. Fig. 9.1 illustrates this process, the Service Deployer receives an application description and has a current view of the available infrastructure. Thus, the Service Deployer will apply some algorithm to choose the nodes to host each component of the application, considering, among others, the access to the different sensors (camera, microphones, smartphones, etc.) and the resources of each node.

We point out hereafter the characteristics which are important when deploying the applications in the Fog infrastructure, and which are relevant for the provisioning problem:

- **Heterogeneity**: is a key characteristic of the Fog infrastructure. These heterogeneous and sometimes constrained resources, leads to a higher complexity on the placement decision.

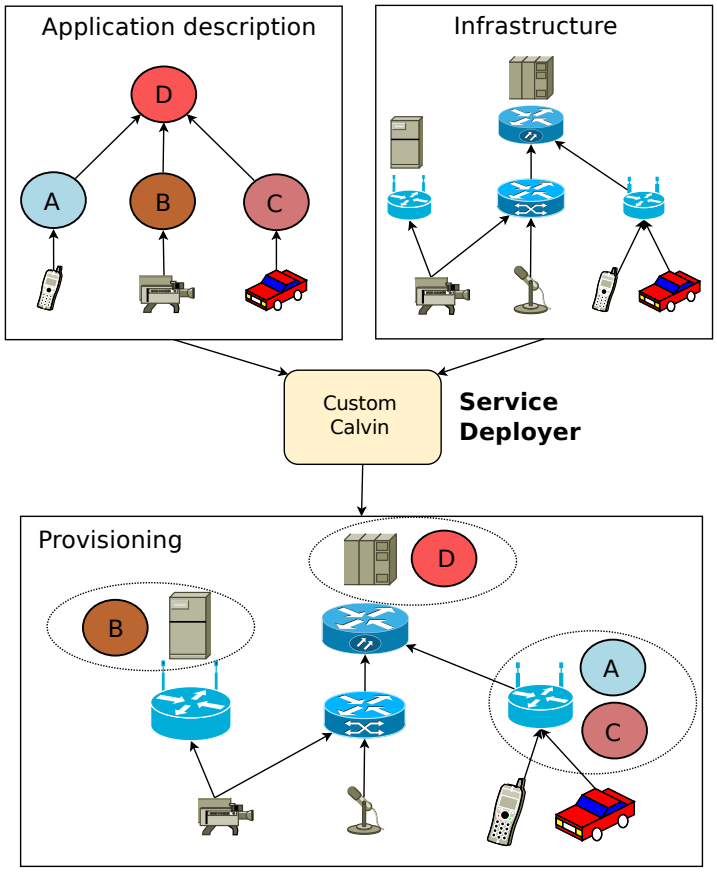


Figure 9.1: Provisioning as a graph matching problem. The Service Deployer is responsible for providing resources to IoT applications, considering its requirements and the infrastructure available.

- **Stringent requirements**: IoT applications frequently have stringent requirements which the Service Deployer must meet. These requirements may include: i) node's capacity, such as CPU and RAM; ii) node's geographical location and iii) network bandwidth and latency.

- **Inaccurate information**: Fog's large scale and heterogeneity make it very difficult to have an accurate view of the infrastructure status. The accuracy depends on the monitoring tools used on the platform and the frequency this information is updated, which may be very frequent in constrained resources.

- **Duality in objectives**: at first glance, IoT applications and infrastructure have contrasting goals. Ideally, IoT applications would like to have dedicated resources for execution, while infrastructure owners prefer to share the available resources among as many applications as possible.

The remaining of the provisioning part is organized as follows. In 9.2 we briefly explain the concept of grasp algorithms, which is used to implement our heuristic. A mathematical formulation, which models the problem as an ILP, is presented in

9.3. Given the mathematical formulation of the problem, we propose a heuristic which minimizes the provisioning cost in Chapter 10. The model and solution are extended in Chapter 11 to take into account the load of nodes, and consequently, improve the number of applications running in the infrastructure.

9.2 Related Work

9.2.1 The GRASP method

GRASP [FR95], which stands for Greedy Randomized Adaptive Search Procedure, is a meta-heuristic commonly used to solve combinatorial optimization problems. Many of these problems are sufficiently large to be intractable by exact algorithms. Hence, heuristics emerge as a reasonable alternative to provide good, but not necessarily optimal solutions. In this context, the GRASP method is an iterative meta-heuristic, in which each iteration generates possible solutions for the optimization problem, consisting in two main phases: the construction of an initial solution and its improvement through a local search procedure. In the end, the best solution found considering all iterations and an evaluation criterion, is selected.

Algorithm 1: GRASP pseudo-algorithm [FR95]

```
1  Function ConstructGreedyRandomizedSolution():
2      Solution = {}
3      for (Solution construction not done) do
4          MakeRCL (RCL)
5          s = SelectElementAtRandom (RCL)
6          Solution = Solution ∪ {s}
7          AdaptGreedyFunction (s)
8      return (Solution)
9  Input: InputInstance()
10 for (GRASP stopping criterion not satisfied) do
11     ConstructGreedyRandomizedSolution (Solution)
12     LocalSearch (Solution)
13     UpdateSolution (Solution, BestSolutionFound)
```

An excerpt from the GRASP pseudo-code is presented in Algorithm 1. The core of the algorithm is the *for* iteration on line 10, where solutions are generated until a specific stopping criterion is reached. Obviously, this stopping criterion is specific of the problem domain, but it can be as simple as a maximum number of iterations or elapsed time reached. The loop consists of three steps: i) the generation of an initial, randomized solution in `ConstructGreedyRandomizedSolution`; ii) `LocalSearch` analyzes the solution's vicinity, looking for possible improvements

in the initial solution; and iii) `UpdateSolution` which updates the final solution following an evaluation criterion.

The function `ConstructGreedyRandomizedSolution` is described in lines 1 to 8, where we detail the procedure to build the initial solution for the problem. The algorithm uses a greedy procedure to iteratively build the solution, adding one element at each iteration. The key point in this procedure is the `MakeRCL` function, which creates the Restricted Candidate List (RCL). The RCL orders all possible candidates for the solution with respect to a greedy and possible short-sighted function. This list is then limited to a subset of its possible values, according to some predetermined criteria. Again, the criteria vary with the problem and can be, for example: i) Cardinality based: maintains the best k solutions or ii) Value based: gets all candidates whose performance is better than $\alpha * \text{bestCandidate}$, for some $\alpha \in [0, 1]$. Afterwards, the solution is updated by adding one element, not necessary the best, from the RCL, in `SelectElementAtRandom`. Finally, the `AdaptGreedyFunction` can eventually be used to adjust the evaluation function according to the partial solution generated.

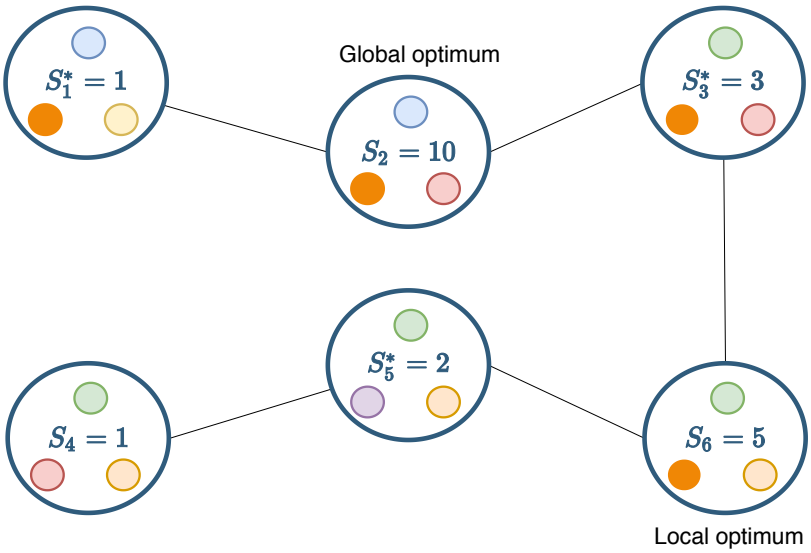


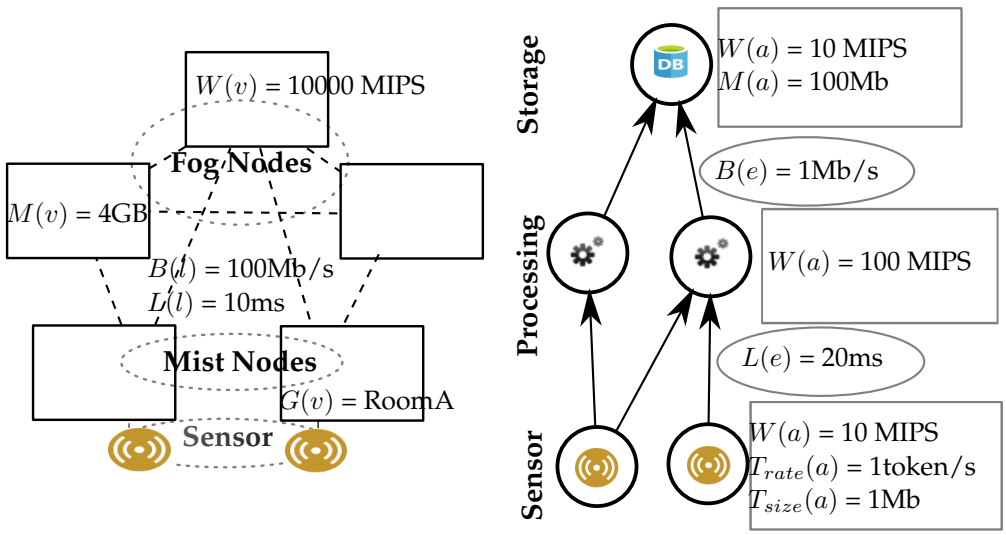
Figure 9.2: Set of solution generated by GRASP for a hypothetical maximization problem (higher values are better). S_1^*, S_3^* and S_5^* are initial solutions generated by `ConstructGreedyRandomizedSolution`.

The initial solution generated by GRASP is not guaranteed to be locally optimal when considering a simple neighborhood. Fig. 9.2 illustrates a set of solutions for a given problem. S^* represents the initial solution created by Algorithm 1. It is straightforward to see that they are not the best solutions, compared to S_6 or S_2, for example. The `LocalSearch` procedure applies a local search to try to improve these initial solutions, generating small disturbances to improve them iteratively. For each initial solution S^*, `LocalSearch` defines the neighborhood N in which it will seek better solutions (in the example, $N(S_3^*) = \{S_2, S_6\}$ and

$N(S_5^*) = \{S_4, S_6\}$). Note that the performance of the algorithm depends on both the initial solution and the neighborhood generation strategy. These are, therefore, the keys to success of GRASP algorithms. A pertinent initial solution along with an intelligent neighborhood generation may cover the most appropriate solution space and lead to optimized solutions.

9.3 Problem Formulation

The Fog landscape $\mathcal{P}$ is modeled as an undirected graph denoted by $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_P)$, as illustrated in Fig. 9.3b, where $V_{\mathcal{P}}$ represents the set of Fog nodes belonging to the different layers and $E_{\mathcal{P}}$ represents the direct communications between them. It is worth noting that $V_{\mathcal{P}} = (F_{\mathcal{P}} \cup S_{\mathcal{P}})$, where $F_{\mathcal{P}}$ corresponds to the set of Fog/Mist nodes, and $S_{\mathcal{P}}$ includes the sensors and actuators in the end device layer. Each node $v \in V_{\mathcal{P}}$ is characterized by its i) available CPU capacity $W(v)$ in MIPS, ii) available memory $M(v)$ in MB, iii) geographic location $G(v)$ and its iii) category $K(v)$[1]. Specifically, two types of Fog resources are considered: edge sensor/actuator (end device) and Mist/Fog node. Besides, we introduce two cost parameters for each Fog node, v: a unit cost of processing $C_W(v)$ and a unit cost of memory $C_M(v)$. Similarly, each physical link $l \in E_{\mathcal{P}}$ is characterized by i) its residual bandwidth $B(l)$, ii) its communication delay $L(l)$ and iii) its communication cost per unit bandwidth $C_B(l)$.



(a) Infrastructure model **(b)** Application model

Figure 9.3: Components of the provisioning problem

[1]The category specifies the hardware properties of the Fog resource (e.g. storage node, network node, sensor, etc.)

Similarly, an application $\mathcal{A}$ is composed of a set of inter-dependent application modules (i.e., micro-services). Fig. 9.3b presents an example of an application with three levels:

1. Sensors that observe the environment and send the collected data to be processed.

2. Processing objects which implement the business model of the application and may have stringent requirements in terms of latency and processing, for example.

3. Storage objects which stock the sensed data for further analysis.

Formally, it is modeled as a directed acyclic graph (DAG), $G_{\mathcal{A}} = (V_{\mathcal{A}}, E_{\mathcal{A}})$, where $V_{\mathcal{A}}$ and $E_{\mathcal{A}}$ correspond to the services and their logical links, respectively. Specifically, each service $a \in V_{\mathcal{A}}$ requires an amount of processing power $W(a)$ and a memory $M(a)$, a geographic location $G(a)$ and a specific category $K(a)$ (e.g. sensing, processing, storage, etc.). Following the dataflow model, $T_{rate}(a)$ is defined as the number of tokens per second sent by service a and $T_{size}(a)$ as the token size. Likewise, each link $e \in E_{\mathcal{A}}$ is characterized by a bandwidth demand $B(e)$ and a tolerable communication delay $L(e)$.

9.3.1 Fog service provisioning problem

The objective of Fog-enabled IoT application provisioning problem consists in optimizing the placement of application components (i.e., micro-services) on distributed Fog nodes while meeting their non-functional requirements. Mist resources are constrained because of their inherent physical structure. Hence, we aim at favoring the usage of higher level Fog nodes since they offer better performance. In doing so, latency-sensitive applications are not obstructed and can be easily provisioned on Mist nodes when needed. However, deploying applications' components far from their data sources may deteriorate the network performance. Consequently, a trade-off between the provisioning cost and applications QoS fulfillment should be assured.

Hereafter, we formulate the provisioning problem of an IoT application $\mathcal{A}$, denoted by the graph $G_{\mathcal{A}}$, in the Fog infrastructure $\mathcal{P}$, modeled by the graph $G_{\mathcal{P}}$. To do so, we introduce:

- α_{av} is a binary variable indicating whether the service, $a \in V_{\mathcal{A}}$, is assigned to the physical node, $v \in V_{\mathcal{P}}$, or not.

- N_{vx} denotes the set of admissible physical paths from v to x, $(v, x) \in V_{\mathcal{P}}^2$.

- $\mathcal{N}$ denotes the set of all admissible paths. Formally, $\mathcal{N} = \bigcup_{\{v,x\} \in V_{\mathcal{P}}^2} N_{vx}$.

- β_{en} is a binary variable indicating whether a logical link $e \in E_{\mathcal{A}}$ is hosted in the physical path $n \in \mathcal{N}$.

- $(a_s(e), a_d(e)) \in V_{\mathcal{A}}^2$ denotes the starting (source) and terminating (destination) component of the logical link $e \in E_{\mathcal{A}}$.

- h_{ln} is a binary coefficient determining whether the physical link $l \in E_{\mathcal{P}}$ belongs to the path $n \in \mathcal{N}$ or not.

- $B(e) = T_{rate}(a_s(e)) \times T_{size}(a_s(e))$ corresponds to the exchanged data rate in logical link e, from $a_s(e)$ to $a_d(e)$. We recall that T_{rate} is the number of tokens sent per second by $a_s(e)$ and T_{size} is the size of each token.

The provisioning of services is constrained so that for each IoT application $\mathcal{A}$, a service must be hosted in one Fog node. Formally,

$$\sum_{v \in V_{\mathcal{P}}} \alpha_{av} = 1, \quad \forall a \in V_{\mathcal{A}} \tag{9.1}$$

A service $a \in V_{\mathcal{A}}$ can be hosted in the physical node $v \in V_{\mathcal{P}}$, if i) the available residual resources (i.e. $W(v)$ and $M(v)$) are at least equal to those required (i.e. $W(a)$, $M(a)$) and ii) a has the same category and geographical location as v. Formally,

$$\forall v \in V_{\mathcal{P}} \begin{cases} \sum_{a \in V_{\mathcal{A}}} W(a)\alpha_{av} & \leq W(v) \\ \sum_{a \in V_{\mathcal{A}}} M(a)\alpha_{av} & \leq M(v) \end{cases} \tag{9.2}$$

$$(K(v) - K(a))\alpha_{av} = 0, \forall v \in V_{\mathcal{P}}, \forall a \in V_{\mathcal{A}} \tag{9.3}$$

$$(G(v) - G(a))\alpha_{av} = 0, \forall v \in V_{\mathcal{P}}, \forall a \in V_{\mathcal{A}} \tag{9.4}$$

We assume that a logical link $e \in E_{\mathcal{A}}$ between a service $a_s(e)$ and a service $a_d(e)$ is hosted in a path $n \in \mathcal{N}$ between v and x. Formally,

$$\sum_{n \in \mathcal{N}} \beta_{en} = 1, \quad \forall e \in E_{\mathcal{A}} \tag{9.5}$$

A logical link $e \in E_{\mathcal{A}}$ must be instantiated in a single path $n \in N_{vx}$. Such as $a_s(e) \in V_{\mathcal{A}}$ is hosted in Fog node $v \in V_{\mathcal{P}}$ and $a_d(e) \in V_{\mathcal{A}}$ is hosted in physical node $x \in V_{\mathcal{P}}$. Formally,

$$\forall e \in E_{\mathcal{A}}, \quad n \in N_{vx} \begin{cases} \beta_{en} \leq \alpha_{a_s(e)v} \\ \beta_{en} \leq \alpha_{a_d(e)x} \end{cases} \tag{9.6}$$

Each physical link $l \in E_{\mathcal{P}}$ is characterized by the consumed bandwidth $B_{used}(l)$ corresponding to the IoT application $\mathcal{A}$. Formally,

$$B_{used}(l) = \sum_{e \in E_{\mathcal{A}}} B(e) \sum_{n \in \mathcal{N}} h_{ln} \beta_{en}, \forall l \in E_{\mathcal{P}} \tag{9.7}$$

Besides, each physical link $l \in E_{\mathcal{P}}$ cannot host more than its capacity. Formally,

$$B_{used}(l) \leq B(l), \forall l \in E_{\mathcal{P}} \tag{9.8}$$

Each path $n \in \mathcal{N}$ is characterized by an end-to-end delay, $L(n)$. The latter corresponds to the sum of delays of its forming $l \in E_{\mathcal{P}}$. Formally,

$$L(n) = \sum_{l \in E_{\mathcal{P}}} h_{ln} L(l), \forall n \in \mathcal{N} \tag{9.9}$$

Finally, a logical link $e \in E_{\mathcal{A}}$ must be hosted in a path n, ensuring an end-to-end delay lower than that required by itself.

$$\sum_{n \in \mathcal{N}} L(n) \beta_{en} \leq L(e), \quad \forall e \in E_{\mathcal{A}} \tag{9.10}$$

Our objective is to generate, for each application $\mathcal{A}$, the best possible provisioning solution while minimizing the placement cost in terms of allocated resources within $\mathcal{P}$. For this reason, we define our objective function as follows,

$$\min_{\forall v \in V_{\mathcal{P}}, \forall l \in E_{\mathcal{P}}} (C_W^{tot} + C_M^{tot} + C_B^{tot}) \tag{9.11}$$

The C_W^{tot} represents the processing cost of the application' components in the infrastructure. C_M^{tot} is related to the cost of the memory required by the components. Finally, C_B^{tot} corresponds the total communication cost for data transfer between applications components. Also, we define the costs associated to the physical infrastructure as i) a cost for processing $C_W(v)$, ii) a cost for memory $C_M(v)$, and iii) a cost for data transfer $C_B(l)$. Formally,

$$C_W^{tot} = \sum_{v \in V_{\mathcal{P}}} \sum_{a \in V_{\mathcal{A}}} C_W(v) W(a) \alpha_{av} \tag{9.12}$$

$$C_M^{tot} = \sum_{v \in V_{\mathcal{P}}} \sum_{a \in V_{\mathcal{A}}} C_M(v) M(a) \alpha_{av} \tag{9.13}$$

$$C_B^{tot} = \sum_{l \in E_{\mathcal{P}}} C_B(l) B_{used}(l) \tag{9.14}$$

The Fog-Enabled IoT application provisioning problem corresponds to an advanced formulation of composable service placement in computer networks problem. As the latter has been proved NP-complete [HGW09], the proposed model in this section can only be solved, in an efficient manner, for small size instances. It is straightforward to see that the Fog service provisioning problem is very hard to solve, due to scalability constraints. In fact, the dimension of the solution space would heavily increase following: i) the number of IoT applications and ii) the size of the Fog infrastructure. In the next chapters, we will see two heuristics to tackle the provisioning problem in an efficient manner.

9.3.2 Summary of notations

Table 9.1 summarizes the notations used in the problem formulation. In the next chapters, we will reuse some of these notations to present the results of the two proposed strategies to solve the provisioning problem.

$\mathcal{P}$	Fog landscape
$V_{\mathcal{P}} = (F_{\mathcal{P}} \cup S_{\mathcal{P}})$	set of infrastructure nodes
$F_{\mathcal{P}}$	set of Fog/Mist nodes
$S_{\mathcal{P}}$	set of sensors
E_P	link between nodes
for each $v \in V_{\mathcal{P}}$	
$W(v)$	available CPU capacity (in MIPS)
$M(v)$	available memory (in MB)
$G(v)$	geographic location
$K(v)$	node's category
$C_W(v)$	unit cost of processing
$C_M(v)$	unit cost of memory
for each $l \in E_{\mathcal{P}}$	
$B(l)$	residual bandwidth
$L(l)$	communication delay
$C_B(l)$	communication cost
$\mathcal{A}$	IoT application
$V_{\mathcal{A}}$	set of application's services
$E_{\mathcal{A}}$	application's logical links
for each $a \in V_{\mathcal{A}}$	
$W(a)$	required amount of processing power (in MIPS)
$M(a)$	required amount of memory (in MB)
$G(a)$	required geographic location
$K(a)$	required category
$T_{rate}(a)$	number of tokens per second sent by service
$T_{size}(a)$	token size
for each $e \in E_{\mathcal{A}}$	
$B(e)$	required bandwidth
$L(e)$	tolerable communication delay
N_{vx}	set of physical paths from v to x
$\mathcal{N}$	denotes the set of all admissible paths
h_{ln}	binary coefficient showing if physical link l belongs to the path n
α_{av}	binary variable showing if service a is assigned to host v
β_{en}	binary variable showing if logical link e is hosted in the path n
C_W^{tot}	total provisioning cost of processing
C_M^{tot}	total provisioning cost of memory
C_B^{tot}	total provisioning cost of bandwidth

Table 9.1: Table of Notations

10 Cost-aware Provisioning

10.1 Proposed Solution: O-FSP

In this chapter, we will detail our proposal named **Optimized Fog Service Provisioning** (O-FSP) to resolve the formulated problem in the previous section. Our strategy is a greedy approach [Cor+09] which aims to incrementally construct an optimized Fog service provisioning solution. To achieve its objective, O-FSP proceeds as follows. First, the problem is split into a set of solution components that are sorted: solving a solution component corresponds to building a small part of the final solution. Then, each solution component is greedily placed while considering its requirements. Finally, the process is repeated until all solution components are provisioned. It is worth noting that O-FSP rejects a Fog service $\mathcal{A}$ as soon as it fails to find a placement to one of its component. O-FSP is summarized in pseudo-code form in Algorithm 2. In the following, we will detail each stage.

10.1.1 Fog service decomposition stage

The Fog-enabled IoT application (i.e., Fog service) $\mathcal{A}$ is subdivided into a set of k components according to $|V_{\mathcal{A}}|$ and $|E_{\mathcal{A}}|$. The aforementioned subsets are called solution components and are denoted by $\{\mathcal{C}_i\}_{1 \leq i \leq k}$. In order to do this, O-FSP selects first the subset of sensor/actuator nodes and their corresponding incoming/outgoing links. The latter are then retrieved from the graph. The remaining $\mathcal{A}$'s topology, devoid of the aforementioned solution components, is called $\tilde{\mathcal{A}}$. It is straightforward to see that $\tilde{\mathcal{A}}$ may hold several nodes lacking their incoming links. We refer to these links as "orphan links". $\mathcal{C}_i$ encompasses i) a central node v_i, and ii) its orphan incoming/outgoing links. $\{\mathcal{C}_i\}_{1 \leq i \leq k}$ is iteratively built on the basis of the number of their orphan links. Indeed, the nodes with the highest number of orphan links are selected first. The process is repeated until $\tilde{\mathcal{A}}$ becomes empty (i.e., $\tilde{\mathcal{A}} = \emptyset$).

Algorithm 2: `O-FSP` pseudo-algorithm

Input: $G_{\mathcal{A}}$, $G_{\mathcal{P}}$, N_{max}
Output: $\mathcal{S}_{best}$
Function `FindProvisioning`$(\mathcal{C}_a)$**:**
    i = 0
    $\{\mathcal{C}_a^i\}_{1 \leq i \leq |V_{\mathcal{P}}|} \leftarrow \emptyset$
    for $v \in V_{\mathcal{P}}$ **do**
        if $W(a) <= W(v)$ *and* $M(a) <= M(v)$ *and* $K(a) = K(v)$ *and* `IsNetworkSufficient` $(v, \mathcal{C}_a)$ **then**
            $\{\mathcal{C}_a^i\} \leftarrow v$ and its set of directed links
            i += 1
    $\{\mathcal{C}_a^i\}_{1 \leq i \leq N_{max}} \leftarrow$ `GenerateSolutions` $(\{\mathcal{C}_a^i\}_{1 \leq i \leq |V_{\mathcal{P}}|}, N_{max})$
    return $(\{\mathcal{C}_a^i\}_{1 \leq i \leq N_{max}})$
$\mathcal{S}_{best} \leftarrow \emptyset$
$\{\mathcal{S}_i\}_{1 \leq i \leq N_{max}} \leftarrow \emptyset$
$\{\mathcal{C}_i\}_{1 \leq i \leq k} \leftarrow$ `FogServiceDecomposition` $(G_{\mathcal{A}})$
$\tilde{\mathcal{A}} \leftarrow \mathcal{A}$
/ Find N_{max} placements of end devices */*
for $(a \in V_A$ *and* $K(a) = "end-devices")$ **do**
    $\{\mathcal{C}_a^i\}_{1 \leq i \leq N_{max}} \leftarrow$ `FindProvisioning` $(\mathcal{C}_a)$
    for $i = 1$ *to* N_{max} **do**
        $\mathcal{S}_i \leftarrow \mathcal{S}_i \cup \{\mathcal{C}_a^i\}$
    $\tilde{\mathcal{A}} \leftarrow \tilde{\mathcal{A}} \setminus \mathcal{C}_a$
$Stop \leftarrow false$
while $(\tilde{\mathcal{A}} != \emptyset)$ *and* $(Stop = false)$ **do**
    Select $a \in V_{\tilde{\mathcal{A}}}$ having the highest number of orphan links
    $\mathcal{C}_a \leftarrow a$ and all its in/out orphan links
    $\{\mathcal{C}_a^i\}_{1 \leq i \leq N_{max}} \leftarrow$ `FindProvisioning` $(\mathcal{C}_a)$
    if $(\{\mathcal{C}_a^i\} != \emptyset)$ **then**
        for $i = 1$ *to* N_{max} **do**
            $\mathcal{S}_i \leftarrow \mathcal{S}_i \cup \{\mathcal{C}_a^i\}$
        $\tilde{\mathcal{A}} \leftarrow \tilde{\mathcal{A}} \setminus \mathcal{C}_a$
    else
        $\mathcal{S}_{best} \leftarrow \emptyset$
        $Stop \leftarrow true$
$\mathcal{S}_{best} \leftarrow$ `SelectBestSolution` $(\{\mathcal{S}_i\}_{1 \leq i \leq N_{max}})$

10.1.2 Solution component's provisioning stage

During the first iteration, `O-FSP` finds the N_{max} best placements of end device solution components $\{\mathcal{C}_i\}_{1 \leq i \leq k_1} \mid k_1 \leq k$. Then, `O-FSP` incrementally constructs the Fog service provisioning solution. Hence, during each iteration, the N_{max} best (partial) placements, $\{S_i\}_{1 \leq i \leq N_{max}}$, maximizing the objective function defined by Equation 9.11) are generated (`GenerateSolutions` in Algorithm 2). To do so,

N_{max} physical nodes are selected for hosting the considered $\mathcal{C}_i$. The latter has sufficient resources (i.e. CPU, memory), and it is connected to one $\{\mathcal{C}_j\}_{1\leq j\leq i-1}$'s placement solution through a shortest path ensuring the required bandwidth and latency (`IsNetworkSufficient` in Algorithm 2). We recall that N_{max} solutions $\{\mathcal{C}_j^i\}_{1\leq i\leq N_{max}}$ have already been generated for the placement of $\mathcal{C}_j$ during the previous iterations. In doing so, we generate the best possible solution while avoiding the exploration of all possible combinations that would make the resolution impractical. Finally, the best solution S_{best}, which corresponds to the placement that maximizes the objective function, is selected (`SelectBestSolution` in Algorithm 2).

