Online Reconfiguration of IoT Applications in the Fog: The Information-Coordination Trade-off

Bruno Donassolo, Arnaud Legrand, Panayotis Mertikopoulos, Ilhem Fajjari

Online Reconfiguration of IoT Applications in the Fog: The Information-Coordination Trade-off

Bruno Donassolo, Arnaud Legrand, Panayotis Mertikopoulos, and Ilhem Fajjari

Abstract—The Internet of Things (IoT) continues its evolution, causing an extraordinary growth of traffic and processing demands. Consequently, 5G players are persuaded to change their infrastructures. In this context, Fog computing emerges as a potential solution, providing nearby resources to run IoT applications. However, the Fog and the IoT raise several challenges which decelerate the adoption of the Fog paradigm. In this paper, we consider the reconfiguration problem, i.e., how to dynamically adapt the placement of IoT applications running in the Fog, depending on application needs and evolution of resource usage. We propose and evaluate a series of reconfiguration algorithms, based on both online scheduling and online learning approaches. Through an extensive set of experiments in a realistic testbed, we demonstrate that the performance strongly and mainly depends on the quality and availability of information from both Fog infrastructure and IoT applications. Finally, 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.

Index Terms—Fog computing, Internet of Things, Reconfiguration, Online learning, Online scheduling

1 Introduction

The rapid increase in the number of IoT (Internet of Things) devices is transforming IoT applications, bringing new challenges to 5G players and their infrastructures. These revolutionary IoT applications have growing traffic and stringent requirements that Cloud infrastructures are struggling to meet. In this context, Fog computing [1] comes to bridge Cloud data centers and edge devices, providing nearby resources to run IoT applications. To do so, Fog computing relies on geographically distributed and heterogeneous devices, called Fog nodes. The latter perform nearby analytics and data storage, and they can be either virtualized or physical, depending on the hardware characteristics. Thus, by taking advantage of all resources available in the spectrum between sensors and the Cloud, the Fog can ensure low delay for latency-sensitive applications and high processing capabilities for data-intensive ones.

The geo-distribution and heterogeneity of Fog nodes raise new challenges for the management of IoT applications. More specifically, one important issue is how to guarantee a high-level quality of service during the application's lifetime. In this perspective, the orchestration of IoT applications in the Fog is the cornerstone that handles the life cycle management of multi-component IoT applications. Orchestrator missions include managing the reconfiguration of IoT applications running in the Fog environment. The reconfiguration aims to adapt the resources used by applications to keep them satisfied and running smoothly.

Several mechanisms, such as the horizontal/vertical scaling or the migration of an application's component, can be leveraged. In this paper we address the reconfiguration of IoT applications in a large, distributed and heterogeneous environment such as the Fog through application migration. We conduct an in-depth and comparative study of diverse reconfiguration strategies in a realistic Fog environment and show how the availability and reliability of information about infrastructure and applications can affect their performance. The main contributions of this paper are summarized as follows:

1) 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.
2) We rely on a unified experimental framework for Fog computing which enables us to evaluate and compare the different strategies in a fair and realistic manner.
3) 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.
4) 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.
5) Finally, 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.

The rest of the paper is organized as follows. In Section 2

- *B. Donassolo was with Orange Labs and Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France*
- *A. Legrand and P. Mertikopoulos are with Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France*
- *I. Fajjari is with Orange Labs*

we give an overview about the Fog, IoT applications and the reconfiguration problem. In Section 3, we describe some related work on online reconfiguration algorithms. The experimental details about the platform, workload and orchestrator are presented in Section 4, while Section 5 describes the rules of the game. In Section 6, we present the reconfiguration strategies along with an analysis of their performance in an faithful context. The analysis of their performance in a scenario with inaccurate information is presented in Section 7. Finally, Section 8 draws out conclusions.

2 Configuration of IoT Applications in the Fog: Context and Challenges

2.1 Fog & IoT

Fog computing has emerged as an alternative to deal with the burden of data-centers and network in Cloud infrastructures. By extending the Cloud towards the edge of the network, the Fog is capable of supporting the geographically distributed, latency sensitive or bandwidth intensive IoT applications. The term Fog was first proposed by Cisco [1], and its name comes directly from nature, as the fog can be seen as clouds near the ground.

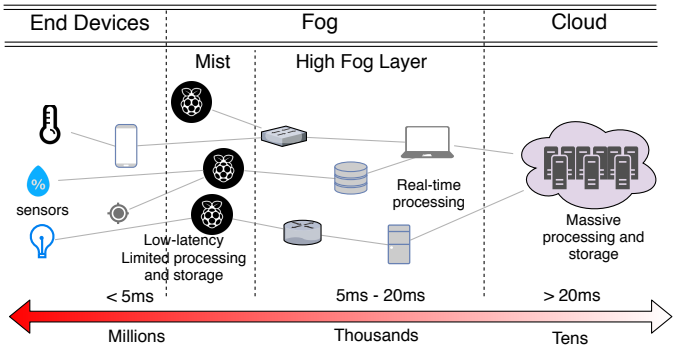

Fig. 1: Fog Infrastructure

Fig. 1 illustrates a typical Fog environment, which provides computing, storage and network services anywhere along the continuum from Cloud to end users. We refer as Fog nodes all devices that can run IoT applications between end devices and the Cloud, either in the Mist or High Fog layers. Fog nodes perform nearby analytics and data storage to IoT applications. They differ greatly in terms of hardware and software capabilities. In this complex, large-scale, distributed and heterogeneous platform, IoT applications must be properly managed to run efficiently.