10.2 Evaluation

In this section, we describe the experimental environment used to evaluate the performance of `O-FSP`. The environment relies on the `FITOR` platform to build the Fog environment and run the IoT applications. Note that evaluation results are averaged over 10 executions and presented with 95% confidence interval.

10.2.1 Describing the environment

In this section, we present the experimental setup used during the study of the provisioning of IoT applications in the Fog. We describe the parameters that characterize platform, workload and orchestrator.

Platform

Our Fog infrastructure $\mathcal{P}$ relies on elements from Grid5000 and FIT/IoT-LAB platforms. The Fog layer is composed of 20 servers from Grid5000 which are part of the *genepi* cluster. They are characterized by 2 CPUs Intel Xeon E5420, with 4 cores per CPU and 8 GB of RAM. Their CPU cost C_W and memory cost C_M are arbitrarily set to 0.1.

It is worth noting that the runtimes are hosted by containers. Hence, to emulate a heterogeneous environment, we define four types of containers:

- Controller: runs the `FITOR`'s node that is responsible for the deployment and control of IoT applications. This node is characterized by a memory capacity of 4GB and can use 100% of the host CPU.

- Fog100: is a more powerful Fog node with 2GB of RAM and 100% of available CPU.

- Fog60: is a middle Fog node with 2GB of RAM. The latter can use only 60% of the CPU.

- Fog30: is a limited capacity Fog node which can use only 30% CPU but has the same 2GB RAM size.

The Mist layer is composed of 50 A8 nodes. These nodes run `FITOR`'s processes and may execute application components. Besides, C_W and C_M are set to 0.9. The cost of links C_B is fixed to 0.1. This higher cost compared to the Fog layer, points out the fact that these resources are less powerful, less available and closer to the end devices and so, must be used cautiously.

Workload

The IoT applications in our workload follow the 3-level model as described in Section 8.1.2. At the bottom, 1 Trigger service periodically sends tokens which represent collected measurements related to the end devices' surrounding environment. These tokens are processed by $[1, 4]$ Burn services, which emulate the performed application treatment and consumes a certain amount of million instructions (MI) per token. Finally, $[1, 2]$ Sink services store the received tokens in memory for further processing (if necessary). Hence, our workload is described by:

- **Application load**: the applications send tokens of a fixed size (T_{size}), equals to 1024 *bytes* at a given rate (T_{rate}), taking values in $[1, 10]$ *tokens per second*. Each token consumes between $[100, 1500]$ *MI*. Note that each parameter is uniformly chosen in the configured range.

- **Application heterogeneity**: the heterogeneity of our workload is characterized by the variability of token rate and processing as described in the application load profile.

- **System Load**: is characterized by 50 applications being deployed in the environment.

- Application **arrival interval**: is fixed to 1 application every 2 minutes.

- Application threshold: no applicable in the provisioning setup.

- Churn: applications are always active.

► **Low-level application description:** the workload characterization comprises these low-level parameters, described hereafter:

- **Requirements for services**:
  - Trigger: requires a memory $M(a)$ equals to 100 bytes and an amount of CPU $W(a)$ proportional to its T_{rate}, in the range $[300, 1000]$ MIPS
  - Burn: CPU is proportional to the processing effort per token, W_{token}, i.e., $W(a) = T_{rate} \times W_{token}$. On the other hand, $M(a)$ is fixed to 100Kb.
  - Sink: requests $W(a)$ equals to 500 MIPS and a $M(a)$ proportional to the token size and rate, such as $M(a) = T_{rate} \times T_{size}$.
- Requirements for links: bandwidth is fixed (100Kb/s) so links have enough capacity to send the data. Latency is set to $100ms$ or $1s$.
- Connection type: we use the 1:N where one service is connected to all actors in top layer.
- Routing method: all tokens are sent to all services.

Orchestrator parameters

- **Provisioning strategy**: `O-FSP` and the baseline strategies described at Section 10.2.2.
- Reconfiguration strategy: not applicable for provisioning.
- Maintenance interval: not applicable for provisioning.
- **Monitoring interval**: information about resource usage (CPU, RAM) is updated every *60 seconds*.

10.2.2 Baseline strategies

In this section, we describe the baseline strategies used to compare the performance of `O-FSP`.

- **Uniform**: distributes each service $a \in V_{\mathcal{A}}$ uniformly in available $v \in V_{\mathcal{P}}$. The uniform heuristic does not optimize the provisioning cost and only considers the requirements given by the user.

- **Best-fit**: places the service a in nodes with the smallest available residual resources, which meets the requirements of the a. This heuristic favors the physical nodes and physical links with the smallest residual resources.

$$\max \left(\sum_{a \in V_{\mathcal{A}}} \sum_{v \in V_{\mathcal{P}}} \left(\frac{W(a)}{W(v)} + \frac{M(a)}{M(v)} \right) * \alpha_{av} + \sum_{e \in E_{\mathcal{A}}} \sum_{n \in \mathcal{N}} \left(\frac{B(e)}{B(n)} + \frac{L(n)}{L(e)} \right) * \beta_{en} \right) \tag{10.1}$$

- **Min-latency**: minimizes the sum of delays between nodes hosting the application's components.

$$\min \sum_{e \in E_{\mathcal{A}}} \sum_{n \in \mathcal{N}} L(n) \beta_{en} \tag{10.2}$$

10.2.3 Performance metrics

To evaluate the performance of `O-FSP` compared with the classical approaches, we consider the following metrics:

- $\mathbb{A}$: is the acceptance rate of IoT applications.

- $\mathbb{T}$: is the total number of processed tokens per second within the infrastructure. This metric represents the applications' throughput.

- $\overline{\mathbb{W}}$: is the average CPU utilization (expressed in percentage) of physical nodes in Grid5000 and FIT/IoT-LAB.

- $\mathbb{C}_{tot}$: is the total provisioning cost related to the consumed resources, as measured by the monitoring tools.

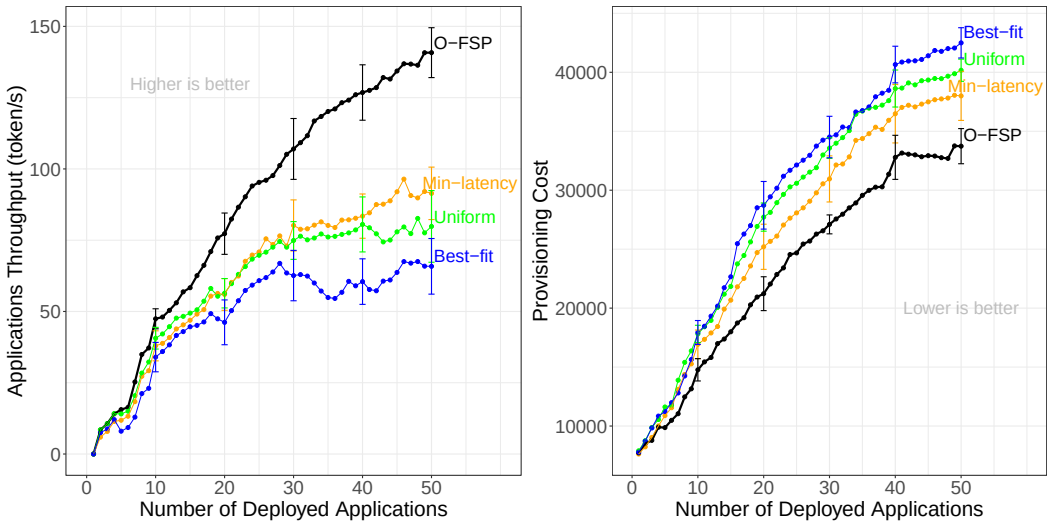
10.2.4 Evaluation results

First, we evaluate the proposed strategy `O-FSP` regarding its acceptance rate and in comparison with the alternative approaches, `Best-fit`, `Uniform` and `Min-latency`. Table 10.1 compares the percentage of accepted IoT applications. We notice that `O-FSP` achieves 77.8% and hence, outperforms the classical strategies.

Provisioning Approach	Acceptance Rate (%)
O-FSP	77.8 ± 5.5
Min-latency	65.6 ± 1.9
Best-fit	69.4 ± 3.4
Uniform	65.2 ± 2.9

Table 10.1: `O-FSP` performance evaluation. $\mathbb{A}$ - Acceptance rate

This is due to the fact that the proposed approach aims to minimize the provisioning cost. In doing so, higher residual resources, specifically network bandwidth, are maintained. It is worth noting that `Best-fit` achieves good results since it favors the selection of less powerful nodes which will lead to a higher number of available powerful nodes.



(a) $\mathbb{T}$ - Applications throughput

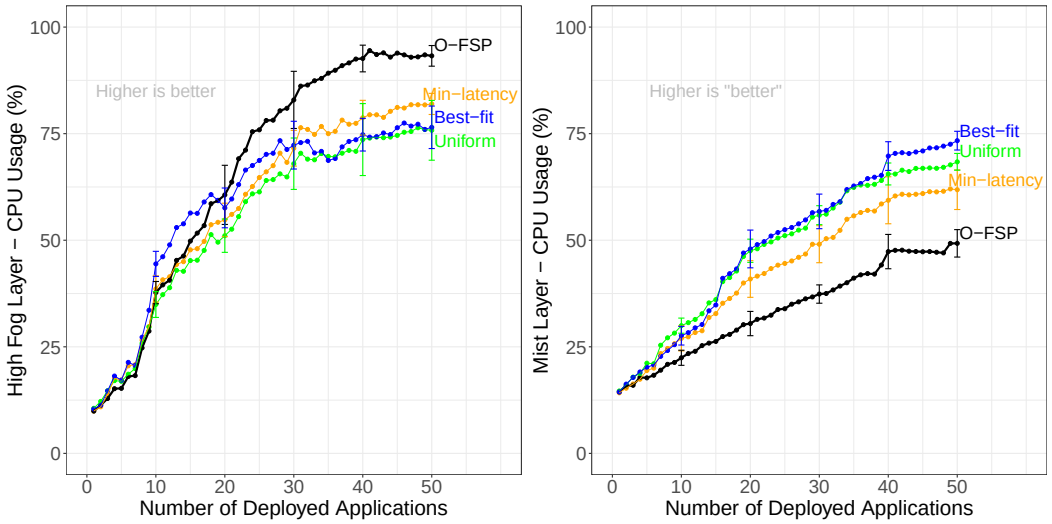
(b) $\mathbb{C}_{tot}$ - Total provisioning cost

Figure 10.1: `O-FSP` performance evaluation: application and provisioning cost results

Fig. 10.1a depicts the rate of processed tokens when increasing the number of provisioned applications. It is straightforward to see that our scheme `O-FSP` outperforms the classical strategies thanks to its capability to accept IoT applications while meeting their requirements. It is worth noting that the gap between the different approaches is insignificant for low number of provisioned applications. This is due to the fact that initially, all provisioning strategies are capable of placing the applications. However, the gap gets wider as soon as the number of applications increases. Such a result proves that `O-FSP` efficiently places the application components which corroborates the results related to the acceptance rate.

Fig. 10.1b illustrates the provisioning cost of accepted IoT applications. It notably shows that `O-FSP` achieves a lower cost compared with the classical strategies and does so throughout the experiment. In fact, at the end of the experiment, the provisioning cost of `O-FSP` is $\approx 13\%$ lower than the one of the second strategy,

`Min-latency` ($33,735 \pm 1,490$ vs. $37,998 \pm 2,078$). Such results are predictable since the proposed approach favors high fog physical nodes as long as the application requirements are satisfied. It worth noting that end devices and mist nodes have limited capacity. Consequently, they incur a high cost compared with fog nodes which are characterized by higher capacities.



(a) $\overline{\mathbb{W}}$ - High fog layer **(b)** $\overline{\mathbb{W}}$ - Mist layer

Figure 10.2: `O-FSP` performance evaluation: infrastructure performance results

In order to gauge the efficiency of `O-FSP` in terms of resource consumption, we evaluate the CPU usage of both end device/mist and fog nodes. Fig. 10.2a and Fig. 10.2b prove that our proposed strategy favors the Fog nodes whenever they meet the application requirements. In fact, it achieves 90% of CPU usage which is 10% higher than the second method `Min-latency`. In doing so, end devices and mist nodes are kept available and used only when it is necessary. We recall that the latter are less powerful and more expensive than Commercial-Off-The-Shelf (COTS) servers. It is worth noting that all strategies reach, at the end of the experiment, a high level of CPU usage on both end device/mist and fog layers. This is due to the system saturation.

10.3 Limitations

`O-FSP` is capable of providing a fair solution to the provisioning problem, optimizing the placement of IoT application components while meeting their non-functional requirements. Despite the good performance of `O-FSP` in terms of acceptance rate and applications throughput, the cost is a metric focused on infrastructure. By minimizing the provisioning cost, `O-FSP` tends to concentrate the applications in the cheaper nodes of the infrastructure. Consequently, bottlenecks can be created in

the infrastructure, which impair not only the performance of applications, but also the number of applications running in the infrastructure.

In the next chapter, we present an alternative approach to cope with this limitation, which aims to improve the acceptance rate of IoT applications while maintaining a minimal infrastructure cost.

11 Load-aware Provisioning

11.1 Extension of the Problem Formulation

In the last chapter, we have seen how to solve the provisioning problem in an optimized way, which minimizes the total cost while meeting application requirements. However, `O-FSP` tends to concentrate the applications in a few hosts, which may lead to sub-optimal infrastructure utilization. In this chapter, we extend the model proposed in Section 9.3 to consider also the load of nodes. Consequently, our decision is guided by two main objectives and the solution is found by solving the problem as a multi-objective problem.

Firstly, we aim to fulfill applications requirements while minimizing the cost of resources in $\mathcal{P}$, as defined in Section 9.3. So, our first objective function is

$$\min_{\forall v \in V_{\mathcal{P}}, \forall l \in E_{\mathcal{P}}} (C_W^{tot} + C_M^{tot} + C_B^{tot}). \tag{9.11}$$

Remember that C_W^{tot} corresponds to the processing cost of the application' components in the infrastructure. C_M^{tot} expresses the cost of the memory required by the components. Finally, C_B^{tot} represents the total communication cost for data transfer between applications components (cf. definitions in Eq. (9.12), (9.13) and (9.14)).

Secondly, we aim to maximize the number of provisioned IoT applications while meeting 100% of their demand. To do so, it is crucial to perform a load-aware application provisioning. The latter should balance the load of Fog nodes while taking into account their properties in terms of processing and memory usage. In this perspective, we formulate our second optimization objective function as follows:

$$\max \min_{v \in V_{\mathcal{P}}} \left(\frac{W(v)}{W_{max}(v)} + \frac{M(v)}{M_{max}(v)} \right), \tag{11.1}$$

where W_{max} and $M_{max}(v)$ correspond to the CPU and memory capacity of node v, respectively. It is straightforward to see that Eq. 11.1 aims at maximizing the

minimal residual resources of Fog nodes, while considering their capabilities. In doing so, an equitable load sharing will be achieved.

11.2 Proposed Solution: GO-FSP

In order to cope with both objective functions, we propose a new strategy which makes use of Greedy Randomized Adaptive Search Procedures [AR95] to optimize the Fog service provisioning.

Our **GRASP-based Optimized Fog Service Provisioning** solution, called `GO-FSP`, initially generates N best initial solutions, $\{S_i\}_{1 \leq i \leq N}$. Then, for each S_i, our strategy iteratively constructs new solutions by performing an efficient local search procedure. The key idea behind our strategy is to generate new solutions which simultaneously enhance the objective functions proposed in Eq. 11.1 and Eq. 9.11. The process will be repeated until all S_i are explored or no new improving solutions are found.

`GO-FSP` has three main stages: i) decomposition of the Fog service; ii) generation of initial solutions; and iii) local search. Algorithm 3 illustrates the pseudo-code of our proposal `GO-FSP`. In the following, we will detail each stage.

11.2.1 Fog service decomposition

The Fog service $\mathcal{A}$, is split up into a set of k components according to $|V_{\mathcal{A}}|$ and $|E_{\mathcal{A}}|$, known as solution components and denoted by $\{\mathcal{C}_i\}_{1 \leq i \leq k}$. The Fog service decomposition is performed as follows. First, the sensor/actuator nodes and their outgoing/ingoing links are selected. Then, these solution components are retrieved from the graph. It is straightforward to see that the remaining $\mathcal{A}$'s topology, called $\tilde{\mathcal{A}}$, will contain several nodes bereft of their attached links. We refer to these links as "orphan links". $\mathcal{C}_i$ encompasses i) a central node a_i, and ii) its orphan attached links. The sequence of $\{\mathcal{C}_i\}_{1 \leq i \leq k}$ is iteratively built in decreasing order of orphan links' number. To do so, the Fog service with the highest number of orphan links is selected. Then, $\mathcal{C}_i$ is constituted using only the Fog service and all its attached orphan links. Afterwards, $\mathcal{C}_i$ is subtracted from $\tilde{\mathcal{A}}$ (i.e., $\tilde{\mathcal{A}} \leftarrow \tilde{\mathcal{A}} \setminus \mathcal{C}_i$). The process is recursively repeated to generate the remaining solution components until $\tilde{\mathcal{A}}$ becomes empty (i.e., $\tilde{\mathcal{A}} = \emptyset$).

Algorithm 3: `GO-FSP` pseudo-algorithm

```
1  Input: G_A, G_P, S_{1..n}, N, K, ε
2  Output: S_best
3  Function LocalSearch(C_a, S̃):
4      RCL ← ∅
5      for v ∈ V_P do
6          if C(v,a) ≤ C(v*,a) × (1+ε) then
7              RCL ← RCL ∪ v
8      r ← FindBetterSolution (C_a, RCL)
9      if r != S̃(a) then
10         S̃(a) ← r
11         return (true)
12     return (false)
13 S_best ← ∅, Optimized ← true
14 for (i = 1 to N) and (Optimized = true) do
15     Ã ← A
16     Stop ← false, Optimize ← false
17     for (j = 1 to K) and (Stop = false) do
18         Stop ← true
19         while Ã != ∅ do
20             Select a ∈ V_Ã having the highest number of orphan links
21             C_a ← a and all its in/out orphan links
22             if LocalSearch (C_a, S_i) then
23                 Stop ← false
24             Ã ← Ã \ C_a
25     if Improved(S_i, S_best) then
26         S_best ← S_i
27         Optmized = true
```

11.2.2 Generation of initial solutions

This stage consists in generating N provisioning solutions $\{S_i\}_{1 \leq i \leq N}$. These S_i are built using our heuristic based provisioning approach, `O-FSP`. The rationale behind the aforementioned strategy is to greedily select N best solutions that minimize the provisioning cost. To do so, `O-FSP` incrementally constructs N optimized solutions while generating during each iteration, N best partial solutions within a fixed neighborhood.

11.2.3 Local search

Starting from an initial S_i, `GO-FSP`, incrementally constructs an improved solution, $\tilde{S}_i$. To do so, it iteratively explores the ordered set of solution components that has

been provided as the outcome of the decomposition stage. During each iteration, a Restricted Candidate List (RCL) is generated for the ongoing solution component, $\mathcal{C}_i$. RCL is composed of nodes within ϵ distance of the best ranked Fog node v^* in terms of $C(v^*, a_i)$. Note that $C(v, a)$ corresponds to the provisioning cost of v for hosting a. The aforementioned cost can be formulated as follows:

$$C(v, a) = (C_W(v) \times W(a) + C_M(v) \times M(a)) \tag{11.2}$$

In this perspective, RCL can be defined as $\{v \in V_{\mathcal{P}} \mid C(v, a_i) \leq C(v^*, a_i) \times (1 + \epsilon)\}$. It is worth noting that the Fog nodes belonging to RCL are potential provisioning candidates since they fulfill the requirements of $\mathcal{C}_i$ in terms of CPU, memory, network latency and bandwidth.

Afterwards, thanks to `FindBetterSolution`, the least loaded node will be selected to host a_i. Formally,

$$\max_{v \in RCL} \left(\frac{W(v)}{W_{max}(v)} + \frac{M(v)}{M_{max}(v)} \right). \tag{11.3}$$

In doing so, we minimize the variance of Fog nodes' load, to achieve an equitable load sharing among them, which amounts to optimize the second objective function formulated in 11.1.

11.3 Evaluation

We detail in this section the experimental environment used to evaluate the performance of `GO-FSP`. The environment relies on the `FITOR` platform to build the Fog environment and run the IoT applications. Note that evaluation results are averaged over 20 executions and presented with 95% confidence interval.

11.3.1 Describing the environment

The environment used to evaluate the `GO-FSP` is similar to the one used for `O-FSP`, described in last chapter. Hereafter, we recall the most important parameters that characterize this environment (for the complete setup, see Section 10.2.1).

Platform

The Fog layer in our infrastructure is composed of 20 servers from Grid5000 which are part of the *parapide* cluster. The latter are characterized by 2 CPUs Intel Xeon X5570, with 4 cores per CPU and 24 GB of RAM. Their CPU cost, C_W, and memory cost, C_M, are arbitrarily fixed to 0.1. Besides, the runtimes are hosted by Docker-based containers, with different CPU availability (30%, 60% and 100% of the node's CPU available for applications).

The Mist layer is composed of 50 A8 nodes. The A8 nodes are characterized by their ARM A8 microprocessor, 600 MHz, and 256MB of RAM. These nodes run `FITOR`'s processes and may execute application components. Besides, C_W and C_M are set to 0.9. The cost of links C_B is fixed to 0.1.

Workload

The IoT applications in our workload follow the 3-level model as described in Section 8.1.2. At the bottom, 1 Trigger service periodically sends tokens to be processed by $[1, 4]$ Burn services. Finally, $[1, 2]$ Sink services store the received tokens in memory for further processing. Hence, our workload is described by:

- **Application load**: the applications send tokens of a fixed size (T_{size}), equals to 1024 *bytes* at a given rate (T_{rate}), taking values in $[1, 10]$ *tokens per second*. Each token consumes between $[100, 1500]$ *MI*. Note that each parameter is uniformly chosen in the configured range.

- **Application heterogeneity**: the heterogeneity of our workload is characterized by the variability of token rate and processing as described in the application load profile.

- **System Load**: is characterized by 50 applications being deployed in the environment.

- Application **arrival interval**: is fixed to 1 application every 2 minutes.

▶ **Low-level application description:** the workload characterization comprises these low-level parameters, described hereafter:

- **Requirements for services**:

- Trigger: requires a memory $M(a)$ equals to 100 bytes and an amount of CPU $W(a)$ proportional to its T_{rate}, in the range $[300, 1000]$ MIPS

- Burn: CPU is proportional to the processing effort per token, W_{token}, i.e., $W(a) = T_{rate} \times W_{token}$. On the other hand, $M(a)$ is fixed to 100Kb.

- Sink: requests $W(a)$ equals to 500 MIPS and a $M(a)$ proportional to the token size and rate, such as $M(a) = T_{rate} \times T_{size}$.

Orchestrator parameters

- **Provisioning strategy**: `GO-FSP` and the baseline strategies described at Section 11.3.2.

11.3.2 Baseline strategies

We consider these three provisioning algorithms as basis of comparison next sections:

- **O-FSP**: the algorithm described in Chapter 10, which considers only the provisioning cost when placing the applications.

- **Uniform**: distributes each service $a \in V_{\mathcal{A}}$ uniformly in available $v \in V_{\mathcal{P}}$.

- **Best-fit**: places the service a in nodes with the smallest available residual resources, which meets the requirements of the a, as described in Eq. 10.1.

11.3.3 Performance metrics

To evaluate the performance of `GO-FSP`, we consider the following metrics:

- $\mathbb{A}$: is the acceptance rate of IoT applications.

- $\mathbb{T}$: is the total number of processed tokens per second within the infrastructure. This metric represents the applications' throughput.

- $\overline{\mathbb{L}}$: is the average **latency** (in seconds) for applications. It measures the latency between the end devices (represented by Trigger services) and the top application layer (represented by Sink services).

- $\overline{\mathbb{W}}$: is the average CPU utilization (expressed in percentage) of physical nodes in Grid5000 and FIT/IoT-LAB.

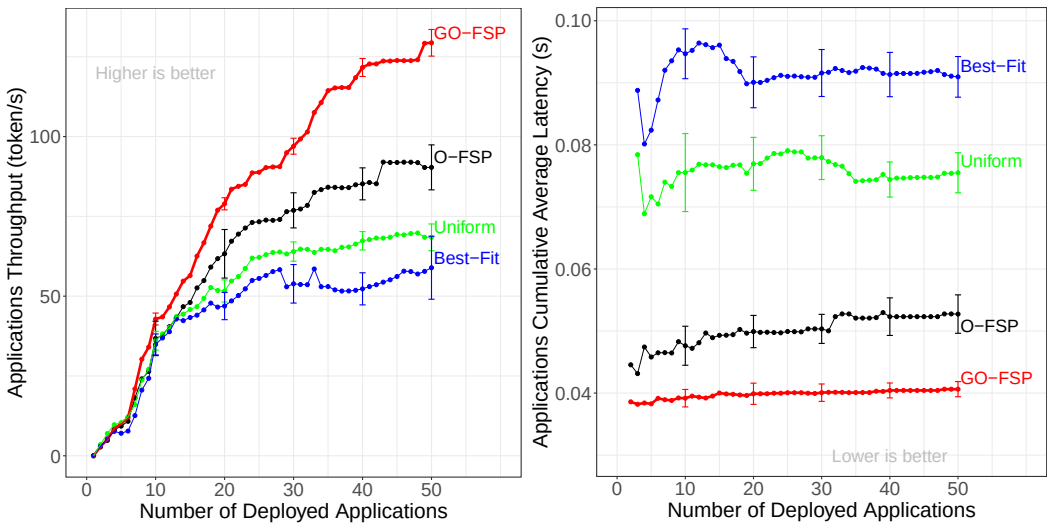
- $\mathbb{C}_{tot}$: is the total provisioning cost related to the consumed resources, as measured by the monitoring tools.

11.3.4 Evaluation results

We evaluate in Table 11.1, the rate of accepted IoT applications. We notice that `GO-FSP` ensures a high acceptance rate compared to `Uniform`, `O-FSP` and `Best-fit`. Indeed, `GO-FSP` accepts 10% more of the second strategy, `Best-fit`. This is due to the fact that our strategy aims at optimizing the load balancing between Fog nodes. In doing so, resources bottleneck is considerably relieved and consequently more applications can be executed on the infrastructure. Note that the results presented in this section cannot be compared with those in Chapter 10, because the environment has changed (especially the machines used in Grid'5000 infrastructure).

Provisioning Approach	Acceptance Rate (%)
GO-FSP	65.5 ± 0.8
O-FSP	56.8 ± 2.4
Best-fit	58.3 ± 3.7
Uniform	55.1 ± 1.6

Table 11.1: `GO-FSP` performance evaluation. $\mathbb{A}$ - Acceptance rate



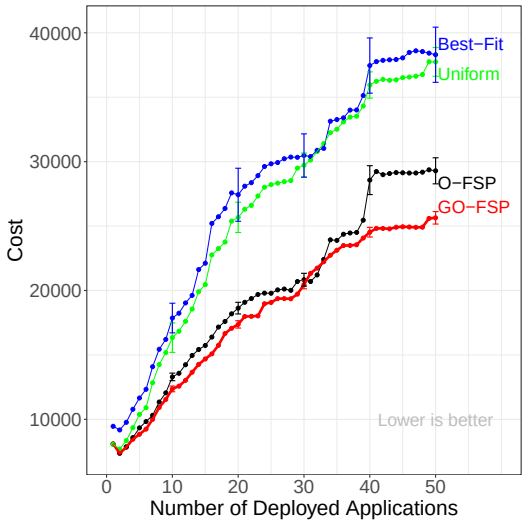
(a) $\mathbb{T}$ - Applications throughput

(b) $\overline{\mathbb{L}}$ - Applications average latency

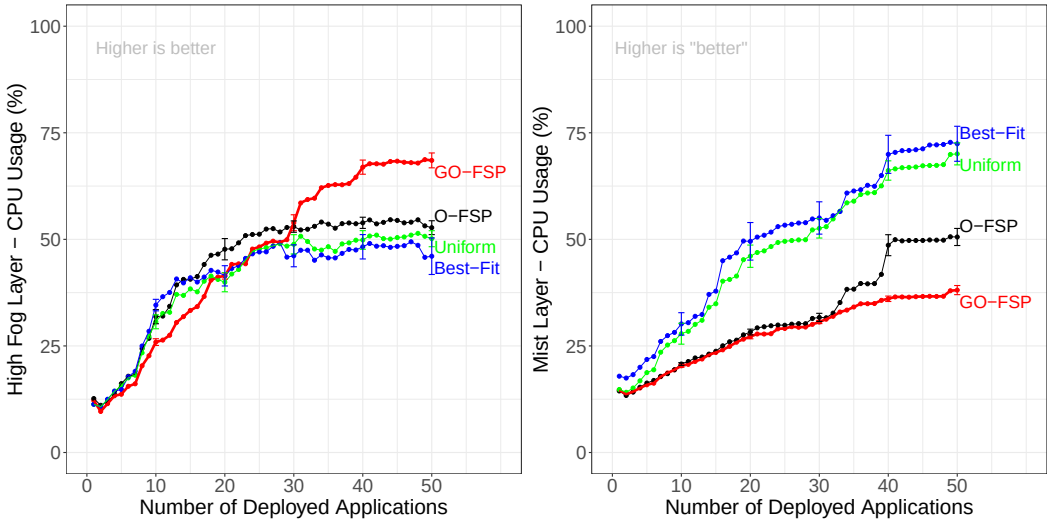
Figure 11.1: `GO-FSP` performance evaluation: application performance results

Fig. 11.1 depicts the performance results in regard to the users' perspective. Specifically, Fig. 11.1a presents the rate of processed tokens while the number of provisioned applications increases. This metric expresses the applications' throughput by

counting the number of tokens that are fully processed and delivered to the Sink services. We note that GO-FSP achieves 30% higher throughput than other strategies thanks to its higher acceptance rate while meeting the applications' requirements. The performance gap gets wider when the number of applications increases. This is due to the fact that at the beginning, most resources are available and thus, all provisioning strategies are capable of placing the applications. In Fig. 11.1b, we evaluate $\overline{\mathbb{L}}$, the network latency experimented by the applications. It is straightforward to see that GO-FSP achieves the lowest network delay. Indeed, by the end of experiments, our proposal decreases the end-to-end latency of applications by respectively 18%, 48%, 59% compared to O-FSP, Uniform and Best-fit. Such an achievement is ensured thanks to the capability of GO-FSP to aggregate application components while jointly optimizing the provisioning cost and load balance.



(a) $\mathbb{C}_{tot}$ - Total provisioning cost



(b) $\overline{\mathbb{W}}$ - High fog layer

(c) $\overline{\mathbb{W}}$ - Mist layer

Figure 11.2: GO-FSP performance evaluation: infrastructure performance results

In order to gauge the efficiency of GO-FSP in terms of resources usage, we evaluate the infrastructure performances in Fig. 11.2. Firstly, Fig. 11.2a depicts the provisioning cost, $\mathbb{C}_{tot}$. We note that GO-FSP decreases $\mathbb{C}_{tot}$ by 15% compared to O-FSP strategy. This can be explained by the fact that our proposal favors cheaper resources and hence places

application components in the High Fog Layer whenever it is pertinent. Secondly, the CPU utilization, for both High Fog and Mist layers, is evaluated in Fig. 11.2b and 11.2c. It is straightforward to see that GO-FSP optimizes the usage of the resources available in the High Fog Layer. The gap is approximately equal to 12.5% compared to the second best strategy, O-FSP. Such a result highlights the utility of the load balance which corroborates the results obtained in Fig. 11.1. Finally, we notice that the CPU utilization in the High Fog layer remains less than 75% due to the heterogeneity of applications profiles.

11.4 Limitations

11.4.1 Experimental limitations

The experimental setup is vital for the correct comprehension and evaluation of provisioning strategies. For this reason, we aggregate resources from Grid'5000 and FIT/IoT-LAB testbeds, along with the FITOR platform, in order to set up our experimental environment. However, our experiments have some downsides which are important to keep in mind.

Probably the most obvious drawback is the network resources in our environment. First, the VPN connection between FIT/IoT-LAB and Grid'5000 creates an artificial link over which we have no control. Second, the dedicated, high-performance network on Grid'5000 provides uniform and oversized network resources, with consistent and reliable performance over time. This is not the expected behavior in a real Fog environment, and although it is not impossible, its customization requires extra effort to recreate a highly heterogeneous and variable environment. In practice, despite our effort to model the network, and its great importance on a Fog platform, we could verify that the network was not the bottleneck in our experiences. Thus, we have decided to focus on other resources, such as the CPU usage, in the next chapters.

Regarding the workload used in our experiments, all applications have the same basic 3-level structure. Although we vary the load induced by each application, more complex scenarios with different IoT applications can be studied. Another

important characteristic of our workload is that applications are stable, always running after their initial deployment. In more complex scenarios, resource usage may evolve over time, leading to different resource availability and consequently different provisioning results.

Finally, the infrastructure scale has its restrictions. Our experiments have a considerable size, about 70 nodes, but they are far from the expected scale of a real Fog environment. Moreover, the heterogeneity of our platform is limited to 2 main types of nodes: A8 nodes from FIT/IoT-LAB and the servers from Grid'5000. Thanks to Docker's settings, we were able to simulate a more heterogeneous environment, but more complex scenarios are possible.

11.4.2 Model limitations

We have proposed two heuristics, `O-FSP` and `GO-FSP`, which solve the provisioning problem in an optimized and effective manner. Throughout extensive experiences based on the `FITOR` infrastructure, we demonstrate their good performance for both applications and infrastructure. Nevertheless, a heuristic has not a guarantee performance in the worst case and may be far from the optimal. In this context, a comparison with the exact solution of the proposed model, which is optimal, could be interesting. The optimal solution is possibly calculable in the relatively small instance of our problem, at least considering the expected size of thousands or millions of nodes in a Fog infrastructure. However, the optimal solution quickly becomes unfeasible in larger instances of the problem.

Furthermore, the optimal solution obtained by solving the mathematical model is valid for the specific scenario of the model, i.e., giving the set of applications and the resources available in the infrastructure. Small variations in the values for applications and resources can invalidate the solution and change the optimal outcome. In this context, a Fog environment, and consequently our experimental setup, has some limitations.

First, measuring the availability of resources depends on the accuracy of tools used, as well as the frequency which we update the information. In a large scale Fog environment, the cost of having a proper and precise view of the whole infrastructure is unbearable. Therefore, we opt to update the available resources only once per minute in our experiments. In addition, the accuracy of retrieved information is not great, mainly for network resources such as bandwidth. Specifically in our experimental scenario, we were unable to track properly the amount of bandwidth available between nodes in our system.

Second, it is extremely difficult to obtain an accurate description of the resources used by the applications. Even if a developer is exceptionally meticulous and provides an accurate picture of the application, its accuracy will be limited to her knowledge of the application's behavior and infrastructure characteristics. In a heterogeneous environment such as the Fog, some nodes may contain specific hardware beyond the developer's knowledge. Moreover, the number of users can increase and vary over time and thus invalidate the application description.

In conclusion, in these last chapters, we have seen how to solve the provisioning of IoT applications in the Fog and the importance of doing it properly. By using the proposed heuristics (`O-FSP` and especially `GO-FSP`), we are able to minimize the provisioning cost and accept more applications on the infrastructure. Also, the non-functional requirements of applications were respected, providing a good performance in terms of applications throughput and latency. Despite the promising results, the provisioning is not sufficient to ensure that applications will be satisfied over the long term. Consequently, the next chapters of this thesis are dedicated to study the reconfiguration problem and how to adapt the initial placement of the applications to keep them satisfied throughout their existence.

Part IV

The Reconfiguration of IoT Applications in the Fog

12 Problem Statement

12.1 Introduction

In this part of the thesis, we study the reconfiguration problem, i.e., how to adapt the placement of IoT applications to keep them satisfied, even when their behavior evolves over time and they need more (or less) resources. We take an in-depth look at what happens when IoT applications are running, sharing and competing for the available resources. In this context, we will see how the environment and the information about it play an important role in the reconfiguration game, influencing the performance of the reconfiguration strategies and IoT applications.

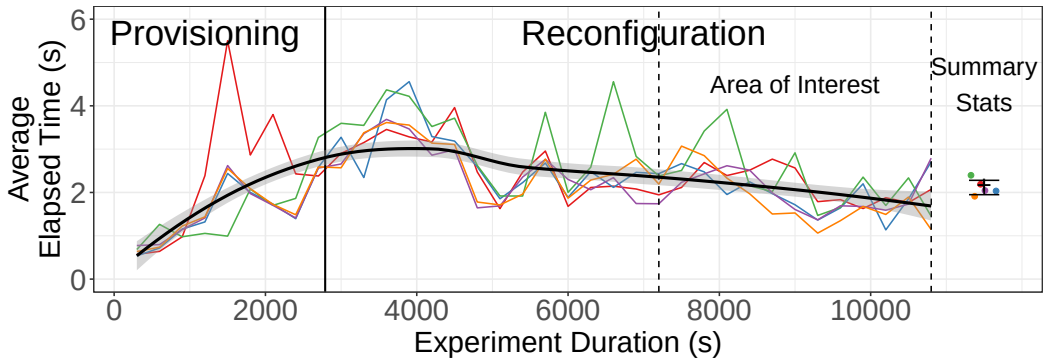


Figure 12.1: Performance evolution over time for a reconfiguration strategy. Each colored line represents the aggregated average elapsed time of all applications in a single experiment. The black line is the average performance for this strategy. In Summary Stats, we summarize the average performance, for each experiment (color points) and in general (black), during the area of interest.

Fig. 12.1 presents a typical scenario addressed in this thesis. It depicts the satisfaction metric corresponding to the average end-to-end elapsed time to process the application messages. The experiment includes two phases: provisioning and online reconfiguration. We assume that all applications arrive during the provisioning stage and are placed by the provisioning algorithm, as seen in the previous chapters. During the online reconfiguration stage, the performances of the applications are optimized considering the resource usage evolution. Note that, even if an application enters the system after the provisioning phase, it would be provisioned by the initial provisioning algorithm and, as typical IoT applications have a long life span, the system will reach a steady-state at some time in the future. In this part, we focus on strategies to be applied in the reconfiguration phase. The "Summary Stats" in

Fig. 12.1 exemplifies the results presented in the next chapters, summarizing the average performance in the last hour of the experiment ("Area of Interest").

The reconfiguration adjusts the placement of applications according to both applications and infrastructure evolution. The reconfiguration process receives as input the application's feedback with its current resources and decides whether the application should be reconfigured or not. Both reconfiguration decision and action vary according to the implemented strategy. The decision policy may be *proactive*, by regularly re-configuring the application to optimize its metric, or *reactive*, where we only reconfigure when its performance becomes unacceptable. Despite the multiple reconfiguration options available, such as horizontal or vertical scaling, in our work, we focus on the migration of the application's components.

We highlight hereinbelow some important characteristics of the interaction between IoT applications and the Fog environment which are relevant for the reconfiguration problem:

- **Distributed**: the Fog environment is distributed by nature. In the same vein, to achieve the scalability, the reconfiguration decision should be as decentralized as possible.

- **Online**: the long-running characteristic of IoT applications brings the online component to this problem, where applications change their behavior, and consequently, their resource usage over time, without any particular notification to the system.

- **Delayed information**: the large scale and resource constrained devices of the Fog environment make it very difficult to have an up-to-date and global view of resource utilization.

- **Inexact information**: the inexact information comes in two forms: i) the application's description, which may contain imprecise details due to human errors and/or shortsighted view; and ii) infrastructure measurements, either by limitation in the tools used to measure or by the delay between the measure and its utilization.

The remaining of this part is organized as follows. We start by presenting some possible approaches to cope with the online reconfiguration problem, such as online scheduling and online learning in 12.2. The details about the game, its components and how they interact are presented in 12.3. With the game in place, we propose and evaluate a variety of reconfiguration strategies in Chapters 13 and 14. Each chapter studies the problem in a different but complementary manner. In Chapter 13, we

consider a scenario with reliable and accurate information about applications and the environment, while in Chapter 14, the strategies must face the uncertainty in this information.

12.2 Related Work

12.2.1 Online scheduling

Online scheduling is a vast research domain which aims to optimize the execution of jobs on hosts. In this domain, the scheduler receives jobs that arrive over time and must be executed on a set of hosts. The arrival time of jobs is unknown, but once they arrive, its size and processing time are generally known. This lack of knowledge about job arrivals prevents the scheduler from finding optimal solutions. Therefore, considerable research has focused on finding solutions which are ρ-competitive i.e., which are never more than ρ times worse than the optimal offline solution. The ρ-competitiveness is thus a very strong worst case analysis of these algorithms.

The tuple $\langle resources|jobs|objective\rangle$ [Gra+79] is usually used to characterize a scheduling problem. For example, $\langle 1|r_j|F_{\max}\rangle$ characterizes an environment with one host running heterogeneous jobs and whose objective is to minimize the maximum flow time of jobs, i.e., the time that the job stays in the system. For this problem, a FCFS (First Come First Served) strategy is known to be optimal [BCM98].

A common HPC exploitation problem for example consists in scheduling parallel jobs in an environment with P identical hosts (e.g., a cluster) while minimizing the makespan, i.e., the time to finish executing all jobs. This problem, $\langle P|size_j|C_{max}\rangle$[1], is NP-hard but when all jobs are available up-front, heuristics with excellent worst-case guarantees are well-known (2 for list scheduling and even a PTAS for a fixed number of machines [Dro09]).

Due to NP-hardness of such class of problems, batches are often used to solve online scheduling problems in a greedy way. In this approach, a good schedule is computed for available jobs (first batch) using a guaranteed algorithm for the offline problem. All jobs arriving during the execution of the first batch are queued and constitute the new batch, which will be processed again using the guaranteed algorithm. The quality of guaranteed algorithm in the offline setting can thus often be transferred to the online setting. In [SWW95] for example, the authors show that given an algorithm $\mathcal{A}$, which is a ρ-approximation for $\langle P|size_j|C_{max}\rangle$,

[1] $size_j$ indicates that jobs are parallel and C_{max} denotes the makespan of the schedule, i.e., the time to finish executing all jobs

the batch procedure provides a 2ρ-competitive algorithm for the online version $\langle R|r_j, size_j|C_{max}\rangle$.

Different metrics and jobs characteristics are studied in other works. In [LSV08], the authors are mainly interested in stretch minimization, where the stretch S_j of a job is the factor by which its response time is slowed down compared to what it would have been if it was alone in the system. The studied problem is thus $\langle 1|r_j, \text{pmtn}|\max S_j, \sum S_j\rangle$ (pmtn means job preemption). Bender, Chakrabarti, and Muthukrishnan [BCM98] proposed heuristics for the online problem which are $\sqrt{\Delta}$-competitive for the max-stretch, where Δ denotes the ratio between the largest and smallest job in the system. The idea behind the proposed heuristics is leaving a slack from optimal max-stretch solution for each new task. This relaxation allows the heuristic to achieve the $\sqrt{\Delta}$-competitiveness. Moreover, the authors in [LSV08] show that we cannot achieve an optimal solution for both metrics (max and sum stretch) simultaneously. Indeed, optimizing for the worst case and the average is in general not possible.

Some works apply a similar approach to deal with the reconfiguration in the Fog. The authors in [Ska+17a], [Ska+17b] and [You+18] study the provisioning problem, i.e., where to run the applications' components in the Fog infrastructure. In their proposals, the provisioning is modeled as an ILP (Integer Linear Programming) problem, considering the constraints in term of resources (e.g. CPU, RAM) used by applications in the model. By solving it, either by an exact solution or a heuristic, they provide satisfactory solutions given a certain objective function. The reconfiguration problem is managed by solving the ILP model periodically. This approach is adequate for applications entering and leaving the system, but it is ill-suited to treat the evolution in resources usage of already running applications. Furthermore, this approach typically assumes that the available information is accurate and up-to-date.

In the same spirit, the authors in [Ait+19] propose the use of constraint programming to study the service placement in the Fog. A set of constraints describes the infrastructure, the applications and their requirements. Applications arrive by batch according a Poisson law and the constraint solver is called in regular period of times to solve the model and optimize the service placement. In the evaluation scenario, the proposal achieves optimal solutions with low solving times compared to a traditional ILP solver, but it strongly depends on the accuracy of the infrastructure and application models.

In [Wan+17], the placement of an application is modeled as a time-graph, where nodes represent possible hosts to run the application and arrows are associated to some predicted cost. In this context, the off-line version of the problem is solved

optimally. By building the time-graph associated to the placement in a certain time window and aggregating new jobs in batches, the authors claims that the online algorithm is $O(1)$-competitive for a broad family of cost functions. However, the effectiveness of this model strongly depends on the quality of the predicted cost for each node and arrow in the graph.

The available information about jobs is the main challenge when applying online scheduling strategies to study the reconfiguration in the Fog. Usually, the proposed approaches in this domain consider a full information scenario, in which the scheduler knows, after the arrival of the job in the system, its exact size and amount of resources needed. Also, this information is stable, allowing the scheduler to apply the offline algorithms to the set of jobs currently available. The uncertainty is mostly concentrated in knowing the arrival date of jobs. However, IoT applications running in the Fog face a much more uncertain environment, having a long lifespan and unpredictable work size. These factors hinder the use of online scheduling algorithms.

12.2.2 Online learning

Online scheduling strategies seen in the previous section are studied in worst-case scenario (they are compared to the best possible offline solution and jobs may arrive at any time) through adversaries. It is also commonly assumed that job characteristics are disclosed at arrival by the scheduling algorithms and that the objective function of each job (stretch, flow, etc.) can be perfectly calculated. However, all this information is not always available in the Fog environment. In this context, alternative approaches may be necessary to cope with the reconfiguration problem.

Some papers propose to solve the reconfiguration problem based on migrations, which are triggered by some threshold based metric. In [SA16], the authors propose the migration of proxy VMs, which link IoT devices to the target application, based on the bandwidth usage by the IoT device and the migration process. The Foglet programming infrastructure is propounded in [Sau+16]. With it, the authors propose two migration strategies: i) QoS-driven which monitors the latency between Foglet agents to initiate the migration process; and ii) Workload-driven which monitors the utilization of resources (CPU, memory, etc.) to trigger the migration. In both papers, the threshold metric relies on the monitoring of resource utilization to react properly to performance degradation.

Moreover, a complementary threshold based strategy is presented in [Ran] and [Wan+18]. In [Wan+18], the auto-scaling mechanism triggers a vertical scaling

by adding more CPU and RAM resources to containers running the application. While in [Ran], although both vertical and horizontal auto-scaling mechanism are supported, application developers usually relies on horizontal scaling, increasing and/or decreasing the number of replicas based on the current resource utilization.

The aforementioned reactive approaches do not optimize any well-defined objective. To close the gap between the exact, but inflexible, objective function from online scheduling and the lack of objective from reactive strategies, online learning allows to study a broader scenario where objective functions may vary over time, e.g., following a (possibly non-stationary) stochastic process. In this context, the Multi-Armed Bandit (MAB) problem has received remarkable attention in the last years, with application in many research fields. MAB inspiration comes from the casinos, where an agent is facing a set of slot machines with unknown probability rewards, and she wants to choose a strategy that maximizes his long-term cumulative income. More precisely, in a MAB problem an agent is offered a set of arms $\mathcal{A} = \{1, ..., A\}$ and at each time step $t = \{1, 2, ..., T\}$, the agent selects an arm $a_t \in \mathcal{A}$ and receives a reward $r_t = u_t(a_t)$. The objective in this game is to maximize the cumulative reward $\sum_{t=1}^{T} r_t$. In order to compare the performance of different strategies, the notion of regret is introduced. Given the best possible arm a^* in the hindsight of the horizon T, the regret is defined as

$$R(T) = \sum_{t=1}^{T} (u_t(a^*) - u_t(a_t)). \tag{12.1}$$

If the agent was given access to an oracle that would indicate which machine is the best, he would only play this machine all time and would have a regret of 0. This would be a pure *exploitation* phase, where the agent knows with very high confidence about his future gains. Unfortunately, most of the time, an oracle isn't available and the agent has to try out all possible slot machines (or arms) to discover which one is the best. This is the *exploration* phase. By exploring, the agent has more information to determine the best arm but loses opportunities to obtain the best reward (i.e., exploiting). The notion of regret is used to compare the proposed strategy for the MAB problem in different scenarios. If the agent fails to identify the best arm, his regret will grow linearly with T. However, if he manages to identify the best arm, his regret will not grow any more. In practice, it is impossible at a given time for an agent to know with certainty which arm is the best since this decision can only be based on the outcomes observed so far; hence the need to find the right trade-off emerges between *exploration and exploitation*. This is why only strategies with sublinear regret are of interest and such strategy should never stop exploring (at least a bit). There are two main classes of MAB problems which

are differentiated according to the behavior of the reward perceived by agents: stochastic and adversarial.

In the stochastic setting, the reward of each arm $a \in \mathcal{A}$ is associated to an unknown probability distribution. In consequence, the goal of the agent is to discover these distributions and exploit the arm with highest expected reward. In this context, the UCB (Upper Confidence Bound) [ACF02] algorithm provides a simple solution for the MAB problem. As the name suggests, the idea of UCB is to compute an upper confidence bound on the mean reward of available arms. UCB is a deterministic strategy which exploits the arm with the highest bound, which will force the exploration of other arms when the uncertainty is too high. Fig. 12.2 illustrates the upper confidence bound for each of the two arms (A and B), in different time steps. UCB will select the arm A in time steps $t = 10$ and $t = 30$, so that it can decrease uncertainty about A's reward. On the other hand, in time step $t = 20$, UCB is selecting the arm B which offers the best reward. It has been proven that UCB achieves a regret in the order of $O(\log T)$, which is optimal [ACF02].

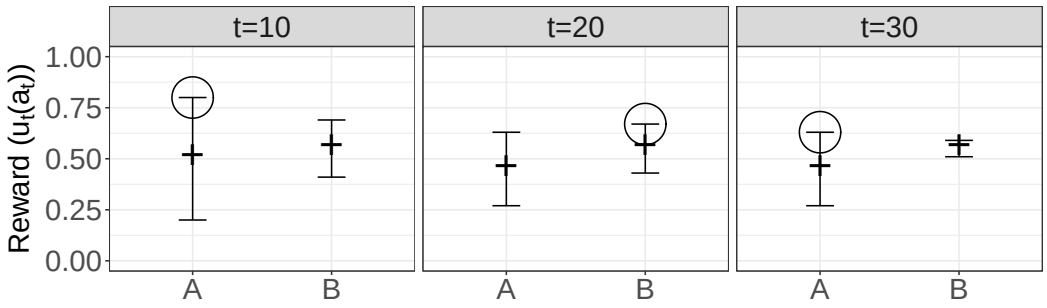


Figure 12.2: UCB arm selection in different time steps. At each time step t, UCB selects the arm with highest reward plus uncertainty factor.

Nevertheless, in many real problems the reward does not follow a stationary probability distribution and instead depends on external exogenous factors. Adversarial bandits address this situation by studying the case where the agent is facing an adversary who tries to minimize the cumulative agent's reward. So, the reward $u_t(a)$ at instant t does not follow a statistical distribution, but it is instead determined by the adversary right before letting the agent decide which arm a_t she will play. In this case, there is no single optimal arm anymore and any deterministic strategy, such as UCB, can be exploited by the adversary to minimize the agent's gain. The state of the art algorithm for adversarial bandits is EXP3 [Aue+02], which stands for Exponential-weight algorithm for Exploration and Exploitation. EXP3 works by maintaining a probability vector with weights for each arm. At each time step t, the agents use this vector to decide randomly which arm a_t she will play next. The received reward $u_t(a_t)$ is then used to update the relevant weight in the vector. In this hard scenario, EXP3 obtains a regret of $O(\sqrt{AT \log A})$ [Aue+02].