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 model to cope with these challenges [2]. 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.

In this scenario, an application can be seen as a DAG (Directed Acyclic Graph), where nodes are micro-services and edges represent the message stream exchanged between them. In this graph, the IoT sensors are typically the source of the information, sensing the environment and sending the measured data to parent nodes for further processing. Fig. 2 illustrates a Smart Traffic Light application, responsible for adjusting the timing of traffic lights following the cars and pedestrians movement. This 3-layer application presents important characteristics of IoT applications running on the Fog, such as longevity and QoS (Quality of Service) requirements. In the bottom, there are sensors and actuators that collect data and act in the environment, requiring low latency link for the modules in the upper layer. The collected data is sent to be analyzed by micro-services in the processing layer. In this layer, for example, the "Car Recognition" module requires high processing and memory capacity. Finally, the top layer is responsible for long-term analysis and optimization. All these requirements must be considered when placing the application since they may impact QoS perceived by the user.

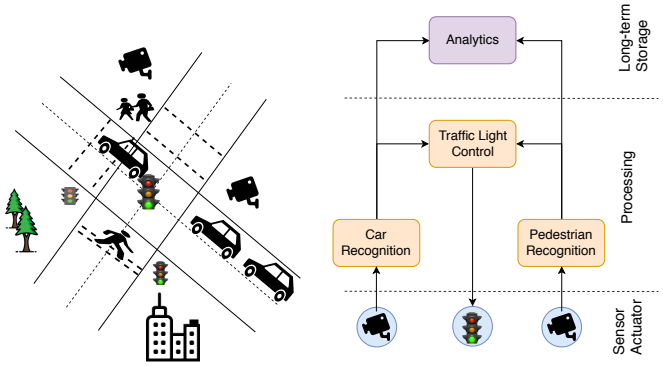

Fig. 2: Smart Traffic Light Application

However, describing precisely the amount of resources required by an application is extremely challenging. In addition, the variability of applications over time makes their specification even harder. In our case, the processing needed by each component is closely related to the pedestrians and cars traffic at that moment. In this context, a new solution is needed to efficiently collect information about the application's requirements, and consequently, fulfill its needs. One potential solution consists in enabling the application to describe its satisfaction level related to the available resources. The used metrics can vary from application to application, such as end-to-end delay, number of received messages, number of missed messages, etc. It is straightforward to see that monitoring an application, a posteriori or in an online manner, is more convenient than describing, a priori, all the possible scenarios and resources the application will need in the future.

2.2 The Reconfiguration

In this section, we describe the main components of the reconfiguration problem that we will tackle later.

Fig. 3 presents a typical scenario addressed in this paper. It depicts the satisfaction metric corresponding to the average end-to-end elapsed time to process the application messages. The experiment includes two phases: ramp-up and online reconfiguration. We assume that all applications arrive during the ramp-up stage and are placed by the

provisioning algorithm. 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 later, 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 paper, we focus on strategies to be applied in the reconfiguration phase. The "Summary Stats" part in Fig. 3 exemplifies the results presented in the next sections, summarizing the average performance in the last hour of the experiment ("Area of Interest").

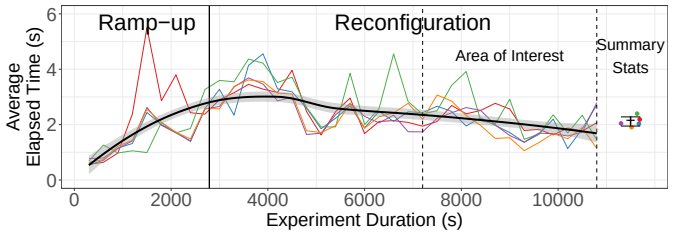

Fig. 3: Performance evolution over time for a reconfiguration strategy. Each colored line represents the aggregated average elapsed time of all applications in a single experiment. The black line is the average performance for this strategy. In Summary Stats, we summarize the average performance, for each experiment (color points) and in general (black), during the area of interest.

The reconfiguration adjusts the placement of applications according to both applications and infrastructure evolution. The reconfiguration process receives as an 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 herein below 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.

3 Related Work

3.1 Online Scheduling

Online scheduling is a vast research domain which aims to optimize the execution of jobs on hosts. 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 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 [3]). Unfortunately, in this context, 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.

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 [4] for example, the authors show that given an algorithm $\mathcal{A}$, which is a ρ-approximation for the makespan, the batch procedure provides a 2ρ-competitive algorithm for the online version.

Some works apply a similar approach to deal with the reconfiguration in the Fog. The authors in [5], [6] and [7] 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 [8] 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 tradicional ILP solver, but it strongly depends on the accuracy of the infrastructure and application models.

In [9], 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 and its quality 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, which allows to apply offline algorithms to the set of currently available jobs. However, IoT applications running in the Fog have a long lifespan and an unstable/unpredictable work size, hindering the use of online scheduling algorithms.

3.2 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 [10], 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 propound in [11]. 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 [12] and [13]. In [13], the auto-scaling mechanism triggers a vertical scaling by adding more CPU and RAM resources to containers running the application. While in [12], 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)[1] problem has received remarkable attention in the last years, with application in many research fields. 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 $R = \sum_{t=1}^{T}(u_t(a^*) - u_t(a_t))$. 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) [14] 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. UCB achieves a regret in the order of $O(\log T)$, which is optimal [14].