In a MAB setting, the agent chooses the next action a_t from a predefined set of discrete actions $\mathcal{A}$. On the other hand, in the Bandit Convex Optimization (BCO) framework, the agent chooses a_t from a continuous space in $\mathbb{R}^n$ and has access only to the bandit feedback $f_t(a_t)$. In this context, the authors in [CG19] uses the BanSaP (Bandit Saddle-Point) algorithm with partial feedback to study the offload of tasks in a Fog environment. BanSaP is also extended to take into account, and minimize, the number of violations of user's defined constraints. In a scenario with one point feedback, BanSaP achieves a regret of $O(T^{3/4})$. Unfortunately, the BCO framework is not suitable for our environment because we have a limited and discrete set of hosts to which applications may migrate.

12.3 Game Overview

From the viewpoint of online learning, we consider a multi-agent setting (a *game*) where J applications share $\mathcal{R}$ hosts ($\mathcal{R} = \{r_1, r_2, .., r_R\}$). In each time step t of T in the game, each *agent* (or application) j selects one *action* (or host) a_t^j among $\mathcal{A}^j \subset \mathcal{R}$ possible actions[2]. Note that each application takes its decision about a_t^j independently, in parallel and without knowing the decision of other applications. The set of all placements at time step t is denoted by $\pi_t = \{a_t^{(1)}, a_t^{(2)}, .., a_t^j\}$. Given the current placement π_t, the applications will execute and measure their incurred cost $C_t(a_t^j|\pi_t)$. This cost not only depends on the current host assigned to the application but also on other active applications at the same time. With this personal feedback, agents restart the process by selecting (or not) a new host to execute.

Algorithm 4: The Game

```
1: for t = 0 to T do
2:    for all j = 1 to J do in parallel
3:       app j chooses host a_t^j = r_i ∈ A^j
4:       app j observes incurred cost C_t(a_t^j|π_t)
5:    end for
6: end for
```

The game is described in Algorithm 4. In this type of game, we are interested in the long-term cumulative performance of the system. Thus, our objective is to minimize the *overall cost* over π: $\sum_{j=1}^{J} \sum_{0}^{T} C_t(a_t^j|\pi_t)$. Different cost functions may be considered, such as cost, end-to-end delay or application throughput. In 12.3.1, we present the performance metrics considered to study the reconfiguration problem in our context.

[2]Note that we adopt a simplified notation as an application has several actors to place. However, we believe that this is enough to convey intuitions without burdening the notations. Also, every host in $\mathcal{A}^j$ has enough capacity to run the application if it was dedicated to it.

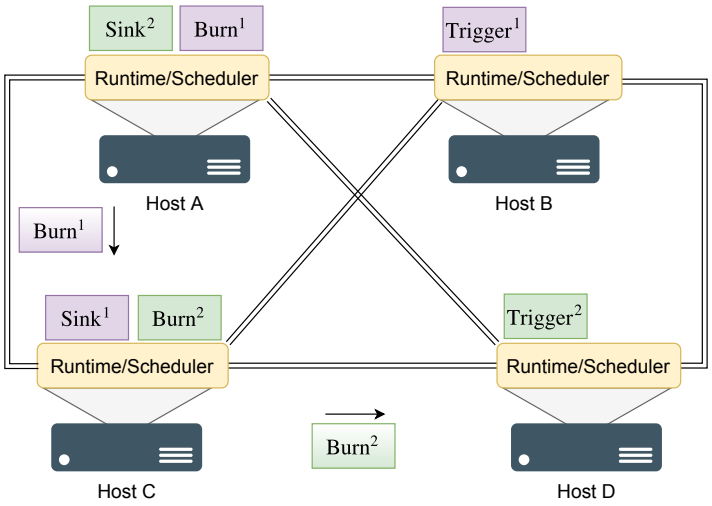


Figure 12.3: The reconfiguration game in place

In Fig. 12.3 we detail a simplified game with 2 applications sharing 4 hosts. At a given instant, the host A decides to send the component Burn from application 1 to host C, while the Burn from application 2 is migrated from host C to D. The scheduler may decide to migrate only one component of the application or all components, depending on its strategy[3]. We must highlight two important points in these interactions:

- **Scheduler**: for simplification, we describe the interactions between hosts and applications, but the scheduler (or Calvin's runtime) is the agent responsible for managing the applications running on the host. Although done independently for each application, it is the scheduler who takes the decision to migrate the application to another host.

- **Non-negotiable** migrations: the scheduler decides to send away applications to other hosts without prior communication. This can be seen in the figure when host C receives Burn^1 even if it is already sending Burn^2 to another host due to its current load.

- **Distributed** decisions: the host (or scheduler) controls its running applications independently. The migrations of Burn^1 and Burn^2 may happens concurrently.

These asynchronous and non-negotiable migrations may lead to extra migrations and, consequently, degraded performance. This effect is amplified due to the characteristics of the Fog environment, such as the delayed and inaccurate information.

[3]In the rest of this part, we use the term application to refer to the migration decision, even if only one of its components is migrated.

12.3.1 Performance metrics

In the rest of the reconfiguration study, we consider three performance metrics:

1. Average **elapsed time**: represents the average end-to-end delay of messages received in last time step.

2. Total **time above threshold**: describes the total time in seconds where the performance of applications wasn't satisfactory and exceeds the threshold.

3. **Number of migrations**: corresponds to the total number of migrations performed by applications, higher numbers incur in important downtime for applications.

The primary cost function considered in our paper is the *average elapsed time* but the other two metrics are also monitored as they reflect interesting aspects of the strategy's performance. In our experiments in the next chapters, we have two application classes (*intensive* and *calm;* more details in Sections 13.1.2 and 13.1.2). When presenting the results, we split the cost among each class of application (in the left the intensive applications which consume a high amount of resources, while in the right, the calm ones which have a small impact over the system load). In the "Summary Stats" part of Fig. 12.1, we present a typical result, where the "+" signal is the mean performance across all experiments, aggregated as explained above. The confidence interval is calculated as $mean \pm 2 * se$ (standard error). Despite the uncertainty of some results due to their high variance, we believe they convey us a good notion of the actual performance of strategies.

13 Reconfiguration in a Well-informed Environment

In this chapter, we evaluate a scenario where users have a good working knowledge of application and infrastructure characteristics, providing an accurate estimation about the resource utilization. In this scenario, resources are shared by applications whose performance depends mainly on concurrent applications on the same host. To analyze the evolution of application's behavior, the experiments last for *1 hour* after the initial deployment of all applications (after provisioning phase on Fig. 12.1).

13.1 Describing the Environment

In this section, we give insights into the experimental setup used during the study of the reconfiguration problem in this well-informed environment. We describe the parameters that characterize platform, workload and orchestrator.

13.1.1 Platform

The platform used in our reconfiguration tests is composed of *67 nodes*. We use *50 nodes* from FIT/IoT-LAB to represent the sensors, and *17 nodes* from Grid'5000, forming the Fog layer. From FIT/IoT-LAB, we select nodes from Grenoble site, each node contains an ARM A8 microprocessor and 256MB of RAM.

The 17 Grid'5000 nodes are part of the *suno* cluster in Sophia Antipolis and they are characterized by 2 CPUs Intel Xeon E5520, with 4 cores per CPU and 32GB of RAM[1]. From them, 1 node is reserved to be our `Prometheus` server, 1 node is the master for Calvin's applications and 15 slave nodes are responsible for running the IoT applications. To obtain an estimation of CPU power in MIPS of our nodes, we run a matrix multiplication application in each node of the infrastructure. Accordingly to this, each Grid'5000 node provides about 170 MIPS of CPU power.

[1]Note that we use Docker runtime options to enforce memory limits to our Fog nodes. In our setup, the Calvin's master node has 4GB of RAM and slaves have 2GB.

13.1.2 Workload

Our workload starts with the description of the application used in our experiments. As seen in Section 8.1.2, the application used follows the 3-level model, with 1 Trigger generating messages to be processed by 1 Burn component, and 1 Sink actor collecting and storing the messages. The characteristics of these actors allow us to describe the applications running on our platform, their heterogeneity and specially their evolution in resource utilization over time. To consider all these parameters, our workload is described by:

- **Application load**: we distinguish two kinds of applications:
  ▶ *Intensive*: these resource-consuming applications send a large amount of messages (*5 messages per seconds* with a payload of *1024 bytes*) to be processed, each incurring *30 MI* (millions of instructions). The intensive application load is calculated so that that each Fog host on our platform can run only one application satisfactorily at once.
  ▶ *Calm*: these applications send fewer messages (only *1 message/s* with the same payload) which require low processing (*10 MI*) capacity.

- **Application heterogeneity**: is the mix of applications present in the workload. In our experiments, we favor the heterogeneity, opting for a $50\%/50\%$ mix between intensive and calm applications.

- **System Load**: we calibrate our setup to have a *heavy* load, where the system is almost saturated, as we can see in Fig. 13.1 which shows that almost all 15 nodes in infrastructure are running intensive applications. In practice, the *heavy* load represents 50 applications concurrently running on the infrastructure.

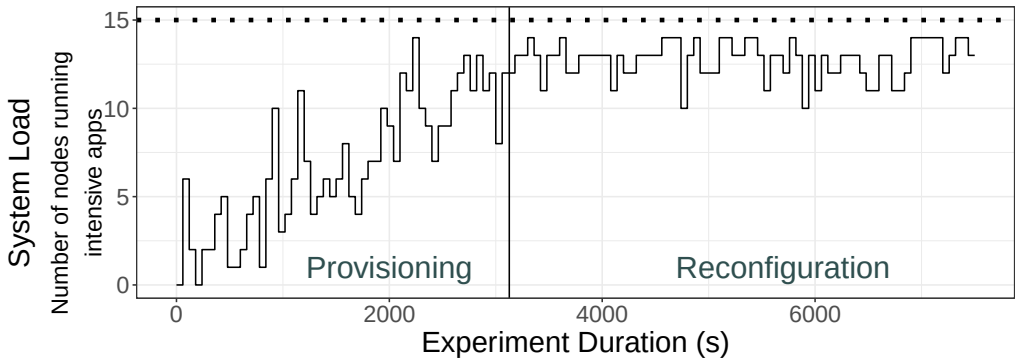


Figure 13.1: Workload - System Load (for an experiment). On the y-axis, we present the number of hosts that are close to saturation and cannot run more intensive applications. The dotted line marks the number of hosts in the system (15)

- Application arrival interval: although not relevant for reconfiguration experiments, applications arrive in a 60 seconds interval during the provisioning phase.

- **Application threshold**: in our setup, the threshold is set to *2 seconds*. Above this threshold, the applications are not satisfied with their current placement and should be migrated.

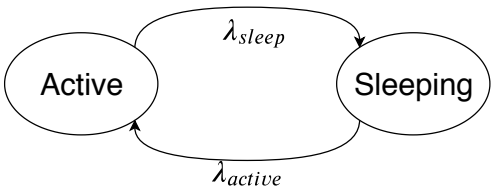


Figure 13.2: Workload - modeling application evolution

- **Churn**: as illustrated in Fig. 13.2, we modeled the churn as a 2-state Poisson process, where state changes are exponentially distributed. The parameters λ_{active} and $\lambda_{\text{sleeping}}$ control the rate of state change. The churn is implemented by activating and disabling the Trigger component in the application. In our experiments, we considered applications with a mean active/sleeping time $(1/\lambda)$ of 300 seconds. Fig. 13.3 illustrates the time spent in each state for an application in our setup.

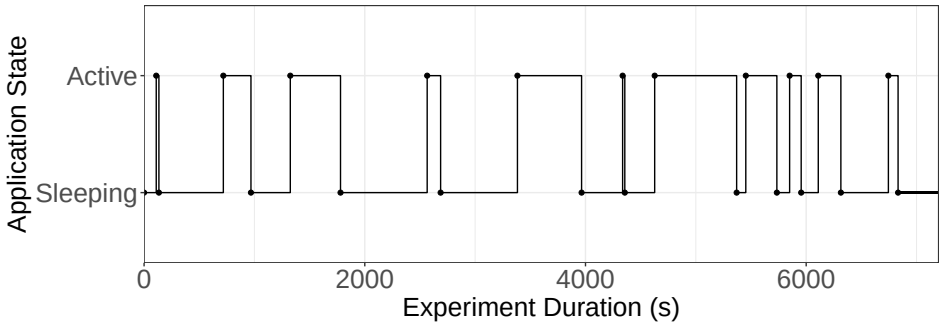


Figure 13.3: Workload - Churn. Example of state transitions for an application with mean active/sleeping time $(1/\lambda)$ of $300s$.

We highlight that the parameters chosen for our workload lead to a quite complex and difficult environment to handle. The elevated number of intensive applications causes a high charge in the infrastructure, consequently leaving a smaller margin for improvements. Considering the number total of applications (50), their heterogeneity (50% calm/50% intensive) and churn, on average, we have 12.5 intensive applications running at the same time, which leads to a high system load as illustrated in Fig. 13.1. Besides, as we can see in Fig. 13.3, the application's behavior is unpredictable and has an important impact on the overall performance of all applications.

► **Low-level application description:** hereafter we describe the main low-level parameters in our setup.

- **Requirements for actors**:
  - Trigger: requires an amount of CPU proportional to its token interval, i.e., 4 MIPS for calm applications and 7 MIPS for intensive.
  - Burn: CPU is proportional to the processing effort per token. Calm applications require 10 MIPS while intensives demand 150 MIPS.
  - Sink: requests memory requirement equals to 9 MB for intensive applications and 1.8 MB for calm ones.

13.1.3 Orchestrator parameters

- Provisioning strategy: corresponds to the algorithm used in the provisioning phase to decide the initial deployment of applications. In this scenario, we use the *GO-FSP* algorithm as proposed in Chapter 11.
- **Reconfiguration strategy**: this parameter is defined later in this chapter.
- **Maintenance interval**: remember that this parameter defines the frequency at which the orchestrator verifies the satisfaction of applications. We chose the *5 seconds* maintenance interval to have a good responsiveness of the reconfiguration algorithms.
- **Monitoring interval**: this parameter is kept the same, updating the information about resource usage (CPU, RAM) each *60 seconds*.

13.2 Evaluation

With the playground set and the game rules defined, we are able to start studying the possible reconfiguration strategies. In the next sections, we will detail each strategy, analyze its performance and, if possible, indicate the improvements that can be made to get better results.

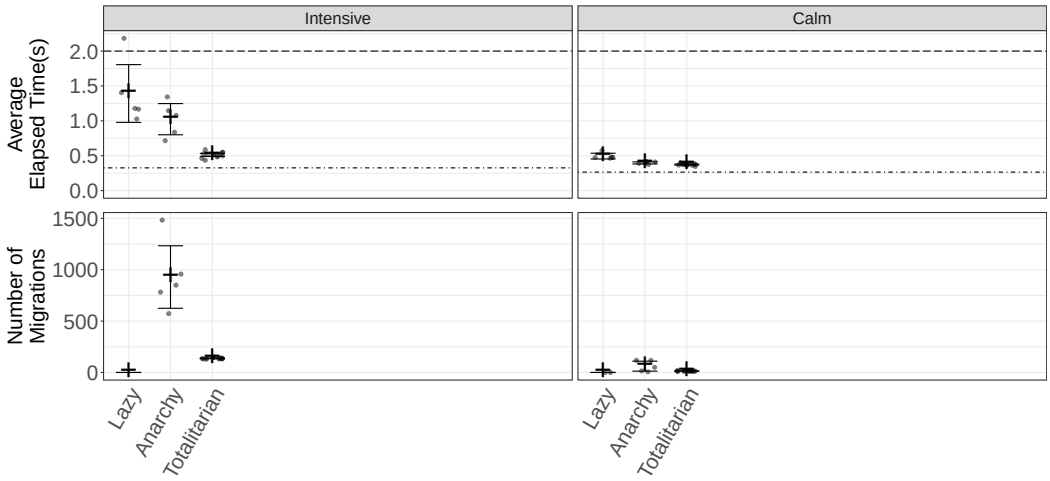


Figure 13.4: Performance evaluation for baseline strategies. The $2s$ horizontal dashed line represents the threshold above which applications request migration, while the bottom line represents the minimum response time when operating on a dedicated server. Both Lazy and Anarchy strategies have a poor overall performance.

13.2.1 Baseline strategies

The first step to proceed with the evaluation of our case study is the definition of the baseline strategies. This section details and presents the result of the three such strategies, which vary in terms of policy and information used, ranging from simply maintaining the initial placement to having a total knowledge of the environment.

Lazy: no reconfiguration

Our first baseline, called Lazy, is the simplest possible strategy: maintaining the initial placement. Consequently, the applications remain in the initial host decided by the provisioning strategy, even if the current performance is degraded and the application requests its migration. In this case, the performance depends on the initial provisioning strategy, `GO-FSP`, and the application's resource usage pattern.

As illustrated in Fig. 13.4, the Lazy strategy performs poorly, as shown by the huge average elapsed time of intensive applications. On the other hand, calm applications suffer only mildly from this situation because they use very few resources.

Anarchy: total freedom

The second baseline strategy, Anarchy, gives total freedom to applications to decide when they should migrate. The anarchy is a **reactive** and **greedy** strategy, which is implemented as follows: i) each application monitors its elapsed time and, when it reaches a predefined threshold, notifies the scheduler of the host on which it currently runs; ii) the scheduler runs a maintenance procedure in a fixed time interval, collecting all applications that notified their bad performance; iii) `GO-FSP` algorithm is executed for each application to find the best host to run it. `GO-FSP` makes its migration decision based on the description of the application provided by user, as well as its current view of the platform's status, with the delay and inaccuracy incurred by the monitoring tools.

We can see in Fig. 13.4 a slight improvement for intensive applications compared to the Lazy strategy. In this case, a visible drawback is the large number of migrations done by these applications.

Totalitarian: clairvoyant dictatorship

The third baseline proposed for strategy comparison is called Totalitarian. In this strategy, an **oracle**, centralized and fully informed, controls the reconfiguration of all applications, dictating where they should run at each moment. Although not implementable in a real Fog scenario, this strategy gives a good target for the best possible performance for applications.

In particular, the Totalitarian is a **proactive** strategy which has a perfect knowledge about both infrastructure (the total of resources of each host is known, as well the remaining available resources) and applications (the strategy knows exactly when the application is active and the amount of resources used by it).

As input, the Totalitarian algorithm receives all wake-up/sleeping events of all applications. Then, just before the application is activated, the orchestrator checks whether the current host running the application is capable of bearing the additional load incurred by this application. If not, the `GO-FSP` algorithm is executed to find a new host. By acting proactively, the strategy can obtain an excellent performance and meet the QoS as required by the user. Note that we ensure this algorithm runs on a sufficiently fast machine.

As expected, Fig. 13.4 shows the excellent performance of the Totalitarian strategy. With a reduced number of migrations, it is capable of satisfying both intensive and calm applications, while improving by a factor of at least 2, the elapsed time for

intensive applications. However, Totalitarian is a centralized and fully informed strategy, which makes it unsuitable for the Fog environment. Therefore, in the next section, we will study some distributed and less informed learning strategies.

13.2.2 Online learning strategies

In this section, we describe some algorithms based on online learning to solve the reconfiguration problem. We evaluate the performance of the algorithms in this realistic Fog environment, identifying the main issues that influence the performance and presenting the improvements we had to make in the algorithms to mitigate them.

UCB: stochastic

The first learning strategy we explored is UCB (Upper Confidence Bound) which has excellent regret properties in the stochastic case [Bel+18]. UCB is a **proactive** algorithm which works with minimal information, trying to optimize the performance of applications by looking only to their feedback. For each application j in our environment, UCB selects the next host a^j_{t+1} to run j, as follows (with tuning parameter $\alpha = 3$) as convergence is guaranteed only when $\alpha > 2$):

$$a^j_{t+1} = \arg\max_{a \in \mathcal{A}^j} \left\{ \hat{\mu}^j_{a,t} + \sqrt{\frac{\alpha \log t}{2 n^j_a}} \right\}, \tag{13.1}$$

where n^j_a is the number of times the action was selected, and where the first term in (13.1), $\hat{\mu}^j_{a,t}$, is the empirical mean reward observed by application j for host a, calculated as

$$\hat{\mu}^j_{a,t} = \frac{1}{n^j_a} \sum_{a \text{ chosen at time } t' < t} \mu^j_{a,t'}.$$

$\hat{\mu}^j_{a,t}$ drives the exploit of the host with the highest empirical reward so far. On the other hand, the second term in (13.1) drives the exploration in the algorithm, indicating the confidence of the algorithm on the current reward for each host.

Equation (13.1) expects a positive reward which it aims to maximize. However, we use the average elapsed time to drive the performance of our applications. To translate the average elapsed time e to a positive reward, we use the following equation

$$\mu^j_{a,t} = \max\left(0, \frac{e_{max} - e}{e_{max}}\right), \tag{13.2}$$

where $e_{max} = 10$ is the highest value for which the host receives a reward for running the application. We have chosen $e_{max} = 10s$ to have a positive reward

and differentiate nodes whose performance is close to the 2-seconds application threshold.

Furthermore, our implementation of UCB only has a partial view of the system since we reduced the number of possible hosts for each application to $|\mathcal{A}^j| = 5$, randomly chosen to avoid selection bias and to distribute the applications among available hosts. Such a strategy was adopted to reduce the search space and to accelerate the learning rate[2]. Moreover, to mitigate the effect of migrations on the elapsed time, we increased the maintenance interval from 5 to $10s$. Note that the same transformation (Eq. (13.2)), reduction of search space and maintenance interval are used in all learning algorithms we present.

Unfortunately, UCB assumes a stochastic setting and may not perform very well in a game context where each agent has to adapt to the others. In our experiments, UCB indeed has bad performance for both intensive and calm applications, as seen in region A of Fig. 13.6. The elapsed time is comparable to Lazy which does nothing, even for calm applications that are less resource demanding. This effect is explained by the high number of migrations done by the applications.

Moreover, the performance of applications running on each host is similar, depending more on the current applications running on the host. So, hosts are indistinguishable in terms of performance and UCB tends to alternate uniformly among all available hosts, unable to learn which are the best.

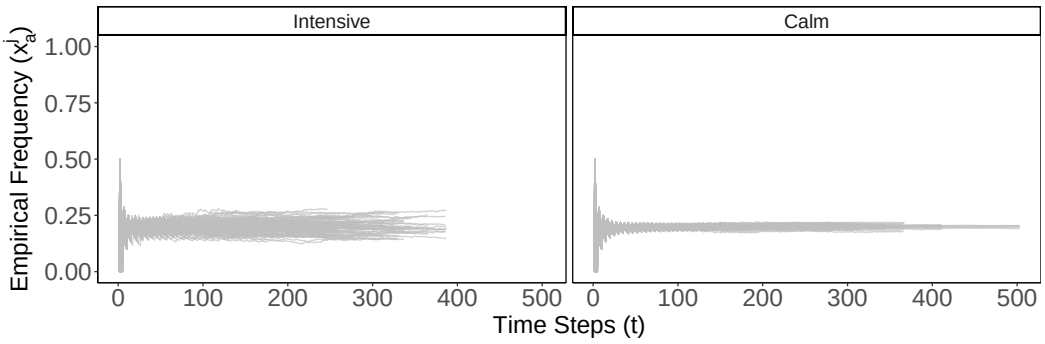


Figure 13.5: UCB learning rate for an experiment in a well-informed environment. Each line represents the frequency an application has selected determined host ($x_a^j(t) = n_a^j/t$). We highlight with a different color, the applications that were able to identify the best hosts to run (none in this case).

In order to analyze the learning process, we introduce the empirical frequency

$$x_a^j(t) = \frac{n_a^j}{t}, \tag{13.3}$$

[2]The reduction in the search space ($|\mathcal{A}^j| = 5$) allowed us to evaluate and study the different learning strategies. A proper study should be carried out to determine possibly better values for $|\mathcal{A}^j|$.

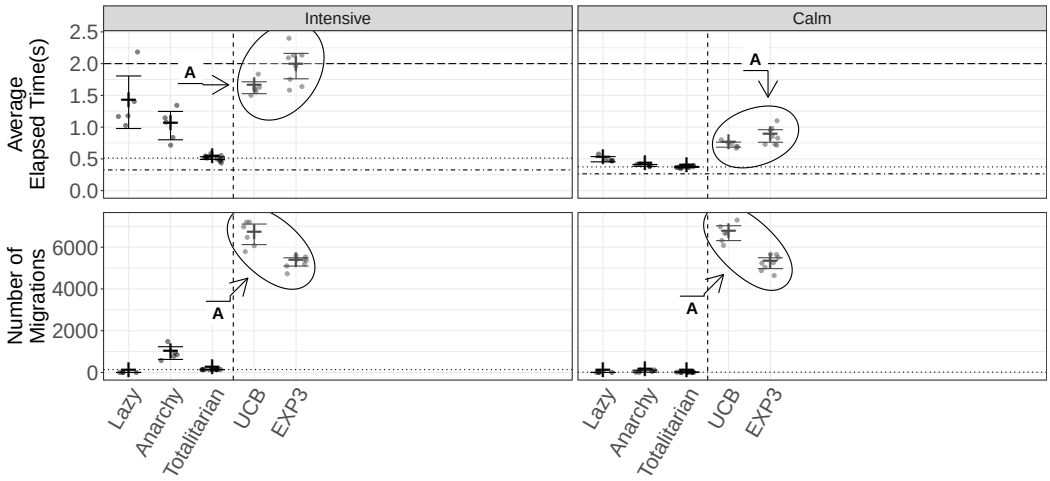


Figure 13.6: Performance evaluation for UCB and EXP3 strategies. This figure complements Fig. 13.4 by adding the results for UCB and EXP3. Note that the y-axis scale has changed and that a horizontal dotted line now indicates the performance of the Totalitarian strategy and serves as a target lower bound.

which represents the frequency each host was selected by each application. Fig. 13.5 presents the empirical frequency for all applications in a single test execution. We can see that none of the applications is able to distinguish the best host to run, selecting each one roughly the same number of times. In conclusion, the results obtained show the unfitness of UCB in our context.

EXP3: adversarial

In an adversarial context, EXP3 (EXPonential-weight algorithm for EXPloration and EXPloitation) is known for having good regret properties. In terms of information used by the algorithm, it is similar to UCB, i.e., a **proactive** algorithm that uses only bandit feedback. In an adversarial scenario, EXP3 randomizes its arm selection to minimize the regret against the adversary. This is done by maintaining a reward vector y and a probability vector p for each application j, as follows

$$y^j_{t+1} = y^j_t + \eta \hat{v}^j_t \tag{13.4a}$$

$$p^j_{t+1} = \Lambda(y^j_{t+1}), \tag{13.4b}$$

where the logit choice map Λ is given by

$$\Lambda(v) = \frac{(\exp(v_a))_{a \in \mathcal{A}}}{\sum_{a \in \mathcal{A}} \exp(v_a)}$$

In each step t, the application selects a host based on the probability vector p and will update its reward vector y, taking a step of $\eta = 0.1$. However, to update the

reward vector, we need an unbiased estimator for the feedback vector $\hat{v}$. This is achieved by the importance sampling technique

$$\hat{v}_{a,t}^{j} = \begin{cases} \frac{\mu_{a,t}^{j}}{p_{a,t}^{j}} & \text{if } a = a_t^j \\ 0 & \text{otherwise} \end{cases} \tag{13.5}$$

By dividing the observed feedback μ by the probability p, we obtain an unbiased estimator of the real feedback vector μ, i.e., $\mathbb{E}[\hat{v}_{a,t}^{j}] = \mu_{a,t}^{j}$.

The results for EXP3 are presented in region A of Fig. 13.6 and its performance is disappointing. In a close analysis, we could observe that the applications keep moving around the available hosts, degrading and creating instabilities in the performance. Consequently, EXP3 has difficulty to learn and to reach the equilibrium in this congestion game.

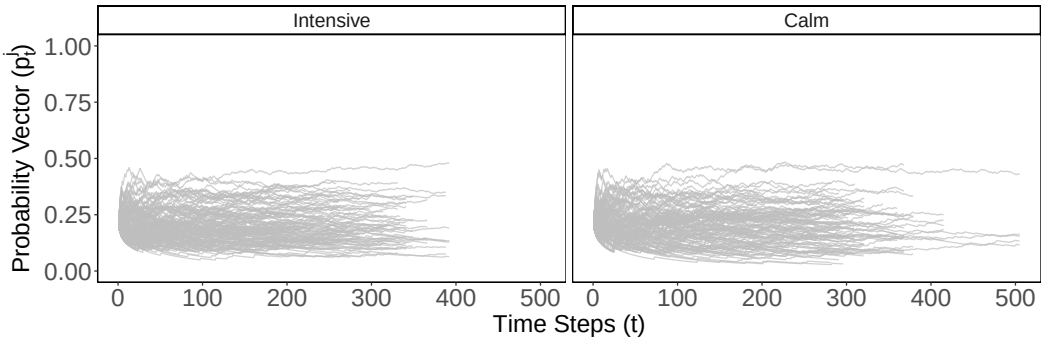


Figure 13.7: EXP3 learning rate for an experiment in a well-informed environment. The figure shows the probability vector p_t^j, each line represents the probability that an application has to select each host. We highlight with a different color, the applications that were able to identify the best hosts to run (none in this case).

In Fig. 13.7, we see for a given experiment, the evolution of the probability vector p, as defined in Eq. (13.4b). Although the greater variability, EXP3's learning behavior is similar to UCB (presented in Fig. 13.5). There is no learning curve, and no host has a distinguishable better performance or greater probability of being selected. This leads to the high number of migrations seen in Fig. 13.6.

From the results of UCB and EXP3, we observe that both strategies undergo the same learning problem. Moreover, we can clearly see the impact of excessive exploration on the overall performance of these algorithms. Hence, in the following, we evaluate two adaptations of UCB and EXP3 that try to limit the number of migrations through a Migration-Control (MC) mechanism.

UCB-MC: migration control

Usually, online learning algorithms, such as UCB and EXP3, consider that the agent can switch arms for free, i.e., without impact over the perceived reward. However, it is natural to see that, in many real cases, this assumption does not hold true and the agent needs to pay a cost $\gamma > 0$ when switching arms ($a^j_{t+1} \neq a^j_t$). Especially in our case, migrations have a non-negligible cost associated with moving application components from one host to another. Moreover, it is extremely difficult to measure this switching cost γ since it depends on the status of both application and platform.

UCB-MC (UCB with Migration-Control) is the implementation of the state-of-the-art UCB2 algorithm [ACF02] to deal with the stochastic scenario with switching costs. UCB2's main difference with UCB is that the plays are divided into epochs so that each arm is played during N consecutive time steps, where N is an exponential function of the number of times the arm was played so far. By doing so, UCB2 reduces the switching cost from $O(T)$ to $O(\log(T))$ [ACF02] while maintaining the $O(\log(T))$ optimal regret.

UCB-MC is still a **proactive** algorithm which tries different configurations to find the best one as follows:

$$a^j_{t+1} = \arg\max_{a \in \mathcal{A}^j} \left\{ \hat{\mu}^j_t + \sqrt{\frac{(1+\alpha)(1+\ln\left(t/\tau(r^j_a)\right))}{2\tau(r^j_a)}} \right\}, \tag{13.6}$$

where $\alpha = 0.1$[3], r^j_a is the number of epochs played by host a so far and τ is the following exponential function

$$\tau(r) = \lceil (1+\alpha)^r \rceil \tag{13.7}$$

We can see in region A of Fig. 13.9 that UCB-MC indeed does fewer migrations compared to UCB and EXP3. However, the elapsed time is not significantly improved. Although the number of migrations is decreased, it reduces slowly in time, since the hosts are chosen in an almost uniform way. In Fig. 13.8, we can see that UCB-MC has the same learning issue as UCB). Consequently, the elapsed time is in the same order of magnitude as for UCB.

[3]The α in Eq. (13.6) has the same status but not exactly the same semantic as the one in Eq. (13.1), wherefore they have different values.

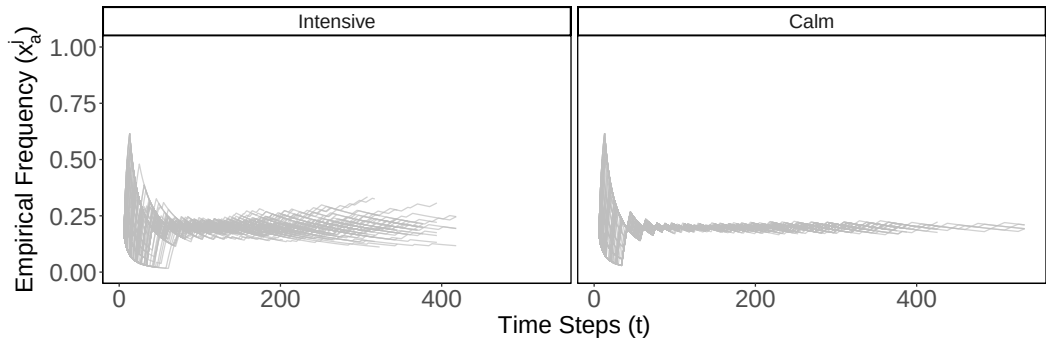


Figure 13.8: UCB-MC learning rate for an experiment in a well-informed environment. Each line represents the frequency an application has selected determined host ($x_a^j(t) = n_a^j/t$). We highlight with a different color, the applications that were able to identify the best hosts to run (none in this case).

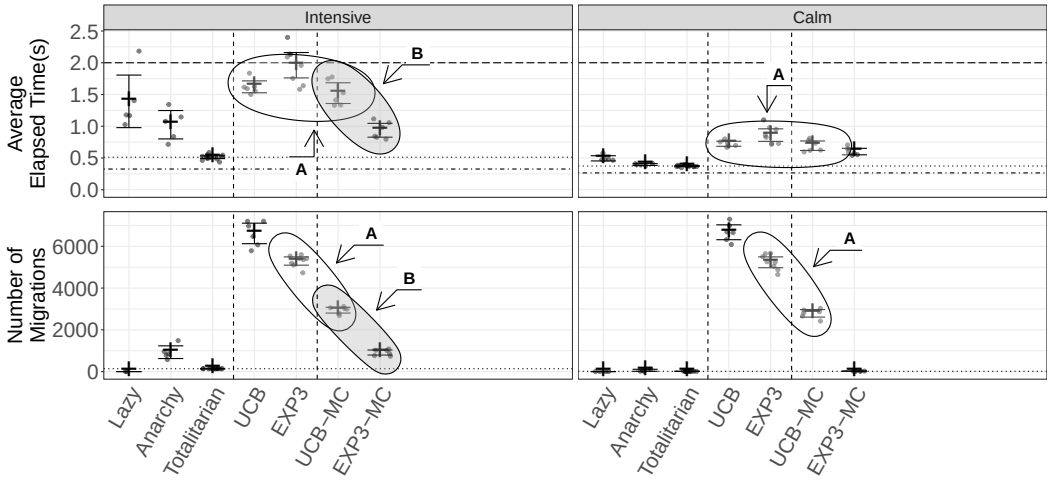


Figure 13.9: Performance evaluation for UCB-MC and EXP3-MC strategies. This figure complements Fig. 13.6 by adding the results for these strategies.

EXP3-MC: reactive migration control

Just like for the stochastic case, variants of EXP3 have been proposed [Aue+02] for the adversarial setup where switching from an arm to another has a cost which should be minimized as well. Unfortunately, the regret for the adversarial case with switching costs is $O(T^{2/3})$ instead of $O(\sqrt{T})$ [CDS13]. For this reason, we propose a different approach for EXP3 to handle with the switching cost. EXP3-MC (EXP3 with Migration Control) follows the same algorithm logic as EXP3, but the explorations are done only when the application has an unacceptable performance, i.e., it exceeds the threshold defined by the user. By doing so, EXP3-MC becomes a **reactive** algorithm, changing the placement of an application only when it is really needed.

Fig. 13.9, region B, shows a significant improvement in the elapsed time of intensive applications, along with a great reduction in the number of migrations done by applications. This result reinforces our understanding about the cost of explorations on the overall performance of applications. By being less aggressive in the learning and doing less migrations, the mean elapsed time for all applications tends to improve.

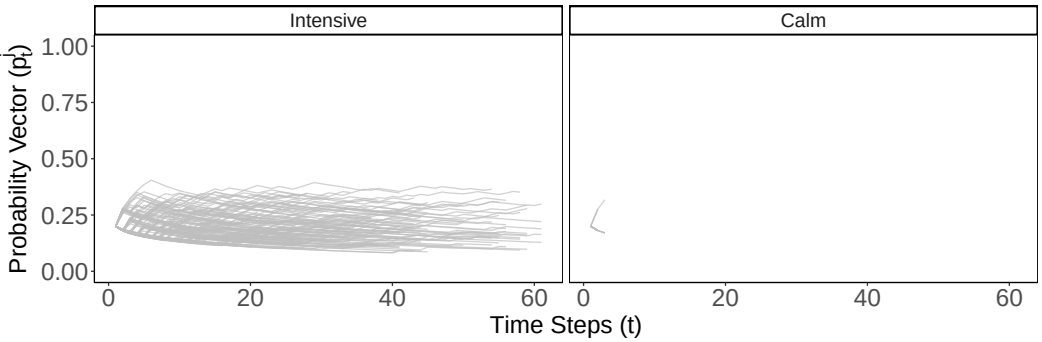


Figure 13.10: EXP3-MC learning rate for an experiment in a well-informed environment. The figure shows the probability vector p_t^j, each line represents the probability that an application has to select each host. Calm applications rarely exceed the threshold and so do not migrate. Consequently, the p_t^j does not evolve over time.

One obvious drawback of such solution is the learning rate, as presented in Fig. 13.10. With the reduced number of explorations applications do, EXP3-MC is not capable of distinguishing among available hosts, and finishes by choosing almost uniformly the next host to run the application. This learning effect is amplified for calm applications, which use very few resources and do not migrate at all.

Hedge-MC: trading feedback for estimates

So far, the learning process of all the algorithms relied on partial information, i.e., they have access only to the feedback of current host running the application. In a full information context, where the algorithm has access to all feedback vector v_t, the Hedge [Aue+95] algorithm has improved performance, achieving a regret of $O(\sqrt{T})$. However, as each application is running on only one host at time, it is impossible to obtain the exact feedback for other hosts at instant t. To cope with this limitation, Hedge-MC (Hedge with Migration Control) uses an estimation function to calculate the expected feedback, using the partial and **inaccurate** information about **applications** and **hosts**.

The Hedge algorithm is similar to EXP3 as described in Eq. (13.4), the only change is the feedback vector $\hat{v}$:

$$\hat{v}_{a,t}^{j} = \begin{cases} \mu_{a,t}^{j} & \text{if } a = a_t^j \\ f_{est}(a) & \text{otherwise} \end{cases} \tag{13.8}$$

Eq. (13.8) relies on the estimation function f_{est} to provide a good estimation of the application feedback. In summary, as illustrated in Fig. 13.11, f_{est} accords a good feedback (close to 1) if the host has enough resources to run the application, considering the requirements of the application and the resources available on the host.

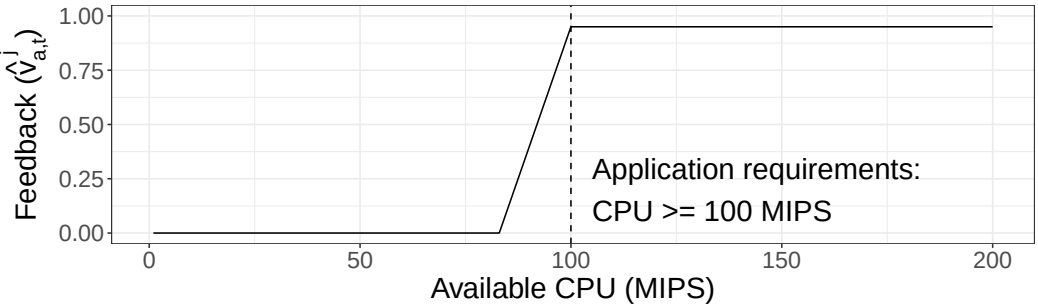


Figure 13.11: Estimation function (f_{est}). Shape of f_{est} for an application that requests a CPU with at least 100 MIPS.

Hedge-MC shares the same migration control mechanism as EXP3-MC, i.e., applications are migrated only when the threshold is exceeded. When the migration must occur, Hedge-MC will follow the probability distribution which reflects the feedback obtained by the estimation function.

Note that both application and host information is noisy and imprecise, since it depends on the accuracy of user description and the update frequency of the monitoring tool. However, it provides a good estimation to improve the learning rate of Hedge-MC algorithm. It is staggering the consequences in the learning of intensive applications, as seen in Fig. 13.12. Many applications, highlighted by colored lines, are able to select the best hosts to run the application. As a result, we can see in region A of Fig. 13.13 a slight improvement in terms of elapsed time and number of migrations.

Furthermore, Fig. 13.13 introduces the results for the "time above threshold" metric which represents the total time that applications were unable to meet the expected threshold defined by users. We point out that this metric follows the behavior of elapsed time in our experimental scenario, and so, Hedge-MC is the one with best

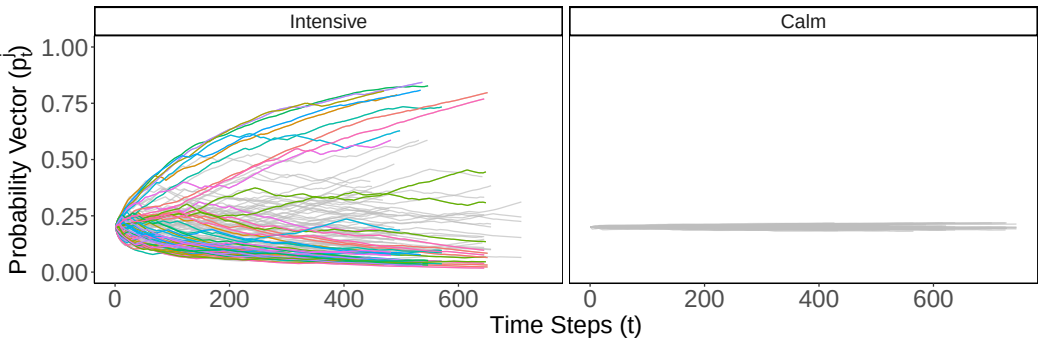


Figure 13.12: Hedge-MC learning rate for an experiment in a well-informed environment. The figure shows the probability vector p_t^j, each line is the probability that an application has to select each host. We highlight with a different color, the applications that were able to identify the best hosts to run (i.e., applications whose 2 best hosts have at least 75% probability of being chosen). We can clearly see the improvement in the learning rate for intensive applications, where hosts are selected more often for some applications.

performance (just after our target Totalitarian). For calm applications, the impact is less noticeable due to the y-scale of this metric.

Finally, another important characteristic of Hedge-MC is its dependency on the estimation function. This is more perceptible when we compare between intensive and calm applications. The estimation function used is more accurate for intensive applications and so, Hedge-MC is able to distinguish among the available hosts. On the other hand, for calm applications, it estimates that all hosts have similar performance and consequently, Hedge-MC is incapable of learning (cf. right part of Fig. 13.12). This can be seen by the similar performance between EXP3-MC and Hedge-MC.

Hedge-MC-TE: encouraging migrations

Despite the better performance of Hedge-MC, it is still a pure reactive algorithm which only migrates when needed. In this context, we may lose optimization opportunities because the elapsed time is just good enough, i.e., below the predefined threshold. On the other hand, we have seen that too many explorations degrade considerably the performance. For these reasons, we introduce Hedge-MC-TE which uses a **mixed** strategy to cautiously select some applications to migrate when we may improve the elapsed time.

Hedge-MC-TE, where TE stands for Trial and Error, is inspired by the work of [You09]. The general idea is to monitor the estimated feedback of all hosts, migrating when the performance of current host is not the optimal for some period of time.

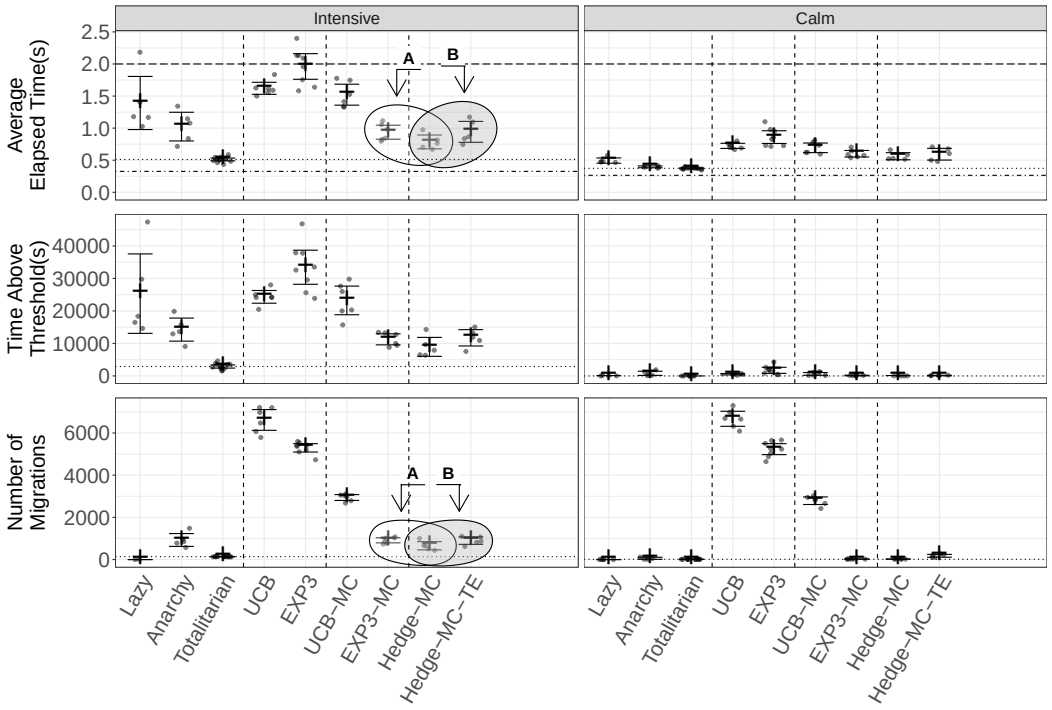


Figure 13.13: Performance evaluation for Hedge-MC and Hedge-MC-TE strategies. This figure complements Fig. 13.9 by adding the results for these strategies. A new metric "Time Above Threshold" is presented.

Fig. 13.14 presents the state machine used to decide when an application should migrate. In the Content state, the application is in the best possible host. The transition to Watchful state happens when some other host has better predicted performance. If this occurs during more than $n_{TE} = 10$ times, the application is considered as Discontent and will trigger the migration. Hedge-MC-TE will choose the next host to receive the application following the vector p, as described in Eq. (13.4). Note the global transition to Discontent which indicates that whenever the application is not satisfied with the current placement, it should be migrated.

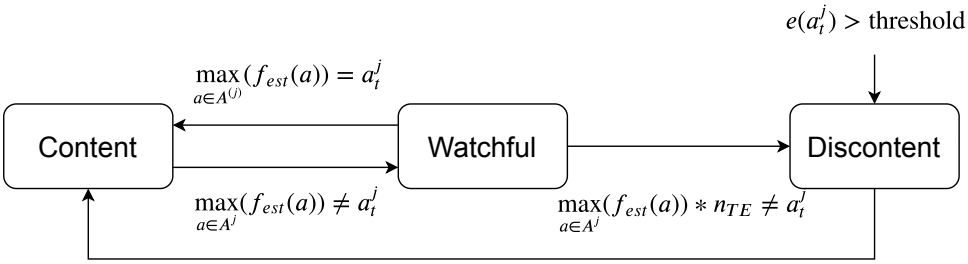


Figure 13.14: Hedge-MC-TE: state machine

Unfortunately, as we can see in Fig. 13.13, region B, the performance is not improved by this algorithm. Despite of the few additional migrations done by Hedge-MC-TE, their migration cost mitigates the possible gain induced by these new explorations. We believe that two main factors lead to this result: i) the estimation function is not

precise enough to correctly identify the performance variation among hosts and ii) the applications and hosts are homogeneous, and so, the elapsed time tends to be similar between hosts, with no significant difference that this algorithm could exploit.

Hedge-MC-TE-DUMP: cordial neighbors

As explained in Section 13.1.2, we ensured that we are in a scenario where the system is very loaded, with few resources available to run intensive applications. In such scenario, it is very difficult for applications, with their partial view of the system, to find the proper placement. To deal with this issue, we propose a strategy based on the cordiality between applications, i.e., voluntary applications accept a temporary poor performance in favor of better performance for everyone.

Fig. 13.15 presents the state machine for Hedge-MC-TE-DUMP strategy. It follows the same general idea as Hedge-MC-TE, where applications try to improve their performance each time they reach the Discontent state. The difference is the new GiveUp state, where cordial applications go to leave free resources to their neighbors in the hosts. The transition to GiveUp state happens after the application is discontent $n_{DUMP} = 10$ times in the last $period_{DUMP} = \{60, 300\}$ seconds. Note that the applications do not stop executing, but they go to a "dump" host where the performance will be degraded. They will leave the GiveUp state and restart executing normally after $period_{DUMP}$ seconds.

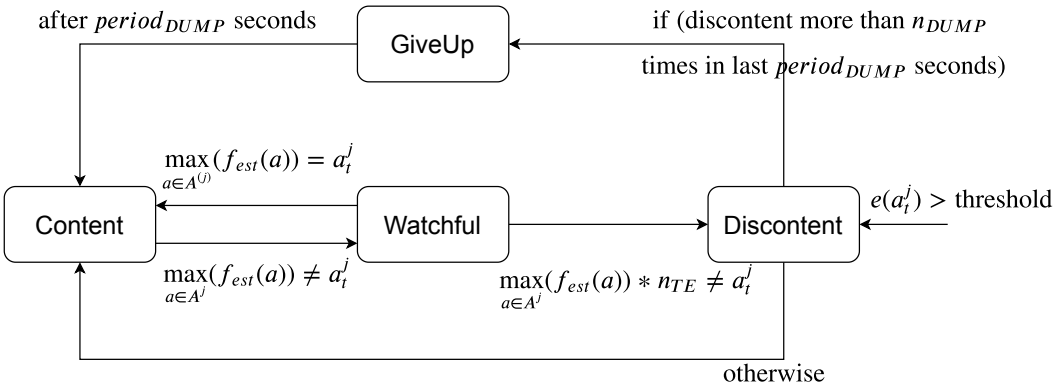


Figure 13.15: Hedge-MC-TE-DUMP: state machine

The results for the Hedge-MC-TE-DUMP strategy are presented in Fig. 13.16. We set the $period_{DUMP}$ parameter for 60 and 300 seconds. In the region A of Fig. 13.16, we can see that, despite the lower number of migrations (mainly for $period_{DUMP} = 300s$), the other metrics (elapsed time and time above threshold) have underperformed compared to Hedge-MC and Hedge-MC-TE. Unfortunately, the gain obtained for the other applications does not compensate for the performance degradation of cordial applications running on the "dump" host. This effect is

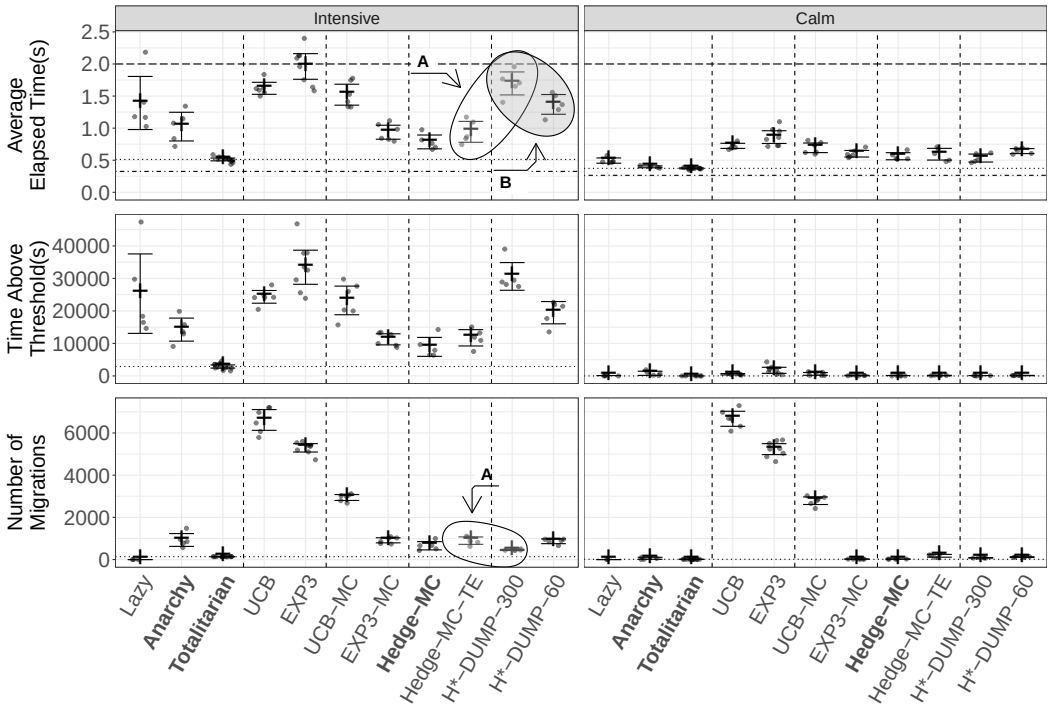


Figure 13.16: Performance evaluation for Hedge-MC-TE-DUMP (H*-DUMP-60 and H*-DUMP-300 in the figure) strategy, varying the amount of time applications stay in dump runtime (300s and 60s). This figure compares all learning strategies. The strategies in bold (Anarchy, Totalitarian and Hedge-MC) have the best performance and will be used as base for comparison in Section 13.2.3.

clearer when we compare the elapsed time between the two Hedge-MC-TE-DUMP strategies, in region B, the longer the applications remain in the "dump" host, the worse the overall performance will be.