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 [15], 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 $O(\sqrt{AT \log A})$ [15].

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

1. 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.

access only to the bandit feedback $f_t(a_t)$. In this context, the authors in [16] 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.

4 Experimental Setup

Fog is a relatively new concept and hence, one of the main challenges when conducting studies is to have a proper experimental setup. To cope with this difficulty, most works rely on simulated environments which may not reflect the complexity of a real Fog environment. In our work, we first designed a realistic experimental environment to perform our reconfiguration studies. In this section, we briefly present how this setup was built. Note that all the traces and scripts used to generate the figures presented in this document are available at: https://gitlab.inria.fr/demourad/tpds-journal.

4.1 The FITOR Platform

The Fog IoT Orchestrator [17], or FITOR for short, is the base for our experimental setup. FITOR's architecture is depicted in Fig. 4 and it is composed of two main parts: i) the physical infrastructure is presented in the bottom of the figure, detailing the machines used and their connectivity; and ii) the Fog IoT Orchestrator in the top, describing the software components.

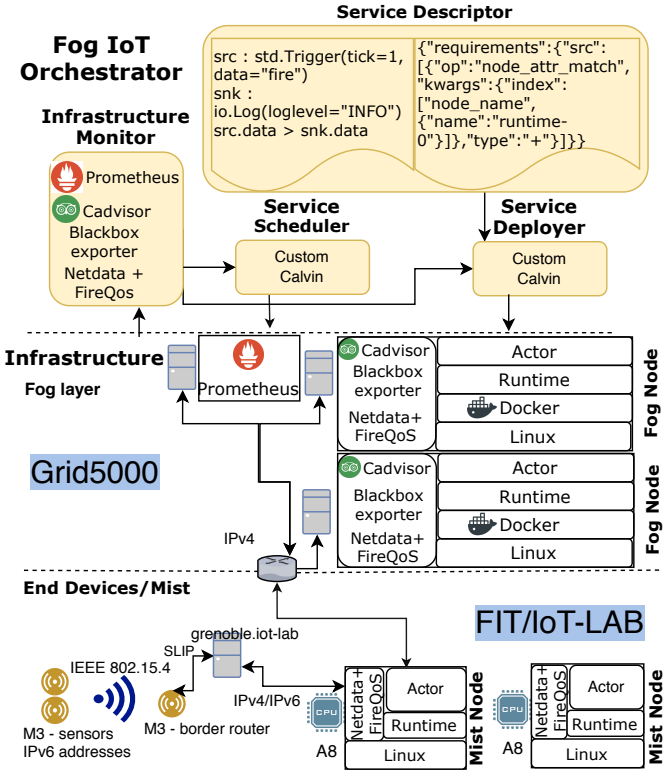

Fig. 4: Fog-IoT Orchestrator Architecture

4.1.1 The Infrastructure

The infrastructure itself is also divided in two parts to be more representative of a real Fog environment: The Fog layer and the End Devices/Mist layer. The Fog layer uses resources from the Grid'5000 platform [18], a large-scale testbed in France whose focus is to allow conducting reproducible experiments on parallel and distributed computing. Grid'5000 comprises a large amount of powerful resources, grouped in homogeneous clusters and dedicated network. In our experiments, we use a subset of available nodes to be our Fog nodes. In each Fog node, the software stack needed for running the IoT applications and for monitoring the node's resources is deployed.

Unfortunately Grid'5000 is not capable of emulating the characteristics present in the edge of the network. So, for the End Devices/Mist layer we had to rely on another platform, called FIT/IoT-LAB [19]. FIT/IoT-LAB is an open platform to perform IoT experiments, providing more than 2000 heterogeneous sensors nodes spread in different sites in France. From these nodes, A8 boards form the Mist layer, on which the software stack (IoT application and monitoring tools) runs directly on bare metal due to their constrained capacities.

4.1.2 The Orchestrator

The Fog IoT Orchestrator is responsible for the lifecycle management of IoT applications in the underlying Fog infrastructure. The upper part of Fig. 4 illustrates the components and the software stack used to build the orchestrator. The **Service Descriptor** enables the description of an IoT application, its building components and its requirements. Following its syntax and the actor and dataflow models, the developer describes the actors, their connections and their requirements in terms of both location and computational effort. Once the description is submitted, the **Service Deployer** handles the initial mapping between the application components and the infrastructure nodes, being responsible for the ramp-up phase as described in Fig. 3. The **Service Scheduler** is the focus of our paper and controls the IoT applications during the reconfiguration phase, as seen in Fig. 3. The scheduler in each host monitors the application's execution, triggering migration actions when necessary to improve application's performance. Finally, the **Infrastructure Monitor** is responsible for sketching out the telemetry information, extracting several resource metrics, such as CPU and RAM utilization, from Fog nodes and sending them to both service scheduler and deployer.

Several tools are used to provide the functionality needed for our experimental setup, such as running the IoT application and monitoring the infrastructure. Among them, a customized version of the Calvin framework [20] is crucial. Calvin is an open source project led by Ericsson that provides a framework for developing IoT applications. Originally, Calvin focuses on the development of IoT applications on the Cloud. In order to use it in our experimental setup, we have extended its different modules (service descriptor, deployer and scheduler) to take into account the specificities of the Fog environment. The source code of the customized version is available at: https://github.com/brunodonassolo/calvin-base/.

4.2 Describing the Environment

In this section, we give insights into the experimental setup used in our tests, and we describe the parameters that characterize both platform, workload and orchestrator. We also present the typical scenario for our experiments and detail its parameters. This scenario is valid, unless explicitly stated otherwise, for the remaining of the paper.