In conclusion, the cordiality and benevolence of the applications did not improve performance, and therefore, Hedge-MC-TE-DUMP does not work in our experimental setup. Furthermore, by comparing all learning strategies in Fig. 13.16, we see that Hedge-MC has the best performance in terms of elapsed time and time above threshold. Along with the Anarchy and Totalitarian baseline strategies (all highlighted in bold), they will be kept as a basis for comparison in the following section, where we will analyze the performance of greedy strategies.

13.2.3 Greedy but informed strategies

The algorithms presented in Section 13.2.2 attempt to boost the applications' performance by learning the best host for each, based on the feedback the applications experience and provide. In this section, we propose a different and more reactive approach, where the algorithms reconfigure the placement of application having

bad performance, but in a controlled and informed manner. All algorithms in this section take advantage of the `GO-FSP` algorithm to select the best host to run the application.

PC: distributed arbitration

PC (Partial Coordination) is motivated by the behavior of applications under Anarchy control. In such a case, many applications tend to migrate to the same host at the same time. Fig. 13.17 illustrates this case in a reduced scenario. Let's consider two applications, A and B, running on hosts 1 and 2. At instant t_1, application A gets active and starts running. Its performance is satisfactory until application B gets active at t_2. At t_3, both applications have a poor performance and decide to migrate to the Host 2 which has 100% of CPU available. However, as both applications are now running at B, in the next step t_4, they will both decide to migrate back to host A. This ping-pong effect happens until t_6, where the application A gets inactive and so, B is capable of running with good performance, making the system stable again.

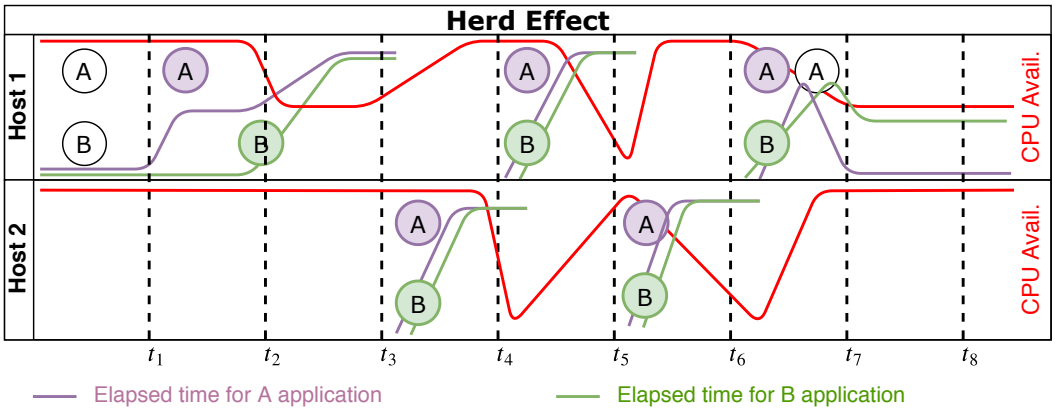


Figure 13.17: The Herd Effect

In this context, we propose two mechanisms to mitigate the "herd effect": i) selective migration: we opt for migrating only 1 application, selected at random among unsatisfied applications, per host per time frame; and ii) migration cooldown: after a migration is done, the source host waits for a period of time (*10s*) to stabilize performance, so that the remaining applications on the host have sufficient time to perceive the improvement in their elapsed time.

Fig. 13.18 shows the effectiveness of the proposed solution when comparing to both Anarchy and the best learning strategy, Hedge-MC. The improvements allow PC to reduce considerably the amount of migrations and the time above threshold (cf. region B). The elapsed time is, in consequence, closer to our target, the Totalitarian approach. Moreover, in region A, we highlight that PC also improves the elapsed

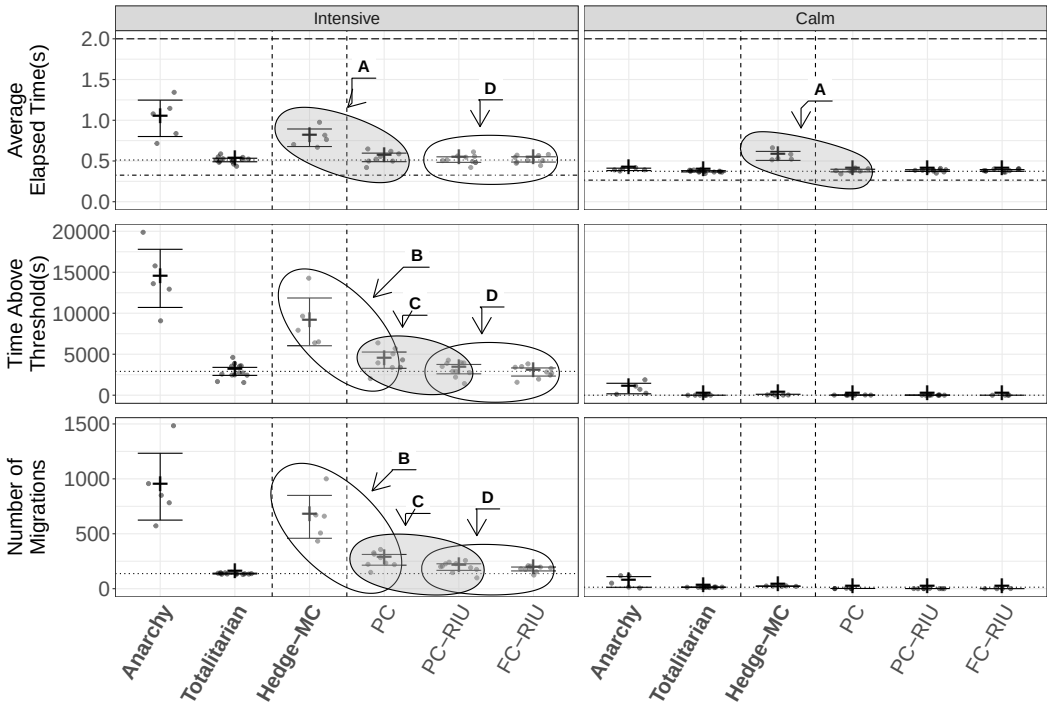


Figure 13.18: Performance evaluation for greedy strategies. This figure compares the performance of greedy strategies with those of learning in Fig. 13.16. The strategies in bold (Anarchy, Totalitarian and Hedge-MC) are kept as a basis for comparison. Note that the y-axis scale has changed.

.

time for calm applications, whose performance was hindered by the learning strategies. The improved performance by PC can be explained by two factors: i) the algorithm is nimble enough to reconfigure the application when needed and ii) the migration cooldown gives the necessary time to applications settle down in their current hosts.

PC-RIU: updating resource information

In the previous section, it has been proven that the performance of the applications can be improved by implementing a control mechanism in each host. This mechanism avoids the "herd effect" in a single host and its results are promising. Nevertheless, we noticed many schedulers were still taking their decision independently and based on not up-to-date information about hosts' resource consumption. This can lead to a distributed "herd effect", where different schedulers send their applications to the same host.

One way to cope with this problem is doing a **partial** update of hosts, selectively updating resource consumption information. PC-RIU (Partial Coordination with Runtime Information Update) solves this problem by requesting the update of

resources utilization for the hosts being used in the migration process. For example, if host 1 decides to migrate an application to host 2, it will also request the host 2 to update its resource utilization. Thus, other hosts may base their migration decision on the up-to-date data from host 2, perhaps avoiding it and choosing another host for their applications.

In our case study, the majority of the "herd effect" was due to applications coming from the same host, as we could see in the previous section. However, despite the uncertainty of the measures, we can still see in Fig. 13.18, region C, a modest reduction in the number of migrations and in the time above threshold of PC-RIU compared to PC and. This is explained by the better decisions taken by hosts thanks to more up-to-date resource information. However, PC-RIU does not show a significant improvement in the elapsed time because we are already very close to the optimal Totalitarian performance (as seen in region A of Fig. 13.18).

FC-RIU: short-sighted dictatorship

Finally, we present the impact of centralizing the reconfiguration decision in a single host. FC-RIU (Full Coordination with Runtime Information Update) extends the proposal in PC-RIU by centralizing the reconfiguration decision in a single entity. In this strategy, the applications request their reconfiguration to a centralized host. FC-RIU will then put together all requests and apply the same criteria adopted for PC and PC-RIU, i.e., one migration per time frame, the migration cooldown and update of resources. It is thus close to the Totalitarian strategy except it cannot foresee when applications will switch from Sleeping to Active and is thus Reactive.

Fig. 13.18, region D shows that the performance of FC-RIU is quite similar to PC-RIU, in both elapsed time, time above threshold and number of migrations. We can conclude that the centralization of the reconfiguration decision offers little benefit compared to those already obtained with the aforementioned strategies. Besides, the characteristics of the Fog environment prohibit the use of a single and centralized entity.

13.2.4 Summary

Table 13.1 summarizes and compares the different strategies presented in this section. The first column shows the strategy's **class** according to its approach, online learning or scheduling. In the second column, we describe the **mode** how each strategy does the reconfiguration, proactively or reactively. We observe that this is an important factor that reflects in the strategy's performance. More precisely, by

	Class	Mode	Information	Coordination	Performance
Lazy	NA	NA	NA	NA	1.39 ●
Anarchy	Greedy	Reactive	Inaccurate	None	1.02 ●
Totalitarian*	Oracle	Proactive	Accurate	Full	0.51 ●
UCB	Learning	Proactive	None	None	1.62 ●
EXP3	Learning	Proactive	None	None	1.96 ●
UCB-MC	Learning	Proactive	None	None	1.52 ●
EXP3-MC	Learning	Reactive	None	None	0.93 ●
Hedge-MC	Learning	Reactive	Inaccurate	None	0.78 ●
Hedge-MC-TE	Learning	Mixed	Inaccurate	None	0.94 ●
PC	Greedy	Reactive	Inaccurate	Partial	0.54 ●
PC-RIU	Greedy	Reactive	Inaccurate	Partial	0.51 ●
FC-RIU*	Greedy	Reactive	Inaccurate	Full	0.51 ●

* - centralized strategies

Table 13.1: Strategies Classification. The performance column summarizes the average elapsed time (in seconds) for intensive applications. (● $\leq 0.75s$, $0.75s <$ ● $\leq 1s$, ● $> 1s$)

passing from a proactive to a **reactive** approach, EXP3-MC considerably improves its performance compared to proactive ones, such as UCB, EXP3 and UCB-MC. This performance gain is also valid for the other reactive approaches. In the **information** column, we describe the level of details available about the infrastructure and applications. Here, we highlight that the use of **inaccurate** information by Hedge-MC and greedy strategies gave a step further in the performance improvement. Although inaccurate and not very up-to-date, the extra information provided by the developer and the monitoring tools are important, impacting positively in the elapsed time of applications. Finally, in the last column, the **coordination** describes how application migration is organized. In the **partial** coordination, each scheduler coordinates the requests from its host, while in the full, all applications are coordinated by a centralized scheduler. Note that the partial coordination used by greedy strategies greatly improves performance. This effect is more noticeable when we compare the performance of Anarchy and PC, since the only difference between them is the partial coordination done by PC.

13.3 Limitations

One of the main limiting factors in our experiments is the duration. Each experiment in Section 13.2 takes at least two and a half hours, between initialization, provisioning and reconfiguration phases, test teardown (closing and collecting the results), etc. Consequently, we had to limit the number of executions and the time spent in the reconfiguration phase.

Moreover, the reconfiguration strategy has only 1 hour to try to learn and adapt the placement of applications. This short period (considering the lifespan of an IoT application), along with the instabilities in the application's performance due to the migrations, explain the difficult of EXP3 in reaching the equilibrium in the proposed congestion game.

Finally, the experimental scenario described in this chapter has the strong assumption that developers are capable of providing a quite accurate estimation of the resources used by their applications. In a real environment, this information is sometimes difficult to obtain.

In Chapter 14, we will increase the duration of experiments and then strategies will have more time to learn the performance of hosts. Also, we will loosen this information constraint, making application performance less predictable.

14 Reconfiguration in an Ill-informed Environment

In Chapter 13, we studied the case where users were able to accurately describe the resources needed by their applications. In this section, on the other hand, we present an affinity scenario, where there is a lack of exactness and accuracy in the description of resource utilization. We will see how incorrect information can affect the strategy's performance and compare this affinity scenario with the previous non-affinity one.

14.1 Describing the Environment

As previously discussed, the Fog environment presents a great heterogeneity in its infrastructure. Consequently, the performance of an application may vary from host to host due to the presence of specialized hardware. In this situation, the requirements described by the application developer may be inaccurate for some machines. To emulate this behavior in this experimental environment, we create an affinity between applications and hosts, where each application has three optimized hosts. When running on these optimized hosts, application performance is greatly improved and processing time at Burn is reduced. Note that, these optimized hosts are unknown for all the strategies.

14.1.1 Platform

The platform used in our reconfiguration tests is composed of *67 nodes*. We use *50 nodes* from FIT/IoT-LAB to represent the sensors, and *17 nodes* from Grid'5000, forming the Fog layer. The 17 nodes from Grid'5000 are part of the *uvb* cluster in Sophia Antipolis and they are characterized by 2 CPUs Intel Xeon X5670, with 6 cores per CPU and 96GB of RAM [3]. In this setup, each Grid'5000 node provides about 240 MIPS of CPU power.

14.1.2 Workload

The application's structure follows the same 3-level model as used in previous chapter and detailed in Section 8.1.2. In summary, our workload is described by:

- **Application load**: contains the same two kinds of applications as in previous chapter, but with few modifications[1]:
  ► *Intensive*: these resource-consuming applications send a large amount of messages (*1 message per seconds* with a payload of *1024 bytes*) to be processed, each incurring *150 MI* per message (or *7.5 MI* if running on an optimized host).
  ► *Calm*: *0.5 message/s* with the same payload, where each message consumes *20 MI* (or *1 MI* if running on an optimized host).

- **Application heterogeneity**: again we opted for an *intense* profile with a $50\%/50\%$ mix between intensive and calm applications.

- **System Load**: the *heavy* load, composed of 50 applications concurrently running on the platform, is kept.

- Application arrival interval: applications arrive in a 60 seconds interval during the provisioning phase.

- **Application threshold**: the threshold to determine application's satisfaction is set to *2 seconds*.

- **Churn**: similar to the churn detailed in Section 13.1.2, we modeled the churn as a 2-state (Active and Sleeping) Poisson process, where state changes are exponentially distributed. In our experiments, we considered relatively *slow* applications with a mean active/sleeping time ($1/\lambda$) of 300 seconds.

► **Low-level application description:** hereafter we describe the main low-level parameters in our setup.

- **Requirements for actors**:
  - Trigger: requires an amount of CPU proportional to its token interval, i.e., 3 MIPS for calm applications and 4 MIPS for intensive.

[1] Although the parameters (token rate and processing) in the application load are changed, the overall load is the same in both scenarios.

– Burn: CPU is proportional to the processing effort per token. Calm applications require 20 MIPS while intensives demand 150 MIPS.

– Sink: requests memory requirement equals to 3.7 MB for intensive applications and 1.8 MB for calm ones.

14.1.3 Orchestrator parameters

- Provisioning strategy: for these experiments, we opted for a *fixed* initial placement for each application. In the fixed setting, applications are distributed to hosts in a round-robin manner. Note that we did not see any impact on results due to the chosen provisioning strategy.

- **Reconfiguration strategy**: this parameter is defined later in the chapter.

- **Maintenance interval**: the *5 seconds* maintenance interval is used in these experiments, in order to have a good responsiveness of the reconfiguration algorithms.

- **Monitoring interval**: this parameter is kept the same, updating the information about resource usage (CPU, RAM) each *60 seconds*.

14.2 Evaluation

The presentation of experimental results in this section follows the same structure as Chapter 13, but in a condensed way to improve readability, since the reader is used to the already mentioned strategies. The experiments in this section last for *2 hours* after the initial provisioning.

14.2.1 Baseline strategies

We start our analysis by inspecting the results for the baseline strategies in Fig. 14.1. In this section, we chose to remove the results of the Lazy strategy from the baseline, due to its extremely poor performance.

As expected, the Totalitarian strategy achieves a quite good performance, especially for the time above threshold metric. By planning ahead, the Totalitarian is able to find acceptable hosts for applications, although not optimal, as we will see in the next sections. For the Anarchy strategy, we call attention to the high variability of

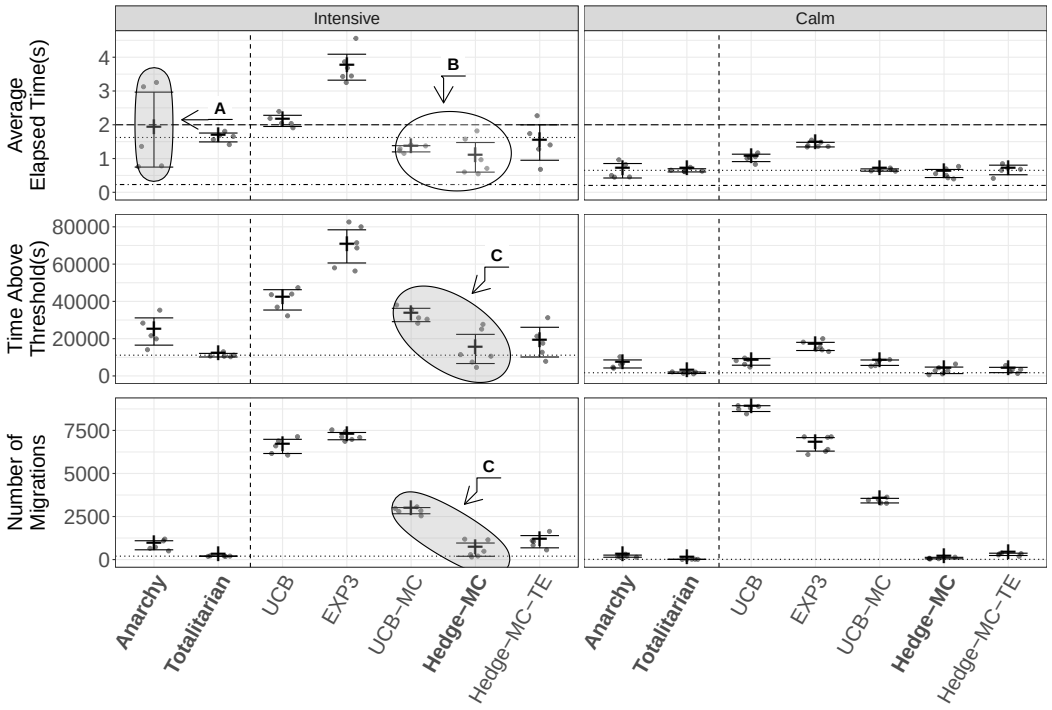


Figure 14.1: Performance evaluation for learning strategies in the ill-informed scenario. This figure compares all learning strategies. The strategies in bold (Anarchy, Totalitarian and Hedge-MC) have the best performance and will be used as base for comparison in Fig. 14.5.

the results, as we can see in the region A of Fig. 14.1. The Anarchy's performance depends on the dynamicity of application migrations. For example, if an application migrates to an optimized host alone, without the "herd effect" described in Section 13.2.3, its performance will be satisfactory and it will stay there. However, if more applications migrate to the same host, the performance will be degraded and the application will migrate again.

14.2.2 Online learning strategies

UCB

The performance for learning strategies is presented in Fig. 14.1. Comparing this result with those obtained in Fig. 13.16, we note the impact of the scenario on the strategies' performance.

Unlike the previous scenario, for UCB, the elapsed time is close to the Totalitarian, despite the large number of migrations carried out by UCB. In fact, in the affinity scenario, UCB is able to learn the real performance of hosts, distinguishing, for

each application, the bad hosts from the optimized ones. Moreover, UCB is able to properly exploit these best hosts, reducing the elapsed time for applications.

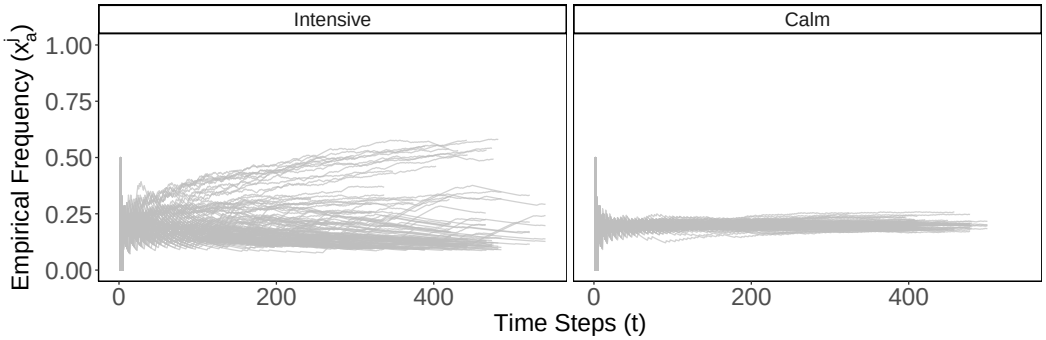


Figure 14.2: UCB learning rate for an experiment in an ill-informed environment. Each line represents the frequency an application has selected determined host ($x_a^j = n_a^j/t$). We highlight with a different color, the applications that were able to identify the best hosts to run (none in this case).

In Fig. 14.2, we analyze the learning behavior for every application in an experiment. For intensive applications, we still cannot highlight applications with 2 hosts that were selected more than 75% of the time. Although, compared to the performance of UCB in Fig. 13.5, it is noticeable that some hosts are selected more frequently for intensive applications ($x_a^j \geq 0.5$). For calm applications, as they already use very few resources, the difference in performance is not so great and so UCB is not able to distinguish the hosts.

The figure shows a higher number of migrations and time above threshold than the Totalitarian strategy, but the tendency is their stabilization over time, at least for those applications that have found optimized hosts to run. This is a consequence of the learning process, as time goes by, the incertitude over the performance of hosts decreases and the UCB algorithm starts exploiting more the optimized hosts.

EXP3

Similar to the non-affinity scenario, EXP3 has the worst performance among learning strategies. In order to be effective against malicious adversaries, EXP3 needs to be conservative in exploiting the best available hosts, and therefore, the elapsed time is considerably higher. This also leads to a larger number of migrations which negatively impact the elapsed time of applications. Finally, we note that stochastic based strategies, such as UCB, perform better in this affinity scenario as there is a clear structure to learn and exploit.

UCB-MC

The effectiveness of UCB-MC can be seen in Fig. 14.1. In this scenario, where hosts have different performances and the strategy has the appropriate migration control mechanism, UCB-MC excels and even outperforms the Totalitarian strategy. This effect is explained not only by the better performance of UCB-MC but also by the fact that the Totalitarian does not know about the optimized hosts and relies only on the erroneous information provided by the user.

The number of migrations is significantly lower than UCB and it has a direct impact over the elapsed time of applications, since the best hosts are selected more often. Therefore, the migration control acts positively and accelerates the stabilization of the performance. However, these initial explorations still impact considerably the time above threshold metric when the optimized host is not selected.

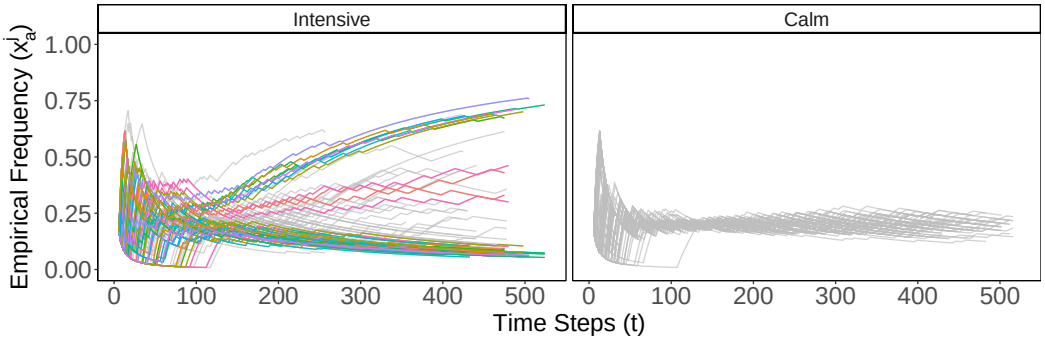


Figure 14.3: UCB-MC learning rate for an experiment in an ill-informed environment. Each line represents the frequency an application has selected determined host ($x_a^j = n_a^j/t$). We highlight with a different color, the applications that were able to identify the best hosts to run (i.e., applications whose 2 best hosts have at least 75% probability of being chosen).

We can how the learning behavior is related with the performance in Fig. 14.3. The greater number of highlighted applications reinforces the better performance of UCB-MC. Once again, the migration control mechanism helps the learning by decreasing the instabilities in the system generated by the migrations. For calm applications, we cannot see any difference but their performance is already acceptable.

Hedge-MC

As in the previous scenario, Hedge-MC is also the best learning strategy for the affinity case, but for different reasons. In the previous scenario, Hedge-MC accelerates the congestion game solution, learning how to dispatch the applications on the available infrastructure. On the other hand, in the affinity scenario, the good perfor-

mance is explained by its ability to find the optimized hosts. Although Hedge-MC uses unreliable user information to estimate the feedback, it eventually selects the optimized host to run the application. When this happens, the application is no longer interested in migrating and therefore remains on the current and optimized host.

We highlight some regions in Fig. 14.1 which show the differences between the proactive UCB-MC and reactive Hedge-MC strategies. In region B, the similar elapsed time indicates their capacity to find the optimized hosts, but for different reasons as explained before. Region C, in turn, shows the strong relation between migrations and time above threshold. In this case, the reactive approach achieves better performance because it disturbs less the environment with unnecessary explorations.

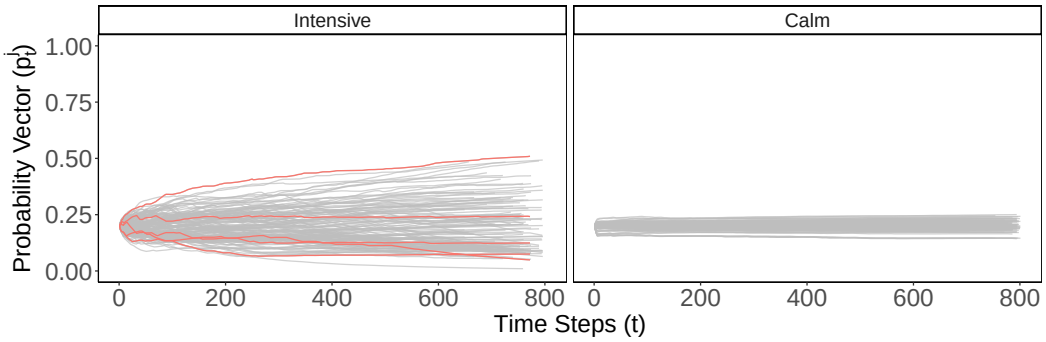


Figure 14.4: Hedge-MC learning rate for an experiment in an ill-informed environment. The figure shows the probability vector p_t^j, each line is the probability that an application has to select each host. We highlight with a different color, the applications that were able to identify the best hosts to run (i.e., applications whose 2 best hosts have at least 75% probability of being chosen).