4.2.1 Platform

The platform can be characterized by its: i) **size**, i.e., the total number of nodes in the platform; and ii) **heterogeneity**, representing the diversity of nodes present in the Fog infrastructure. In practice, these parameters are translated to a given number of nodes from both testbeds we use. In our experiments, we use *50 nodes* from FIT/IoT-LAB to represent the sensors, and *17 nodes* from Grid'5000 (2 of which being dedicated to the monitoring and experiment management), forming the Fog layer where applications run.

4.2.2 Workload

The first step when defining the workload for our experiments is deciding the application we will use. Inspired by the use case described in Section 2.1, we opt for a 3-level application, with one agent per level, as illustrated in Fig. 5. 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 an heterogeneous, complex but still manageable workload.

Fig. 5: 3-level Application

The characteristics of these actors allow to describe the applications running on our platform, their heterogeneity and especially their evolution in resource utilization over time. To consider all these parameters, our workload is described by:

- **Application load**: 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 resource consumption, of the applications in the workload. We distinguish two kinds of applications:
  ► *Intensive*: these resource-consuming applications send a large amount of data (*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 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.
- **Satisfaction threshold**: the acceptable end-to-end delay for applications. Above this threshold, the applications are not satisfied with their current placement and should be migrated. In our setup, the threshold is set to *2 seconds*.

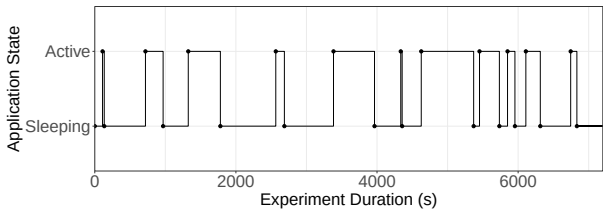

Fig. 6: Workload - Churn. Example of state transitions for an application with mean active/sleeping time ($1/\lambda$) of $300s$.

- **Churn**: defines the evolution over time of applications and consequently their resource usage. We modeled the churn as a 2-state (Active and Sleeping) 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. 6 illustrates the time spent in each state for an application in our setup.
- **System Load**: represents the charge induced by applications to the platform. This parameter is correlated with the platform size and application heterogeneity. We calibrate our setup to have a *heavy* load, where the system is almost saturated. In practice, the *heavy* load represents 50 applications concurrently running on the infrastructure.

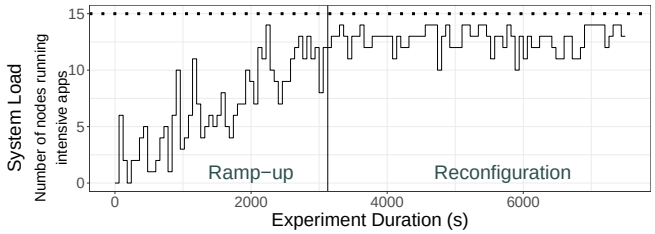

Fig. 7: Workload - System Load (for an experiment). On the y-axis, we present the number of hosts that are close to saturation and cannot run more intensive applications. The dotted line marks the number of hosts in the system (15)

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. 7. Besides, the application's behavior is unpredictable and has an important impact on the overall performance of all applications.

4.2.3 Orchestrator

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 ramp-up phase to decide the initial deployment of applications. In Section 6, we use the *GO-FSP* algorithm as proposed in [21], while in Section 7, applications are distributed to hosts in a round-robin manner. Note that after waiting for the system to stabilize (Figure 3), we did not observe any significant influence of the provisioning strategy on overall performance.
- **Reconfiguration strategy**: describes the algorithm used in the reconfiguration phase to adapt the placement of applications when necessary. Many strategies, inspired from online scheduling and online learning, are evaluated. These strategies are detailed later in Sections 6 and 7.
- **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. In our setup, 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 a obtain a difficult setup.

5 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 1 The Game

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

The game is described in Algorithm 1. 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: $\sum_{j=1}^{J}\sum_{0}^{T} C_t(a_t^j|\pi_t)$. In the rest of the paper, 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. Moreover, since we have two application classes (*intensive* and *calm;* cf. Section 4.2.2), we will split the performance among each class of application when presenting the results in the next sections (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. 3, 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.

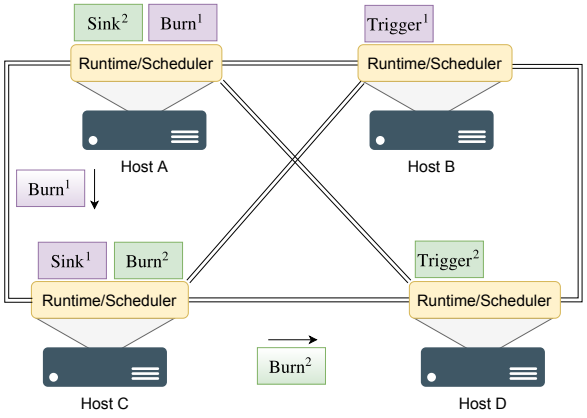

Fig. 8: The Game in Place

In Fig. 8 we detail a simplified game with 2 applications sharing 4 hosts. At a given instant, the host A decides to send

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.

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.

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.

6 Evaluation

In this section 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 applications' behavior, the experiments last for *1 hour* after the initial deployment of all applications (after ramp-up phase on Fig. 3).

6.1 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.