Nevertheless, the learning capacity of Hedge-MC in the affinity is worse than in the non-affinity scenario, as we can see in Fig. 14.4. This can be explained by the inadequacy of the estimation function which uses the imprecise information provided by user. This wrong feedback conflicts with the real feedback provided by the host when the application is running, disturbing the learning process. However, it is important to observe that the bad learning does not incur in more migrations because, as cited above, the applications tend to stay in the good hosts.

Hedge-MC-TE

Finally, the Hedge-MC-TE has a similar (but slightly worse) performance to Hedge-MC and comparable to that from the non-affinity scenario. In terms of elapsed time, the good performance is explained in the same way as for Hedge-MC, i.e., the inertia of applications once on a good host. As in the non-affinity scenario, the

few more migrations done by Hedge-MC-TE do not lead to better performance. This is, once again, caused by the inability of the estimate function to give good approximation for the performance of applications.

Global Analysis

Note that the scenario has an important impact on the performance of the different strategies. In the affinity case, where strategies have a clear difference in the performance of hosts to exploit, UCB-MC has an average elapsed time close to Hedge-MC and Hedge-MC-TE, but with access to less information. More precisely, in the learning process, Hedge-MC and Hedge-MC-TE use the user and host information to estimate the complete feedback vector for all hosts, while UCB-MC relies only on the bandit feedback of the current host. However, the migrations made by UCB-MC worsen the performance of the other two metrics: time above threshold and number of migrations. For this reason, we keep the Hedge-MC when analyzing the greedy strategies in Section 14.2.3.

In addition, we emphasize that the results obtained in the affinity scenario point out the dependence of Hedge-MC and Hedge-MC-TE to a good estimation function. The learning process is linked to the capacity of this function to reflect the reality, even if there is some uncertainty. On the other hand, we see that the proposed migration control is able to circumvent this problem if there are efficient hosts to run the applications smoothly.

14.2.3 Greedy but informed strategies

In Fig. 14.5, we can see the performance for PC and PC-RIU greedy strategies. Note that we opt to remove from these tests the FC-RIU strategy, since it has the drawbacks brought by the centralization and its performance results do not pay off this extra cost. Analyzing the region A of the figure, we see that PC has a good performance and it is more stable than Hedge-MC in terms of elapsed time. In the first sight, this result is surprising since the strategies have no information about the best hosts to run the applications. Although PC and PC-RIU take their decisions based on the inaccurate application description provided by the user, eventually the application is placed in an optimized host. Also, these reactive approaches have the good property of staying in it once this happens.

Moreover, the effect of updating the resource utilization performed by PC-RIU is clear in the affinity scenario. Region B shows the lower number of migrations carried out by PC-RIU compared to PC, while region C presents the effect of this

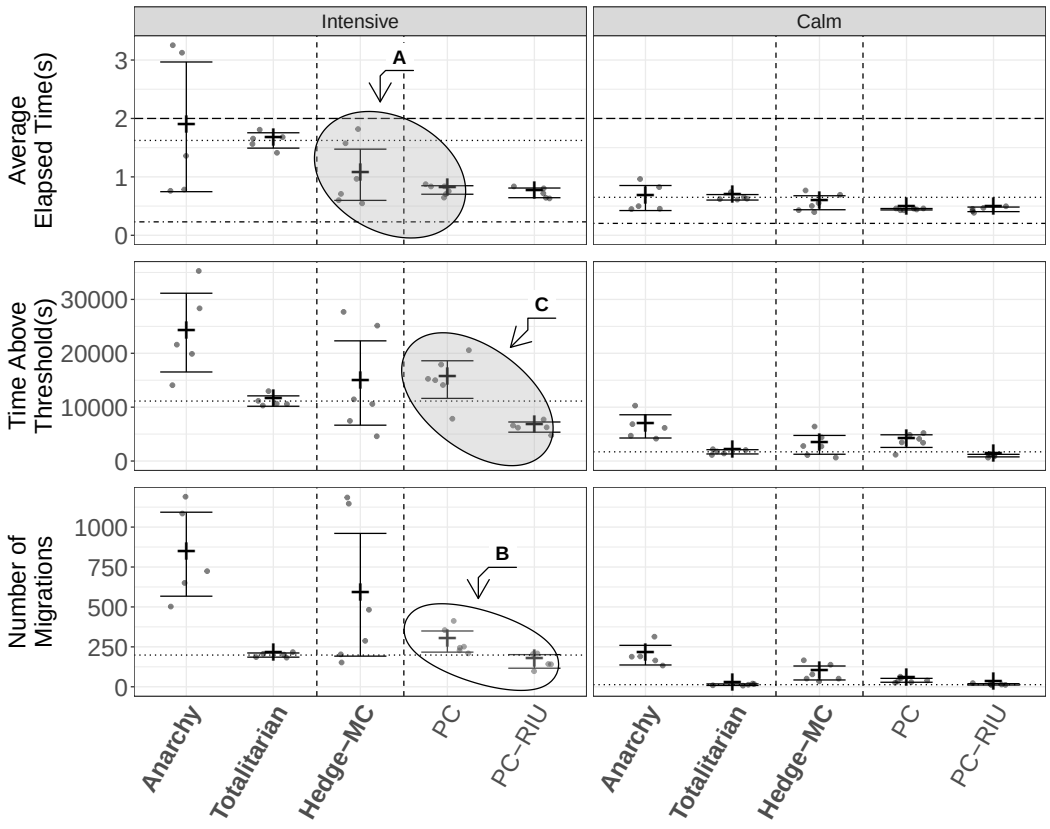


Figure 14.5: Performance evaluation for greedy strategies in the ill-informed scenario. This figure compares the performance of greedy strategies with those of learning in Fig. 14.1. The strategies in bold (Anarchy, Totalitarian and Hedge-MC) are kept as a basis for comparison. Note that the y-axis scale has changed.

.

migration reduction in the time above threshold metric. In conclusion, we note that the policies implemented by PC-RIU (notably the selective migration, the migration cooldown and partial update of resource information) help the system to stabilize faster, achieving a very good overall performance for all metrics. Compared to classical learning strategies, the reactive strategies using informed placement mechanisms may not identify directly optimal placements but they quickly filter out bad decisions without resorting to a costly exploration.

14.3 Limitations

14.3.1 Experimental limitations

We start the analysis of the limitations in our experiments with the duration of the experiments. We have seen an improvement in the learning for the online learning strategies in this chapter. However, we believe that a longer execution

could bring closer the performance of online learning strategies to greedy strategies. To illustrate this fact, we present the results of a single execution for UCB-MC, in the same experimental environment described in this chapter, but during 5 hours instead of 2 hours.

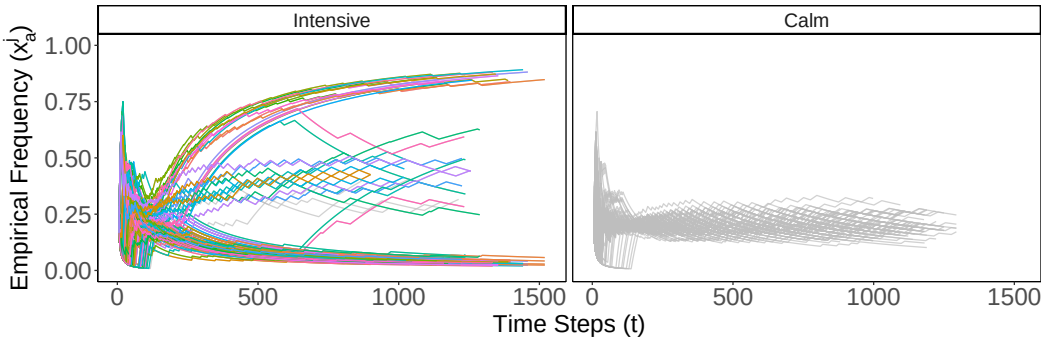


Figure 14.6: UCB-MC learning rate for a long duration (5h) experiment in an ill-informed environment. Each line represents the frequency an application has selected determined host ($x_a^j = n_a^j/t$). We highlight with a different color, the applications that were able to identify the best hosts to run (i.e., applications whose 2 best hosts have at least 75% probability of being chosen).

Fig. 14.6 presents the learning behavior for UCB-MC. We can see the stabilization and improvement for intensive applications which have optimized hosts to run. Furthermore, the average elapsed time for intensive applications in the last hour of the experiment is $0.777s$, very close to the performance of PC-RIU in Fig. 14.5 ($0.726s$).

14.3.2 Platform

Despite the efforts to create a realistic environment, relying on `FITOR`, Grid'5000 and FIT/IoT-LAB to recreate a proper Fog environment, our experimental environment has some drawbacks. The first is the homogeneity of Fog nodes, since we use several nodes from the same Grid'5000 cluster with similar performances. It is possible to increase the heterogeneity of nodes, by choosing nodes from different clusters or by limiting the resources available for containers in Docker. Second, it would be interesting to evaluate the performance in larger scenarios, possibly with hundreds or thousands of machines. Unfortunately, although technically possible, this type of experiment is extremely time and resource consuming.

Nevertheless, the main limitation in our platform is related to the network connectivity. The network within Grid'5000 can be customized, increasing latency or reducing the bandwidth available to the nodes. However, the VPN connection between Grid'5000 and FIT/IoT-LAB creates an artificial tunnel which is not

present in a real Fog environment. In addition, the VPN makes it very difficult any customization in network resources.

14.3.3 Workload

Our workload tries to mimic the main characteristics of IoT applications, but we had to set many of the parameters that characterize our workload. A straightforward extension to our work is considering variations in the workload, such as rapidly evolving applications (churn), variety of application heterogeneity with more calm or intensive applications, or yet, other load profiles that consume different amounts of resources.

In addition, the workload used in the experiments is mainly CPU-intensive, consuming most of CPU resources in the infrastructure. It is possible to extend this by considering other network or memory intensive profiles. Finally, these profiles could be mixed to study a more complete and complex scenario.

Finally, we have studied the impact of an imprecise description of application requirements in our scenario in this chapter. However, it is possible to study the impact over the performance of other application specific requirements, either in terms of location or network latency/bandwidth constraints for example.

14.3.4 Orchestrator

The configuration used in orchestrator settings is important and may have an influence on our experimental results. The monitoring interval, for example, determines the correctness of information about the use of resources on the platform. All strategies that rely on this information to take their decision can potentially take advantage (or be negatively affected) by different update rates.

Furthermore, the maintenance interval, which controls how often hosts check application satisfaction, plays a key role in the performance of strategies. A short maintenance interval is advantageous for reactive strategies, but induces a greater number of migrations. On the other hand, a longer maintenance interval helps the system stability, generating fewer migrations but affecting the satisfaction of the applications. Therefore, the fine-tuning of this parameter, studying the trade-off between short and long intervals, deserves an in-depth and cautious research.

Finally, each strategy has its own parameters that can be fine-tuned for a better performance. Regarding the online learning strategies, an obvious parameter to

configure is the set of hosts available for each application ($|\mathcal{A}^j|$). It defines the search space for learning algorithms and was set to 5 in our experiments. However, a proper study should be carried out to determine the optimal value, considering the trade-off between learning rate and performance. Moreover, each strategy has its own configuration parameters, such as α for UCB, η for EXP3, etc., whose impact can be studied and depends on the environment being studied.

15 Conclusion and Future Work

15.1 Conclusion

Fog computing is shaping the future infrastructure for IoT, meeting the strict requirements of new IoT applications. However, several challenges still remain due to the characteristics of the Fog. In this thesis, we tackled part of these challenges, namely the orchestration of IoT applications in the Fog environment. More specifically, we have studied two main aspects of orchestration: the provisioning and reconfiguration of multi-component IoT applications.

In a first step, we proposed FITOR, a new orchestration solution for IoT applications in the Fog environment. Our goal was to create a realistic Fog-IoT environment, on which we could run real IoT applications, while monitoring and collecting data about applications and infrastructure. The collected data were used to analyze and evaluate the proposed orchestration mechanisms. FITOR puts together several open source tools and two test platforms: Grid'5000 and FIT/IoT-LAB. Moreover, we have extended the Calvin IoT framework to remove the limitations that preclude its use in a Fog environment.

In a second step, we addressed the provisioning problem of IoT applications in a Fog computing platform. We put forward two novel strategies, O-FSP and GO-FSP, which optimize the placement of IoT application components while meeting their non-functional requirements. O-FSP is a greedy algorithm that incrementally builds a valid solution by placing one component of the application at time. To avoid sticking with invalid or bad solutions, several solutions are generated and the best one, considering the provisioning cost, is selected. Extensive experiments show that the O-FSP strategy makes the provisioning more effective and outperforms classical strategies in terms of: i) acceptance rate, ii) provisioning cost, and iii) resource usage.

We also propose a load aware provisioning strategy named GO-FSP. This novel approach adopts the GRASP methodology to optimize the provisioning of Fog services. GO-FSP iteratively improves the initial solutions of the problem generated by O-FSP, considering multi-objective criteria: provisioning cost and infrastructure load balance. By making use of the experimental environment available in FITOR,

we conducted extensive experiments to measure the effectiveness of our proposal. The obtained results prove that GO-FSP achieves better performances compared with O-FSP and classical strategies.

Finally, we have studied the reconfiguration of IoT applications in a Fog environment. To do so, we rely on a unified experimental framework which allowed us to evaluate different reconfiguration strategies in a fair and realistic manner. We have modeled our workload to reflect the main characteristics of an IoT application and application performance was analyzed using three relevant metrics: end-to-end delay, migrations and time above threshold.

Through an extensive set of experiments in two different scenarios, we have investigated which factors impact the performance of twelve reconfiguration strategies, based on both online scheduling and online learning paradigms. These two kinds of strategies differ wildly in how they handle information and uncertainty: the first ones assume faithful load information about applications and platform is available to build good placements while the former ones mostly rely on measured end-to-end application performance.

None of the classical online learning strategies was able to obtain satisfying performance, in particular because in our context, the numerous migrations required by the exploration are prohibitive. Our in-depth analysis of these strategies allowed us to mitigate this problem by designing a mildly informed and reactive learning strategy. We have also observed that a greedy strategy, which uses load and application information to recalculate a good placement when performance is unsatisfying, could obtain an overall performance comparable to the one a fully clairvoyant strategy, provided a minimal coordination between applications was implemented. Surprisingly, this good performance remains even in a scenario with inaccurate information. Our analysis shows that the surprising robustness of these informed strategies stems from the fact that they can quickly filter out bad deployments.

15.2 Future Work

As any research work, this study has its limitations and extensions are possible. In this section, we identify several important perspectives that could be studied in future work.

Provisioning: metrics and fine-tuning

In the first part of this thesis, we studied the provisioning problem, proposing a mathematical model and two algorithms to solve it. At first, it would be interesting to compute the exact solution for the ILP model for small scale problem instances, using the CPLEX solver for example. The main objective is to compare the results with the proposed heuristics while emphasizing the decrease of the convergence time achieved by our proposed approaches, especially for larger scenarios. Despite the promising results we had in our experiments, this would give us a lower bound on the performance of our algorithms.

Another interesting metric to be properly evaluated is the time required to place an application. During our experiments, we did not notice a major performance impact when placing applications. However, this effect can be relevant when the algorithm runs on a low power device in the distributed Fog environment.

Finally, some parameters of our model must be fine-tuned to achieve the best performance with shortest processing times. In particular, two parameters are important here: N and ϵ (see Section 11.2). N dictates the number of solutions generated for the problem. Consequently, higher numbers help to find better solutions, but lead to high processing times. In its turn, ϵ controls the size of our RCL, i.e. the list of possible nodes to place the application component. Again, higher values increase the search space, and thus, the possibility of finding better results.

Reconfiguration: metrics and fine-tuning

In the last part of this thesis, we looked at the reconfiguration problem, evaluating various strategies for migrating applications. We used some evaluation metrics that are usually relevant in the Fog (elapsed time, time above threshold and number of migrations). Nevertheless, other metrics are possible, such as number total of messages processed. In particular, the time needed to decide where to migrate the application may be relevant in this context. In the same way that an elevated number of migrations degrade the performance, a high decision time harms the application's performance if it is running on a faulty node.

Throughout Chapters 13 and 14, we saw several parameters that can influence the performance of applications, either by changing the quality of available information (e.g. monitoring interval), or the performance of a strategy (e.g. α for UCB). All of these parameters can be adjusted to improve performance, but we would like to highlight one parameter: $\mathcal{A}^j$. This parameter is responsible for delimiting the nodes

to which our learning algorithms have access. The search space can have a major impact on performance and learning rate. Ideally, we would like to include as many nodes as possible, within a reasonable learning rate. Although an adequate study is needed to decide $\mathcal{A}^j$, we believe that, given our initial results, adding more nodes to $\mathcal{A}^j$ can be problematic for the learning rate.

Reconfiguration: horizontal scaling

An open perspective relates to the reconfiguration mechanism. The strategies we studied so far rely solely on migration, but other alternatives are possible. For example, in the context of cloud-native applications, horizontal scaling is widely used, replicating application's data and services in local data centers to improve QoS [GBS17]. A study to determine how to scale the number of resources and what is the optimal number of replicas for each application would thus be quite instructive.

Reconfiguration: workload and churn

In the reconfiguration study, our applications evolve according to a 2-state Poisson process, going from sleeping to 100% active immediately and remaining on average 300 seconds in each state. This leads to application behaviors which are quite hard to predict. In future work, it would be interesting to see the behavior of the system when mixing applications with different churn profiles, for example, fast vs. slow state transitions or a smoother transition from sleeping to active states.

In Chapter 13, we saw how knowing these transitions can affect the performance of a strategy (Totalitarian has this information). Moreover, in many cases, IoT applications have their load impacted by human interaction. Therefore, an attractive perspective is to try to estimate the load required by applications, analyzing historical information and/or user mobility. Similar works have been done in the context of Cloud computing [LaC+14] and user mobility [CML11].

Scalability

The evaluation methodology is another challenging point that deserves a special attention. Thanks to FITOR and the two experimental testbeds (Grid'5000 and FIT/IoT-LAB), we were able to execute synthetic applications in our experiments but only at a relatively limited scale. The lessons learned when building and experimenting with this environment could be used to calibrate and design a

faithful simulation environment that would allow conducting realistic evaluations in large-scale scenarios.

Fog environment

Scalability is an important factor for the Fog, but not the only. Other improvements can be made in the Fog infrastructure. In this context, some of the characteristics of the Fog environment, such as geographic distribution, mobility and heterogeneity are underrepresented on our platform. Regarding the geo-distribution, nodes from different sites in Grid'5000 could be used or different latency could be emulated via software (e.g. Distem [Mad19]). Likewise, if the information about the Fog infrastructure is available, heterogeneity can be emulated by restricting resources on each machine, through Distem or Docker, for instance.

The support of mobility, on the other hand, is quite challenging. Indeed, FIT/IoT-LAB has a limited mobility support through two robots that can move in a predetermined route. However, their restricted number and limited mobility makes them unrepresentative of a real Fog infrastructure. Due to the complexity of the subject, the use of simulation, along with real mobility traces from user, appears as an interesting approach to cope with the mobility problem.

IoT applications

Finally, we also think that, different and more complex applications should be considered to gain more insights on the efficiency of provisioning and reconfiguration strategies. In our scenario, we opted for a relatively simple workload, with a 3-tier application that reflects the main characteristics of an IoT application. However, as the IoT and the Fog evolve and mature, more complex scenarios, possibly involving mobility and complex quality of service requirements, should be envisioned.

Bibliography

[ACF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Machine Learning* 47 (May 2002), pp. 235–256 (cit. on pp. 131, 145).

[Adj+15] C. Adjih, E. Baccelli, E. Fleury, et al. "FIT IoT-LAB: A large scale open experimental IoT testbed". In: *IEEE World Forum on Internet of Things*. Dec. 2015, pp. 459–464 (cit. on p. 75).

[AH15a] M. Aazam and E. Huh. "Dynamic resource provisioning through Fog micro datacenter". In: *2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*. Mar. 2015, pp. 105–110 (cit. on pp. 39, 43).

[AH15b] M. Aazam and E. Huh. "Fog Computing Micro Datacenter Based Dynamic Resource Estimation and Pricing Model for IoT". In: *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*. Mar. 2015, pp. 687–694 (cit. on pp. 39, 43).

[Ait+19] F. Ait Salaht, F. Desprez, A. Lebre, C. Prud'homme, and M. Abderrahim. "Service Placement in Fog Computing Using Constraint Programming". In: *2019 IEEE International Conference on Services Computing (SCC)*. 2019, pp. 19–27 (cit. on pp. 40, 45, 128).

[AOL19] E. Ahvar, A. Orgerie, and A. Lébre. "Estimating Energy Consumption of Cloud, Fog and Edge Computing Infrastructures". In: *IEEE Transactions on Sustainable Computing* (2019), pp. 1–1 (cit. on p. 13).

[AP17] Ola Angelsmark and Per Persson. "Requirement-Based Deployment of Applications in Calvin". In: *Interoperability and Open-Source Solutions for the Internet of Things: Second International Workshop, InterOSS-IoT 2016, Held in Conjunction with IoT 2016, Stuttgart, Germany, November 7, 2016, Invited Papers*. Ed. by Ivana Podnar Žarko, Arne Broering, Sergios Soursos, and Martin Serrano. Cham: Springer International Publishing, 2017, pp. 72–87 (cit. on p. 65).

[AR95] Thomas A. Feo and Mauricio Resende. "Greedy Randomized Adaptive Search Procedures". In: 6 (Mar. 1995), pp. 109–133 (cit. on p. 112).

[ATT17] AT&TLabs and AT&T Foundry. *AT&T Edge Cloud(AEC) - White Paper*. Tech. rep. 2017 (cit. on p. 47).

[Aue+02] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. "The Nonstochastic Multiarmed Bandit Problem". In: *SIAM Journal on Computing* 32.1 (2002), pp. 48–77 (cit. on pp. 131, 146).

[Aue+95] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. "Gambling in a rigged casino: The adversarial multi-armed bandit problem". In: *Proceedings of IEEE 36th Annual Foundations of Computer Science*. 1995, pp. 322–331 (cit. on p. 147).

[AV16] M. A. Al Faruque and K. Vatanparvar. "Energy Management-as-a-Service Over Fog Computing Platform". In: *IEEE Internet of Things Journal* 3.2 (Apr. 2016), pp. 161–169 (cit. on p. 20).

[Azz+14] A. Azzarà, D. Alessandrelli, S. Bocchino, M. Petracca, and P. Pagano. "PyoT, a macroprogramming framework for the Internet of Things". In: *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. 2014, pp. 96–103 (cit. on p. 59).

[Bal+13] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Ed. by Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20 (cit. on p. 75).

[BCM98] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. "Flow and stretch metrics for scheduling continuous job streams". English (US). In: Proceedings of the 1998 9th Annual ACM SIAM Symposium on Discrete Algorithms ; Conference date: 25-01-1998 Through 27-01-1998. Dec. 1998, pp. 270–279 (cit. on pp. 127, 128).

[Bel+18] Elena Veronica Belmega, Panayotis Mertikopoulos, Romain Negrel, and Luca Sanguinetti. "Online convex optimization and no-regret learning: Algorithms, guarantees and applications". In: *CoRR* abs/1804.04529 (2018). arXiv: 1804.04529 (cit. on p. 141).

[BF17] A. Brogi and S. Forti. "QoS-Aware Deployment of IoT Applications Through the Fog". In: *IEEE Internet of Things Journal* 4.5 (Oct. 2017), pp. 1185–1192 (cit. on pp. 39, 44).

[BG17] Tayebeh Bahreini and Daniel Grosu. "Efficient Placement of Multi-component Applications in Edge Computing Systems". In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. SEC '17. San Jose, California: ACM, 2017, 5:1–5:11 (cit. on pp. 38, 42).

[BGT16] B. Butzin, F. Golatowski, and D. Timmermann. "Microservices approach for the internet of things". In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. Sept. 2016, pp. 1–6 (cit. on p. 48).

[Bon+12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. "Fog Computing and Its Role in the Internet of Things". In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. Helsinki, Finland: ACM, 2012, pp. 13–16 (cit. on pp. 3, 9).

[Bon+14] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. "Fog Computing: A Platform for Internet of Things and Analytics". In: *Big Data and Internet of Things: A Roadmap for Smart Environments*. Ed. by Nik Bessis and Ciprian Dobre. Cham: Springer International Publishing, 2014, pp. 169–186 (cit. on p. 20).

[Bri+17] M. S. de Brito, S. Hoque, T. Magedanz, et al. “A service orchestration architecture for Fog-enabled infrastructures”. In: *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*. May 2017, pp. 127–132 (cit. on pp. 30, 31, 33).

[Brz+16] R. Brzoza-Woch, M. Konieczny, P. Nawrocki, T. Szydlo, and K. Zielinski. “Embedded systems in the application of fog computing — Levee monitoring use case”. In: *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*. May 2016, pp. 1–6 (cit. on p. 17).

[BS15] H. P. Breivold and K. Sandström. “Internet of Things for Industrial Automation – Challenges and Technical Solutions”. In: *2015 IEEE International Conference on Data Science and Data Intensive Systems*. Dec. 2015, pp. 532–539 (cit. on p. 18).

[CDS13] Nicolò Cesa-Bianchi, Ofer Dekel, and Ohad Shamir. “Online Learning with Switching Costs and Other Adaptive Adversaries”. In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’13. Lake Tahoe, Nevada: Curran Associates Inc., 2013, pp. 1160–1168 (cit. on p. 146).

[CG19] T. Chen and G. B. Giannakis. “Bandit Convex Optimization for Scalable and Dynamic IoT Management”. In: *IEEE Internet of Things Journal* 6.1 (Feb. 2019), pp. 1276–1286 (cit. on pp. 40, 45, 132).

[Cha+16] S. Chauhan, P. Patel, A. Sureka, F. C. Delicato, and S. Chaudhary. “Demonstration Abstract: IoTSuite - A Framework to Design, Implement, and Deploy IoT Applications”. In: *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. 2016, pp. 1–2 (cit. on p. 59).

[CML11] Eunjoon Cho, Seth A. Myers, and Jure Leskovec. “Friendship and Mobility: User Movement in Location-Based Social Networks”. In: *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’11. San Diego, California, USA: Association for Computing Machinery, 2011, pp. 1082–1090 (cit. on p. 174).

[Con17] OpenFog Consortium. *OpenFog Reference Architecture for Fog Computing*. eng. 2017 (cit. on p. 13).

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. Chap. 16 (cit. on p. 101).

[Den+15] R. Deng, R. Lu, C. Lai, and T. H. Luan. “Towards power consumption-delay tradeoff by workload allocation in cloud-fog computing”. In: *2015 IEEE International Conference on Communications (ICC)*. June 2015, pp. 3909–3914 (cit. on pp. 31, 32).

[Den+16] R. Deng, R. Lu, C. Lai, T. H. Luan, and H. Liang. “Optimal Workload Allocation in Fog-Cloud Computing Toward Balanced Delay and Power Consumption”. In: *IEEE Internet of Things Journal* 3.6 (Dec. 2016), pp. 1171–1181 (cit. on pp. 31, 32, 38, 42).

[Diz+19] Jasenka Dizdareviundefined, Francisco Carpio, Admela Jukan, and Xavi Masip-Bruin. “A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration”. In: *ACM Comput. Surv.* 51.6 (Jan. 2019) (cit. on p. 48).