6.1.1 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. 9, 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.

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

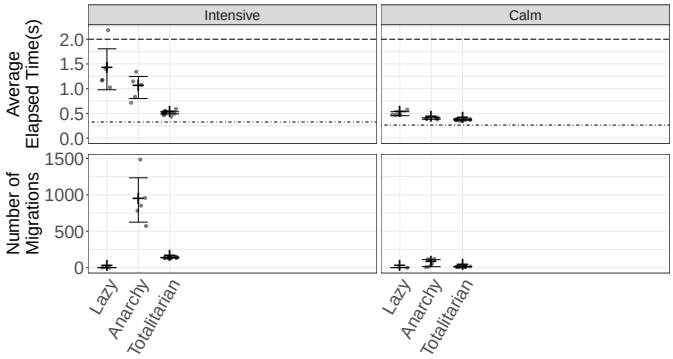

Fig. 9: Performance Evaluation for Baseline Strategies. The $2s$ horizontal dashed line represents the threshold above which applications request migration, while the bottom line represents the minimum response time when operating on a dedicated server. Both Lazy and Anarchy strategies have a poor overall performance.

6.1.2 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. 9 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.

6.1.3 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. 9 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.

6.2 Online Learning Strategies

In this section, we describe the online learning based algorithms used to solve the reconfiguration problem. We evaluate the performance of the algorithms in our 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.

6.2.1 UCB: stochastic

The first learning strategy we explored is UCB (Upper Confidence Bound) which has excellent regret properties in the stochastic case [22]. 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 $\alpha > 2$):

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

where n^j_a is the number of times the action was selected, and where the first term in (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 (1) drives the exploration in the algorithm, indicating the confidence of the algorithm on the current reward for each host.

Equation (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

$$\mu^j_{a,t} = \max\left(0, \frac{e_{max} - e}{e_{max}}\right), \tag{2}$$

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 satisfaction 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[4]. 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. (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 Fig. 10, region A. 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. In conclusion, the results obtained show the unfitness of UCB in our context.

6.2.2 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

$$y^j_{t+1} = y^j_t + \eta \hat{v}^j_t \tag{3a}$$

$$p^j_{t+1} = \Lambda(y^j_{t+1}), \tag{3b}$$

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

$$\hat{v}^j_{a,t} = \begin{cases} \frac{\mu^j_{a,t}}{p^j_{a,t}} & \text{if } a = a^j_t \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

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

The results for EXP3 are presented in region A of Fig. 10 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

4. 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 $|\mathcal{A}^j|$ but this is out of the scope of this paper.

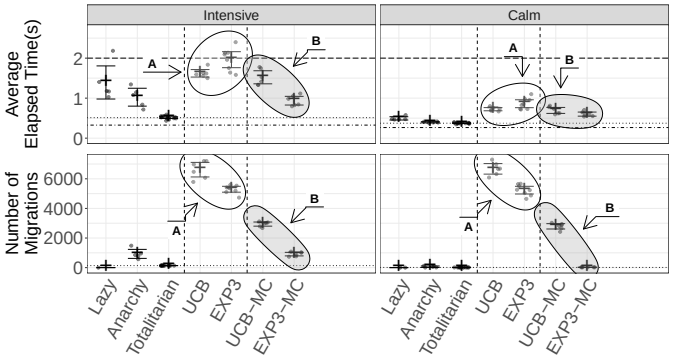

Fig. 10: Performance Evaluation for UCB, EXP3, UCB-MC and EXP3-MC Strategies. This figure complements Fig. 9 by adding the results for these strategies. Note that the y-axis scale has changed and that an horizontal dotted line now indicates the performance of the Totalitarian strategy and serves as a target lower bound.

performance. Consequently, EXP3 has difficulty to learn and to reach the equilibrium in this congestion game.

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.

6.2.3 UCB-MC: migration control

UCB-MC (UCB with Migration-Control) is the implementation of the state-of-the-art UCB2 algorithm[14] whose 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 $O(\log(T))$ [14] 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:

$$a_{t+1}^j = \underset{a \in \mathcal{A}^j}{\arg\max} \left\{ \hat{\mu}_t^j + \sqrt{\frac{(1+\alpha)(1+\ln(t/\tau(r_a^j)))}{2\tau(r_a^j)}} \right\}, \quad (5)$$

where $\alpha = 0.1$[5], r_a^j is the number of epochs played by host a so far and τ is the following exponential function

$$\tau(r) = \lceil (1+\alpha)^r \rceil \quad (6)$$

We can see in Fig. 10, region B, that UCB-MC indeed does fewer migrations compared to UCB. 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 (UCB-MC has the same learning issue as UCB). Consequently, the elapsed time is in the same order of magnitude as for UCB.

5. The α in Eq. (5) has the same status but not exactly the same semantic as the one in Eq. (1), wherefore they have different values.

6.2.4 EXP3-MC: reactive migration control

Unfortunately, the regret for the adversarial case with switching costs is $O(T^{2/3})$ instead of $O(\sqrt{T})$ [23]. 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. 10, 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.

One obvious drawback of such solution is the learning rate. 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.

6.2.5 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 [24] 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. (3), the only change is the feedback vector $\hat{v}$:

$$\hat{v}_{a,t}^j = \begin{cases} \mu_{a,t}^j & \text{if } a = a_t^j \\ f_{est}(a) & \text{otherwise} \end{cases} \quad (7)$$

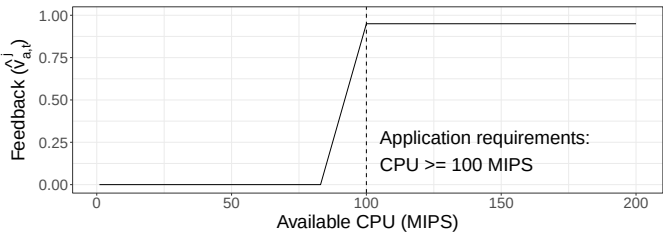

Fig. 11: Estimation function (f_{est}). Shape of f_{est} for an application that requests a CPU with at least 100 MIPS.

Eq. (7) relies on the estimation function f_{est} to provide a good estimation of the application feedback. In summary, as illustrated in Fig. 11, f_{est} accords a good feedback (close

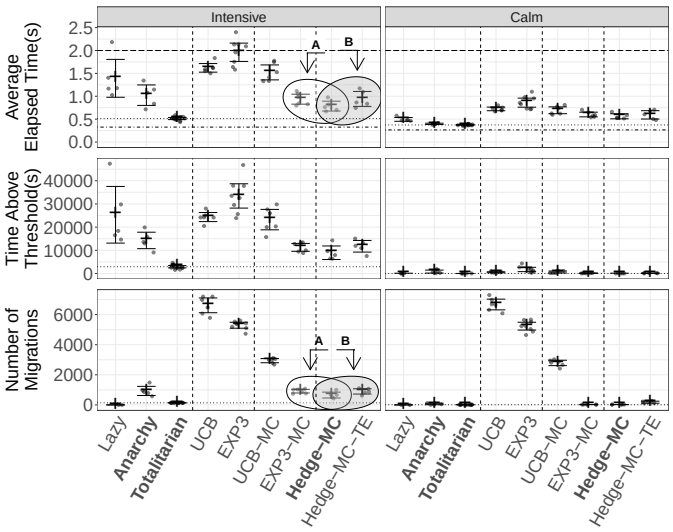

Fig. 12: Performance Evaluation for Hedge-MC and Hedge-MC-TE Strategies. This figure compares all learning strategies. A new metric "Time Above Threshold" is presented. The strategies in bold (Anarchy, Totalitarian and Hedge-MC) have the best performance and will be used as base for comparison in next figures.

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.

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. As a result, we can see in region A of Fig. 12 a slight improvement in terms of elapsed time and number of migrations.

Furthermore, Fig. 12 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 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. This can be seen by the similar performance between EXP3-MC and Hedge-MC.

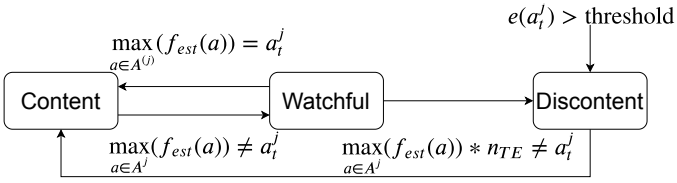

Fig. 13: Hedge-MC-TE: State Machine

6.2.6 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 [25]. 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.

Fig. 13 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. (3). Note the global transition to Discontent which indicates that whenever the application is not satisfied with the current placement, it should be migrated.

Unfortunately, as we can see in Fig. 12, 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.

6.3 Greedy but Informed Strategies

The algorithms presented in Section 6.2 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 [21] to select the best host to run the application.

6.3.1 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

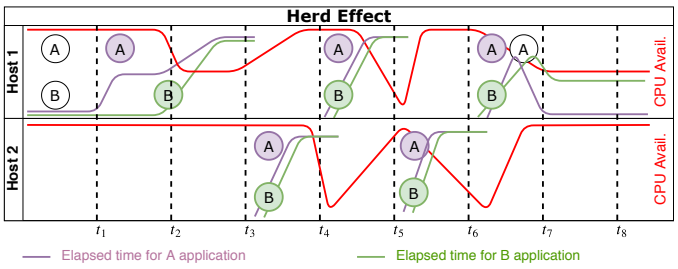

Fig. 14: The Herd Effect

time. Fig. 14 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.

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. 15 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 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.

6.3.2 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

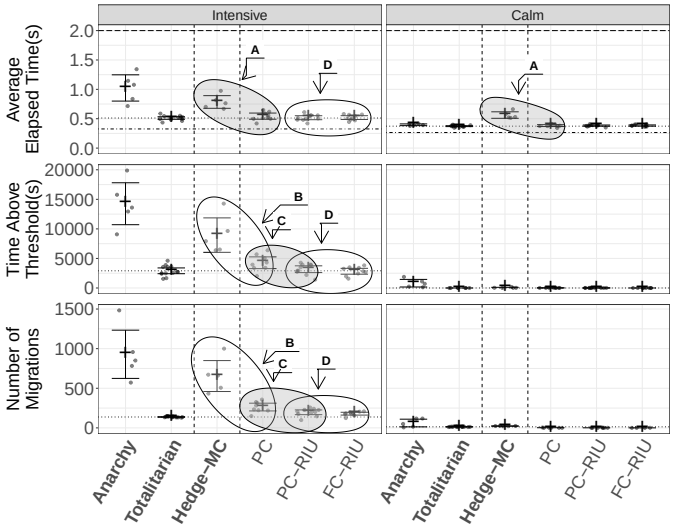

Fig. 15: Performance Evaluation for Greedy Strategies. This figure compares the performance of greedy strategies with those of learning in Fig. 12. The strategies in bold (Anarchy, Totalitarian and Hedge-MC) are kept as a basis for comparison. Note that the y-axis scale has changed.