[Dro09] Maciej Drozdowski. “Scheduling for Parallel Processing”. In: 1st. Springer Publishing Company, 2009. Chap. 5 (cit. on p. 127).

[ElS+18] H. El-Sayed, S. Sankar, M. Prasad, et al. “Edge of Things: The Big Picture on the Integration of Edge, IoT and the Cloud in a Distributed Computing Environment”. In: *IEEE Access* 6 (2018), pp. 1706–1717 (cit. on p. 24).

[Eva12] David Evans. *The Internet of Everything: How More Relevant and Valuable Connections Will Change the World. Cisco Internet Business Solutions Group (IBSG)*. 2012 (cit. on p. 1).

[Fan+12] X. Fang, S. Misra, G. Xue, and D. Yang. “Smart Grid — The New and Improved Power Grid: A Survey”. In: *IEEE Communications Surveys Tutorials* 14.4 (Fourth 2012), pp. 944–980 (cit. on p. 20).

[Fan+17] Weidong Fang, Wuxiong Zhang, Jinchao Xiao, Yang Yang, and Wei Chen. “A Source Anonymity-Based Lightweight Secure AODV Protocol for Fog-Based MANET”. In: *Sensors* 17.6 (2017) (cit. on p. 24).

[FR95] Thomas A. Feo and Mauricio G. C. Resende. “Greedy Randomized Adaptive Search Procedures”. In: *Journal of Global Optimization* 6.2 (Mar. 1995), pp. 109–133 (cit. on p. 93).

[Fre20] Fredrik Jejdling et al. *Ericsson Mobility Report*. Tech. rep. June 2020 (cit. on p. 2).

[GBS17] D. Gannon, R. Barga, and N. Sundaresan. “Cloud-Native Applications”. In: *IEEE Cloud Computing* 4.5 (2017), pp. 16–21 (cit. on p. 174).

[Gha+17] A. Gharaibeh, M. A. Salahuddin, S. J. Hussini, et al. “Smart Cities: A Survey on Data Management, Security, and Enabling Technologies”. In: *IEEE Communications Surveys Tutorials* 19.4 (Fourthquarter 2017), pp. 2456–2501 (cit. on p. 17).

[Gia+15] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C. M. Leung. “Developing IoT applications in the Fog: A Distributed Dataflow approach”. In: *5th International Conference on the Internet of Things, IOT 2015, Seoul, South Korea, 26-28 October, 2015*. 2015, pp. 155–162 (cit. on pp. 12, 39, 43, 59).

[Gra+79] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. “Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey”. In: *Discrete Optimization II*. Ed. by P.L. Hammer, E.L. Johnson, and B.H. Korte. Vol. 5. Annals of Discrete Mathematics. Elsevier, 1979, pp. 287–326 (cit. on p. 127).

[Gu+17] L. Gu, D. Zeng, S. Guo, A. Barnawi, and Y. Xiang. “Cost Efficient Resource Management in Fog Computing Supported Medical Cyber-Physical System”. In: *IEEE Transactions on Emerging Topics in Computing* 5.1 (Jan. 2017), pp. 108–119 (cit. on pp. 39, 44).

[GVS16] R. Geissbauer, J. Vedso, and S. Schrauf. “Industry 4.0: Building the digital enterprise”. In: *PwC 2016 Global Industry 4.0 Survey* (2016) (cit. on p. 18).

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. “Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence”. In: *Advance Papers of the Conference*. Vol. 3. Stanford Research Institute. 1973, p. 235 (cit. on p. 61).

[Hew10] Carl Hewitt. *Actor Model of Computation: Scalable Robust Information Systems*. 2010. arXiv: 1008.1459 [cs.PL] (cit. on pp. 55, 61).

[HGW09] X. Huang, S. Ganapathy, and T. Wolf. "Evaluating Algorithms for Composable Service Placement in Computer Networks". In: *2009 IEEE International Conference on Communications*. June 2009, pp. 1–6 (cit. on p. 99).

[HL09] Hannes Hartenstein and Kenneth Laberteaux. *VANET: Vehicular Applications and Inter-Networking Technologies*. Jan. 2009, pp. 1–435 (cit. on p. 19).

[Hon+13] Kirak Hong, David J. Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. "Mobile fog: a programming model for large-scale applications on the internet of things". In: *MCC@SIGCOMM*. 2013 (cit. on pp. 30, 31, 34).

[Hon+17] H. Hong, P. Tsai, A. Cheng, et al. "Supporting Internet-of-Things Analytics in a Fog Computing Platform". In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Dec. 2017, pp. 138–145 (cit. on pp. 38, 42).

[Hou+16a] X. Hou, Y. Li, M. Chen, et al. "Vehicular Fog Computing: A Viewpoint of Vehicles as the Infrastructures". In: *IEEE Transactions on Vehicular Technology* 65.6 (June 2016), pp. 3860–3873 (cit. on pp. 31, 34).

[Hou+16b] Xueshi Hou, Yong Li, Min Chen, et al. "Vehicular Fog Computing: A Viewpoint of Vehicles as the Infrastructures". In: *IEEE Trans. Vehicular Technology* 65.6 (2016), pp. 3860–3873 (cit. on p. 11).

[IEE18] IEEE. *IEEE Std 1934-2018: IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing*. eng. 2018 (cit. on p. 15).

[Ior+18] M. Iorga, L. Feldman, R. Barton, et al. *Fog Computing Conceptual Model*. Tech. rep. SP 500-325. Gaithersburg, MD, United States: National Institute of Standards and Technology (NIST), Mar. 2018 (cit. on pp. 9, 15).

[ITU17] ITU-R Radiocommunication Sector of ITU. *Minimum requirements related to technical performancefor IMT-2020 radiointerface(s)*. Technical Report. Sophia Antipolis - FR: ITU - International Telecomunication Union, Nov. 2017 (cit. on p. 2).

[Jal+16] F. Jalali, K. Hinton, R. Ayre, T. Alpcan, and R. S. Tucker. "Fog Computing May Help to Save Energy in Cloud Computing". In: *IEEE Journal on Selected Areas in Communications* 34.5 (May 2016), pp. 1728–1739 (cit. on p. 13).

[Jaz14] N. Jazdi. "Cyber physical systems in the context of Industry 4.0". In: *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*. May 2014, pp. 1–4 (cit. on p. 18).

[JHT18] Y. Jiang, Z. Huang, and D. H. K. Tsang. "Challenges and Solutions in Fog Computing Orchestration". In: *IEEE Network* 32.3 (May 2018), pp. 122–129 (cit. on p. 4).

[JRL08] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. "Choco: an Open Source Java Constraint Programming Library". In: *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*. Paris, France, France, 2008, pp. 1–10 (cit. on p. 45).

[Kap+17] A. Kapsalis, P. Kasnesis, I. S. Venieris, D. I. Kaklamani, and C. Z. Patrikakis. "A Cooperative Fog Approach for Effective Workload Balancing". In: *IEEE Cloud Computing* 4.2 (Mar. 2017), pp. 36–45 (cit. on pp. 31, 33).

[KC03] J. O. Kephart and D. M. Chess. "The vision of autonomic computing". In: *Computer* 36.1 (Jan. 2003), pp. 41–50 (cit. on p. 21).

[KCT16] Kang Kai, Wang Cong, and Luo Tao. "Fog computing for vehicular Ad-hoc networks: paradigms, scenarios, and issues". In: *The Journal of China Universities of Posts and Telecommunications* 23.2 (2016), pp. 56–96 (cit. on pp. 19, 24).

[KDB15] F. Khodadadi, A. V. Dastjerdi, and R. Buyya. "Simurgh: A framework for effective discovery, programming, and integration of services exposed in IoT". In: *2015 International Conference on Recent Advances in Internet of Things (RIoT)*. 2015, pp. 1–6 (cit. on p. 59).

[KKL13] Thorsten Kramp, Rob van Kranenburg, and Sebastian Lange. "Introduction to the Internet of Things". In: *Enabling Things to Talk: Designing IoT solutions with the IoT Architectural Reference Model*. Ed. by Alessandro Bassi, Martin Bauer, Martin Fiedler, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–10 (cit. on p. 1).

[KL14] J. Kim and J. Lee. "OpenIoT: An open service framework for the Internet of Things". In: *2014 IEEE World Forum on Internet of Things (WF-IoT)*. 2014, pp. 89–93 (cit. on p. 59).

[KS18] Minhaj Ahmad Khan and Khaled Salah. "IoT security: Review, blockchain solutions, and open challenges". In: *Future Generation Computer Systems* 82 (2018), pp. 395–411 (cit. on p. 25).

[LaC+14] Katrina LaCurts, Jeffrey C. Mogul, Hari Balakrishnan, and Yoshio Turner. "Cicada: Introducing Predictive Guarantees for Cloud Networks". In: *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*. Philadelphia, PA: USENIX Association, June 2014 (cit. on p. 174).

[LSV08] Arnaud Legrand, Alan Su, and Frédéric Vivien. "Minimizing the stretch when scheduling flows of divisible requests". In: *Journal of Scheduling* 11.5 (2008), pp. 381–404 (cit. on p. 128).

[Man+17] Simone Mangiante, Guenter Klas, Amit Navon, et al. "VR is on the Edge: How to Deliver 360° Videos in Mobile Networks". In: *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*. VR/AR Network '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 30–35 (cit. on p. 2).

[Mas+16] X. Masip-Bruin, E. Marín-Tordera, G. Tashakor, A. Jukan, and G. Ren. "Foggy clouds and cloudy fogs: a real need for coordinated management of fog-to-cloud computing systems". In: *IEEE Wireless Communications* 23.5 (Oct. 2016), pp. 120–128 (cit. on pp. 31, 32).

[Mou+18] C. Mouradian, D. Naboulsi, S. Yangui, et al. "A Comprehensive Survey on Fog Computing: State-of-the-Art and Research Challenges". In: *IEEE Communications Surveys Tutorials* 20.1 (Firstquarter 2018), pp. 416–464 (cit. on p. 22).

[MSW18] M. Mukherjee, L. Shu, and D. Wang. "Survey of Fog Computing: Fundamental, Network Applications, and Research Challenges". In: *IEEE Communications Surveys Tutorials* 20.3 (thirdquarter 2018), pp. 1826–1857 (cit. on p. 22).

[Nah+18] R. K. Naha, S. Garg, D. Georgakopoulos, et al. “Fog Computing: Survey of Trends, Architectures, Requirements, and Research Directions”. In: *IEEE Access* 6 (2018), pp. 47980–48009 (cit. on p. 22).

[Nas+13] S. Nastic, S. Sehic, M. Vögler, H. Truong, and S. Dustdar. “PatRICIA – A Novel Programming Model for IoT Applications on Cloud Platforms”. In: *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. 2013, pp. 53–60 (cit. on p. 59).

[NHE14] I. Nakagawa, M. Hiji, and H. Esaki. “Dripcast – Server-less Java Programming Framework for Billions of IoT Devices”. In: *2014 IEEE 38th International Computer Software and Applications Conference Workshops*. 2014, pp. 186–191 (cit. on p. 59).

[Nis+13] Takayuki Nishio, Ryoichi Shinkuma, Tatsuro Takahashi, and Narayan B. Mandayam. “Service-oriented Heterogeneous Resource Sharing for Optimizing Service Latency in Mobile Cloud”. In: *Proceedings of the First International Workshop on Mobile Cloud Computing & Networking*. MobileCloud ’13. Bangalore, India: ACM, 2013, pp. 19–26 (cit. on pp. 39, 43).

[OO16] F. Y. Okay and S. Ozdemir. “A fog computing based smart grid model”. In: *2016 International Symposium on Networks, Computers and Communications (ISNCC)*. May 2016, pp. 1–6 (cit. on p. 20).

[Ord+14] Juan Luis Pérez Ordóñez, Álvaro Villalba, David Carrera, Iker Larizgoitia, and Vlad Trifa. “The COMPOSE API for the internet of things”. In: *WWW ’14 Companion*. 2014 (cit. on p. 59).

[OSB15] J. Oueis, E. C. Strinati, and S. Barbarossa. “The Fog Balancing: Load Distribution for Small Cell Cloud Computing”. In: *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*. May 2015, pp. 1–6 (cit. on pp. 39, 44).

[Ott+13] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, and Umakishore Ramachandran. “MigCEP: Operator Migration for Mobility Driven Distributed Complex Event Processing”. In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. DEBS ’13. Arlington, Texas, USA: ACM, 2013, pp. 183–194 (cit. on pp. 39, 44).

[PA15] Per Persson and Ola Angelsmark. “Calvin – Merging Cloud and IoT”. In: *Procedia Computer Science* 52 (2015). The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015), pp. 210–217 (cit. on pp. 59, 60, 62).

[Per+17] Charith Perera, Yongrui Qin, Julio C. Estrella, Stephan Reiff-Marganiec, and Athanasios V. Vasilakos. “Fog Computing for Sustainable Smart Cities: A Survey”. In: *ACM Comput. Surv.* 50.3 (June 2017), 32:1–32:43 (cit. on pp. 17, 18).

[Pul+19] Carlo Puliafito, Enzo Mingozzi, Francesco Longo, Antonio Puliafito, and Omer Rana. “Fog Computing for the Internet of Things: A Survey”. In: *ACM Trans. Internet Technol.* 19.2 (Apr. 2019), 18:1–18:41 (cit. on pp. 18, 22).

[Rah+18] Amir M. Rahmani, Tuan Nguyen Gia, Behailu Negash, et al. “Exploiting smart e-Health gateways at the edge of healthcare Internet-of-Things: A fog computing approach”. In: *Future Generation Computer Systems* 78 (2018), pp. 641–658 (cit. on pp. 31, 35).

[SA16] X. Sun and N. Ansari. “EdgeIoT: Mobile Edge Computing for the Internet of Things”. In: *IEEE Communications Magazine* 54.12 (Dec. 2016), pp. 22–29 (cit. on pp. 30, 31, 129).

[Sau+16] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. “Incremental Deployment and Migration of Geo-distributed Situation Awareness Applications in the Fog”. In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS ’16. Irvine, California: ACM, 2016, pp. 258–269 (cit. on pp. 30, 31, 34, 38, 42, 129).

[Sch+12] Eric Schulte, Dan Davison, Thomas Dye, and Carsten Dominik. “A Multi-Language Computing Environment for Literate Programming and Reproducible Research”. In: *Journal of Statistical Software, Articles* 46.3 (2012), pp. 1–24 (cit. on p. 87).

[Sch+17] P. Schulz, M. Matthe, H. Klessig, et al. “Latency Critical IoT Applications in 5G: Perspective on the Design of Radio Interface and Network Architecture”. In: *IEEE Communications Magazine* 55.2 (Feb. 2017), pp. 70–78 (cit. on pp. 2, 20).

[SCM18] S. Sarkar, S. Chatterjee, and S. Misra. “Assessment of the Suitability of Fog Computing in the Context of Internet of Things”. In: *IEEE Transactions on Cloud Computing* 6.1 (Jan. 2018), pp. 46–59 (cit. on pp. 31, 32).

[Ska+17a] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar. “Towards QoS-Aware Fog Service Placement”. In: *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. May 2017, pp. 89–96 (cit. on pp. 38, 41, 128).

[Ska+17b] Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner. “Optimized IoT service placement in the fog”. In: *Service Oriented Computing and Applications* 11.4 (Dec. 2017), pp. 427–443 (cit. on pp. 38, 41, 128).

[SM16] S. Sarkar and S. Misra. “Theoretical modelling of fog computing: a green computing paradigm to support IoT applications”. In: *IET Networks* 5.2 (2016), pp. 23–29 (cit. on pp. 31, 32).

[SWW95] David B. Shmoys, Joel Wein, and David P. Williamson. “Scheduling Parallel Machines On-Line”. In: *SIAM Journal of Computing* 24.6 (1995), pp. 1313–1331 (cit. on p. 127).

[Sys09] ETSI Technical Committee Intelligent Transport System. *Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Definitions*. Technical Report. Sophia Antipolis - FR: ETSI - European Telecommunications Standards Institute, June 2009 (cit. on p. 19).

[Sys18] ETSI Technical Committee Intelligent Transport System. *Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Facilities layer protocols and communication requirements for infrastructure services*. Technical Specification. Sophia Antipolis - FR: ETSI - European Telecommunications Standards Institute, Aug. 2018 (cit. on p. 19).

[Tal+20] T. Taleb, R.L. Aguiar, I. Grida Ben Yahia, et al. *WHITE PAPER ON 6G NETWORKING*. und. 2020 (cit. on p. 3).

[Tan+15] Bo Tang, Zhen Chen, Gerald Hefferman, et al. “A Hierarchical Distributed Fog Computing Architecture for Big Data Analysis in Smart Cities”. In: *Proceedings of the ASE BigData & SocialInformatics 2015*. ASE BD&SI ’15. Kaohsiung, Taiwan: ACM, 2015, 28:1–28:6 (cit. on pp. 31, 32).

[Tan+17] B. Tang, Z. Chen, G. Hefferman, et al. “Incorporating Intelligence in Fog Computing for Big Data Analysis in Smart Cities”. In: *IEEE Transactions on Industrial Informatics* 13.5 (Oct. 2017), pp. 2140–2150 (cit. on pp. 17, 31, 32).

[TLG16] L. Tong, Y. Li, and W. Gao. “A hierarchical edge cloud architecture for mobile computing”. In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. Apr. 2016, pp. 1–9 (cit. on pp. 30, 31, 40, 45).

[Tom+17] Slavica Tomovic, Kenji Yoshigoe, Ivo Maljevic, and Igor Radusinovic. “Software-Defined Fog Network Architecture for IoT”. In: *Wireless Personal Communications* 92.1 (Jan. 2017), pp. 181–196 (cit. on pp. 31, 34).

[VR14] Luis M. Vaquero and Luis Rodero-Merino. “Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing”. In: *SIGCOMM Comput. Commun. Rev.* 44.5 (Oct. 2014), pp. 27–32 (cit. on pp. 9, 22).

[Wan+17] S. Wang, R. Urgaonkar, T. He, et al. “Dynamic Service Placement for Mobile Micro-Clouds with Predicted Future Costs”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.4 (Apr. 2017), pp. 1002–1016 (cit. on p. 128).

[Wan+18] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos. “ENORM: A Framework For Edge NOde Resource Management”. In: *IEEE Transactions on Services Computing* (2018), pp. 1–1 (cit. on pp. 30, 31, 34, 129).

[Wen+17] Z. Wen, R. Yang, P. Garraghan, et al. “Fog Orchestration for Internet of Things Services”. In: *IEEE Internet Computing* 21.2 (Mar. 2017), pp. 16–24 (cit. on pp. 31, 34).

[Xia+18] Ye Xia, Xavier Etchevers, Loïc Letondeur, Thierry Coupaye, and Frédéric Desprez. “Combining Hardware Nodes and Software Components Ordering-based Heuristics for Optimizing the Placement of Distributed IoT Applications in the Fog”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC ’18. Pau, France: ACM, 2018, pp. 751–760 (cit. on pp. 38, 41).

[Xio+18] Y. Xiong, Y. Sun, L. Xing, and Y. Huang. “Extend Cloud to Edge with KubeEdge”. In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. 2018, pp. 373–377 (cit. on pp. 31, 35).

[Xu+18] W. Xu, H. Zhou, N. Cheng, et al. “Internet of vehicles in big data era”. In: *IEEE/CAA Journal of Automatica Sinica* 5.1 (Jan. 2018), pp. 19–35 (cit. on p. 11).

[Yan+16] S. Yangui, P. Ravindran, O. Bibani, et al. “A platform as-a-service for hybrid cloud/fog environments”. In: *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. June 2016, pp. 1–7 (cit. on pp. 30, 31, 33).

[Yan+17] M. Yannuzzi, R. Irons-Mclean, F. van Lingen, et al. “Toward a converged OpenFog and ETSI MANO architecture”. In: *2017 IEEE Fog World Congress (FWC)*. Oct. 2017, pp. 1–6 (cit. on pp. 31, 33).

[Yi+15] S. Yi, Z. Hao, Z. Qin, and Q. Li. "Fog Computing: Platform and Applications". In: *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. Nov. 2015, pp. 73–78 (cit. on pp. 30, 31).

[YLL15] Shanhe Yi, Cheng Li, and Qun Li. "A Survey of Fog Computing: Concepts, Applications and Issues". In: *Proceedings of the 2015 Workshop on Mobile Big Data*. Mobidata '15. Hangzhou, China: ACM, 2015, pp. 37–42 (cit. on p. 2).

[You+18] Ashkan Yousefpour, Ashish Patil, Genya Ishigaki, et al. "QoS-aware Dynamic Fog Service Provisioning". In: *CoRR* abs/1802.00800 (2018). arXiv: 1802.00800 (cit. on pp. 38, 41, 128).

[You09] H. Peyton Young. "Learning by trial and error". In: *Games and Economic Behavior* 65.2 (2009), pp. 626–643 (cit. on p. 149).

[YQL15] Shanhe Yi, Zhengrui Qin, and Qun Li. "Security and Privacy Issues of Fog Computing: A Survey". In: *Wireless Algorithms, Systems, and Applications*. Ed. by Kuai Xu and Haojin Zhu. Cham: Springer International Publishing, 2015, pp. 685–695 (cit. on p. 25).

[Zen+16] D. Zeng, L. Gu, S. Guo, Z. Cheng, and S. Yu. "Joint Optimization of Task Scheduling and Image Placement in Fog Computing Supported Software-Defined Embedded System". In: *IEEE Transactions on Computers* 65.12 (Dec. 2016), pp. 3702–3712 (cit. on pp. 39, 43).

[Zha+19] Z. Zhang, Y. Xiao, Z. Ma, et al. "6G Wireless Networks: Vision, Requirements, Architecture, and Key Technologies". In: *IEEE Vehicular Technology Magazine* 14.3 (2019), pp. 28–41 (cit. on p. 3).

Webpages

[Ard] Arduino. *Arduino Products*. URL: https://www.arduino.cc/en/Main/Products (visited on July 23, 2020) (cit. on p. 22).

[Coc05] Alistair Cockburn. *Hexagonal architecture*. Jan. 2005. URL: https://alistair.cockburn.us/hexagonal-architecture/ (visited on June 14, 2020) (cit. on p. 48).

[Dme20] Anasia D'mello. *Global IoT market to grow to $1.5trn annual revenue by 2030*. May 2020. URL: https://www.iot-now.com/2020/05/20/102937-global-iot-market-to-grow-to-1-5trn-annual-revenue-by-2030/ (visited on June 19, 2020) (cit. on p. 1).

[FIT] FIT/IoT-LAB. *FIT/IoT-LAB: What is IoT-LAB?* URL: https://www.iot-lab.info/what-is-iot-lab/ (visited on June 2, 2020) (cit. on p. 75).

[Fow15] Martin Fowler. *Microservice Trade-Offs*. July 2015. URL: https://martinfowler.com/articles/microservice-trade-offs.html (visited on June 12, 2020) (cit. on pp. 48, 50).

[Gri] Grid'5000. *Grid'5000: Hardware*. URL: https://www.grid5000.fr/w/Hardware (visited on June 2, 2020) (cit. on p. 75).

[IEA20] IEA. *Global Energy Review 2019*. Apr. 2020. URL: https://www.iea.org/reports/global-energy-review-2019 (visited on June 5, 2020) (cit. on p. 20).

[JS 13] JS Foundation. *Node-RED - Low-code programming for event-driven applications*. Sept. 2013. URL: https://nodered.org/ (visited on July 25, 2020) (cit. on p. 43).

[Mad19] Madynes team - LORIA. *DISTEM - DISTributed systems EMulator*. Mar. 2019. URL: http://distem.gforge.inria.fr/index.html (visited on June 25, 2020) (cit. on p. 175).

[Pet20] Christo Petrov. *47 Stunning Internet of Things Statistics 2020 [The Rise Of IoT]*. June 2020. URL: https://techjury.net/blog/internet-of-things-statistics (visited on July 7, 2020) (cit. on p. 1).

[Ran] Rancher. *K3s: Lightweight Kubernetes*. URL: https://k3s.io/ (visited on Feb. 27, 2020) (cit. on pp. 30, 31, 35, 129, 130).

[Ras] Raspberry Pi Foundation. *Raspberry Pi Products*. URL: https://www.raspberrypi.org/products/ (visited on July 23, 2020) (cit. on p. 22).

[SIL] SILECS. *SILECS: Super Infrastructure for Large-Scale Experimental Computer Science*. URL: https://www.silecs.net/ (visited on June 4, 2020) (cit. on p. 78).

[The98] The CMU CS Department Coke Machine. *CMU SCS Coke Machine*. June 1998. URL: https://www.cs.cmu.edu/~coke/ (visited on June 18, 2020) (cit. on p. 1).

Software

[Doc13] [SW] Docker Inc., *Docker*, Mar. 20, 2013. URL: https://www.docker.com/.

[Goo14] [SW] Google, *cAdvisor (Container Advisor)*, June 9, 2014. URL: https://github.com/google/cadvisor.

[Net13] [SW] Netdata Inc., *Netdata*, June 17, 2013. URL: https://github.com/netdata/netdata.

[Ope01] [SW] OpenVPN Inc., *OpenVPN – A Secure tunneling daemon*, May 13, 2001. URL: https://github.com/OpenVPN/openvpn.

[Pro12] [SW] Prometheus Authors, *Prometheus*, Nov. 24, 2012. URL: https://prometheus.io/.

[Pro15] [SW] Prometheus Authors, *Blackbox exporter*, Sept. 5, 2015. URL: https://github.com/prometheus/blackbox_exporter.

[Red12] [SW] Red Hat, *Ansible*, Feb. 20, 2012. URL: https://www.ansible.com/.

[TW02] [SW] Costa Tsaousis and Phil Whineray, *FireQoS*, Sept. 5, 2002. URL: https://firehol.org/.

My Papers

[Don+18] Bruno Donassolo, Ilhem Fajjari, Arnaud Legrand, and Panayotis Mertikopoulos. *FogIoT Orchestrator: an Orchestration System for IoT Applications in Fog Environment*. 1st Grid'5000-FIT school. Apr. 2018 (cit. on p. 4).

[Don+19a] B. Donassolo, I. Fajjari, A. Legrand, and P. Mertikopoulos. "Demo: Fog Based Framework for IoT Service Orchestration". In: *2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC)*. 2019, pp. 1–2 (cit. on p. 4).

[Don+19b] B. Donassolo, I. Fajjari, A. Legrand, and P. Mertikopoulos. "Fog Based Framework for IoT Service Provisioning". In: *2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC)*. 2019, pp. 1–6 (cit. on p. 5).

[Don+19c] B. Donassolo, I. Fajjari, A. Legrand, and P. Mertikopoulos. "Load Aware Provisioning of IoT Services on Fog Computing Platform". In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. 2019, pp. 1–7 (cit. on p. 5).

[Don+20] Bruno Donassolo, Arnaud Legrand, Panayotis Mertikopoulos, and Ilhem Fajjari. "Online Reconfiguration of IoT Applications in the Fog: The Information-Coordination Trade-off". working paper or preprint. May 2020 (cit. on p. 5).

List of Figures

List of Tables