.

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. 15, 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. 15).

6.3.3 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. 15, 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.

6.4 Summary

	Class	Mode	Information	Coord.	Perf.
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 1: Strategies Classification. The performance column summarizes the average elapsed time (in seconds) for intensive applications. (🟢 $\leq 0.75s$, $0.75s <$ 🟠 $\leq 1s$, 🔴 $> 1s$)

Table 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 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.

7 Evaluation: Affinity Scenario

In Section 6, 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 the description of resource utilization is inaccurate. We will see how incorrect information can affect the strategy's performance and compare this affinity scenario with the previous non-affinity one.

7.1 Experiment Description

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.

The experiment follows the description provided in Section 4.2, with a few exceptions listed below:

- **Duration**: each experiment lasts for *2 hours* after the initial provisioning.
- **Application load**: although the overall charge of the system remains the same, the application load changes: i) *Calm*: *0.5 message/s* where each message consumes *20 MI* (or *1 MI* if running on an optimized host); ii) *Intensive*: *1 message/s* and *150 MI* per message (or *7.5 MI* if running on an optimized host).

7.2 Baseline Strategies

We start our analysis by inspecting the results for the baseline strategies in Fig. 16. 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 the results, as we can see in the region A of Fig. 16. 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 6.3.1, 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.

7.3 Online Learning Strategies

7.3.1 UCB

The performance for learning strategies is presented in Fig. 16. Comparing this result with those obtained in Fig. 12, 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.

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

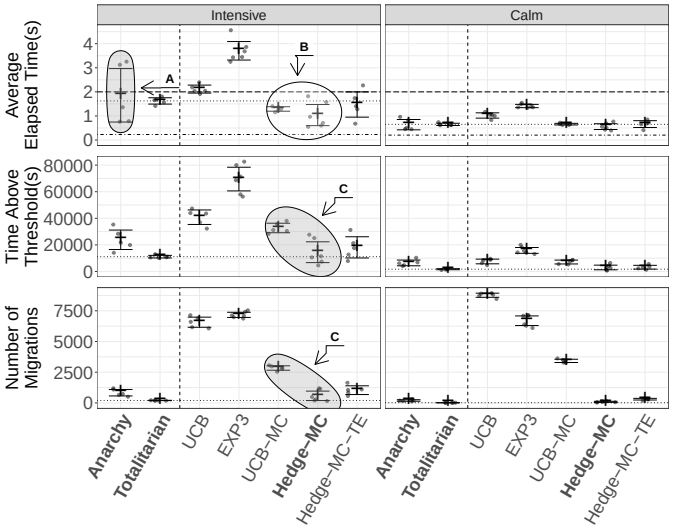

Fig. 16: Performance Evaluation for Learning Strategies in the Affinity Scenario. This figure compares all learning strategies. The strategies in bold (Anarchy, Totalitarian and Hedge-MC) have the best performance and will be used as base for comparison in Fig. 17.

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.

7.3.2 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.

7.3.3 UCB-MC

The effectiveness of UCB-MC can be seen in Fig. 16. 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.

7.3.4 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 performance 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. 16 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.

Nevertheless, the learning capacity of Hedge-MC in the affinity is worse than in the non-affinity scenario. 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.

7.3.5 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.

7.3.6 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 7.4.

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.

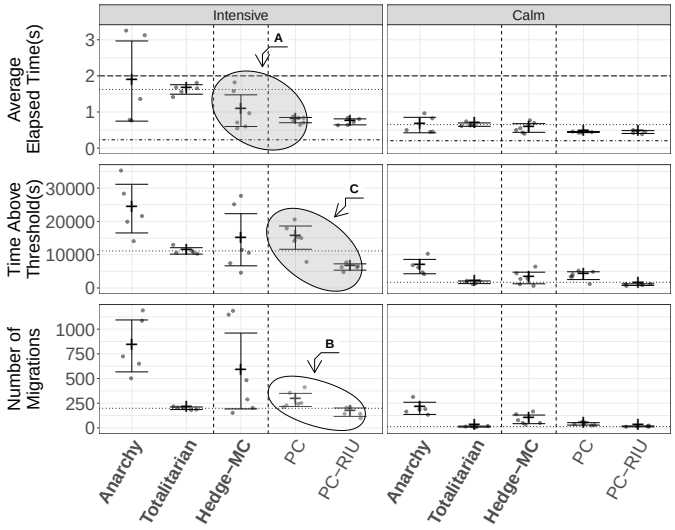

Fig. 17: Performance Evaluation for Greedy Strategies in the Affinity Scenario. This figure compares the performance of greedy strategies with those of learning in Fig. 16. The strategies in bold (Anarchy, Totalitarian and Hedge-MC) are kept as a basis for comparison. Note that the y-axis scale has changed.

.

7.4 Greedy Strategies

In Fig. 17, 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 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.

8 Conclusion

In this paper 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 kind 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 recompute 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.

We identify several important perspectives that should be studied in future work. The first 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. A study to determine online how to scale the number of resources and what is the optimal number of replicas for each application would thus be quite instructive. Secondly, we also think that, different and more complex workloads should be considered to gain more insights on the efficiency of 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. Finally, the evaluation methodology is another point that deserves 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 to conduct realistic evaluations in large-scale scenarios.

Acknowledgements

This work has been done in the context of the Inria/Orange joint laboratory. Experiments presented in this paper were

carried out using the Grid'5000 and the FIT/IoT-lab testbeds, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr and https://www.iot-lab.info/).

REFERENCES

[1] F. Bonomi, R. Milito, J. Zhu, and S. 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*. New York, NY, USA: ACM, 2012, pp. 13–16.

[2] 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)*, Sep. 2016, pp. 1–6.

[3] M. Drozdowski, *Scheduling for Parallel Processing*, 1st ed. Springer Publishing Company, 2009, ch. 5.

[4] D. B. Shmoys, J. Wein, and D. P. Williamson, "Scheduling parallel machines on-line," *SIAM Journal of Computing*, vol. 24, no. 6, pp. 1313–1331, 1995.

[5] 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.

[6] O. Skarlat, M. Nardelli, S. Schulte, M. Borkowski, and P. Leitner, "Optimized iot service placement in the fog," *Service Oriented Computing and Applications*, vol. 11, no. 4, pp. 427–443, Dec 2017. [Online]. Available: https://doi.org/10.1007/s11761-017-0219-8

[7] A. Yousefpour, A. Patil, G. Ishigaki, I. Kim, X. Wang, H. C. Cankaya, Q. Zhang, W. Xie, and J. P. Jue, "Qos-aware dynamic fog service provisioning," *CoRR*, vol. abs/1802.00800, 2018. [Online]. Available: http://arxiv.org/abs/1802.00800

[8] 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.

[9] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. K. Leung, "Dynamic service placement for mobile micro-clouds with predicted future costs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1002–1016, April 2017.

[10] X. Sun and N. Ansari, "Edgeiot: Mobile edge computing for the internet of things," *IEEE Communications Magazine*, vol. 54, no. 12, pp. 22–29, December 2016.

[11] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. 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*, ser. DEBS '16. New York, NY, USA: ACM, 2016, pp. 258–269. [Online]. Available: http://doi.acm.org/10.1145/2933267.2933317

[12] "K3s: Kightweight kubernetes," https://k3s.io/, accessed: 2020-02-27.

[13] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos, "Enorm: A framework for edge node resource management," *IEEE Transactions on Services Computing*, pp. 1–1, 2018.

[14] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, pp. 235–256, 05 2002.

[15] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "The nonstochastic multiarmed bandit problem," *SIAM Journal on Computing*, vol. 32, no. 1, pp. 48–77, 2002. [Online]. Available: https://doi.org/10.1137/S0097539701398375

[16] T. Chen and G. B. Giannakis, "Bandit convex optimization for scalable and dynamic iot management," *IEEE Internet of Things Journal*, vol. 6, no. 1, pp. 1276–1286, Feb 2019.

[17] 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)*, Jan 2019, pp. 1–6.

[18] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, L. Niclausse, Nicolas an d Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science. Springer International Publishing, 2013, vol. 367, pp. 3–20.

[19] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. P.-G. et, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne, "Fit iot-lab: A large scale open experimental iot testbed," in *IEEE World Forum on Internet of Things*, Dec 2015, pp. 459–464.

[20] P. Persson and O. Angelsmark, "Calvin – merging cloud and iot," *Procedia Computer Science*, vol. 52, pp. 210 – 217, 2015, the 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015).

[21] 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)*, May 2019, pp. 1–7.

[22] E. V. Belmega, P. Mertikopoulos, R. Negrel, and L. Sanguinetti, "Online convex optimization and no-regret learning: Algorithms, guarantees and applications," *CoRR*, vol. abs/1804.04529, 2018. [Online]. Available: http://arxiv.org/abs/1804.04529

[23] N. Cesa-Bianchi, O. Dekel, and O. Shamir, "Online learning with switching costs and other adaptive adversaries," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'13. Red Hook, NY, USA: Curran Associates Inc., 2013, p. 1160–1168.

[24] 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.

[25] H. P. Young, "Learning by trial and error," *Games and Economic Behavior*, vol. 65, no. 2, pp. 626 – 643, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0899825608000614

Bruno Donassolo is a research engineer at Orange Labs and a PhD candidate at University Grenoble-Alpes. His research is on reconfiguration strategies in fog environments. He obtained his M.S. and Engineering degree from both Federal University of Rio Grande do Sul and University Grenoble-Alpes in 2011 and 2007, respectively. He worked as a software engineer at Datacom/Brazil from 2011 to 2017.

Arnaud Legrand is a senior CNRS researcher at University Grenoble-Alpes since 2004. His research interests encompass the study of large scale distributed systems, theoretical tools (scheduling, combinatorial optimization, and game theory), and performance evaluation, in particular through simulation. He obtained his M.S. and Ph.D. in computer science from the Ecole Normale Supérieure de Lyon, France in 2000 and 2003, and his Habilitation Thesis in 2015 from University Grenoble-Alpes.

Panayotis Mertikopoulos is a senior CNRS researcher at University Grenoble-Alpes since 2011. His research interests lies at the he interface of game theory, learning and optimization, with a special view towards their applications to networks, signal processing, and machine learning. He obtained his M.S. in mathematics in 2005 fro Brown University, his Ph.D. in computer science from University of Athens in 2010, and his Habilitation Thesis in 2019 from University Grenoble-Alpes.

Ilhem Fajjari is a tenured research project leader on cloud-native network function orchestration in Orange Labs. Her main research interests include cloud, network function virtualization, orchestration and optimization of communication networks. She obtained her PhD in computer science from Pierre & Marie Curie university (Paris 6) in 2012.